Escape Simulation Suite

Thomas Y Merrell

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Engineering

Dr. Pushkin Kachroo,Chair
Dr. A Lynn Abbott
Dr. Binoy Ravindran
Dr. M . P. Singh

April 15, 2005
Blacksburg Virginia

Keywords: simulation, real-time, 3D graphics, networking, Win32, DirectX, dynamically linked library (DLL)

Escape Simulation Suite

Thomas Y. Merrell

Abstract

Ever since we were children the phrase “In case of an emergency, walk, DON’T run, to the nearest exit” has been drilled into our heads. How to evacuate a large number of people from a given area as quickly and safely as possible has been a question of great importance since the first congregation of man; a question that has yet to be optimally answered. There have been many attempts at finding an answer and many more yet to be made.

In light of recent world events, 9/11 for instance, the need for a better answer is apparent. While finding a solution to this problem is the end objective, the goal of this thesis is to develop an application or tool that will aid in the search of an answer to this problem.

There are several aspects of traditional evacuation plans that make them inherently suboptimal. First among these is that they are static by nature. When a building is designed, there is some care taken in analyzing its floor plan and finding an optimal evacuation route for everyone. These plans are made under several assumptions and with the obvious constant that they cannot be modified during the actual emergency. Yes, it is possible for such a plan to actually end up being the optimal plan during any given evacuation, but the likelihood of this being the case is most definitely less then 100%. There are many reasons for this. The most obvious is this: the situation that the plan is trying to solve is a very dynamic one. People will not be where they should be or in the quantities that the static plan was prepared for. Many of them will probably not know what they should do in an emergency and so most likely will follow any large group of people, like lemmings. Finally, most situations that require the evacuation of a building or area occur because all or part of the building has become, or is becoming, unsafe. It is impossible for a static evacuation plan to take into account the way a fire or poisonous gas is spreading, or the state of the structural stability of the building.

What is needed during a crisis is an artificially intelligent and dynamic evacuation system that is capable of (1) analyzing the state of the building and its occupants, (2) coming up with a plan to get everyone out as fast as possible, and (3) directing all occupants along the best exit routes. Furthermore, the system should be able to modify its plan as the evacuation progresses.

This application is intended to provide researchers in this area the means to quickly and accurately simulate different evacuation theories and ideas. That being the case, it will have powerful graphical capabilities, thus allowing the researchers to easily see the real-time results of their work. It will be able to use diverse modeling techniques in order to

handle the many different ways of approaching this problem. It will provide a simple way for equations and mathematical models to be entered which can affect the behavior of most aspects of the world being simulated. This work is in conjunction with, and closely tied to, Dr Pushkin Kachroo's research on this same topic.

The application is designed so that future developers can quickly add to and modify its design to meet their specifications. It is not the goal of this work to provide an application that directly solves the optimal evacuation problem, or one that inherently simulates everything perfectly. It is the job of the researchers using this application to define the specific physics equations and models for each component of the simulation. This application provides an easy way to add these definitions into the simulation calculations.

In brief, this Escape Simulator is a client server application. All of the graphics and human interaction are handled client side using Win32 and Direct3D. The actual simulation world calculations are handled server side, and both the client and server communicate via DirectPlay. The algorithm being used to model the objects and world by the server will be completely configurable. In fact, everything in the world, including the world physics, will be completely modifiable. Though the researchers will need to write the necessary pluggins that to define the actual models and algorithms used by the agents, objects, and world, ultimately this will give them much more power and flexibility. It will also allow for third parties to develop libraries of commonly used algorithms and resources that the researchers can use.

Acknowledgments

The author would like to thank Dr. Kachroo for his unending encouragement and support throughout the research process, and also for his work in the area of evacuation which in large part was the motivation and inspiration for the author's initial choice to pursue this thesis.

The author would like to thank Dr. Lynn Abbott and Dr. Binoy Ravindran for participating in the advisory committee for this thesis.

The author would like to also thank Greg Dean, Ryan Wilson, and Jad Meouchy for their independent research studies and help.

Finally, the author would like to thank his wife, whose support and encouragement were instrumental in the completion of this thesis.

Table of Contents

Table of Figures

Introduction

Computer simulation plays a critical role in the development of any large or important project. This is especially true when human life is concerned. Long before a new aircraft design is ever built and tested by professional pilots and wind tunnels, the model has undergone thousands of hours of computer simulation and testing. Long before the latest model of a car hits the pavement; it has already logged thousands, if not millions, of miles along computerized tracks and streets.

Buildings are no exception to this rule. Whenever a large structure is to be erected for general public use it must provide its occupants a quick and safe way to evacuate in the event of an emergency. According to [1] this has in large part been achieved in one of two ways.

First is the strict adherence to established building codes that have been designed to ensure, for the general case, safety in an evacuation. Unfortunately these codes cannot take into account every possible building structure, nor the constant changes and improvements in architecture. Finally, they cannot take into account the possible worst case and dynamic situations that can happen in any emergency.

The second method comes closer to achieving the desired goal of ensuring safety in an emergency. This method is computer simulation. Since introduction of simulating building designs into the design process of a structure, it has quickly become an essential and important part of any large-scale building project. This is for many reasons, money not being least of which. It costs quite a bit less to move a stairwell in a computer generated world than in the real world. Computer simulation of a building's structure in an evacuation situation has become so important that there have been a very large number of simulators that have already been developed and are in use. At the time [1] was written there were over 22 models either in use or being developed.

There are several critical differences between this application and those mentioned in [1]. First, each of the models described were only interested in improving the actual structure of the buildings being designed in order to offer the safest and quickest paths for the occupants in case of an emergency. None were interested in developing a dynamic feedback driven system that not only monitors occupants in real-time during an emergency, but actually directs the occupants and possibly controls the building's automated structures, such as doors and ducting, to get all the occupants out in the least time possible. Nor are they capable of adding such a system easily.

Second, all of the above applications used statically defined modeling and algorithms, thus restricting their abilities.

1.1 Common application development issues

1.1.1 Optimization problem

As with almost anything in life, "you can't have it all" in computer programming. There are no sports cars in programming, small and fast. Likewise, there are no universal tools, infinitely flexible with great performance. When programming, you have to make tradeoffs. On one hand you can make your code incredibly fast, but to do so you have to increase your memory usage. In order to decrease the memory requirement of your program you have to add many extra and often inefficient steps to your program's algorithm to compensate for the lack of ready information. Similarly, if you want a program that takes into account all possibilities, you cannot expect it to make decisions or react very fast.

So the question is, where is that "happy medium"? There are many methods to aid designers in finding the right answer to this question. Some of the best involve getting out a pencil and good old pad of engineering paper and writing/drawing. For the rest of this section, background information will be given on many of the different issues and considerations that were addressed in designing this application.

1.1.2 Language

Though it may seem a trivial thing, which programming language to use is an important consideration. The language one uses can have a major impact on the different optimization issues raised in the first section. There are plusses and minuses for each.

Because of the original specifications for this application, something fast and maintainable was needed. Of the different programming languages in use, the most maintainable ones fall under the object-oriented group. There are several different languages that fall under this category, C++, Java, J#, C#… From this list most of these languages cannot be considered because of their speed. This is because they are mainly "interpreted" languages, meaning they do not compile down to byte code that is run directly by the computer. Instead, they are compiled to intermediate code that is "interpreted" and run by some other application. Such languages are for the most part platform independent, but such freedom comes at the cost of extra code and slower execution time.

From the list that remains, C++ is one of the best options as it is, in general, very fast. Even though C is faster and hand written assembly faster still, assuming that the programmer writes good code, C++ was designed to allow pure C code and assembly to be interjected at time critical points to allow those points to be optimized.

Though the majority of the code written for the Escape Simulator will be C++, it is strongly recommended that a reader not familiar with C briefly look over [2] as many of the commonly called C++ calls actually call the C equivalent, and it would be a good idea to understand what those functions or methods are actually doing.

1.1.3 Object oriented code

In a nutshell, object oriented code is, as its name implies, code that has been divided up into different functional groups or objects. These groups are designed so that they can be easily connected together in order to add to the overall functionality of the application.

For example: A modern watch can be made up of several components. It may or may not have a band; it may or may not have hands; it may or may not have a stopwatch; and it may or may not have something that tells you the current day of the month. Each of these things by themselves do not constitute a watch, and you do not need all of them to have a watch, but each has a distinct job to do that is unique and, for the most part, independent of the other components. If your watch's band breaks, your watch will still tell time.

There are several benefits to using object-oriented code. First, by breaking the application into its functional components it allows for easy creation and maintenance of each part. Each component can be developed independent of the other components. If one component is broken, the other components can still function while the broken component is fixed. Upgrades can also be made without breaking the overall application. For instance, if you create a graphics application that uses a DirectX interface to handle all the rendering, and then wish to change your rendering to OpenGL, this change is just a matter of writing a new rendering Object with the same functionality as the DirectX one, but in OpenGL. As far as the rest of the application is concerned, nothing has changed even though you are now using a completely new graphics interface.

There is one downside to object-oriented code. In general, it is not as efficient as straight liner code. However, modern compilers are getting much better at creating machine code that is closer to the linear runtime equivalent.

For a project as large as this, using an object-oriented approach is a must. It would be possible to make a linear version, but the code would be insanely complicated and inefficient! The big question is how to divide the different specifications for this application into appropriate objects.

There are some obvious places where the code can be divided. One component will handle the graphics. Another component will keep track of the world's state and calculate whatever changes happen as a simulation progresses. A third component will directly interact with the user. If the application is networked it will need a component of code to handle the network communication. It will need to be able to read and write information to a file….

For those not familiar with object-oriented coding and C++ please refer to [3]

1.1.4 Multithreading

Computers are linear by nature. They lack the ability to truly multitask. However, computers also are very fast, at least modern ones. Because of this, modern operating systems are able to fake multitasking. To do this they use time division multiplexing, or

more simply put, they jump between each of the running applications, or processes, giving each a short chunk of time, or time slice, in order to get the work done. Because they jump around so fast it seems to us as if they are actually doing many tasks at once.

A program is made up of a series of instructions that are executed in a linear fashion. This path of execution has been nick-named a "thread". By default there is initially one thread created for each process. For many linear applications it is sufficient to only use this single thread, but there are instances where it is very beneficial to create multiple threads of execution in an application. When multiple threads are created within a process, the operating system treats them as individual tasks within the process, and gives each a portion of the application's original time slice. Again, this makes it appear as if the program is doing many things at once.

A good example of multithreading, is a server which needs to handle several different client requests at once. If the server were singly threaded it would only be able to handle one request at a time on a first come first serve basis. All other clients would have to wait for the server to finish handling the client in front of it. This would be similar to waiting in line at a grocery store with only one checkout. On the other hand, if the server were multithreaded it would dedicate the main thread to looking for clients requesting a connection. When it received one it would spawn another thread to handle the request while it went back to looking for more requests. This way there is no line and all the clients can be handled at the same time.

As you might suspect, multithreading is a very powerful and useful tool when used correctly. There are several things that one must consider when deciding to make an application multithreaded.

- Is the benefit seen from using multiple threads greater then the overhead created by the operating system switching between the different threads?

- Is it possible to ensure that the mutual data used by the different threads will stay safe and there will be no mutual exclusion problems?

- How time-critical is each thread? If a thread must be executed for a specified amount of time at a certain frequency, then multithreading might not be a possibility because there are no guarantees as to how much of each time slice each thread gets.

- Finally, the programmer must ensure that there are no "deadly embraces" in his/her multithreaded code. This is when one thread locks up a resource that another thread needs, but the other thread also locks up a resource that the first thread needs, so they both sit and wait forever for their needed resource to become free. Another way this happens is if one thread uses a resource and forgets to free it once done. From that point on no one can use this resource. It is highly likely that eventually some or all of the threads will halt progression and just sit waiting on the un-freed resource.

Multithreading makes a program in some ways much more complicated, and in others far simpler. It is up to the designer to decide whether or not it is warranted.

In the Evacuation Simulator multithreading is used sparingly. The majority of the code is linear execution. Only the network component is fully multithreaded.

1.1.5 Client/Server vs. single application

Many projects require that the workload and functionality be divided between multiple applications or computers. A good example of this is a simple FTP program. One single application by itself is not enough to provide the functionality of an FTP program. In order to perform an FTP transaction there must be at least two applications running, each on a separate computer. The first is a "server" which resides on the computer that contains the desired file. This application's job is to listen for other applications that want to have a file transferred to them, and when a request is received, to send the file. The second is a "client" which resides on the computer requesting the file. It is his job to connect to the server, request the file, and handle it when it is received. This type of structure is called a "client/server" model.

Another reason for separating a project into two applications is to divide the computation load between two processors. Often times there are several segments of code that are very computationally intensive. Running all these segments on a single computer can very easily bring a single processor to its knees, and obviously result in very poor overall application performance. If the workload could be distributed between two or more processors then things likely would run more smoothly. A programmer needs only to decide how best to divide the workload. However, it might turn out that it is impossible to divide the load in a way that would allow for significant benefits.

Aside from the advantages noted above, there are also disadvantages with distributing an application's workload across multiple processors. There is a significant amount of overhead associated with setting up network communication. There is also overhead associated with the type of protocol chosen. Some protocols are worse than others. Finally, there is the network reliability, bandwidth, and throughput to consider. All of these issues are ones for which there is no "right" answer. Part of deciding whether or not to network an application requires the designer to address each of these issues.

Depending on the type of protocol being used, there may or may not be a way to greatly reduce the overhead associated with establishing a connection, but you can reduce the times it is required to establish or reestablish a connection.

Information that is sent back and forth over a network is grouped into packets. An application might have to send several packets over a network in order to send one large chunk of data, or it might only need to send one packet for several small chunks. With each packet that is sent there is a certain amount of extra information that must be added to the application data. This extra information is used to properly route the packet from

the sender to receiver. Along with this required information there is more overhead that may or may not be required depending on the protocol being used.

Two of the most used protocols are TCP and UDP. As always, both have their advantages and disadvantages. TCP guarantees the data will be both intact and in order if it arrives at its end destination. Unfortunately this requires that TCP make a connection with the receiving application. Also, the TCP header that is added to each packet increases the packet overhead. Finally, it can take longer for a TCP packet to completely arrive at its destination. UDP, on the other hand, is very fast. It requires almost no extra packet overhead, and it does not require a connection to be established between the two applications. However, there are no guarantees on either the data integrity or order upon arrival. If integrity and order are important then it is up to the receiving application to find and fix any problems.

1.1.6 Graphics

1.1.6.1 Basic 3D graphics information

There are several topics that are more or less universal no matter which graphics API one uses, even though the individual API's handle them differently. This section tries to address many of these topics. Afterwards, several API's available for windows programmers are discussed.

1.1.6.1.1 Views

Views deal with what the user sees, meaning even though there may be many different objects defined in a world, only the ones that the user is looking at are actually seen. Also, a programmer can create several different views, assigning a different camera or even a completely different rendering to each, and have each be rendered to a different part of the render window.

1.1.6.1.2 Coordinates

This is a big one. Within a graphics engine there are usually many different types of coordinates defined and used. A single object commonly has two to three different sets of coordinates that define its location and orientation relative to the world, other objects, etc. There are many different types of coordinates that are used in the Escape Simulation Suite. DirectX has its own typedefs, as does Win32. Finally, the coordinate information can be stored in many different forms. For instance: Cartesian – standard x,y,z Polar – r,?Rcylindrical – r,?Rz and spherical – r,?RR Each is interchangeable, meaning that the same data is stored in each, but each encoding has different benefits and drawbacks.

1.1.6.1.3 Matrices

Matrices and basic linear algebra are essential for designing a fast and efficient graphics program. One of the Escape Simulator engines is dedicated totally to math. In this engine there is a component dedicated to matrices and linear algebra. If the reader wants more information on using this component please see the documentation written on the math engine included on the Escape Simulation Suite Website (please see the appendix for the address). For the most part, though, the graphics engine uses the D3DXMATRIX

class. The reason for this is that many of the Direct3D function calls require this type of matrix and there are a significant number of method and optimized helper functions that have already been written for this type of matrix. One important difference between the math engine and the D3DXMATRIX class is the math engine can handle matrices of any given size and shape, whereas D3DXMARIX classes and the associated functions are only 3x3, i.e.:

$$\begin{bmatrix} \_11 & \_12 & \_13 \\ \_21 & \_22 & \_23 \\ \_31 & \_32 & \_33 \end{bmatrix}$$

1.1.6.1.4 Vertices, primitives and meshes

A vertex is one point in 2D or 3D space. It can have different properties associated with it, such as color, texture mapping, normal, etc. A primitive is usually made up of three or more vertices. Many primitives are combined together to form a mesh.

1.1.6.1.5 Sprites

A sprite is a unique type of primitive. It only has a single vertex associated with it. This allows for much faster rendering by the graphics card.

1.1.6.1.6 Textures

A texture is a 2D picture that is stretched across one or more primitives. Textures aid significantly in creating the illusion that an object is real. Not only can a texture be used to give a primitive color, but it can also be used to modify the texture a primitive appears to have. There are some very impressive results that can be achieved by using advanced texture mapping.

There are many more topics that could be introduced under this heading, but these should be sufficient.

1.1.6.2 Win32

Win32 in and of itself is not really a graphics API. Rather, it is the lowest level of windows API calls that can be used to generate windows and everything associated with them. That being the case, there are certain simple graphics calls that are available to the Win32 programmer. These calls are all grouped into what is known as the GDI. Using GDI calls, a programmer can change any pixel within a window to any color desired, draw primitives, render pictures, and accomplish many other basic things. Along with this, Win32 is very good at handling the graphics involved with rendering menus and toolbars. All that being said, Win32 does not come anywhere close to the speed and performance of other graphics API's. This is partially because it was never intended to provide a complete rendering solution. The GDI calls it supports are just for doing quick and dirty projects, not high end work. The main reason it cannot compare is because it cannot take advantage of the hardware acceleration that exists on most modern graphics cards. If a developer wants to utilize the full capabilities of his/her graphics card they will have to use something else.

Another benefit of using Win32 is the common look and feel of applications created in Win32. When a programmer wants to have menus and tool bars, as are common in many applications today, Win32 is a good option.

1.1.6.3 OpenGL

OpenGL was written a while back to specifically answer the problem of not being able to access hardware acceleration in a standard way. In the past, every application developer had to rewrite the graphic portion of his/her code for each type of graphics card he/she decided to support. If a new card came out, the application would not be able to use the new hardware acceleration until a new component was written to access it. Obviously this was the cause of many sleepless nights and frustration for many a graphics programmer.

What OpenGL and DirectX did was provide a common interface for the programmer. No matter what graphics card was being used, all he/she had to do was write one graphics component in OpenGL or DirectX and, provided the card's drives supported OpenGL and DirectX, the application would work fine.

One of the benefits of OpenGL is it has been designed to be cross platform. An application written in OpenGL can be compiled and run for either Windows or Linux. Performance-wise it used to be far better than DirectX, but as of the last three to four years they have become virtually equivalent. One of the places it falls short, though, is it can only handle graphics, rather than all types of multimedia.

1.1.6.4 DirectX

Where DirectX excels over OpenGL is as follows. It is made up of several components that each handle different aspects of multimedia. Some of these components are DirectDraw for 2D drawing, Direct3D for 3D rendering, DirectSound for sounds, DirectPlay for networking, and DirectInput for all types of user input. Each of these components have the ability to provide a standard API to the different interfaces allowing the programmer to directly access the hardware accelerators for each device regardless of make or model. Any functionality that is missing from the hardware is faked in software, so the programmer does not need to worry as much about missing functionality.

The problem with DirectX is that it only works on Windows. So any application that uses it is restricted to Windows only.

1.1.6.5 DirectX/Win32 hybrid

In the end the choice was made to go with a DirectX/Win32 hybrid. This is because the application needed to have menus and tool bars that would look and feel intuitive, but at the same time it needed to be able to access the hardware acceleration capabilities of the hosting computer. DirectX was chosen instead of OpenGL because it could be used for all of the following areas: multimedia, I/O, networking, sound, and graphics. Choosing OpenGL would have necessitated still other API's to handle the other non-graphical

components listed above, and the only advantage would have been that the graphics would now be portable to Linux. Even with this there are certain design issues that arise from this hybrid that will be discussed in a later chapter.

1.1.7 User interface and input handling

The user interface (UI) of a program can be as important to the usefulness of an application as the internal functionality. There are many aspects that must be considered in designing a good UI. What are the input and output types required? What interfaces would be best suited for each? What should the overall layout and presentation be? Too many options can be just as bad as too few.

Under the category of input/output types you have such things as command line consoles, windows, and rendering pages. Command line consoles are useful for linear applications where there is very little user input required, and where the program's output will be fairly linear and straightforward. Windows, on the other hand, are very useful for nonlinear programs where menus and toolbars are needed and where the output can be moderately complex and nonlinear. Windows are also good for outputting the status of applications which do multitasking and parallel processes. Windows applications, however, typically fall short when complex graphics rendering is required. In such situations a full blown page rendering API is required, such as the DirectX or OpenGL, which were discussed above.

There are many devices which are categorized as input peripherals, though they generally fall under three headings: keyboard devices, mice, and game-pads/joysticks. Each of the input/output types above lends itself more or less to a specific input peripheral. Command line consoles are ideally used with keyboard peripherals. Windows work well with both keyboard and mouse input. DirectX and OpenGL applications typically are used for keyboard and game-pad/joystick input, and can be used for mouse input with a little amount of extra work.

This application uses a simple command line console for the server side component. This is because the server, for the most part, requires very little direct user input, and receives most of its commands via the network communication to the client. As far as outputs are concerned, the server requires only simple basic status messages to be reported to the user.

The client, on the other hand, requires multiple types of input from multiple peripherals and reports back through many different types of output. There are special dialogue boxes which provide for strictly text based output to report on the client's behavior and status. There is a basic menu, toolbars, and dialog box structure which allows the user to easily modify the behavior and properties of the application. Finally there is a DirectX page being rendered within the window's client area. The menu, toolbars, and dialog boxes will accept both mouse and keyboard input. The rendering page accepts keyboard, game pad/joystick, and mouse input.

Once again, from a user interface standpoint, the server is implemented as a command console, and the client as the Win32/DirectX hybrid

1.1.8 Application types

Although they have already been presented to some degree in the sections above, the different types of possible applications that were considered for this program should be addressed in a little bit more depth.

1.1.8.1 Console

Among the most basic, and most widely used, programs is the command console application. It supports the most basic input and output types and peripherals, and adds very little overhead to run the application code. It is also the simplest to set up and code for. Such an interface is ideal for linear applications, and also is very good for multithreaded applications where minimal user output is required.

1.1.8.2 Win32

Within windows there are two main types of windows applications. The first is a pure Win32 application. Win32 is the underlying API upon which windows applications are built. It allows the user to be able to create windows with menus, toolbars, buttons, dialogues, text output, and draw basic graphics.

A Win32 application requires a decent amount of extra code in order to get even the most basic application running. It also adds some overhead to the overall execution of the code.

Win32 applications are by nature nonlinear. They require a callback function that the operating system calls whenever a message is sent directly to the application. Within the callback function the different messages must be either handled or passed along to the default system handler. The windows messages can originate from many sources. They can be generated from the operating system itself when the application is created, destroyed, or its status is changed, such as the window being minimized, maximized, or changed in size. These messages can originate from key presses from the keyboard, from mouse activity within the window's client area, or from clicks on menus, buttons, or toolbars. The user can even create application specific messages which are unique to the given application itself.

The Win32 window itself is divided into several sections. There are two required sections for a new window: the title bar, and the client area. The windows title bar exists at the top of the application window, and may contain such things as maximizing, minimizing, and closing buttons. Immediately below the title bar there may or may not be a menu. There also may or may not be a status bar located at the bottom of the window. The rest of the space within the window is considered to be the client section. This is the section of the window where text, primitives, and basic graphics are usually drawn. Also, if toolbars are used they are placed within this portion of the window.

1.1.8.3 MFC

MFC was designed to allow for rapid development of windows applications. Internally it uses Win32 commands to get a lot of its work done, but wraps these commands in a very powerful and object oriented API. For the most part it is best to use MFC when quick and dirty applications need to be written where performance is not a concern. MFC tends to add significant size and overhead to an application. As a result this slows the application down and increases the memory and storage requirements. Due to the speed constraints of the Escape Simulation, MFC was not a good choice for this particular application.

1.1.9 Platform

Choosing the target platform for an application is another important decision for a program developer. In designing this particular application two particular operating systems were considered: Windows, and Linux. Both of these operating systems offer solutions to the specific requirements for this application.

1.1.9.1 Windows

There are many advantages to programming in Windows. First, and most obvious, is that Windows is one of the most widespread operating systems in use today. If a programmer writes an application for the Windows operating system, it is guaranteed that the majority of users would be able to run the code.

Windows also offers a complete solution to graphics, input, and networking requirements—namely, DirectX. It also provides a convenient way for researchers to write pluggins and have them quickly and easily added to the main application. This is accomplished through the use of dynamically linked libraries (DLLs).

Along with these advantages, coding in Windows allows for the use of Visual Studio, which is one of the best integrated development environments (IDEs) in use today.

1.1.9.2 Linux/Unix

In today's R&D environment some flavor of Linux is often the operating system of choice. Linux is a very powerful and configurable operating system.

Like Windows, Linux offers solutions to many of the program's requirements. It also provides an environment where there would be less overhead during the application's actual runtime. This would be especially important for the server side portion of the application. Unfortunately, unlike the Windows' DirectX API, the solutions are not packaged together..

Ideally it would be nice to be able to run the server on a Linux box and run the client on the windows machine, but due to time constraints, the initial Alpha and Beta versions of this application are restricted solely to Windows. Future versions can be made to be cross platform.

1.1.10 Contributions

The goal for this thesis is to create a software tool for simulation and experimentation to be used for the development of dynamic evacuation systems. The name for this tool is the Escape Simulation Suite.

The main goal of the Escape Simulation Suite is not to solve the evacuation problem, but rather to provide a tool that researchers can use to test their theories quickly. This tool must be both flexible and easy to use.

One of the main characteristics of the Escape Simulation Suite is its adaptability. Researchers are able, through pluggins, to create objects for the simulated world giving them whatever level of AI or behavior they want. Some of the objects that might be rendered might be; people, chairs, cameras, lights… It is also possible to add non-rendered objects, such as high level controllers and data collectors. This adaptability is one of the key features that makes the Escape Simulation Suite different from all the other simulators currently available.

There are several other goals that need to be met. These goals are as follows:

- The ESS must be fast. Meaning there must be as little overhead as possible. The algorithms and data structures used must be ones that provide for the best overall performance.
- The ESS must be extendable. Meaning that future work and improvements must be possible without completely overhauling the application code.
- The ESS must be intuitive. The user interface and overall runtime flow for the suite must be easy to understand and use.

CHAPTER 2: Preliminary Research

In researching the different components that ultimately would go into the Escape simulation, many sample and test application were written. This chapter covers many of these applications and the information learned from them..

2.1 Research work on graphics

2.1.1 Win32

In order to test the graphics and window capabilities of Win32 there were several applications written. The first few applications were very simple and for the most part only rendered different primitives and text to the window client area. The next applications were more complex, rendering a lot of multiple moving objects, and stress testing the a APIs rendering capabilities. Finally, a simple artificial intelligence application was created to simulate a mock "fishpond," to stress test and evaluate the APIs performance under a more heavy load.

2.1.1.1 Other small applications

As stated above there were many simple programs written at the beginning of the author's excursion into the wonderful world of Win32 programming. These programs range from being able to generate thousands of random pixels or primitives, to creating the equivalent of a command line console application.

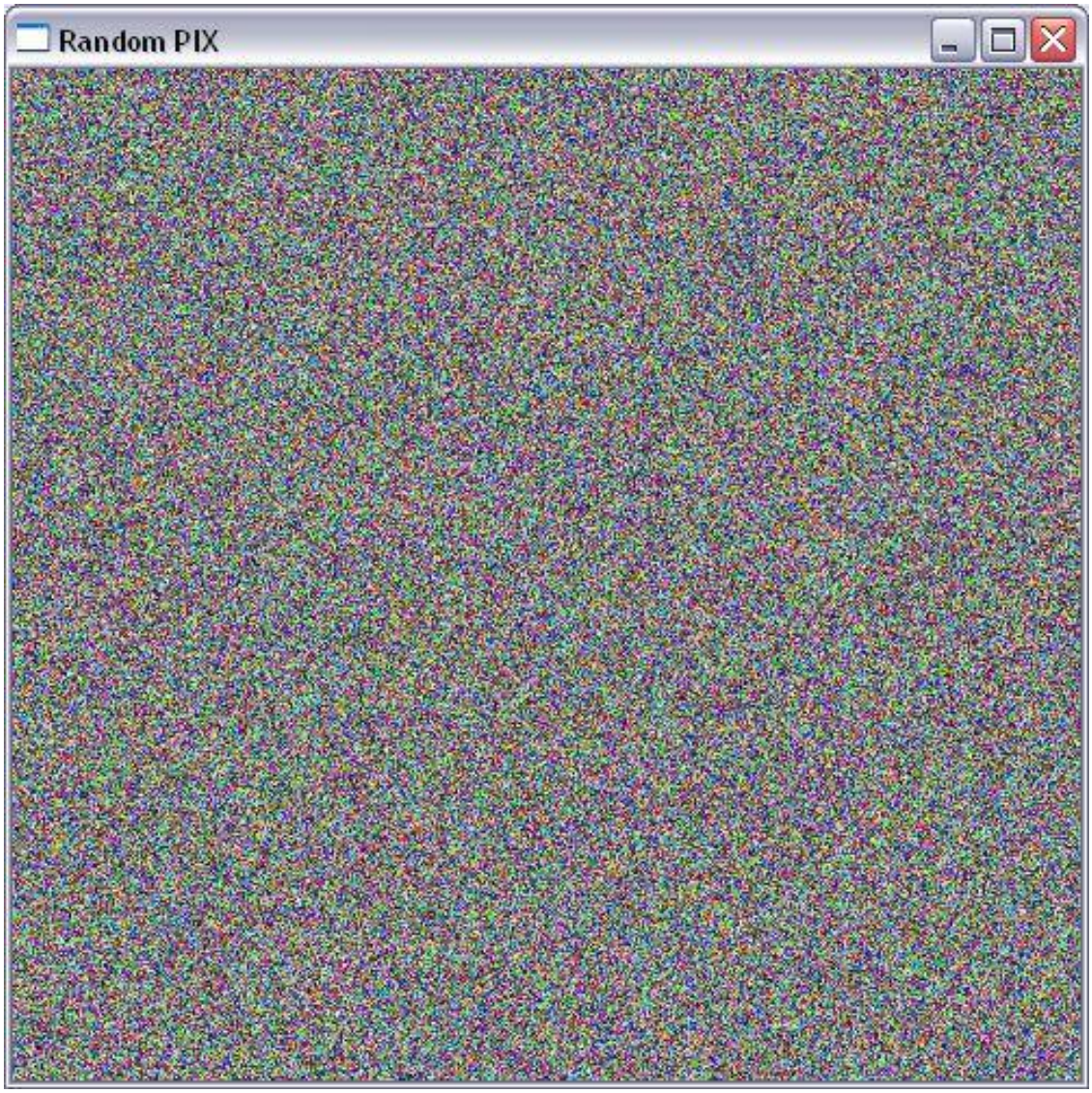

Figure 1 Screen Shot of Random Pixel Win32 Application

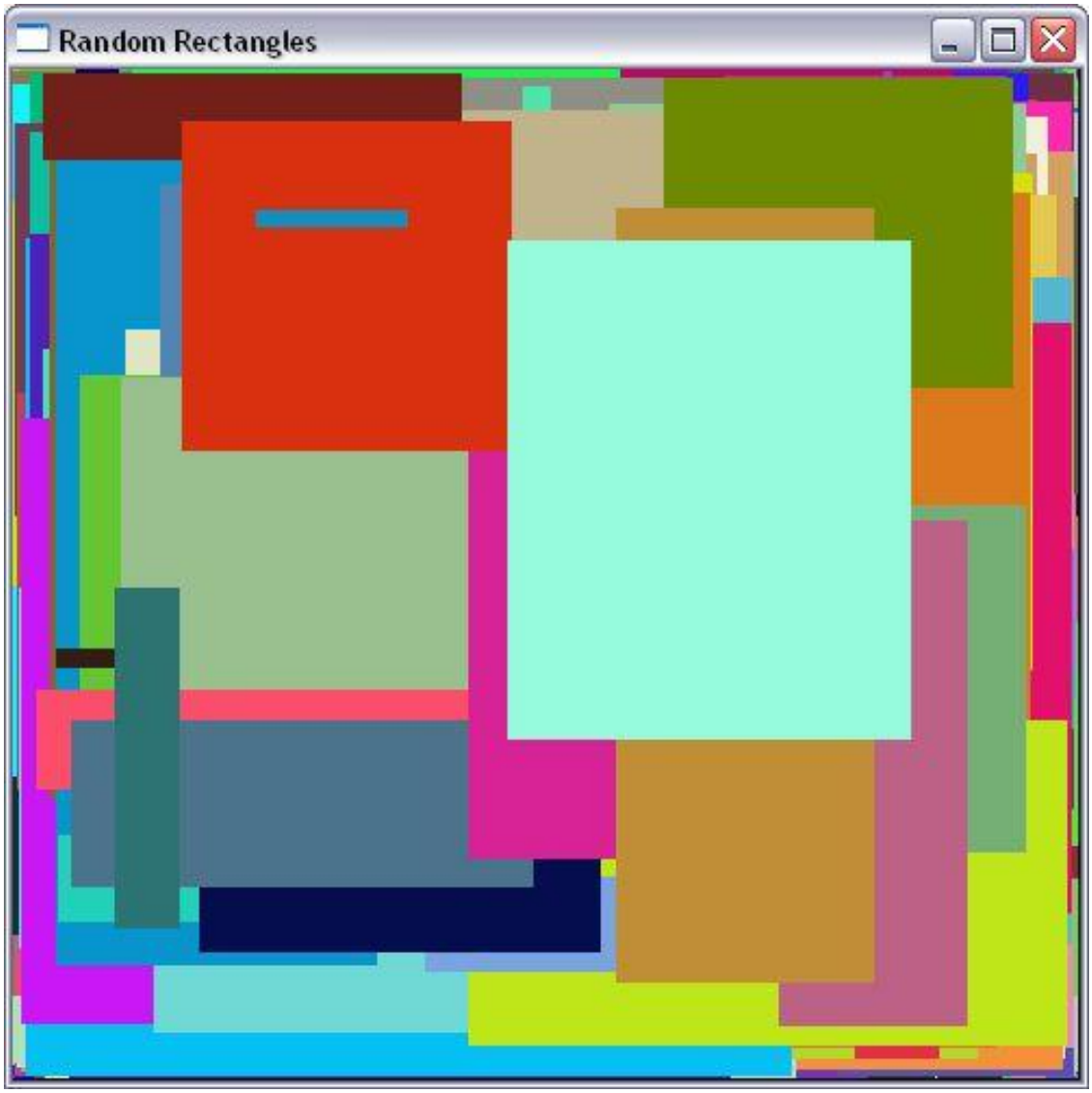

Figure 2 Screen shot of Random Rectangles Win32 Application

There were also many applications written for testing out menus, toolbars, buttons, child windows, and dialog boxes.

The simplicity and cleanliness of the Win32 code proved, for the most part, very pleasing. One of the problems with picking Win32, MFC or some other kind of window API is that they can add a significant amount of clutter and overhead to the end application. So far Win32 has proved to be a good candidate.

2.1.1.2 Charged particles application

This application was designed to stress test the graphics environment , and perform some simple mathematical calculations. It allows the user to create multiple charged particles being represented as simple circle primitives. These charged particles use a simple calculation based on the inverse of their distance from each other to decide their current

force vector. This vector is then applied to modify their current velocity accelerations and momentum. Along with placing these particles within the world, the user can place poles and zeros which effectively attract or repel these particles. The size of the actual rendered particle can be varied to increase the stress on the graphics rendering. Also, an unlimited number of particles, nodes, and poles may be placed to increase the computational load.

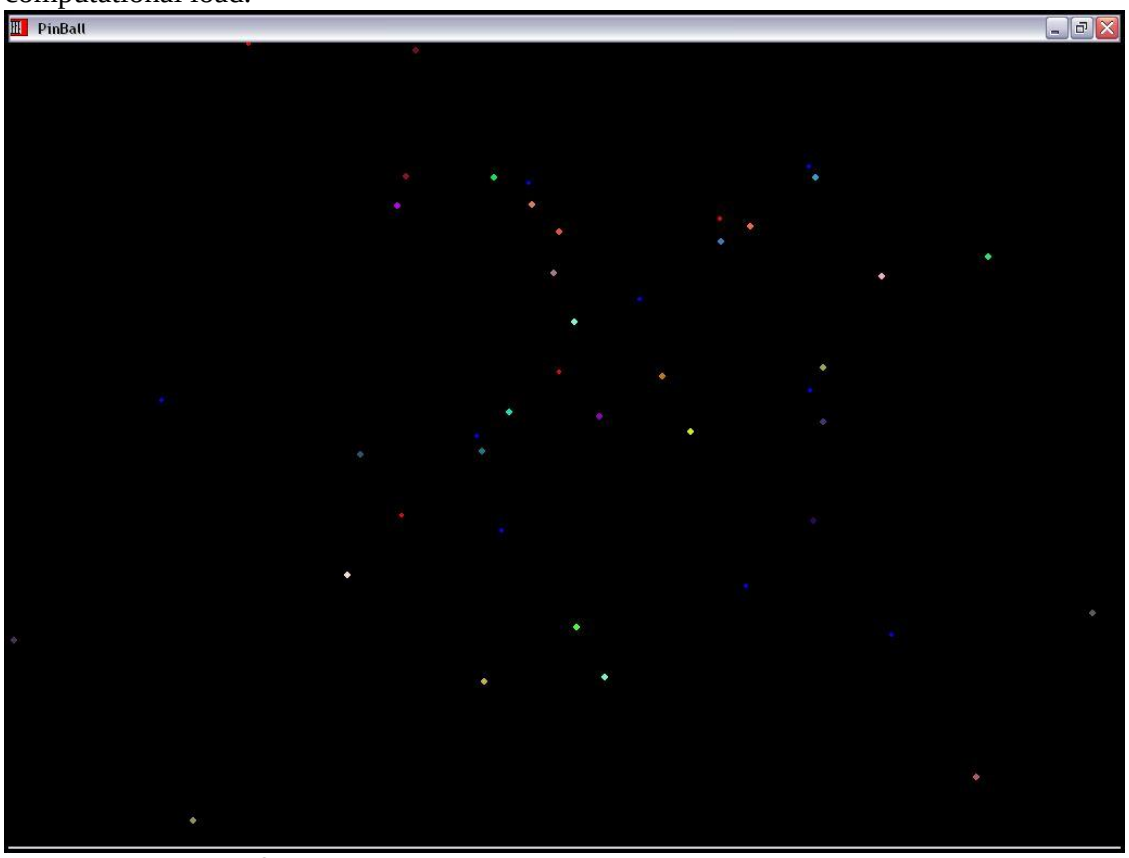

Figure 3 Screen shot of charged particles Win32 application called Pinball. In this the blue and red dots represent zeros and poles respectively. The other dots are particles

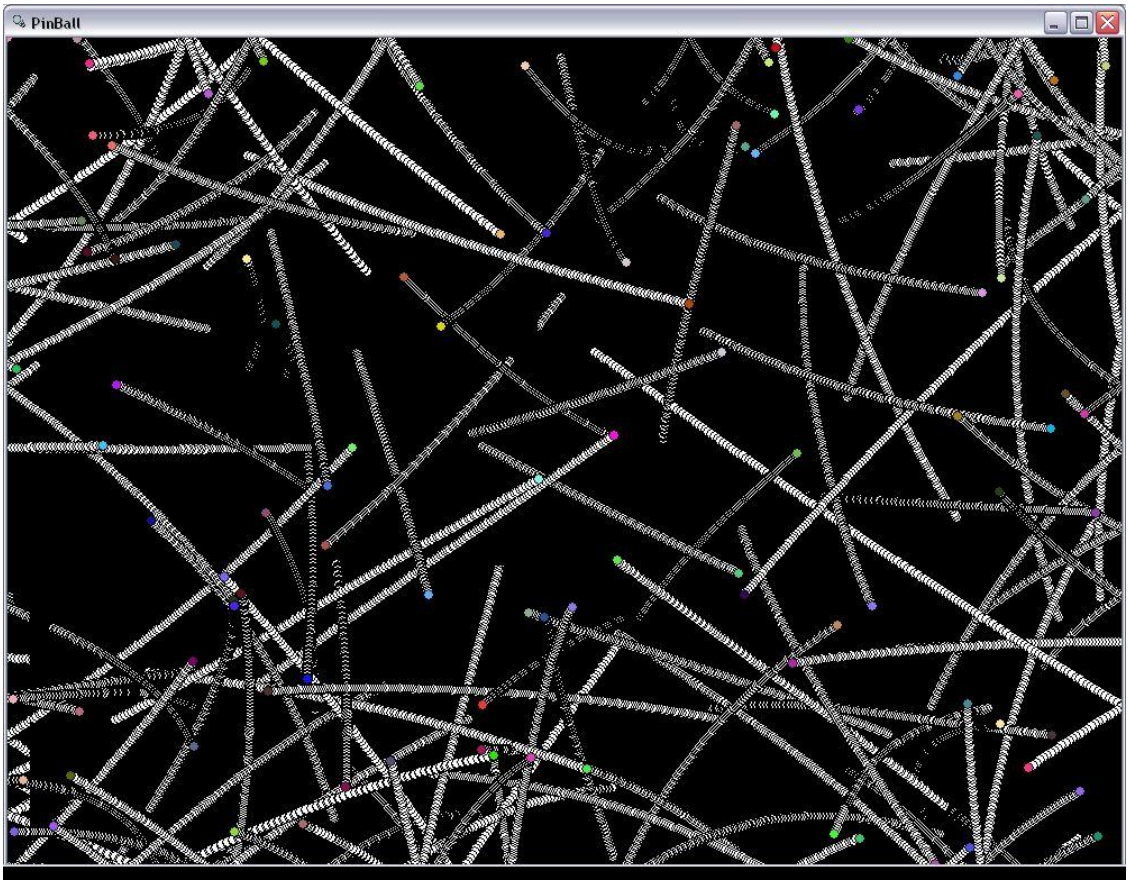

Figure 4 Another version of Pinball, this time without poles and zeros, but with each particle having a repelling force on all other particles. This force is inversely proportional to distance.

For the most part, the API seemed to hold up fairly well under a small number of particles. However, when a large number of particles were placed the particles began to be choppy and the rendering and overall engine slowed down significantly.

Unfortunately, the requirement to calculate and render even a reasonable number of two dimensional objects appears to be fairly taxing on the system when using Win32 GDI calls. It would also seem that some other solution would have to be found for a three dimensional environment.

2.1.1.3 Fishpond Application

After writing the simple applications described above and doing some preliminary graphics stress tests, a more involved test which would combine both graphics and a simple artificial intelligence was needed. This test would use mathematical calculations similar to those used by the Evacuation Simulator in a macroscopic mode. This test application also was used to test out other Win32 functionalities such as toolbars, menus, and child windows.

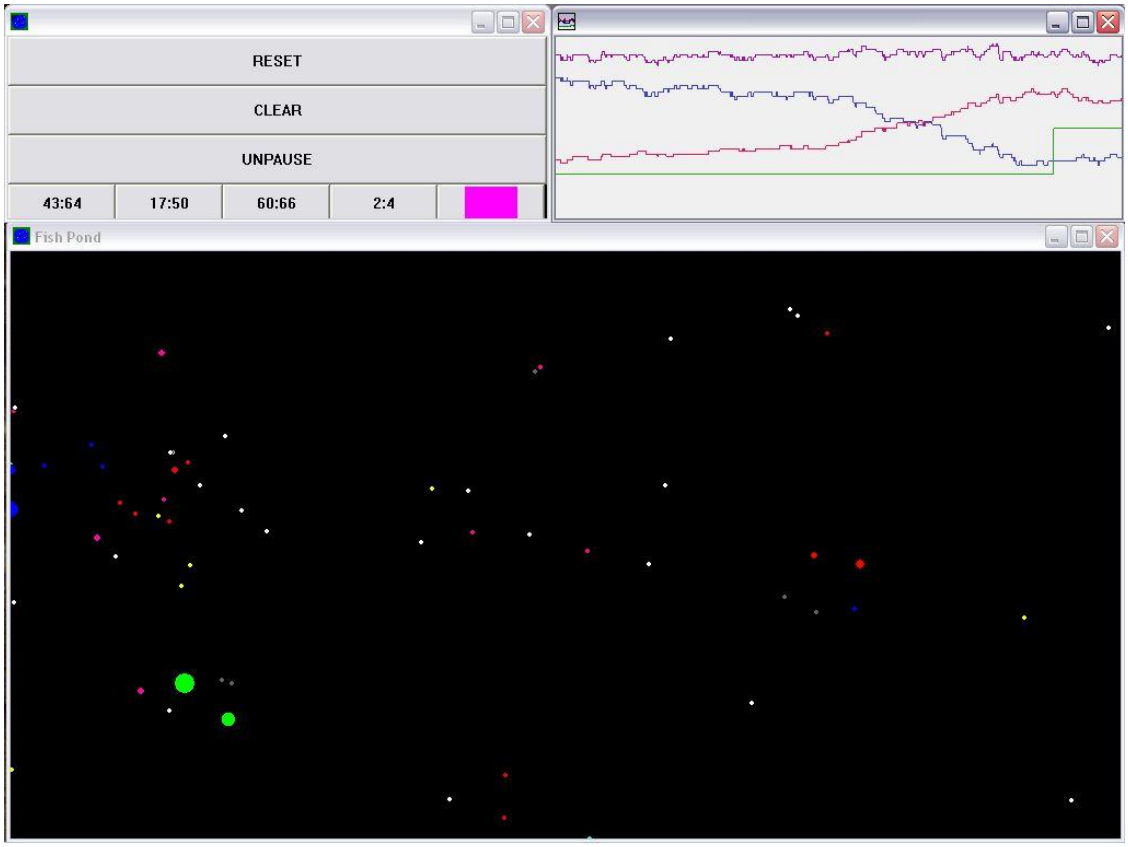

Figure 5 Screen shot of Fish Pond Win32 Application. Here all the colored dots, except green, represent different types of fish. Some are carnivores and some herbivores, and the window at the top right gives the current populations of the different types of fish.

For a two dimensional environment, the results from this application were interesting. Again, the general rendering speed and quality was unacceptable for the finished Escape Simulator, but the menus and buttons, as well as the general nonlinear behavior of the application were very encouraging.

2.1.2 DirectX

DirectX provides a powerful interface API to many of the computer's hardware systems. Among these components are DirectDraw, Direct3D, DirectInput, and DirectPlay.

DirectX implements each of these components as COM objects. These objects allow multiple users to share the API's functionality with very little overhead. Also, Microsoft has worked with the hardware companies to make sure that all new hardware devices used by DirectX, such as video cards, sound cards, network cards, etc, are completely compatible with the DirectX API. This collaboration has allowed DirectX to provide the least amount of overhead possible when accessing a computer's hardware. It also allows DirectX to access all the hardware acceleration available for the current hardware device. Finally, DirectX is designed to provide a software solution to any missing hardware functionality. For instance, if you write a program that uses the DirectDraw's rotation

command which your graphics card happens to support, but then try to run this same application on your friend's computer which does not support the same, then DirectX will provided the missing functionality in software. Obviously, this will make the application run slower, but at least it will work.

A full description of DirectX and its components is well beyond the scope of this thesis. Most of the details necessary for understanding its use in the Escape Simulator are included throughout the thesis and other documentation, but it is strongly recommended that the reader refer to and become familiar with three books which are excellent references for DirectX and developing graphics applications using it. These books are [5], [6] and [7]. Also, as with Win32, the MSDN provides a wealth of information on DirectX. Please refer to [10] & [11].

2.1.2.1 Spaceflight program

This application was designed to test the amount of work involved with using DirectX's 2D API DirectDraw. It draws a 3D space flight world in which the user can fly around and explore. All the calls used are 2D. Basic trig and geometry were used to display the objects in a way that appears 3D.

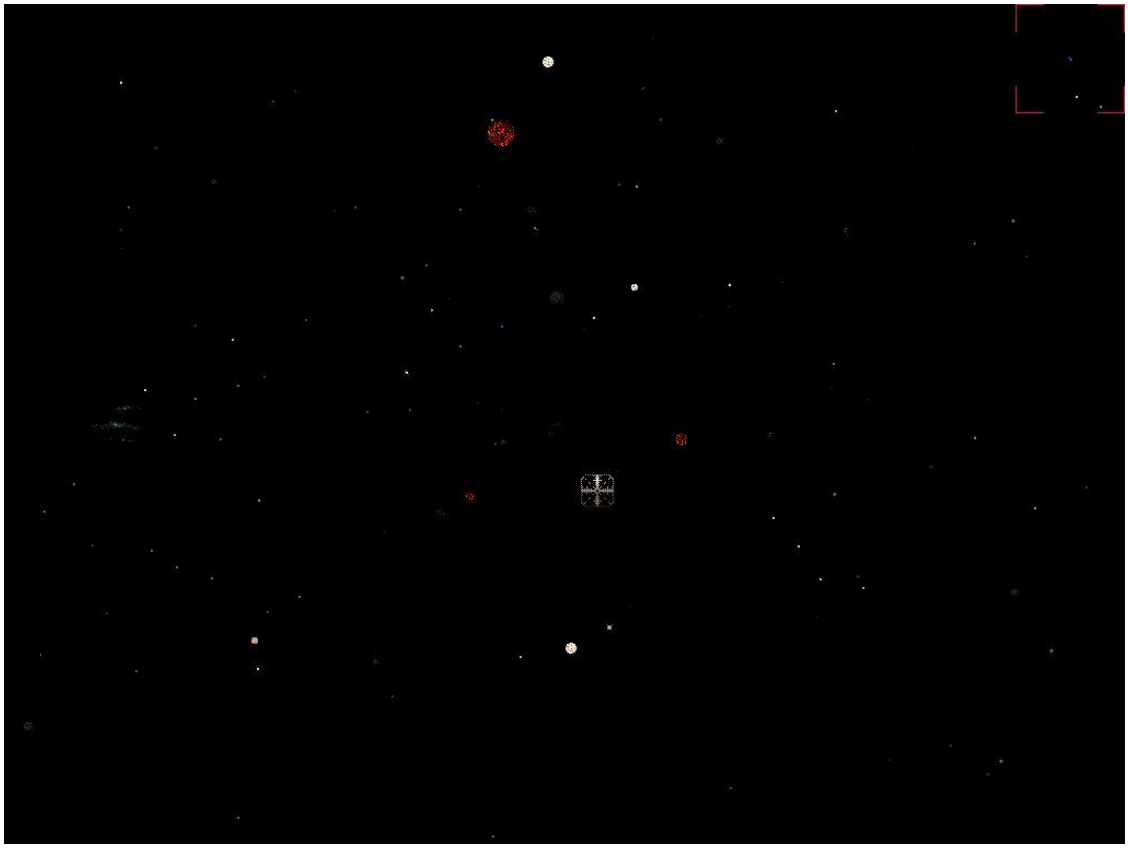

Figure 6 Screen shot of the Space Flight DirectDraw application. In this application a 3D space world was rendered using 2D DirectX calls. The user is able to both "fly" around this world and change the FOV characteristics.

Among the obstacles that had to be overcome was sorting the objects in the world based on their current depth relative to the camera's current position and look direction. For small numbers of objects this sorting is not a big deal. However, when you are displaying a 1,000,000 by 1,000,000 world with 100,000 objects that must be sorted 60 times a second, you need a very fast algorithm.

The algorithm used in the end was a basic quick sort, calling the sort every render loop. Due to the fact that the application randomly distributes the objects when the field is first created, not doing any sorting as it goes, the first sort is usually the worst one, since after this first sort the objects are in order and only small changes are made every time slice to update the list before rendering.

The math involved with displaying the 2D objects and making them appear 3D is fairly simple, but can be a pain to setup at first. The reader might be wondering why DirectX 2D graphics is being covered here when the actual Evacuation Simulation is based on DirectX's 3D API. It is essential to understand the math used here in order to correctly know how to use the 3D functions properly. It should be noted that none of the math discussed here uses quaternions. Though there are several advantages to using them, they are not without disadvantages, and for this particular application, quaternions are not really needed. If the reader is interested in learning about them please refer to [6] pages 317-318, 325, and [7] pages 100-103.

2.1.2.1.1 Coordinates

In the Graphics section of the first chapter, coordinates were mentioned as one of the important topics that a graphics programmer must consider and keep track of. For any object in 3D space there may be several different coordinates assigned to it. For instance, there should be one set relative to the world as a whole, one set relative to the object's center of mass, and one set relative to the camera's position and orientation. There could be several more relative to different groupings of objects, etc… All a set of coordinates tells us is an object's position and orientation in space.

For this program the camera's coordinate position was more important than the actual orientation. The reason for this is that the objects being displayed are completely 2D, and therefore the orientation is always with the image on the plane parallel to the user's screen with the front facing the screen. The world's coordinates also were not of real importance. This is because there is no interaction between the different objects.

2.1.2.1.2 Render Loop

Within a graphics program there is always a render loop. After finishing all the initial required setup, the application enters the render loop. There are several required parts for a render loop. First, any user input that exists must be taken in. Second, any changes to the world as a result of the input, AI, or some other algorithm must be calculated. The world's data structures must be updated, and finally, the frame must be painted and rendered on the screen.

This application accepted two input sources, keyboard and game pad. Depending on the input, the camera's position and orientation was changed.

After changing the camera's position, all of the objects in the world needed to be resorted based on its new position relative to the camera. In order for depth to be faked in a 2D environment there are several tricks that can be done. The most popular is to inversely vary the size proportional to the distance and render the objects in farthest to nearest order. The distance is easily calculated by finding the magnitude of the vector formed from the object's position and the camera's position:

$$d = \sqrt{(objectX - cameraX)^2 + (objectY - cameraY)^2 + (objectZ - cameraZ)^2}$$

In order to make the actual rendering as fast as possible the object list must be resorted and the size adjusted to match based on this distance to meet the above requirements. So the objects farthest away from the camera are placed first in the render list and the rendered size is scaled inversely.

2.1.2.1.3 Field of View FOV

The field of view is the last topic that must be addressed. A FOV is similar in concept to looking out a window. The farther away the viewer is to the window, the smaller his/her FOV is of the outside world. As the viewer gets closer, the FOV appears to expand and allow the viewer to see a wider area of the world on the other side of the glass. Likewise, the size of the window makes a big difference in the FOV.

The computer rendering's FOV is the same. Instead of a window of glass you have the screen of your computer. The area of objects that can be seen is dependent on the calculated distance of the viewer from the screen and the actual screen's size.

Why do we care? It is not efficient to simply attempt to draw every object every time through the render loop and let the hardware decide what it can actually display. Doing this usually means trying to render objects that cannot be seen on the screen and this results in a lot of wasted function calls and time. So the last thing that must be calculated in order to correctly render the objects on the screen for a given frame is which objects can actually be seen.

As stated above, there are several factors that determine what the FOV for a rendering is. The person's and camera's position in the world, the screen's width, height, and position and orientation relative to the camera's, and the object's position in the world.

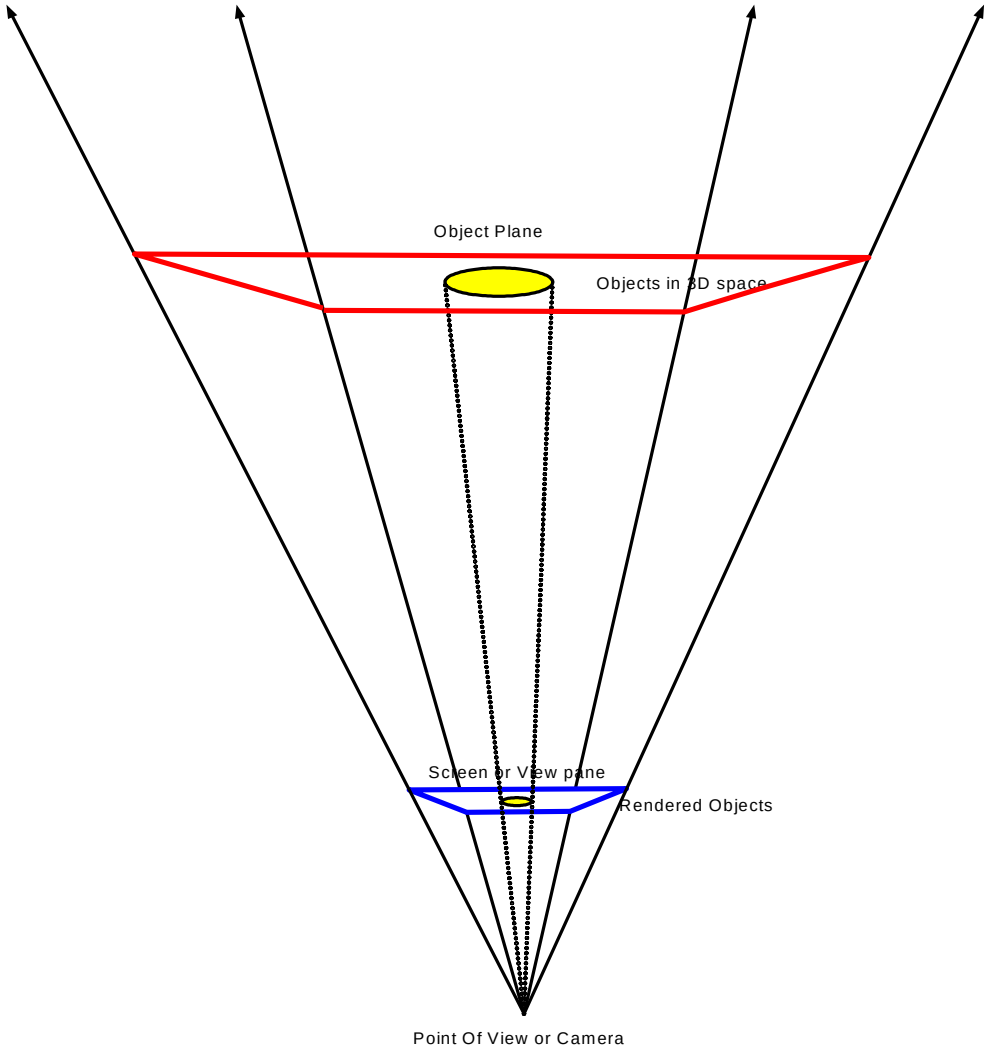

Figure 7 Diagram of the Field of View

If one were to start drawing a line from the camera's position through each of the corners of the screen and then on to infinity he/she would get a pyramid shape which forms the FOV. Each of the four side faces would form a triangle from the point to the screen. And if one creates a field parallel to the screen including a given object in 3D space he/she would get another pyramid whose four sides are proportional to the first pyramid's sides. Using this relation, it is easy to see if a particular object falls in the FOV of the camera. The base of the larger pyramid will also be proportional to the base of the smaller. Since the dimensions of the first rectangle are known, multiplying the top left and bottom right corner's x and y positions will give you the corresponding position of the larger base's corner. Once these two positions are known the object in question can be checked to see if it falls in the rectangle defined by the two points. If it does then it is in the FOV of the camera and should be drawn.

2.1.2.1.4 Moving the Camera

There are two ways the camera moves. First, its position can be moved around the world, and second, the camera's orientation can be changed so that the view is changed.

When the camera is moved throughout the world the camera's world position is what is modified. It is translated along some vector which is created based on its current orientation. There is a small amount of work that is needed in order to get the correct vector and translate the camera based on it. The camera's orientation vector's direction is from the camera's position to the center of the FOV screen, and it has magnitude 1. If the camera is moved forward or backward, then its world position is translated along a positively or negatively scaled version of this vector respectively. Similarly, if moved up, down, left, or right, its world position is translated along a scaled vector perpendicular to the orientation vector.

The orientation vector is stored in Cartesian, but changes to the orientation are made in spherical, thereby allowing rotation around any specific axis as being a simple matter of adding or subtracting to the appropriate degree.

Overall, this program was very successful. It does quite well when it comes to displaying a large number of objects correctly, even with all the overhead of the 3D calculations. The performance was much better than the Win32 programs from the section before. Unfortunately, with a more significant load than simply making and drawing a large number of primitives, along with the extra network and AI calculations that the Escape Simulation requires, even this would not be a feasible solution for drawing the bulk of the graphics for this application. Also, texture and rotation support would have to be done by hand because there is none for 2D.

2.1.2.2 Density program

There were two objectives when this test was written. The first was to actually code up one of the macroscopic models that would be used in the finished Escape Simulation Suite. It was important to get an idea of how much stress this would put on the application. The second reason this application was coded was to figure out how to create a way of rendering the density field that would be useful to the researchers.

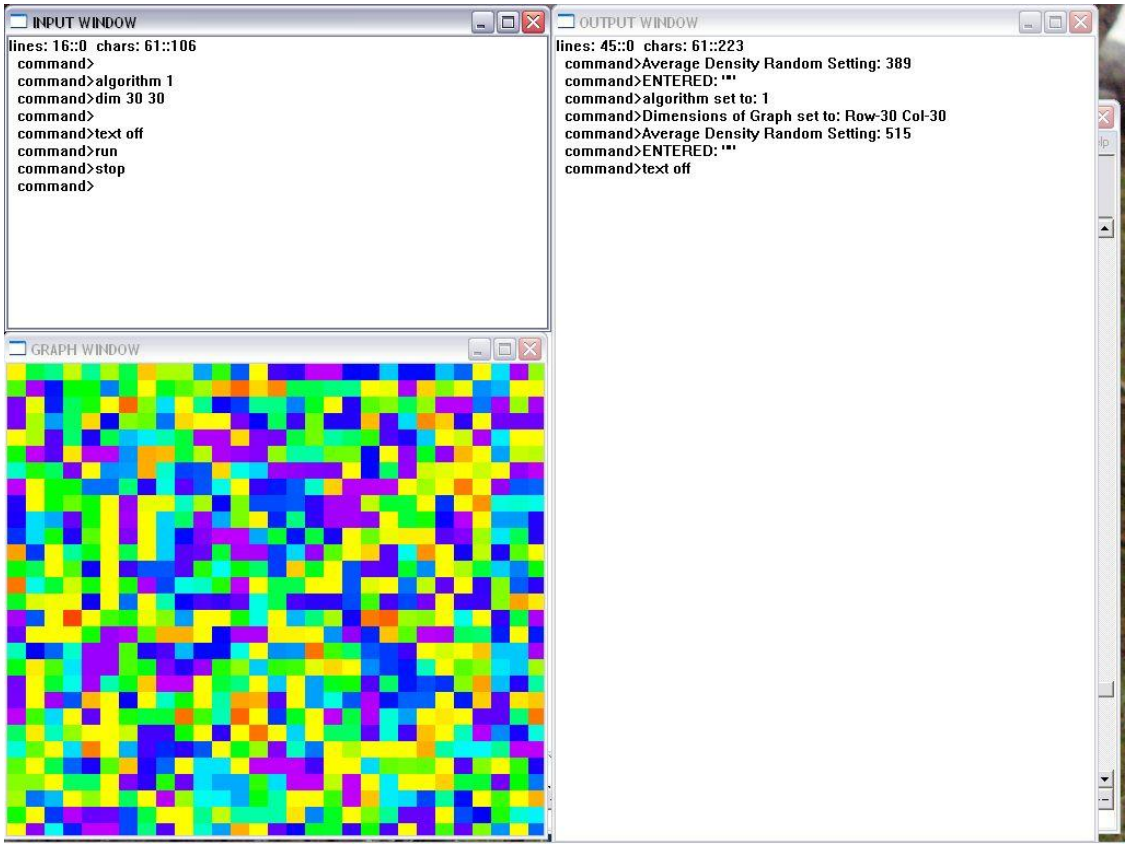

Figure 8 Screen shot of the Density Win32 application. The top left window is the command window; the bottom left is the density field being rendered. The different colors represent different densities, ranging from low (purple), to high (red). The right window is the output window.

For the model used in this application please refer to [21] which was included with the source files zip that accompanies this document. It took a little work to code up correctly. Because the changes in any given block are coupled with those of all the others and everything was obviously being calculated discreetly, it was impossible to get a perfect answer. That being said, the solution that was reached is one that takes care of many of the potential problems with discreet calculations of continuous systems, jumping boundaries for instance, and one that, with a small enough step size, provides a close enough approximation to the actual answer, at least for the purposes of this test application.

The graphics proved to be an interesting problem to solve. In the beginning it was decided to go with displaying the densities as a height map grid across the floor. The heights were displayed as different colors with purple representing a density of 0, red representing a density equal to whatever the maximum allowable density is set to, and all the other colors being evenly distributed between.

For this application everything was done in 2D, but for the Escape Simulation Suite the field not only has a color based on the density, but it also is given a height based on the same.

All in all, this program was relatively good. The algorithm turned out to be not too coastly, and the graphics worked out very well. However, one tricky part was putting together the array of color values that was indexed based on the density in order to give the proper color. This was a little complicated because the different values—purple->blue->green->yellow->red— at first do not seam to transition in an intuitive manner, and also because not all the color bands are the same width. So there was a little work to get everything uniform and even.

2.1.2.3 Flying Cube stress test

This was the first real test of Direct3D. All the tests up to this one were done using the DirectX 2D API, DirectDraw. This application was intended to stress test the capabilities of Direct3D under a very large load and various settings.

The application in a nutshell created x^3 cubes each with six large, different textures mapped on each side with different lighting. All the cubes are arranged to form a larger cube x-by-x-by-x large. The formation as a whole rotated each times step along two axes. At the same time each cube is rotated along two different axes. The camera is designed to be able to rotate along any axis, orbit the cube formation, and zoom in and out. Finally, the frames-per-second and other statistics were also rendered each time through the loop.

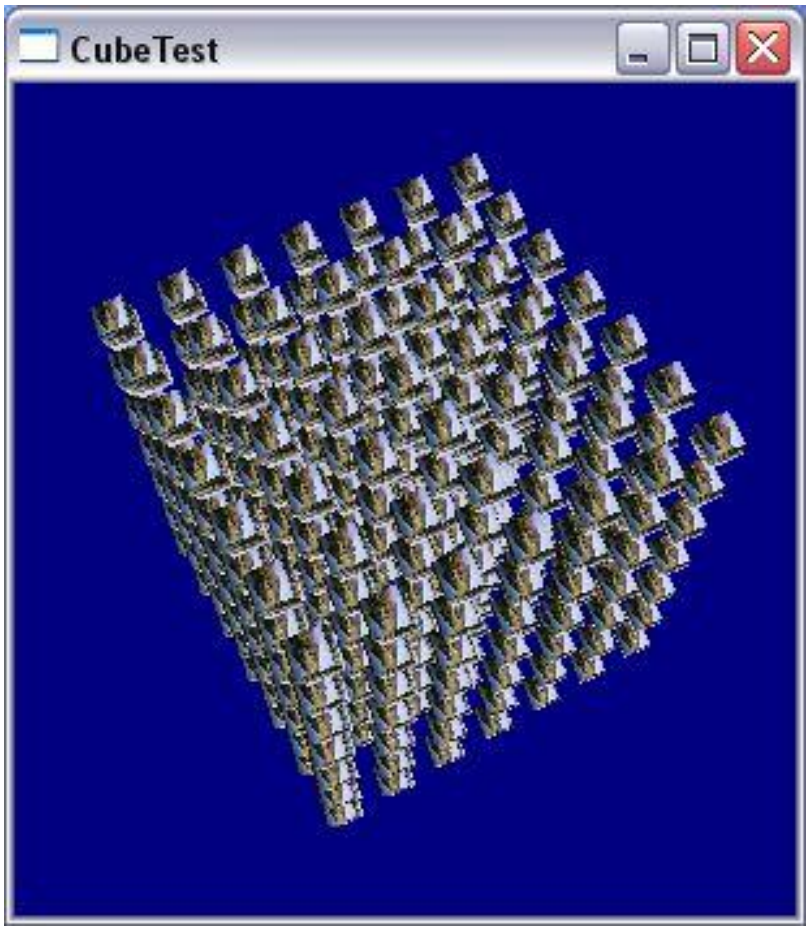

Figure 9 CubeTest application screen shot.

First off, it was pleasantly surprising how easy it was to set everything up. The hardware acceleration also made a huge difference in the overall performance. It was possible to take x up to 15 and still get around 60 fps. That means Direct3D was able to render 3,375 boxes = 20,250 textures = 40,500 primitives = 121,500 vertices 60 times a second. Not too shabby

2.1.2.4 Network Chat Program

Along with testing the other aspects of DirectX, a few applications were created that tested the client/server capabilities of DirectX's networking API, DirectPlay. In particular, a client/server chat application was coded. More will be covered about this application in the next section, "Research work on networks."

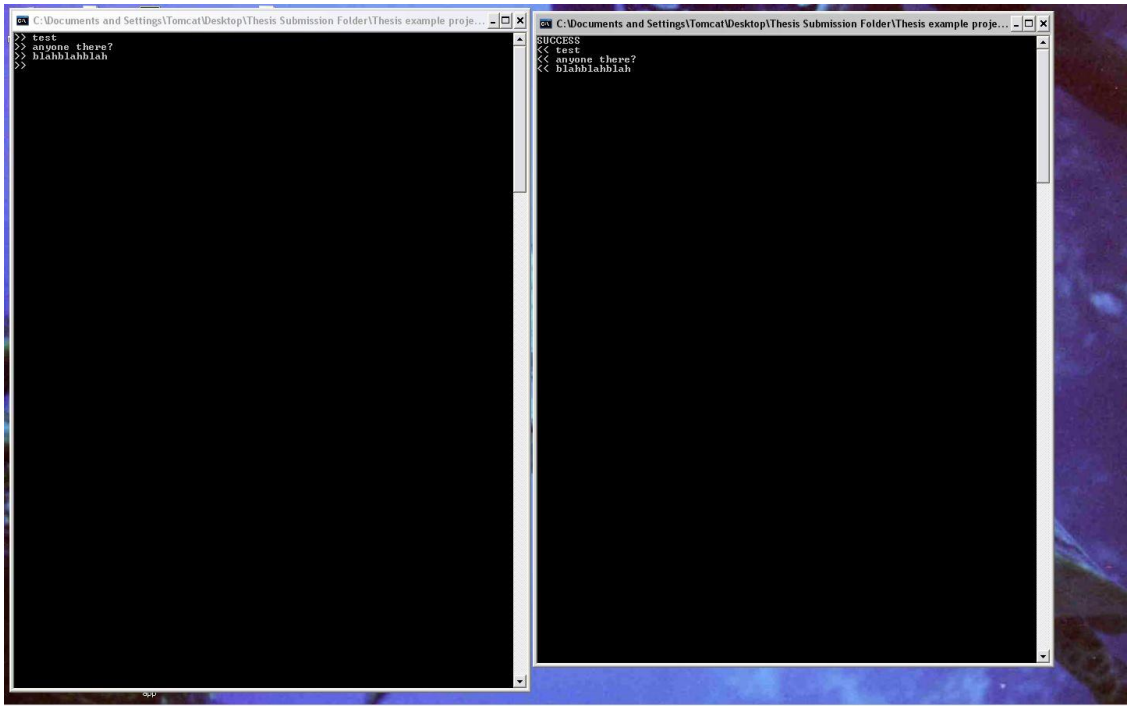

Figure 10 DirectPlay network application

In brief, the overall performance of DirectPlay was good. Little is still known about how much the overhead will affect the overall throughput and reliability.

For a good explanation of the workings of DirectPlay please refer to [12].

2.1.2.5 DirectInput

The final aspect of DirectX that was tested was DirectInput. This is the API responsible for allowing the user access to the different input devices. As stated in the last chapter, it was important to be able to get input from the keyboard, mouse, and game pad. A separate application to test these things was not created. Instead, DirectInput was used throughout. In all the above DirectX programs, different combinations of input were used to figure things out. In all cases, DirectInput turned out to be very easy to use. The only concern that arose is how to integrate properly with Win32's input accelerator. Please refer to [5] chapter 9 for more information on how to setup and use DirectInput.

2.2 Research work on networks

From very early in this application's design, the decision was made to network it. At a minimum it was be desirable to separate the application into a client/server module with the graphics and user interface being handled by the client, and the world calculations being on the server. Depending on how complex the world calculations got it might be nice to eventually cut the server up and distribute it across a cluster.

2.2.1 Personal Winsock Library

2.2.1.1 Written by hand TCP/UDP applications for network applications class

When the decision was made to network this application, several important design issues arose. First and foremost was how to do the networking. Should a network library be developed from scratch, or should some other API be used? In an effort to answer this problem and to understand networking in general, several test applications were written. The applications ranged from simple chat programs and message boards, to a web server with simple cgi support. In order for the reader to understand some of the logistics of network communication, some of the basics will be covered here. The reader is also encouraged to refer to [8]. Also, Microsoft's MSDN is an excellent source of very in-depth descriptions of the various functions needed to create a Winsock application [13].

TCP and UDP applications have certain aspects that are similar. Both require you to call WSAStartup and WSAShutdown to start and stop the windows socket interface. In both cases you must create and destroy a windows socket to get your work done. After this point they are very different.

As stated in the first chapter, TCP is a reliable way to send data, meaning that when you send a packet from one socket to another, if it is possible to make it, it is guarantied to get there and in the correct order. Part of having this guarantee, however, means that a connection must be established between the client and server before any data can be sent. Once a connection is established data can be sent in either direction. After all data has been sent, the connection is closed and the socket is destroyed.

In order for a connection to be made you need the server to be bound to an interface:

```
int bind(SOCKET s,const struct sockaddr* name,int namelen);
```

and put into listen mode:

```
int listen( SOCKET s, int backlog);
```

The client then can request a connection using:

```
int connect(SOCKET s, const struct sockaddr* name, int namelen);
```

Finally, the server accepts the connection using:

```
SOCKET accept( SOCKET s, struct sockaddr* addr, int* addrlen);
```

Both the connect() and accept() functions take in a pointer to a generic struct of type sockaddr. This struct was designed so that different protocols can extend its data fields to meet their specific needs. For a typical TCP/IP connection this struct would look like this:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Once the connection is established the two can communicate back and forth using the Winsock send()/sendto() and recv()/recvfrom() functions.

When the two applications are done the connection is broken down.

For a UDP based application, a connection is not required but can still be established. The packets are simply sent to the target address port pair with no guarantee that they will get there intact or in order, if they get there at all. If data and packet integrity is needed, then the programmer must add his/her own code to ensure it.

Even after the brief top level description above, there are still many small details that a Winsock developer needs to keep in mind, for instance: blocking verses nonblocking calls.

At the end of the above mentioned test application, a fairly decent TCP/UDP library had been developed. In order to add some contrast, another library was developed based on DirectPlay. The next section is about this library. For the most part, though, this library worked very well.

2.2.2 DirectPlay

There were several reasons DirectPlay was chosen to be the second network option. First, many of the components that DirectX offers were already going to be used by the application, so the idea of com objects was already familiar. Second, DirectX is a well established API and has had significant work done to it to ensure compatibility and reliability. There are many big-name applications that use DirectPlay for their network interface. Finally DirectPlay is setup structurally in a way that would fit well with the way we wanted to handle our network interface.

Like the Winsock section above there will be only a brief description of DirectPlay here. For a better description of how to develop DirectPlay applications please refer to Microsoft's MSDN articles on the topic.

Creating a client/server connection in DirectPlay is very similar to doing it using Windows sockets directly. First, the server must be created and placed in host mode; this is equivalent to listen mode. Then, the client must be created and the target address information must be filled out. Finally, the client must call connect to establish a connection to the server.

In order to send a message using DirectPlay the client must use the **`send()`** function. The server uses the **`sendto()`** function.

Receiving messages is handled differently by DirectPlay. Unlike Windows sockets you do not call recv() or get, or any function for that matter. Rather, incoming messages are handled similarly to the way a Win32 program handles them. There is a call back function that the client and server are given when they are created that is called any time a message is received. Inside that handler the user can put whatever code is needed to handle the messages being received.

This method of handling messages was exactly what was desired for the Escape Simulation Suite framework. The plan was to have one central place where all messages would be handled, as opposed to having the handling spread throughout the entire application. This would help in making the code much more manageable and modular.

In order to test the DirectPlay interface a simple chat application was written, which worked very well for the most part. There was one negative aspect: when the client would attempt to connect to a server that was not in existence there would be a significant amount of lag while DirectPlay did some extra work to attempt to search for the missing server.

2.3 Research work on application types

2.3.1 MFC

MFC, as stated above, is an API that was built on top of Win32. It allows for rapid application development. Unfortunately this comes at the cost of efficiency. MFC applications tend to have more overhead than a typical Win32 equivalent. They do allow for much more complex menus, user interfaces, and dialogs, though.

The test MFC application was intended to just test what the logistics would be to get an MFC application up and running with menus, etc… and also to get that application working with DirectX.

2.3.2 Win32

Throughout this research many different Win32 applications have been written. This section will cover some of the applications that were not already mentioned in previous sections, It will also give a brief description of Win32 basics.

2.3.2.1 Win32 Basics

"All journeys start with a single step." Or in this case, all Win32 programs start with WinMain().

2.3.2.1.1 WinMain()?!

For the C purist it will feel a bit unsettling the first time one codes or looks at Win32 code. This is in part because one the of first things written or looked for is main(). Unlike normal console applications, Win32 applications do not have main() as their entry point. Instead, they have:

```
int WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
```

As far as we are concerned, the most important of the arguments in WinMain() is hInstance. This is a handle to this instance of the application. A handle is just a unique number that Windows will use to reference this instance in a lookup table somewhere. Windows loves to use handles for everything, so get used to them.

The second argument is not important to us. Older versions of Windows used this argument in handling multiple instances of an application.

The argument lpCmdLine is a pointer to the command line string that can be parsed to find command line arguments.

Finally, the nCmdShow parameter tells us what the current visibility of the application is. Since we will be changing this it really is not that important.

After WinMain, the programmer must define the characteristics of the main window. This is done by creating and filling out a WNDCLASS structure:

```
typedef struct {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

Once that class is filled out then the class is registered with the operating system using:

```
ATOM RegisterClass( CONST WNDCLASS *lpWndClass);
```

and windows of this new type can be created with:

```
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
DWORD dwStyle, int x, int y, int nWidth, int nHeight,

HWND hWndParent, HMENU hMenu, HINSTANCE hInstance, LPVOID lpParam );
```

Notice this function returns another handle. This one is a handle to the window we just created. The first was just to the instance of the application. Using this new handle we can now tell our main window to display itself with:

```
ShowWindow(hwnd, nCmdShow);

UpdateWindow(hwnd);
```

Okay. We are almost done. We need only two more components and we will have all the basic things needed for a minimal Win32 application. The first component we need is the message handler. You might have noticed the **`WNDPROC lpfnWndProc`** field in the WINCLASS structure. This field is a pointer to the callback function that this particular window type will use whenever it receives a message. A callback function has a very specific format that it must match. The prototype looks like this:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
LPARAM lParam );
```

The actual name, WindowProc, and the variable names can be changed to whatever you want. One of the unusual things you might notice is the CALLBACK distinction that is after the return type and before the function name. This is a special windows distinction, and identifies the function as a callback function.

A callback function is a one that is called by the operating system when a message has been received and needs to be handled. This is similar to the message handlers used by DirectPlay's receive. As mentioned in the last chapter, there are thousands of potential messages that can be generated and sent from the operating system to your application. For the most part you can be safe by passing these messages to the Windows default handler and only handling the few that you actually care about.

The second thing still needed is the main message loop at the end of WinMain(). This loop, with very little deviation, always looks like this:

```
while (GetMessage(&msg, (HWND) NULL, 0, 0) != 0 && GetMessage(&msg,
(HWND) NULL, 0, 0) != -1)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
```

There are three important functions here. The first is GetMessage(). The while loop will continue to spin as long as this function does not return either 0 or -1. It will return 0 if the application receives a WM_QUIT message. It will return -1 if a major error occurs.

The GetMessage() function takes in two arguments of interest. First is the msg argument. This argument will contain the actual message that was received by the GetMessage call. This variable will then be passed on to the other two functions for further processing. The second argument is the handle to the particular window of interest. This means that if your application has multiple windows, each window can receive messages targeted specifically for that window. Also, if you have multiple windows it is a good idea to make the application multithreaded and give each window its own thread.

The second function of interest is TranslateMessage(). This function translates virtual-key messages into character messages. These messages are then posted onto the calling threads message queue.

The last function is DispatchMessage(). This function actually tells the application to call up the appropriate callback function and handle the message.

There you have it. Those are the basic components that must be included in all Win32 applications. Even your most basic and small Win32 application must have these components. Even “Hello World”!

Luckily there are very nice benefits that are gained by making programs windowed that, depending on the program, can balance out the overhead required. For example, “Hello World” is not an application that would benefit from being made windowed. An application that needs a menu and has many toggles and switches would greatly benefit from being windowed. Menus and switches are clunky at best in a command line application. In general, if an application is linear and requires little, if any, user interaction then it is best created as a console application. On the other hand, if the application is very nonlinear and requires lots of user interaction then you probably are safe making it a windowed application.

The above discussion on Win32 applications has been very basic. Please refer to [4] for an excellent guide to Win32. For the topics covered in the section please look specifically in chapters 1-7 in section 1. Please also refer to [9]

2.3.2.1.2 Win32 and Consoles

At times it is nice to get the simple text input/output of a console application from within a Win32 application. If it were not possible to do this then users would be stuck doing all their own text management, which can be rather annoying to setup.

Here are some examples of times it might be nice to have a console window attached to a Win32 application.

- *Status/Error message output*. Many times it is nice to have a separate status window open that just reports on the general status of the application. This window is purely output, stderr and possibly stdout would be redirected to this window.
- *Console IO for graphics rendering engine*. Aside from basic keyboard input for a graphics rendering window, which usually constitutes single key commands, it may be nice to allow for much more powerful commands. In order to do this it would be nice to spawn off a console to take in command line style input. This is ideal when the programmer is developing the engine and wants to be able to tweak values and things quickly. It only takes a very short amount of time to get a console up and running. Later, the console can be replaced with a dialog box or some other graphical user interface GUI.
- *Windows handle needed but full window is not*. Some API's require a windows handle to work properly. Often these API's may be needed in an application where a full blown windowed application is overkill and a console is all that is needed. In this case it is possible to create a Win32 application to get the windows handle from the WinMain() arguments and then attach a console to get the work done.

Only one console can be attached to any window at any given point in time. You can create multiple consoles and disconnect/connect them as needed.

In order to add a console to a Win32 application you must first create one using:

```
BOOL AllocConsole(void);
```

This function creates a new console and attaches it to the calling process. Again, as stated above, there can only be one console attached to a given process. Child processes inherit attached consoles from their parents. In order to detach a console so another can be created you must call:

```
BOOL FreeConsole(void);
```

Provided that the console is still attached to some process, it is not destroyed by this call. So for example, let's say a simple chat program is being developed. The main console may be reserved purely for status messages or connect/disconnect type of command line input. When a connection is made to another chat program the application would spawn off another thread to handle that connection. By default that thread would share the main console with the parent process. In order to keep the main window reserved for status messages and the such, a call to FreeConsole() would be made from within the new thread to detach itself from the parent's console. Then the thread would call AllocConsole() to attach a new console solely for the child's use, and in this particular example this new console would handle all the "chatting" communication over this connection.

Along with detaching a console from a processes/thread with FreeConsole() you can also attach an existing console to a process using:

```
BOOL AttachConsole( DWORD Error! Hyperlink reference not valid. );
```

It is important to note the difference between AttachConsole() and AllocConsole(). The first attaches an existing console to a process whereas the latter creates a new console and then attaches it to the process.

Just creating and attaching a new console may not be enough to get things working. There are two more important functions that are very useful. The first is

```
HANDLE GetStdHandle( DWORD Error! Hyperlink reference not valid. );
```

This function is used to get the handles to the current stdin, stdout, and stderr. The DWORD value is what tells the function which of the three is desired. It can take one of three values:
STD_INPUT_HANDLE - gets the handle to stdin
STD_OUTPUT_HANDLE - gets the handle to stdout
STD_ERROR_HANDLE - gets the handle to stderr
These handles can then be used by the main process just like any normal c-standard file.

The second useful function is:

```
BOOL SetStdHandle( DWORD Error! Hyperlink reference not valid., HANDLE
Error! Hyperlink reference not valid. );
```

There may be instances when you want to completely overwrite the default stdin, stdout, and or stderr handles with an existing handle. For example you might have a global error handle that all exceptions and errors write to. In this case you would pass this function the handle to that file and pass in STD_ERROR_HANDLE as the nStdHandle value.

Several Win32 applications have been written that have used the above technique to add a console to a windowed application. One example of this is the chat application written to test DirectPlay. This is a great example of the third case given above when you might want to use attached consoles. In this instance, DirectPlay needed a window handle to work properly. So a Win32 application was created that never actually created a window. Instead, a console was attached to the process and was used for all the IO. (It should be noted this is not the only way this could have taken care of DirectPlay's requirement for a window handle. It is possible to create a normal console application and then get the window handle that windows assigned to the console window.)

2.3.2.1.3 Win32 text interaction

While a console is often times a quick and efficient solution to an application's user interface needs, especially when text parsing is required, it can fall short to a windows managed text interface. All you get with a console is the standard console functionality. If you wanted any extra bells and whistles you are out of luck. One of the major benefits of a windows managed text interface is that the sky's the limit as far as the cool features and things that can be added. The major down point is that it must be written all from scratch, and depending on what exactly needs to be done, this may take a decent amount of coding to get it right. It could take a couple hundred lines of code just to get the basic functionality of a standard console working.

In order to create a windows managed interface you start by creating a normal window. From here you need to add code to the message handler to catch key presses and mouse clicks to the window. This is important because the message handler needs to be able to act appropriately depending on the combination of presses and clicks it receives. For instance, if we are going for basic console style interaction, then the user enters an alpha-numeric value. This letter needs to be placed on the current command line string and the current value of the sting needs to be updated on the screen. If the key press is "enter" then the current command line string needs to be processed and cleared. The results need to be added to the output portion of the windows managed console. Whenever the window needs to repaint itself it is up to the programmer to tell the application which characters are to be displayed, where they are to be displayed, and how they are to be displayed. This is an overly simplified example. It should be apparent though from this simple example that there is a lot more work that is involved in creating a windows managed text interface, but again the rewards are that you can do some very cool and useful things when you handle all the text input/output yourself.

For a good example of windows managed text see the "Simulator First Run" application included in the appendix zip file.

Windows managed text interfaces are a very useful and powerful option when more than the common console is desired. Unfortunately, there is a significant amount of extra work that falls on the programmer's shoulders if this route is taken.

Along with the extra effort involved in getting one of these interfaces up and running, efficiency is another thing to consider. If you are writing a windows managed text interface that functionally is more or less the same thing as a standard console, you are adding a lot of unnecessary overhead to your application. You should probably just use a standard console and save clock cycles for something more useful.

2.3.2.2 Win32 menu test

There are a lot of menu functionalities, such as dockable toolbars, which are becoming standard in today's applications. This is one of the areas where MFC excels above basic Win32. In MFC it is easy to create menus and toolbars that intrinsically have these "standard" capabilities. If a programmer wants these kinds of bells and whistles without using MFC, he/she has to code it all up by hand. All this being said, we should not just abandon Win32 for the extra features of MFC. Remember, even though the programmer doesn't have to code up the functionality, the computer still needs to execute some code somewhere to get the job done. So an interface designer needs to decide if the extra overhead is really needed. In the case of the Escape Simulation Suite these kinds of bells and whistles are really not needed and add nothing to the application.

The other thing to be remembered is that the majority of the standard user interface functionality is available in Win32. It was anticipated when the application was being initially designed that all of the following would be needed: a menu structure, toolbars (floating and/or static), dropdown menus on right mouse click, dialog boxes with tabs and

all the usual buttons, text fields, etc… Luckily, Win32 handles all these resources very well.

2.3.2.2.1 Windows Resources

This section will briefly go over how each of these interfaces is coded up correctly. First, here is a little general background. Windows treats all of the interfaces listed at the end of the previous section as resources. In order to speed up the development and manageability of the resources in a program, the Win32 developers created a two file structure that defines all resources in an application and how they are to be used/accessed/modified. The first is the resource.rc file. This file is written in a fairly simple scripting language. Each of the different resource types has its own options and syntax. Everything is contained in this file except the actual #define values that are used to know when something is accessed, clicked, moved, etc. You have full control over the resources in this file. That being said, most of us do not want to write this file from scratch. Luckily Visual Studio has provided a nice GUI interface that allows you to create the various resources graphically and then it generates/modifies the resource.rc file accordingly. This is nice for two reasons, the first of which, as already mentioned, is that you don't have to create and modify this file by hand, although you still can if it is needed. Secondly it allows you to design these resources and actually know exactly how they are going to look as you build them.

The second file needed is the resource.h file. This file is also generated automatically by the Visual Studio resource generator, and can be modified by hand. This file is included in the project cpp files that actually use the resources and contain all the #defines that are needed to access and respond to the resources. For example, if a menu tab called ***file*** is created and one of the options under the ***file*** tab is ***exit,*** then a specific message would be generated containing the ID value that was assigned the ***exit*** entry every time the entry is slected. This ID value could be changed to be whatever the developer wanted. For instance, in this example he/she might choose that the file->exit selection would have the ID value of ID_MENU_FILE_EXIT. In the resource.h file this value would be #defined as a unique numerical value.

For each of the sections below the creation and setup of resources by using the resource editor will be covered. Other methods exist for creating windows resources. For instance, the resource.rc file can be edited/written directly using a simple text editor. No matter which of the resources it is you want to create, you will always at least have to follow these steps.

- Under the ***View*** menu tab at the top select ***Resource View***
- In the window that pops up find the specific project you want to add the resource to and expand that portion of the tree.
- Right click on the .rc file name and select ***Add Resource***
- Pick the resource you wish to add
- Build your resource in the GUI
- Add the needed hooks to your main code.

For the purpose of teaching how to create each of these, the following sections will step the reader through the creating of a fictitious application call Resources In Use or RIU for short.

Throughout this tutorial please refer to [17] for a more in-depth explanation of different resources. Also please refer to [20] in general for more information about different function calls, types, and definitions.

2.3.2.2.2 Menus

Menus are one of the easier resources to add to a project. First follow the steps above and add a new menu to your project.

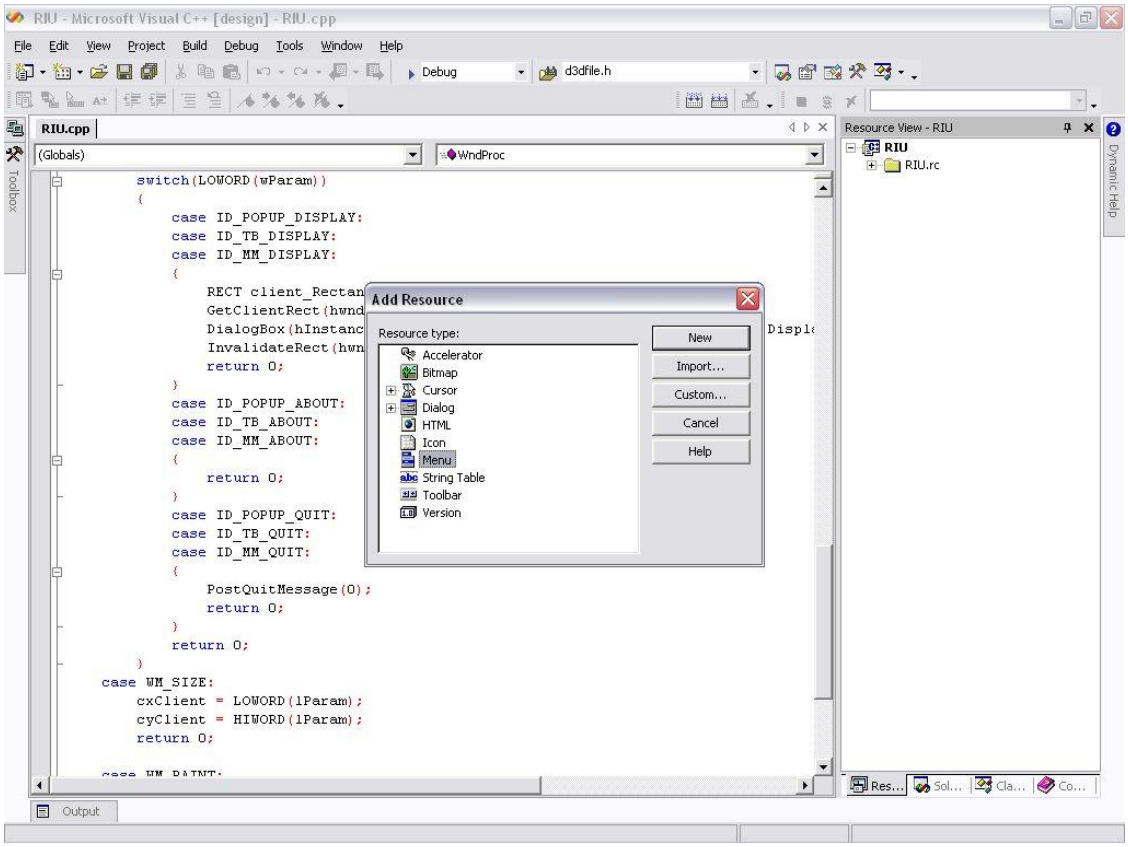

Figure 11 Adding a resource to a Win32 project

From here on out you can always come back and modify any of the resources you have created by following the first two steps above, expanding the .rc branch and the particular resource's folder type, and selecting the resource's name.

After you add the menu to the project you need to fill out the different menu options. You will see in the GUI a mock version of the menu that you are creating. To add more tabs to the main part of the menu just click on the last spot and type in the desired tab name. To add options to each main tab just select the desired tab and type the new entry

name into the bottom slot. You can add popup menus to each entry by clicking on an entry and then adding entries to the right of it.

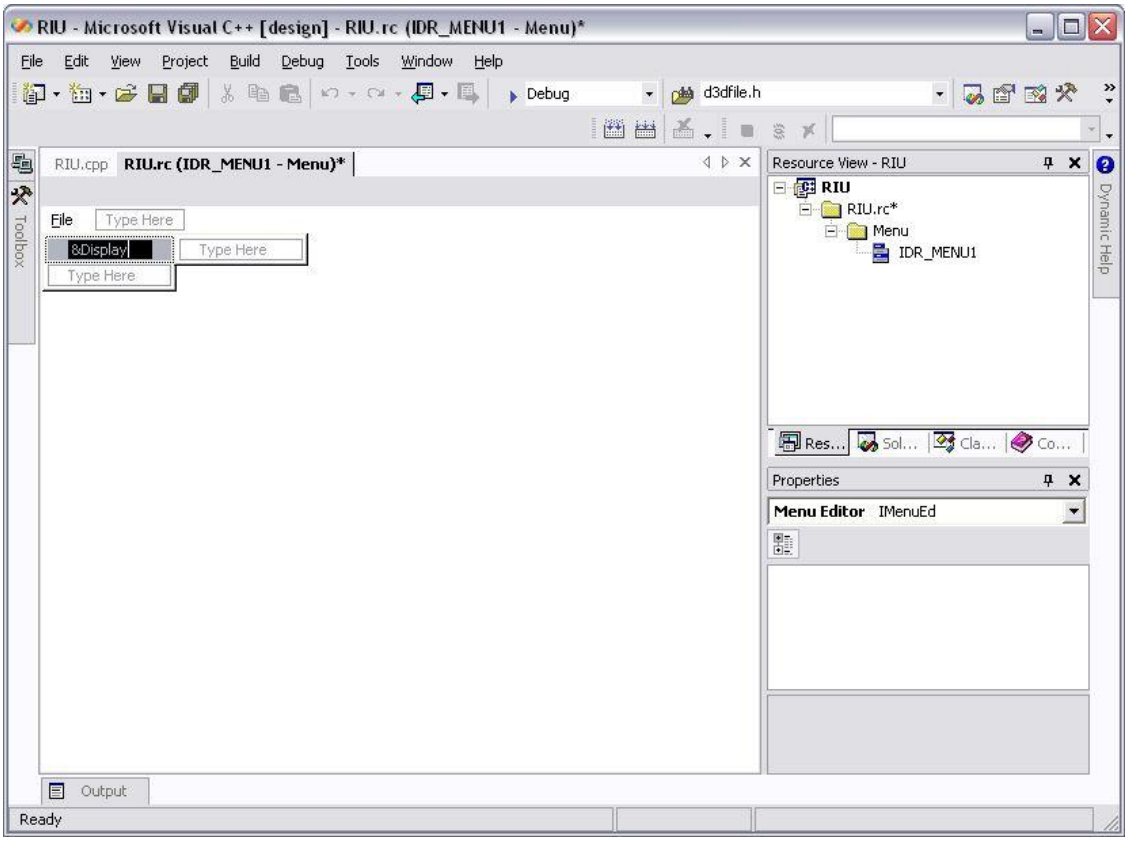

Figure 12 Adding an entry to a Win32 menu resource.

As with all windows menus, you can add accelerators to the different menu options in order to allow for quick keyboard access. Adding an '&' immediately before a specific letter when you are entering the name for the entry makes that letter the quick access value for that entry. For example:

If the value "&File" were entered (minus the quotation marks) into an entry then it will be displayed as ***File*** in the menu and the user only has to press 'f' in order to select this option. Another example is this: perhaps under the ***File*** tab we already have ***Display*** using the letter 'd,' and adding ***Done*** would cause problems. Instead, we can enter "Do&ne". Now this entry will show up as ***Done*** and the letter 'n' is the quick access value.

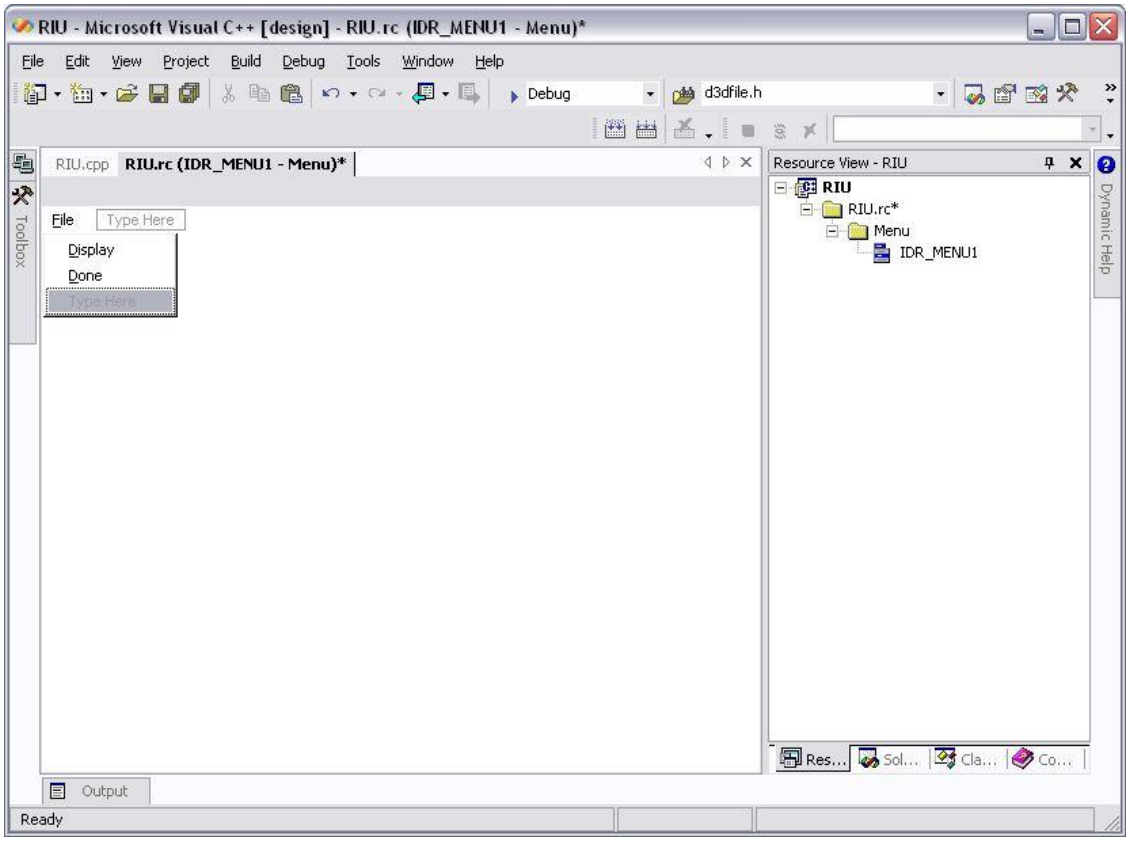

Figure 13 Screen shot of the menu before the change from "&Done" to "Do&ne".

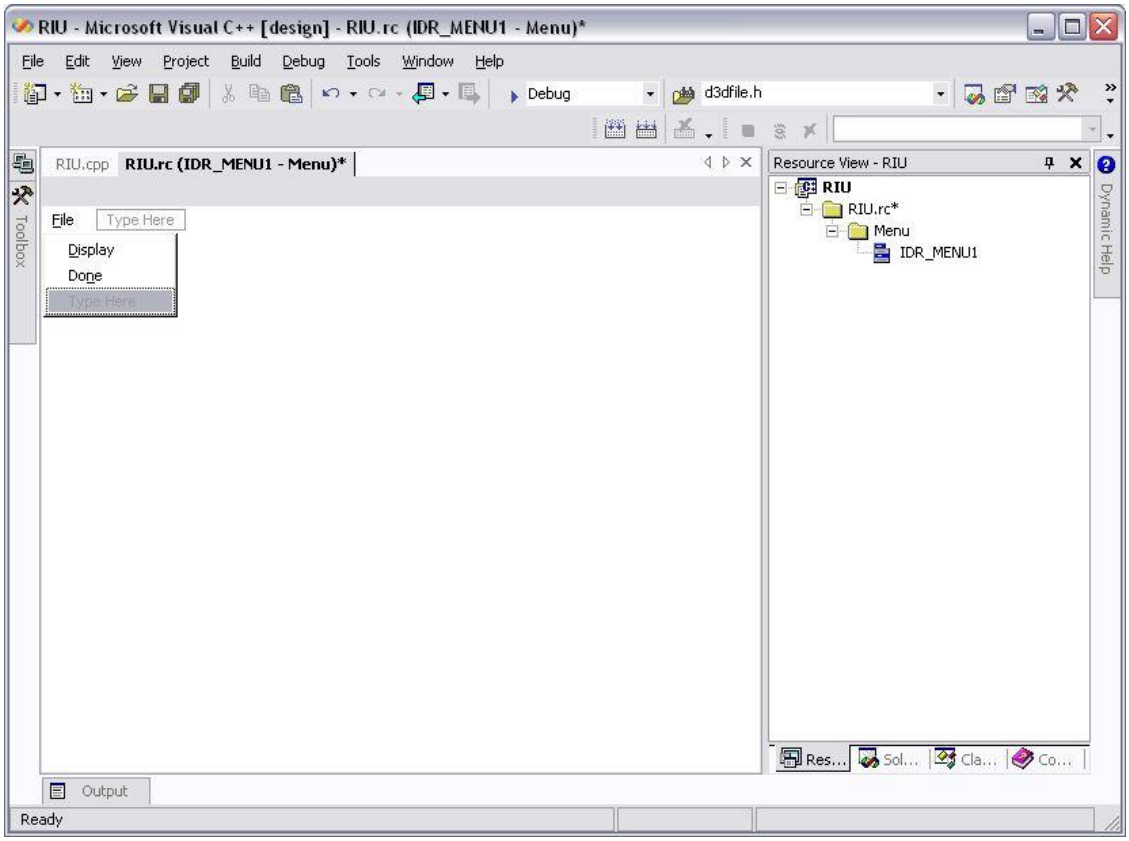

Figure 14 Screen shot of after the change

Aside from the '&' all the usual text escape values are valid here, like '\t' for instance. When we add the hooks to the code later we can add special key combinations such as "Ctrl+D" for ***Display***. If you are going to add a special key combination it is a good idea to add that to the entry string so that the user would be able to easily see the option. For example:
"&Display\tCtrl+D" would be displayed as "***Display Ctrl+D***"

If you now open up the resource.h file that has been added to the project directory and take a look at its contents you will notice that there are several new entries.

Now add two more entries to the ***File*** menu tab: "&Quit" and "&Help". At this point your menu should look like this:

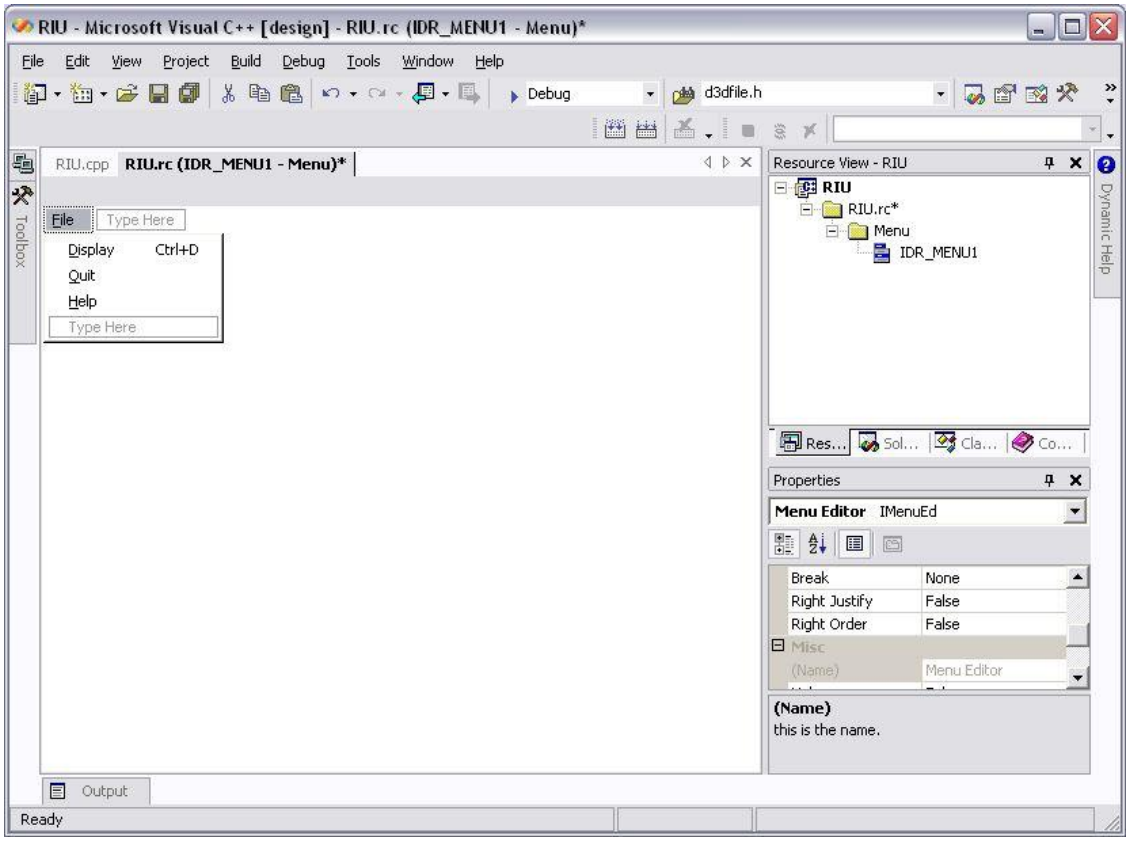

Figure 15 Screen shot of the finished menu.

Now we need to change the ID's for these entries. First highlight the **Display** entry. Open up the properties view, and in the view find the ID field and add the value of ID_MM_DISPLAY. Do this for each of the other entries, giving them the values of ID_MM_QUIT and ID_MM_HELP respectively.

Once the ID's are set, we can modify the WinProc() callback function to handle these event ID's. Note: If you open the resource.h file you will see these ID's #defined as unique numerical values.

In the WinProc() callback function we need to add a new case to the main switch statement. Whenever a button is pressed, a WM_COMMAND message is sent, so you need to add a case to catch this message.

Inside the case you have to switch on the low word value in order to see which event ID was passed. Make the case look like this:

```
case WM_COMMAND:
      switch(LOWORD(wParam))
      {
      case ID_MM_DISPLAY:
            {
```

```
                return 0;
        }
case ID_MM_ABOUT:
        {
                return 0;
        }
case ID_MM_QUIT:
        {
                PostQuitMessage(0);
                return 0;
        }
}
break;
```

Notice we did not give the cases any functionality yet. That will be done later.

2.3.2.2.3 Toolbars

Next we are going to add a toolbar to the RIU application. Toolbars are slightly more complicated then regular menus. As usual, start by going to the resource editor and adding another resource, this time of type *toolbar*. This time, instead of seeing the menu-creation GUI you will see the toolbar GUI. This GUI lets you create the actual toolbar graphics. You add these in a similar way to adding entries to a menu. Click on the blank square on the end and instead of typing the label, you draw the button.

For our example let's add three buttons. You can copy the graphics in the picture below, or create your own. We will tie the first button to the display message event. The second button will quit the application, and the last one will print out the *about* dialog box.

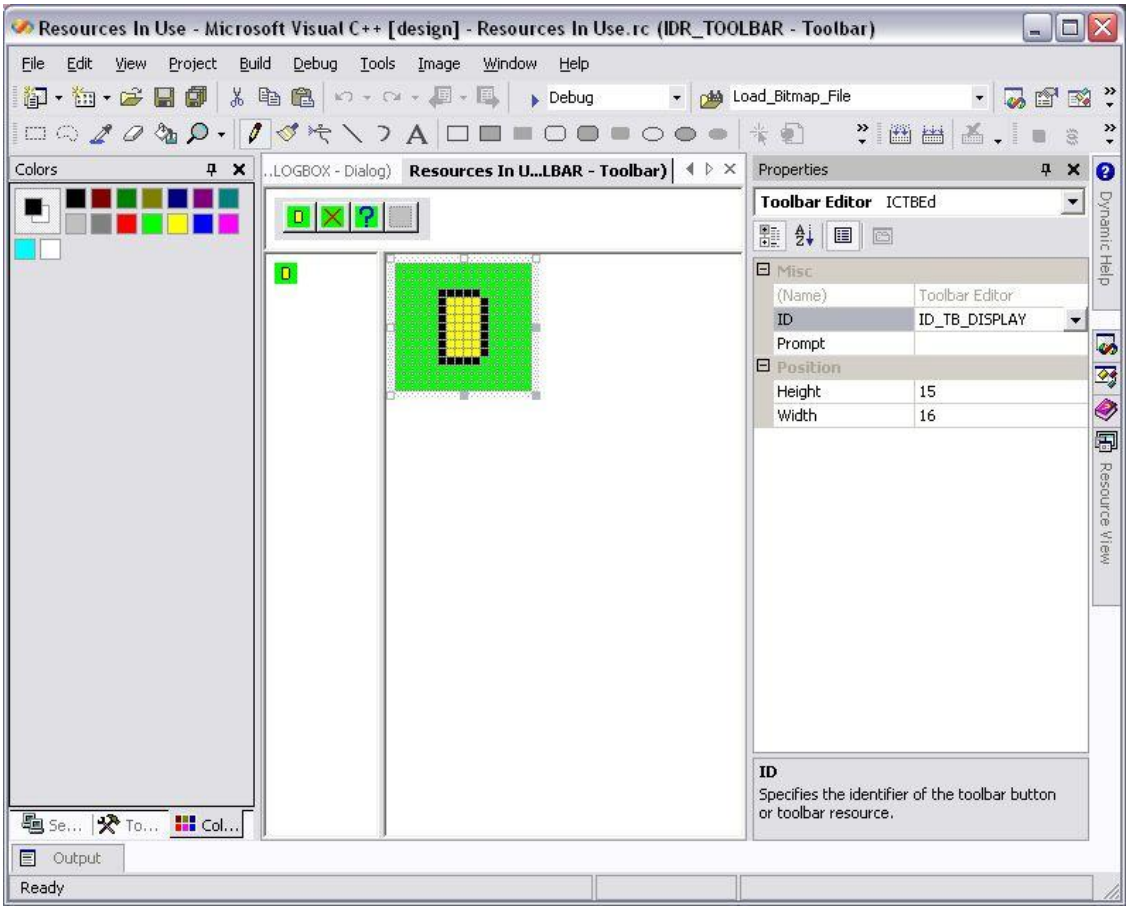

Figure 16 Toolbar creation tool and finished toolbar bitmap.

Once the buttons are drawn, open up the properties window as described in the menu section above. Look for the ID field and give each button a descriptive value. For the sake of this test application please use the following:

Button	ID Value
Display	ID_TB_DISPLAY
Quit	ID_TB_QUIT
About	ID_TB_ABOUT

You might want to get more descriptive in a more complex example so as to avoid accidentally using an identifier that is already in use.

Once you have your buttons drawn and they have all been assigned unique ID's we are ready to hit save and add the toolbar hooks to our application.

Start by going to WinMain() and below, where you created the main class and loaded the menu into the class instance, add the following code:

```
TBBUTTON tbrButtons[3];

tbrButtons[0].iBitmap   = 0;
```

```
	tbrButtons[0].idCommand = ID_TB_DISPLAY;
	tbrButtons[0].fsState   = TBSTATE_ENABLED;
	tbrButtons[0].fsStyle   = TBSTYLE_BUTTON;
	tbrButtons[0].dwData    = 0L;
	tbrButtons[0].iBitmap   = 0;
	tbrButtons[0].iString   = 0;

	tbrButtons[1].iBitmap   = 1;
	tbrButtons[1].idCommand = ID_TB_QUIT;
	tbrButtons[1].fsState   = TBSTATE_ENABLED;
	tbrButtons[1].fsStyle   = TBSTYLE_BUTTON;
	tbrButtons[1].dwData    = 0L;
	tbrButtons[1].iString   = 0;

	tbrButtons[2].iBitmap   = 2;
	tbrButtons[2].idCommand = ID_TB_ABOUT;
	tbrButtons[2].fsState   = TBSTATE_ENABLED;
	tbrButtons[2].fsStyle   = TBSTYLE_BUTTON;
	tbrButtons[2].dwData    = 0L;
	tbrButtons[2].iString   = 0;

	hWndToolbar = CreateToolbarEx(hwnd, WS_VISIBLE | WS_CHILD |
WS_BORDER | TBSTYLE_ALTDRAG,IDR_TOOLBAR, 3, hInstance,
IDR_TOOLBAR,tbrButtons, 3, 16, 16, 16, 16, sizeof(TBBUTTON));
```

These lines create a temporary array of TBBUTTONS structs. Each struct in the array describes a different button. They should be entered in the order they appear in the toolbar from left to right. Also notice the use of the ID_TB* #defines here. This is the first place these values will be used. The other place is in the handler for each specific button event. The CreateToolbarEx function will not be covered here. Suffice it to say that it is responsible for actually creating the toolbar and returning a handle to the toolbar. Error checking can be performed on the handle to ensure that the toolbar was created correctly.

The last step is to add the handlers for each of the button events. This is not too difficult. Whenever one of these buttons are pressed the assigned ID_TB_* value is sent as the low word value in a WM_COMMAND message. So we just need to add three more entries to our winProc WM_COMMAND switch statement. Here is what was added:

```
	case ID_TB_DISPLAY:
	case ID_MM_DISPLAY:
		{
			return 0;
		}
	case ID_TB_ABOUT:
	case ID_MM_ABOUT:
		{
			return 0;
		}
	case ID_TB_QUIT:
	case ID_MM_QUIT:
		{
			PostQuitMessage(0);
			return 0;
		}
```

The only case that really has anything filled in is the ID_TB_QUIT case. The ones that are missing will be filled in later. We don't know enough to call a dialog box or handle the display message. Remember the only requirement in handling a message is that you must return either 0 or the result from the default handler. So as far as the user will be able to tell, the first two buttons will not do anything at this point in the example.

The last button handler uses the PostQuitMessage(0) function. All this function does is puts a Quit message on the message queue that will tell the program to shutdown gracefully. So at the moment, clicking on this button in the toolbar is identical to hitting the close button on the top right corner of the window.

2.3.2.2.4 Dropdown menus

Popup menus are only slightly more complicated than toolbars. Lets start by adding another resource in the usual manner by picking menu type again. Give it the *ID* value of IDR_POPMENU. As far as the resource manager is concerned, there is no difference between a popup menu and a normal menu.

Let's label the first menu tab as ***Popup***. After that, add three entries under that tab. Call them ***Display***, ***Quit***, and ***About***.

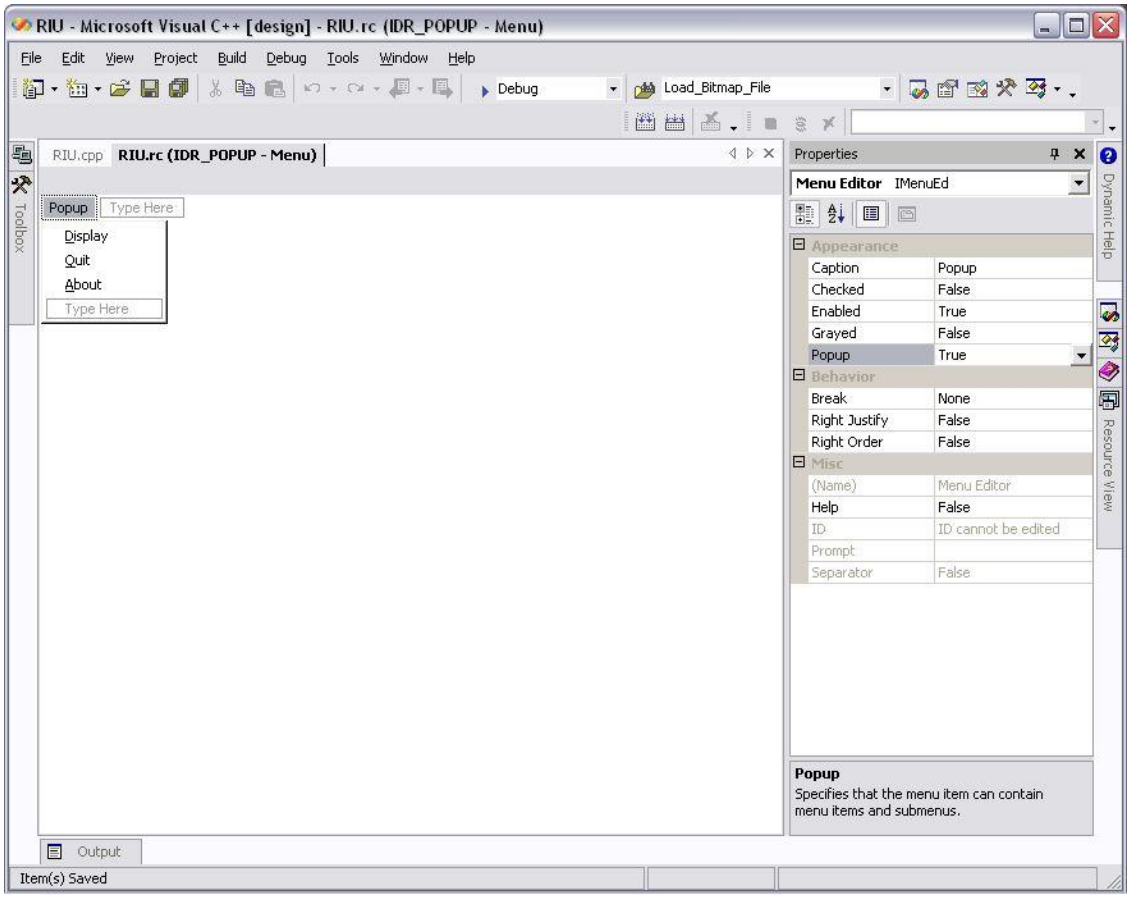

Figure 17 Completed popup menu and properties.

After the tab and entries are done we need to make a few changes to the properties of each. In the ***Popup*** tab's property window look for the *popup* option and set it to **true**. Next, go to each of the entries and in its *ID* field give them a nice descriptive ID value. For instance:

Entry	ID Value
Display	ID_POPUP_DISPLAY
Quit	ID_ POPUP_QUIT
About	ID_ POPUP_ABOUT

Okay, now the resource is ready to be hooked into the program. Unlike the other two resource types all the extra code will be placed directly into the WinProc() callback function. There will be no extra code added to WinMain().

When a window is first created there is a special `WM_CREATE` message that is sent to the WinProc() function. Any special instantiation work should be placed in this case handler. In our case we need to tell the window that the popup menu exists here. Add the following code to the message handler:

```
case WM_CREATE:
      // Save important values for future use
      …
      // Load and create up popup menu
      hpopmenu = LoadMenu(hInstance,MAKEINTRESOURCE(IDR_POPUP_MENU));
      hpopmenu = GetSubMenu(hpopmenu,0);
      …
```

The first of these two lines loads the menu we created. Then the second allows us to grab all the information under the first tab. You will notice the IDR_POPUP_MENU value is used here with the MAKEINTRESOURCE() function to actually access the resource we just created. Remember that after you create this resource you can change the default identifier it assigned be going to the resource tree and selecting the menu name and then finding the *ID* value in the properties window. In both cases a handle to the resource or the desired portion of the resource is returned. We will use this handle to actually make the menu popup.

We can tie the displaying of a popup menu to any event we want. Since a popup menu typically "pops up" when the user right clicks the mouse, lets do the same. In order to do this, we need to add a new case to our message handler to catch right mouse clicks. We actually have several options here. Do we want to grab the right mouse down event, right mouse up event, or the right mouse double click event? Usually you will want to grab the up event, so here is how to do it:

```
case WM_RBUTTONUP:
      {
            POINT point;
            point.x = LOWORD(lParam);
            point.y = HIWORD(lParam);
```

```
            ClientToScreen(hwnd,&point);

            TrackPopupMenu(hpopmenu,TPM_RIGHTBUTTON,point.x,point.y,0,
            hwnd,NULL);

            return 0;
      }
```

There are a couple of things happening here that should be explained. First, in thinking about how popups usually behaves, when the button is clicked the popup window appears, but not just anywhere, it will pop up where the mouse curser currently is. In order to get this behavior we need to tell the popup menu where the mouse is. Conveniently enough when the operating system sends a mouse click event it stores the current curser's x and y positions in the low and hi word respectively of the left parameter value sent along with the message.

In the above code we store these values in a POINT structure. We do this so that we can use the ClientToScreen() function that translates the client (x,y) positions to the actual screen (x,y) positions. We need to do this because TrackPopupMenu() expects screen coordinates.

The last part of this code that needs an explanation is the TPM_RIGTHBUTTON flag that is passed into the TrackPopupMenu() function. This flag tells the popup menu that it should accept both right and left mouse clicks.

Now we are ready to modify the WM_COMMAND message switch case statement to take into account the new menu messages that will be sent when one of the selections are clicked. Here we will just add *fall through* cases to the toolbar cases. That way we can avoid adding as much extra code as possible. So, our switch statement now looks like this:

```
      case ID_POPUP_DISPLAY:
      case ID_TB_DISPLAY:
      case ID_MM_DISPLAY:
      {
            return 0;
       }
      case ID_POPUP_ABOUT:
      case ID_TB_ABOUT:
      case ID_MM_ABOUT:
       {
            return 0;
       }
      case ID_POPUP_QUIT:
      case ID_TB_QUIT:
      case ID_MM_QUIT:
       {
            PostQuitMessage(0);
            return 0;
       }
```

Now the RIU application as a working popup menu! On to dialog boxes.

2.3.2.2.5 Dialog boxes

These are perhaps the most complex of the resources that can be created. In our simple example we will only scratch the surface of the possibilities available with dialog boxes.

We are going to create two boxes for our simple example. The first will just display some information about this example. After that we will make a slightly more complex box that we will attach to the ID_TBDISPLAY and ID_POPUP_DISPLAY messages.

To make our first box we go through the same routine as creating any other resource, but when you chose to create a new dialog box you will be asked what type of box you would like to make. You should have the following options:

IDD_DIALOGBAR
IDD_FORMVIEW
IDD_OLE_PROPPAGE_LARGE
IDD_OLE_PROPPAGE_SMALL
IDD_PROPPAGE_LARGE
IDD_PROPPAGE_MEDIUM
IDD_PROPPAGE_SMALL

The option we want is the IDD_PROPPAGE_SMALL dialog type. After selecting this option and giving it the *ID* IDD_ABOUT_DIALOGBOX, you will see a nice blank rectangle in the resource editor. We now need to add a few things to it.

Under the ***View*** tab at the top, select the ***Toolbox*** option, or you can click *Ctrl+Alt+X*. As the name implies, the toolbox has a bunch of different tools that you can drag into your blank dialog box to get it to do what you want. Since this is an *about* dialog box it will need limited functionality. It will need to have a line or two about what the application is, and it should have a button that the user can click to close the dialog.

To get the text into the dialog box look for the *Static Text* tool in your toolbox and drag it onto the dialog box. Next, find a *Button* tool and do the same. Once you have a tool in the dialog box you can move it and resize it as you like. After you have both tools where you want them, change their properties so that they will do what we want:

First, select the *Static Text* tool, and in the property menu look for the *caption* option. Enter some kind of descriptive string here, for instance:

“This application tests simple resource functionality”

You can also change the *ID* field to something unique. However, this is not as important because unless you are planning on changing the text in this tool dynamically, there is no need to access it.

Now let's setup the button. Select it in the dialog box and edit its caption to something descriptive—"OK" perhaps. Also, give it a good *ID* value like, ID_ABOUT_OK. This is what the dialog looks like at the moment.

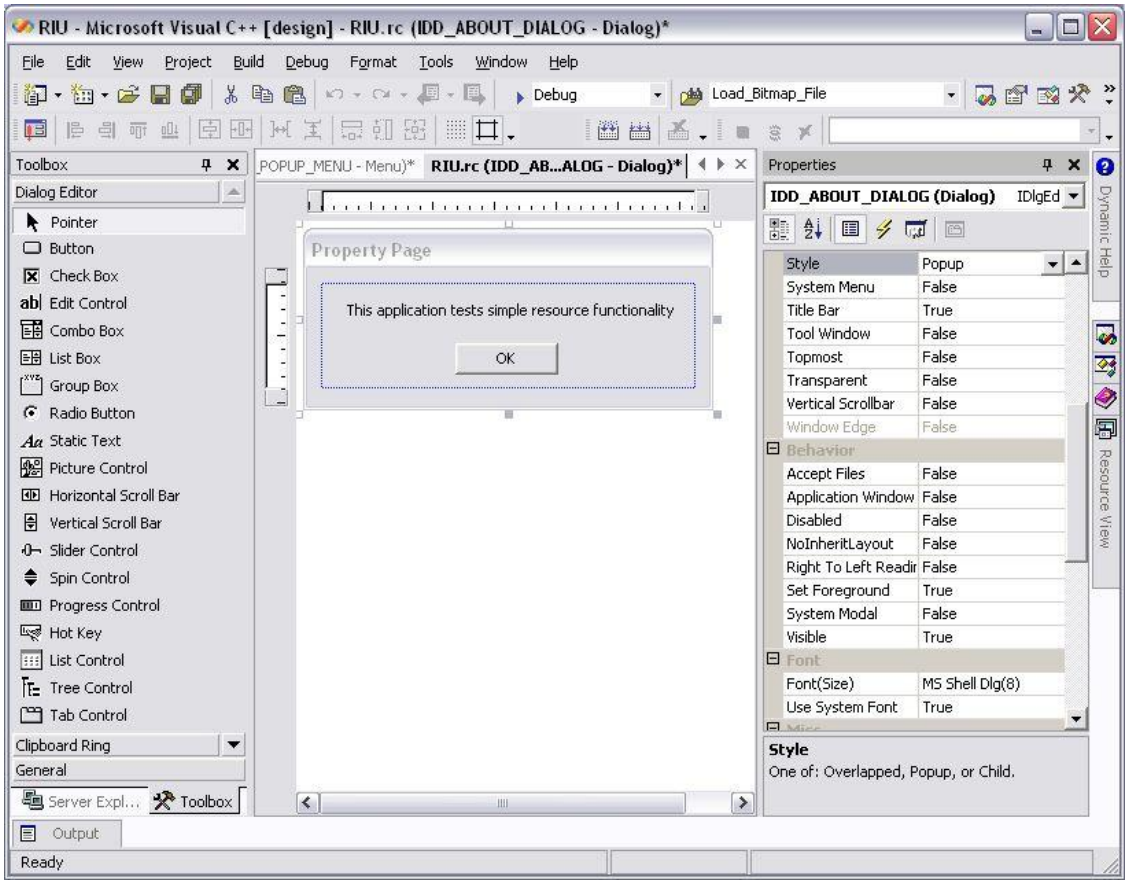

Figure 18 Completed "About" dialog box.

Finally, make sure both tools are set to visible, and the dialog boxes properties are set to:

Property	**Value**
Style	Popup
Set Foreground	True
Visible	True
Title Bar	True
Caption	Something informative. Ex. "About Resources in use Application"
Border	Thin
ALL OTHERS	False

You can play with all these options later. The three most important options are: Style, Set Foreground, and Visible. If these are not set correctly then the dialog box will not work correctly.

Now that the dialog box is created and has the correct properties, we need to create a callback function for this dialog. Dialogs can be viewed as a special type of window and like all other windows it receives messages from the operating system and other sources.

Like the regular Windows callback functions, this one follows a simila pattern. In fact, the only difference is that this callback function should return a BOOL value instead a HRESULT. The prototype looks like this:

```
BOOL CALLBACK AboutDialogBoxProc(HWND hDlg, UINT message, WPARAM
wParam, LPARAM lParam);
```

By now everything in this prototype should be pretty standard. The function behaves exactly like the main WindProc() callback function. It receives a WM_INITDIALOG message when the dialog pops up. And when the user interacts with the dialog it gets a WM_COMMAND message with the actual event id being sent in the wParam argument's low word.

The actual definition looks like this:

```
{
	switch(message)
	{
	case WM_INITDIALOG:
		return TRUE;
	case WM_COMMAND:
		switch(LOWORD(wParam))
		{
		case IDC_ABOUT_OK:
			EndDialog(hDlg,0);
			return TRUE;
		}
		break;
	}
	return FALSE;
}
```

The EndDialog() function is similar to PostQuitMessage(). It sends an end message to the dialog box telling it to close.

Luckily, in this case the dialog box is not doing anything that requires information from the application and the application does not receive any output from this dialog. So the callback function is very simple and the hooks into the WinProc() function are a single line.

Now we just need to change the WinProc() function to make it actually call this dialog box when any of the about buttons or menu options are clicked. To do this, simply make the following change to the callbacks WM_COMMAND switch statement:

```
	case ID_MM_ABOUT:
	case ID_TB_ABOUT:
	case ID_POPUP_ABOUT:
```

```
{
        DialogBox(hInstance,MAKEINTRESOURCE(
        IDD_ABOUT_DIALOGBOX),hwnd,AboutDialogBoxProc);
        return 0;
}
```

The DialogBox() function looks up the resource identified by its second argument, assigns the callback function passed to it in the fourth argument to the resource, and attaches the dialog to the specified window and application instance. This is why the dialog's properties are very important. If we had not specified visible as true then after the above function call we wouldn't see anything. If we had not specified foreground then the dialog would not have focus and we would not be able to give it input. Finally, if we had not made it popup then it would not be visible or take in input. However, as it is, when the user click/selects the about option from the toolbar or popup menu then the dialog will be displayed and stay in the foreground until the user clicks on the OK button, at which time the dialog will receive a close message and shut down, returning execution to the WinProc() function. Running the application and clicking on any of the about buttons or menu options results in the following:

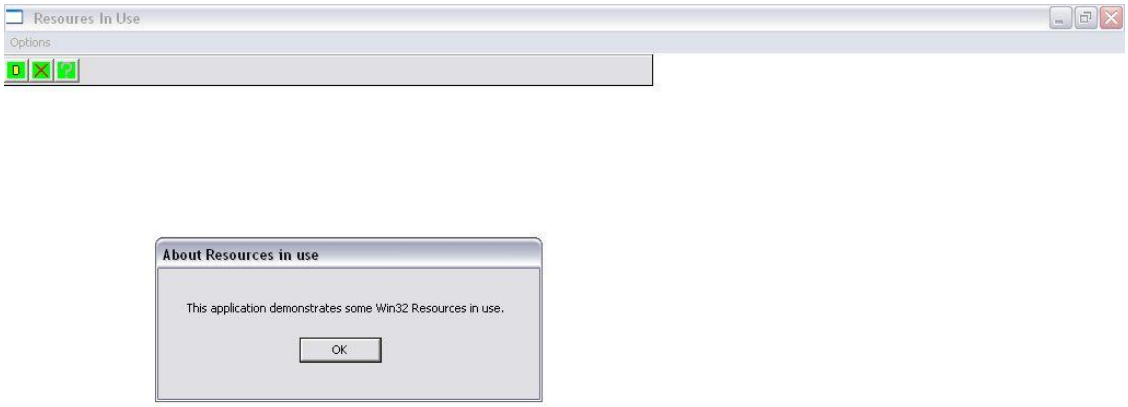

Figure 19 Screen shot of RIS application using the "About" dialog box.

Now let us take a look at what it takes to do something a little more involved with dialogs. Let's make the display dialog interact with the WinProc() and the main application. Also, let's make the main application modify the dialog as well.

In tribute to the age old application "*Hello World,*" we will be making a dialog box that allows the user to select whether or not the phrase "Hello World" is displayed on the screen or not. Start by creating another dialog just like the first. Assign this one an ID of IDD_DISPLAY_DIALOGBOX. Add to this box two buttons, labeling one "Cancel" and the other "OK". Give them the ids IDC_DISPLAY_CANCEL and IDC_DISPLAY_OK.

Now find the check box tool and add one of these to the dialog box. Give it a descriptive caption such as "Display "Hello World"", and ID it as IDC_DISPLAY_CHECK.

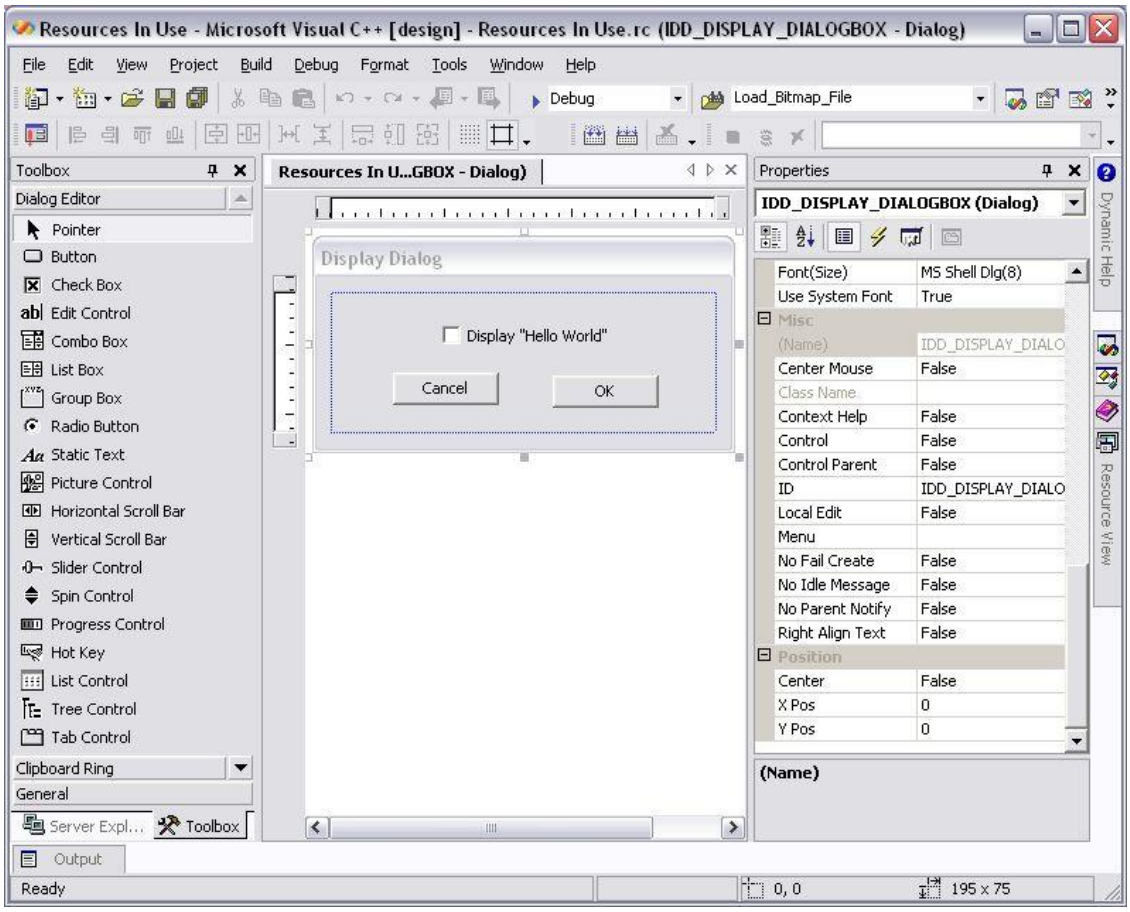

Figure 20 Finished "Display" dialog box creation.

So far nothing much has been new. Before we create the callback function for this dialog box we need to decide exactly how we will get data in and out of this dialog. Typically, there are a couple ways to do this. The first and easiest is to give the dialog box access to some variable that has scope outside the dialog box.

While many beginning programmers will argue that this option is not the correct one because their programming teacher told them global variables are bad, this is not true. When used properly, global variables are a very useful programming tool. In fact, most large programs, e.g. games, have at least a few global variables in them, if not a large number of them.

Another option is to have the main window send a special message to the dialog with the needed information in it, and have the dialog send a special message back to the main window with whatever output it generates.

Because this is meant to be a simple example, the easy way out will be taken and a global variable will be used. So at the top of the .cpp file add the following booling variable:

```
bool display_hello_world;
```

Now that we know which variable to check and modify, we can go ahead and write the dialog box's callback function. To some degree it is the same as the first dialog box, but we need to add more code to the initialization function, and also add code to make it update display_hello_world based on the state of the checkbox when the *OK* button is clicked and to discard any changes if the *Cancel* button is clicked.

So the WM_INITDIALOG case becomes:

```
	case WM_INITDIALOG:
	{
		// Check the global flag.  If true set the check box.
		// Otherwise uncheck it.
		if(display_hello_world)
		{
			CheckDlgButton(hDlg,IDC_DISPLAY_CHECK,BST_CHECKED);
		}
		else
		{
			CheckDlgButton(hDlg,IDC_DISPLAY_CHECK,BST_UNCHECKED);
		}
		return TRUE;
	}
```

For the most part this should be self explanatory. The only new function introduced is the CheckDlgButton() function. It takes in the handle to the desired dialog, the *id* value of the checkbox to modify, and the desired setting to make it. This last argument can be one of three values:

```
BST_CHECKED
BST_UNCHECKED
BST_INDETERMINATE
```

The first two are obvious but the third is only used if the check box was set to be a 3-state check box. If it is, then the last value will set the checkbox to grayed.

To finish up the callback add the following code to the WM_COMMAND case:

```
	case WM_COMMAND:
	{
```

```
            switch(LOWORD(wParam))
            {
            case IDC_DISPLAY_OK:
                  {
                  // Check the Check box.
                  // If it is checked then set the global display
                  // flag to true.  Other wise to false

                        if(IsDlgButtonChecked(hDlg,IDC_DISPLAY_CHECK)
                              == BST_CHECKED)
                        {
                              display_hello_world = true;
                        }
                        else
                        {
                              display_hello_world = false;
                        }
                  }
            case IDC_DISPLAY_CANCEL:
                  {
                        EndDialog(hDlg,0);
                        return TRUE;
                  }
            }
            break;
      }
```

If the command is IDC_DISPLAY_OK then the IsDlgButtonChecked() function is called. This function takes as its last argument the same three possible values as the last argument of CheckDlgButton(). In our case we want to see if the button is checked. If the function returns BST_CHECKED then it is. If it returns anything else then we assume that it is not checked.

Now in an effort to shrink the amount of code we have to write we make the IDC_DISPLAY_OK case fall through to the IDC_DISPLAY_CANCEL case since that portion of code is common to both cases.

Now the dialog box is ready to be hooked into the main application. We will start by modifying the main callback function WM_CREATE case to initialize the display_hello_world variable by adding, anywhere in the case statement, the line:

```
display_hello_world = false;
```

Next, we need to add some code to get the dialog box to display when it is needed. To do this we modify the WM_COMMAND case statement for the display case like this:

```
case ID_POPUP_DISPLAY:
case ID_TB_DISPLAY:
case ID_MM_DISPLAY:
{
      DialogBox(hInstance,MAKEINTRESOURCE(IDD_DISPLAY_DIALOGBOX)
                  ,hwnd,DisplayDialogBoxProc);
      return 0;
}
```

Now our dialog box will display and modify the state of the display_hello_world variable correctly. There are only two more things to add.

First, we need to add the code to the WM_PAINT case statement to actually display "Hello World!" At the moment, all we are doing is getting a handle to the device context using the BeginPaint() function and then immediately releasing it. Remember all this does is validate the entire device context region. So, we now want to add the following code between the BeginPaint() function call and the EndPaint() call:

```
if(display_hello_world)
{
      RECT text_Rectangle;
      char text[] = "Hello World!";

      text_Rectangle.top            = (cyClient >> 1) - 10;
      text_Rectangle.left           = (cxClient >> 1) –
                                     ((int)strlen(text) << 2);
      text_Rectangle.bottom         = (cyClient >> 1) + 10;
      text_Rectangle.right          = (cxClient >> 1) +
                                     ((int)strlen(text) << 2);

      DrawText(
            hdc,                    // handle to DC
            "Hello World!",         // text to draw
            (int)strlen(text),      // text length
            &text_Rectangle,        // formatting dimensions
            DT_CENTER               // text-drawing options
            );
}
```

To begin, let me explain DrawText(). This function draws the specified text (second parameter), of the specified width (third parameter), on the specified device context (first parameter), within the given rectangle (fourth parameter), aligning that text based on the given flags (last parameter).

In our case we want to print "Hello World!" centered in the defined rectangle. In order to define a rectangle all we need are two points. Windows RECT structure wants us to define the top left point and the bottom right point. We define the top left point as having an x-value of half the width of the client rectangle minus the length of the string we are printing multiplied by 4. The y-value is half the context height minus 10. The bottom right point is similarly defined with plusses instead of minuses. This particular box definition creates a box the correct size to display all the text and centers it in the middle of the context screen.

For those not familiar with the << and >> notation, they represent left and right bitwise shifting respectively, which is the same thing as multiplying or dividing by 2, raised to the number of positions shifted. These operations are only valid for integers and are much faster than doing actual multiplication and division.

Remember that windows will only send a WM_PAINT message when a portion of the window has become invalid and only the portion of the device context that is invalid will be redrawn. So, unless the display dialog box covers the middle of the window, when it closes the middle of the window will not be redrawn. In short, changes to the display_hello_world variable will not be seen.

The last thing we need to do is fix this problem. The easiest way to do this is to go back and add a few lines to the case where we call the dialog box. These lines will get the entire device context window and invalidate the whole thing. Here is the code:

```
case ID_POPUP_DISPLAY:
case ID_TB_DISPLAY:
case ID_MM_DISPLAY:
{
      RECT client_Rectangle;
      GetClientRect(hwnd,&client_Rectangle);

      DialogBox(hInstance,MAKEINTRESOURCE(IDD_DISPLAY_DIALOGBOX),
                hwnd,DisplayDialogBoxProc);

      InvalidateRect(hwnd,&client_Rectangle,true);
      return 0;
}
```

The GetClientRect() function gets the entire device context rectangle and returns the rectangle in the &client_Rectangle variable. The InvalidateRect() function invalidates the region of the device context defined by &client_Rectangle.

There you go! That is it. If you have been following along then you should now have an application that has a menu, popup menu, toolbar, and two dialogs. The first simply displays an *about* message and the second allows the user to turn the “Hello World” message on and off. If you were not following along or just do not want to type everything out then the fill .cpp code can be found in the appendices of this document. And the source code, including the resource.h and .rc files, is available for download with this paper. The results from creating the last dialog box should look something like this:

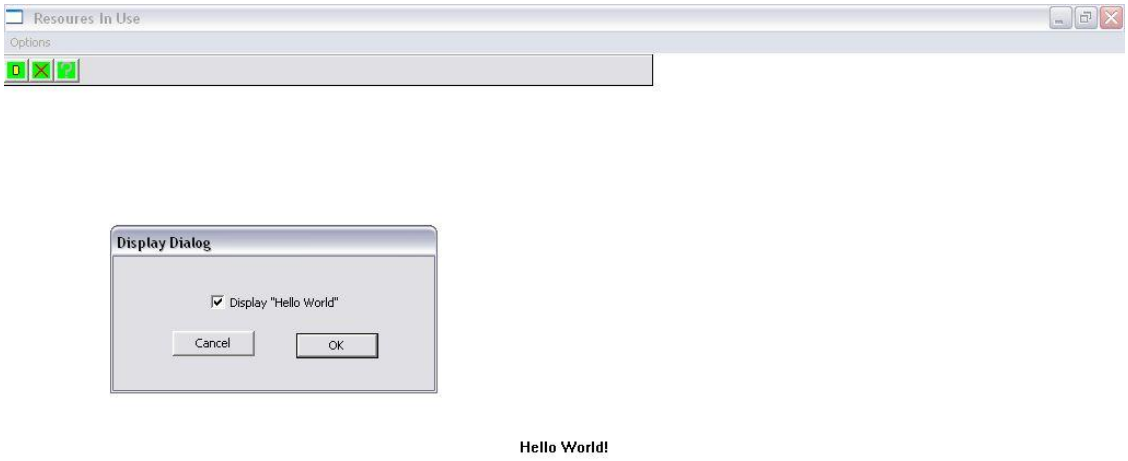

Figure 21 Screen shot of RIS application using th "Display" dialog box.

Again, this code is only a very simple example. There are hundreds more functions, tools, option, etc, that are associated with resources and managing/using them. The above information should be sufficient for someone to understand the Escape Simulation Client code and what it is doing. If you want more information please refer to *Programming Windows: The definitive guide to the Win32 API* by Charles Petzold, chapters 9-11.

2.3.2.3 Win32/directX hybrid

How to integrate the DirectX API's with a Win32 application was one of the concerns raised in developing this application. To that end several test applications have been written that have incorporated some or all of the different DirectX API's in Win32 applications. Most of these have already been discussed to some degree in previous sections so, in order to avoid too much redundancy, only the integration method that was finally decided upon for the Escape Simulation Suite will be covered here.

About 3-5 years ago the developers of DirectX came out with a standard development interface to aid in the creation of Direct3D applications. It was designed as a base class which developers could inherit from, overloading whichever methods they wanted to extend beyond the basic default functionality .

This framework is used by the Escape Simulation. It will not be covered in this document, but [5] is a good reference, particularly chapter 5 of [5].

2.3.3 Console

For most C or C++ programmers, console applications are their starting point and bread and butter. It is assumed that any reader of this thesis has written many of these applications. For this reason, the basics of console applications will not be covered here at all. Rather this section will cover only one aspect of console applications, and that is how to get the hInstance of a console application. The reason this is useful is that it allows a programmer to use many of the DirectX components that normally would require a Win32 application at the least.

2.3.3.1 Acquiring a console's hwnd

In several DirectX components it is necessary for the component to be given the hwnd of the window or the instance of the component it is being tied to. If one is using Win32, acquiring this handle is a simple matter, but if the application is a pure console application then the developer is given no easy way to do this. At one point a sound module was developed to test out the possibility of adding sound to the Escape Simulation. This may still happen in the future, but the addition of sound will have to wait on more important features.

One of the results of creating this module, though, was to learn how to acquire the hwnd of a console application.

To do this the following code should be used:

```
GetConsoleTitle(oldWindowTitle,1028);// Grab the current console title
memset(newWindowTitle,'\0',1028);    // Clear out newWindowTitle buffer
strcpy(newWindowTitle,"Please Wait: Getting Sound Window Handle....");
                                     // Set value for newWindowTitle
                                     // buffer
SetConsoleTitle(newWindowTitle);     // Set the console title to
                                     // newWindowTitle
Sleep(40);                           // Sleep for 40 msec to make sure
                                     // the title change takes effect

hwndFound = FindWindow(NULL,newWindowTitle); // Find the desired hwnd

SetConsoleTitle(oldWindowTitle);     // Restore the console's original
                                     // title
```

The comments here do a pretty good job of describing what is going on. The important thing is that the value of newWindowTitle must be unique, otherwise FindWindow() is not guaranteed to find the right window.

2.3.4 DLLs

The last area of research was on the creation and use of dynamically linked libraries (DLLs). These are used in the Escape Simulation Suite to give users a way to extend the default functionality to meet their specific needs.

DLLs are, as the name implies, libraries that can be added to an application at runtime instead of at the initial compilation. There are many benefits to using DLLs, as well as many down points. The assumption has been made that the reader has access to and is familiar with Visual Studio.net 2003 or can look up the way to tell their favorite compiler how to compile a project as a DLL.

In order to illustrate the DLL basics, a simple DLL application will be created here. The main application will consist of a base class that the DLL will extend. It will also consist of main(), that will simply load up the DLL, run the class exporter function to get an instance of the extended class, run a few methods directly from the class, and call the second exported function to destroy the class instance.

Although this topic will be covered in more depth than other topics thus far in this paper, the reader is still encouraged to refer to [18] and [19].

The model is very similar to the one used in the Escape Simulation.

Start by creating a blank solution. Call it whatever you would like ("bob" or "blah" are the author's personal favorites). Next, add two projects to this solution. The first is a normal blank console application. The second should be the same, except that in the application settings window, where you check *blank project*, you need to also check the *DLL* box. This tells Visual Studio that this project should be compiled and linked as a DLL.

Start by creating the base class that we will extend. Do this by adding a .h file to the first project. Again, call it whatever you want. This file will be referred to hereafter as base_class.h. In this file enter the following:

```
#ifndef BASE_CLASS_H
#define BASE_CLASS_H

class base_class
{
public:
	virtual void set_base_string(char *str) = 0;
	virtual int get_base_sting_length() = 0;
	virtual void print_base_string() = 0;

	virtual void set_derived_string(char *str) = 0;
	virtual int get_derived_sting_length() = 0;
	virtual void print_derived_string() = 0;

protected:
	char base_class_string[1028];
```

```
};

#endif
```

Notice that this is a purely virtual class. Therefore you cannot actually create an instance of it. It must be inherited. In fact, this class is completely virtual so there is no .cpp file for it nor and inlined definitions for the methods in this class.

This file needs to be included in both projects. It is the link between them. So, you can put it in a common folder and modify your project's *include p*ath, if you know how, or you can just copy this file into both project directories and include it in both projects.

Next, we need to create the derived class. Create a new .h file in the DLL project. You would normaly call it whatever you want, but for the purposes of this sample, please call these files derived_class.h and the corresponding .cpp, to be created in a second, derived_class.cpp. In the .h add the following:

```
#ifndef DERIVED_CLASS_H
#define DERIVED_CLASS_H

#include "base_class.h"

class derived_class : public base_class
{
public:
	derived_class();
	~derived_class();
	virtual void set_base_string(char *str);
	virtual int get_base_sting_length();
	virtual void print_base_string();

	virtual void set_derived_string(char *str);
	virtual int get_derived_sting_length();
	virtual void print_derived_string();

private:
	char derived_class_string[1028];

};

#endif
```

Finally add the following to the derived_class.cpp:

```
#include <iostream>

using namespace std;

#include "derived_class.h"
#include <string.h>

derived_class::derived_class()
{
```

```
	cout << "Derived Class Constructor" << endl;
}
derived_class::~derived_class()
{
	cout << "Derived Class Destructor" << endl;
}


void derived_class::set_base_string(char *str)
{
	strcpy(this->base_class_string,str);
}

int derived_class::get_base_sting_length()
{
	return (int)strlen(this->base_class_string);
}

void derived_class::print_base_string()
{
	cout << this->base_class_string;
}

void derived_class::set_derived_string(char *str)
{
	strcpy(this->derived_class_string,str);
}

int derived_class::get_derived_sting_length()
{
	return (int)strlen(this->derived_class_string);
}

void derived_class::print_derived_string()
{
	cout << this->derived_class_string;
}
```

Hopefully nothing in the above code is that unusual. Now we need to add the code to actually export this class. To do this, create two more files in the DLL project called dll_exporter.h & .cpp.

In the .h put the following:

```
#ifndef DLL_EXPORTER
#define DLL_EXPORTER


#include "base_class.h"

#define EXPORT __declspec(dllexport)

EXPORT base_class *	Create(void);
EXPORT void Destroy(base_class *ptrBC);
```

```
#endif
```

Here is the first unusual thing we have seen. Windows has created this special identifier __declspec(dllexport) that tells the compiler that the given function with the identifier is to be added to the export table. In order to make your code more readable you can go ahead and make a #define as above.

Next lets add the following to the .cpp:

```
#include "dll_exporter.h"
#include "derived_class.h"

EXPORT base_class *     Create(void)
{
      derived_class     *ptrDC;
      ptrDC = new derived_class();
      return (base_class *)ptrDC;
}

EXPORT void Destroy(base_class *ptrBC)
{
      derived_class     *ptrDC = (derived_class *)ptrBC;
      delete ptrDC;
}
```

Here we are doing a little polymorphism so that the caller of these functions do not need to include derived_class.h only base_class.h.

We are almost finished with the DLL project. There are two ways to export DLL symbols. In this example, the long way is shown. Even though it is longer, it actually gives you a lot more power.

We need to create a .def file for the DLL project and tell the project to use it. To do this just add a new file to the project. Normally this file could be called whatever you like, but for our purposes, call this file dll_example.def. In the file add these lines:

```
LIBRARY dll_example

EXPORTS
      Create      @1
      Destroy     @2
```

The words LIBRARY and EXPORT are special. The first tells the compiler what to call the library. Make this value the same as the project name. The second tells the compiler you are listing the function names to be exported. In our case these are Create and Destroy. The @1 and @2 values after the function names are called ordinal numbers. These are used because it is faster for the application that loads your DLL up to look up functions by ordinal. One of the benefits of using a .def file is you can pick these values; otherwise they are generated for you and can change from compile to compile so you can

not hard code them in your calling application. In our case we will be looking them up by name so it really does not matter for us.

In order to tell the project to use the .def fil,e open up the project's properties. Under the *linker* branch on the left, select *input*. Now look for the field on the right labeled *Module Definition File*. Type the name of the .def file into this line, in this case dll_example.def.

Okay, the DLL project is done. It should be able to compile now without any problems. If not, please go back over this section and make sure all steps were followed.
Now all we have to do is write the main application and use the DLL. Start by adding a .cpp file to the main application project. Include the base_class.h file which should already be a part of this project.

To make things that follow easier, we are going to create two type definitions. These are going to be type definitions for two function pointer types. Please add these two lines to the .cpp:

```
typedef base_class *(*CREATE)(void);
typedef void (*DESTROY)(base_class *);
```

Next we need two function pointers of these types:

```
CREATE create_class;
DESTROY destroy_class;
```

These are the two functions we will need in order to load the DLL and access the two exported functions. To load the DLL we call:

```
HMODULE LoadLibrary(LPCTSTR lpLibFileName);
```

This function takes a normal null terminated c-string containing the DLL name, including path if needed, and returns a handle to the module. If the DLL could not be found it will return NULL. The second function is:

```
FARPROC GetProcAddress(HMODULE hModule, LPCTSTR lpProcName);
```

This function takes in the handle to the DLL which was returned by LoadLibrary() and a c-string containing the name of the exported function you want. This name is the same one used in the .def file.

Okay, we now have everything ready and just need to add the code for main():

```
int main()
{
      HMODULE dll_handle;      // Holds the handle to the dll when it is
opened
      base_class *ptrBC;       // Will point to the class when it is
created
```

```
        // Now we need to load the dll.
        dll_handle = LoadLibrary("../dll_example/debug/dll_example.dll");
        if(dll_handle)
        {
           create_class = (CREATE)GetProcAddress(dll_handle,"Create");
           destroy_class = (DESTROY)GetProcAddress(dll_handle,"Destroy");
           if(create_class && destroy_class)
              {
                    // Create an instance of the derived class
                    ptrBC = (*create_class)();

                    // Give the two strings approprate values
                    ptrBC->set_derived_string("This is the Derived
                                               Class' String");
                    ptrBC->set_base_string("This is the Base Class'
                                           String");

                    // Ok output everything
                    ptrBC->print_base_string();
                    cout << "\nThe base string has "
                         << ptrBC->get_base_sting_length()
                         << " characters."
                         << endl;
                    ptrBC->print_derived_string();
                    cout << "\nThe derived string has "
                         << ptrBC->get_derived_sting_length()
                         << " characters."
                         << endl;

                    // Destroy the instance of the derived class
                    (*destroy_class)(ptrBC);
              }
        }
        return 0;
}
```

Please note that as long as we plan out the base class correctly, we are able to manipulate the derived class's data through accessors and mutators. Also, the derived class has no problems accessing data passed to it by the main application.

Well, there you have it. That is one way to make and use a DLL. If you followed all the above directions then the output you should get from the executable is:

Derived Class Constructor
This is the Base Class', String
The base string has 31 characters.
This is the Derived Class' String
The derived string has 34 characters.
Derived Class Destructor

2.4 Final decisions

After all the research and test applications were written, the following decisions were made about the different components for the Escape Simulation Suite:

2.4.1 Client

The client application was chosen to be Win32 based for the main graphical user interface. This option was chosen because of the ease with which very good and intuitive user interfaces can be created with Win32. Also, the amount of overhead involved with using Win32 is relatively low when compared with other solutions.

The rendering engine chosen was pure Direct3D. The only other viable option for this choice was OpenGL, but it was desirable to keep as many components in the same API as possible, and since the input and network were also going to be DirectX, this choice was pretty straightforward.

For the input, as the above paragraph hints, DirectInput was decided upon for the 3D world interface. Otherwise standard Win32 input handling is used for the GUI input, and DirectInput for the input to the 3D world.

DirectPlay was chosen for the network module. This was the hardest of the components to decide upon. Creating a Windows socket library from scratch was tempting because much of the extra overhead that comes with using another API would be avoided, but at the same time, because of how standard DirectPlay is, and the fact that the architecture already conformed so well with the desired messaging architecture, DirectPlay was the end winner. (For the time being that is.)

2.4.2 Server

The server is a standard console application. It could have been made Win32, but for its purposes that would have been overkill, not to mention keeping it as a console application makes it that much easier to move it to a Linux/Unix platform later.

Since the IO is minimal, it will all be kept just the standard console IO.

Again, like the client, the network interface is handled by DirectPlay.

CHAPTER 3: Issues

After the final decisions were made at the end of chapter 2, the next question was: what issues are created by these choices and how are these to be resolved? This chapter will attempt to address many of these issues and describe their resolution.

3.1 State machine

The first major issue that had to be resolved was the overall state machine. This was made more complex than usual because of the hybrid nature of the application make up. The state machine had to be able to manipulate and call all the internal components of the application. It needed to be able to be easily maintainable, and finally, it needed to be efficient.

Because of our use of DirectX and Win32 there are certain tasks that need to be done in a timely manner. It is not acceptable for the state machine to take complete control of what is done and when. It needs to allow for these rendering and input tasks to have time to do whatever they need to do.

After all the above requirements, the following state machine design was chosen. Each state will have a state handler method that is a private method to the main application class. Within a given handler the state will have its own states. At a minimum the state must have a beginning and ending state that takes care of initialization and cleanup. No state is allowed to block for any reason. For example, if a state must wait on some kind of outside event, acknowledgment from first the client/server, and then the handler must return execution to the rest of the client/server.

The current state, next state, and so on are kept track of by a global state stack. If one state needs to transition to another state to get something done then it pushes the new state handler's identifier onto the stack. When a state completes its work it pops its identifier off the top of the stack. The idle state is always the bottom state on the stack and is never removed.

The overall application uses a modified version of the DirectX application SDK that was introduced in the second chapter. Remember that in this architecture, after the application has been setup, it enters the run() function that will loop until it is told to quit the application. Then it gets a chance to clean up any memory, save data, etc. The current state machine handler is called each time through the run() loop after all messages from the network and Win32 were handled and the current frame has been rendered. Because no handler is allowed to block, this solution allows for the critical graphics and IO functions to be called in a timely manner and avoids the potential of deadlock.

This solution also is very easy to maintain. If a new state is needed, then the developer writes a handler method to do whatever the state requires, adds a new identifier to the handler ID table, and finally adds the handler's function call to the state case switch statement in the application's run() method. Because the handler is a private method in

the main application class it has the rights to use, look at, and modify everything in the application.

This solution to the state machine meets all the requirements for this particular application.

3.2 Inter-module communication

Inter-module communication, unfortunately, is an issue that needs to be addressed. In an ideal application all modules would be completely independent, and for the most part the different modules in this application have been designed to be as independent as possible. There are three points where there is a significant amount of inter-module communication. The Win32 WinProc() and other dialog callback functions, the network receive callback function, and the state machine. To some degree it may have been possible to completely get rid of the dependencies here, but this would have meant adding a lot of extra accessors and mutators and, in general, extra overhead.

There are several obvious concerns with having this type of communication. Since the network callback function is multithreaded, data security must be guaranteed. This is done through standard Windows semaphores. Obviously it was desirable to avoid any deadly embraces. These can be avoided with good programming. The last problem is how to actually make the data accessible to these components.

A commonly used solution for this last problem is the use of a singleton class, which only allows for a single instance of this class to exist and allows for global access to the class. So in order to give the callback functions access to the main application class then we make the main application class as a singleton and then we can access it directly. As long as these functions are declared as friends ,they will have access to all public and protected data in the singleton class.

Both the client and server applications have their main classes declared as singleton. Within the callback functions semaphores are used to avoid data corruption problems and, using careful programming, deadlock can be avoided.

3.3 Win32 vs. DirectX

There are several issues that are raised when combining DirectX with Win32. When it is run in windowed mode DirectX will, by default, take over then entire client window. This means any menus, toolbars, etc, will be completely covered. Also, any GDI graphics drawn to the client area will be covered. Input can also be a problem. If both DirectX and Win32 are gathering input from the same source then some unexpected situations can arise.

To solve the rendering problem you need to modify the rectangle that you pass to the 3D engine. You are required to calculate the amount to reduce it by so that all the Win32 resources can be viewed. At worst, this will require some trial and error to get the values right.

The input problem can be solved by allowing DirectX to normally have all the input from the game pad at all times, to have keyboard input unless a menu or dialog is selected, and to have mouse input as long as the mouse is in the render window or, like the keyboard, there is no menu or dialog open. This works because Win32 never gets input from the game pad. In this application, the mouse input is never directly used by Win32. This means we never look at mouse clicks or position messages, we only look for messages from menu selections or button clicks. Menus, tool bar buttons, and dialog buttons can only be selected or clicked if the mouse is not within the rendering window or if a dialog is open. This logic also works for the keyboard because the keys that are used by both the rendering engine and the Win32 interface are only used in the interface when a menu or dialog is open.

3.4 Console vs. DirectX

This section was already partially covered above. The main issue with using DirectX with console applications is that many of the DirectX components require a handle to an application window or instance. WinMain() has the instance value as its first argument, but a normal console application's main() functions does not. In order to take care of this problem we need to either start a Win32 application and immediately attach a console, redirecting all output to the console, or we need to be able to get the instance and window handles from the operating system.

The second of these two options is the more desirable, mainly because it avoids as much excess windows code as possible.

Getting the window handle of a console was covered in more depth above, but in a nutshell, getting the instance handle boils down to these three function-calls:

```
GetConsoleTitle()
SetConsoleTitle()
FindWindow()
```

The actual code for using these calls was covered in 2.3.3.1 please refer to this section for the example.

3.5 Client Server

The main issues with the server are network communication with the client, and making the world engine capable of using DLL pluggins.

With the server it is important that we guarantee the packets being transmitted. If any packets are lost it could mean very bad things for the application. Also, it is important to get a very high throughput on the number of packets we are able to send. Obviously we cannot have both, so we will try and find a happy medium.

As far as reliability is concerned we will be using TCP in order to guarantee that everything makes it to the destination safe and sound. As far as throughput we first try to make the packets as small as possible. Second, we try to reduce the number of packets that need to be sent by grouping them together when possible.

Solving the DLL issue was a major thing. This application would not be nearly as useful if the researchers were not able to modify whatever they needed to get their work done. That being said, it is not an easy thing to get the pluggins to work correctly. Because we wanted to allow the researcher to create objects that know about and have the ability to see and work with all other objects, we needed to make each of the objects capable of accessing all the known object data. In order to do this, almost all the .h and .cpp files must be linked into each pluggin. This requirement is a very annoying one that is difficult, if not impossible, to get around. Furthermore,there is also the issue of actually making the application code general enough to allow for taking in multiple DLLs of many different types and requirements. For the most part, this problem has been resolved with careful programming, virtual classes, and polymorphism.

3.6 In general

As noted above, one of the general issues that has had to be resolved is how to make the program as general and flexible as possible, while still abstracting as many of the little nitty-gritty details from the developer. As a result, they do not have to be experts in graphics programming, DLLs, and the general internals of the Escape Simulation Suite to be able to write working pluggins. That being the case, we are restricted from using DLLs for object pluggins except to add server-side functionality.

Because of this restriction, special care has had to be made to make the client very graphically verbose and flexible. The simulation file has had to be designed to be able to be very descriptive so that the client can know exactly what to expect and how to respond.

CHAPTER 4: Escape Simulator

4.1 Application Classes

The client and server are designed in a similar hierarchical structure. The main tier is the class ESCAPE_SIMULATION_CLIENT or ESCAPE_SIMULATION_SERVER respectively. The client class inherits from the graphics and network engines. It is created and called by the Win32 code. Server side, the class is called by main() and it inherits from the network engine and the world engine. Both engines have access to and can use the math engine.

In the next sections each of the components that make up the Escape Simulator will be covered. Though there is text, code, diagrams, and screen shots included here, the code in its entirety is included in the appendices along with complete UML diagrams. Also included with this document is a zipped version of the Visual Studio.net 2003 solution.

4.1.1 Setting up your computer

Before you jump into the source code , you need to do a few things to get your computer ready to compile and/or run the code. First and foremost, the assumption was made that most of the readers will be running windows XP. If that is not the case, you may or may not be able to get this working on your computer. You will need to install the latest version of the DirectX SDK on your computer. This can be downloaded from the Microsoft's homepage. If all you want to do is run the executables on your computer, then you only need to download and install the runtime libraries. If you want to compile the code or write your own pluggins then you will need the entire 250+MByte SDK.

If you are going to create your own project from scratch there are many library files that you will have to link into your project. You need to add the following libraries to your projects:

For the client:
comctl32.lib ws2_32.lib d3dx9dt.lib d3d9.lib d3dxof.lib dinput.lib dinput8.lib dplayx.lib winmm.lib dxguid.lib odbc32.lib odbccp32.lib

For the server:
dplayx.lib dxguid.lib comctl32.lib winmm.lib odbc32.lib odbccp32.lib

If you just want to follow along in the solution that is provided, you do not need to worry about these libraries. They are already set to be linked properly.

In its current state the Escape Simulation spans over 77+ files and over 16,000 lines of code. It would be impossible to cover every line. If nothing more was written, cutting and pasting the code into this thesis would make it over 300 pages long. Also, every block of code will not be covered. The code will be covered on a component-by-component basis showing its structure, and pointing out its critical or complicated elements. It is up to the reader to look through the files that are being covered and grab

the usual components of code, i.e. include blocks, #defines, forward declarations, etc. All this code should be self-explanatory. If you are not familiar with any of the above terms, please familiarize yourself with a reference on C & C++ programming, preferably one that covers object oriented programming style and techniques. It is recommended that the reader first look through [3].

Finally, because of the shear volume of information to be covered, this chapter will be fairly fast paced.

4.1.2 Escape Simulation Client Class

This is one of the largest of the classes that are used by the Escape Simulation Client. This class inherits from the graphics and network classes. It contains the state machine and all high level application code, such as application initialization, setup, run, and shutdown. Each of these function and code segments will be covered in the sections that follow. This section will only cover this class. Discussion on the graphics and network will be addressed in later sections.

4.1.2.1 Escape Simulation Client Class header

The first line of importance is the class prototype:

```
class ESCAPE_SIMULATION_CLIENT : public
Singleton<ESCAPE_SIMULATION_CLIENT>, public D3D_GRAPHICS, public EVC
```

As stated in the Inter-module communication section of Chapter 3, one of the issues is being able to communicate information throughout the application. For example, the network message handler needs to be able to make changes to the graphics data based on the messages it is receiving. It is also important that we only have one instance of this ESCAPE_SIMULATION_CLIENT.

In order to take care of both of these problems we also have this class inherit from the Singleton template class:

```
// Macros
#define     SINGLETON(sing_name)    sing_name::GetSingleton()

namespace Es
{
      // SingletonException class
      class SingletonException : public Exception
      {
      public:
            // Constructors/Desturctor
            SingletonException(std::string msg) : Exception(msg) { }
            virtual ~SingletonException() { }
      };
}

// Singleton class
template <typename T>
class Singleton
```

```
{
public:
	// Destructor
	virtual ~Singleton()
	{
		if(!IsInitialized()) throw
		Es::SingletonException("Encoutered an uninitialized
		Singleton instance upong destruction.");
		Singleton<T>::s_ptSingleton = NULL;
	}


	// Reporters
	static bool IsInitialized()
	{
		return NULL!=Singleton<T>::s_ptSingleton;
	}

	static T& GetSingleton()
	{
		if(!IsInitialized()) throw
		Es::NullReferenceException("Singleton referenced without
		being initialized.");
		return *Singleton<T>::s_ptSingleton;
	}


protected:
	// Constructor
	Singleton()
	{
		if(IsInitialized()) throw Es::SingletonException("Singleton
		instance already exists.");
		// Determine the derived object's starting memory location
		int offset = (int)(T*)1 - (int)(Singleton<T>*)(T*)1;
		Singleton<T>::s_ptSingleton = (T*)((int)this + offset);
	}


private:
	// Variables
	static T*		s_ptSingleton; // Points to the only instance
					               //of this class
};


// Static members
template <typename T> T* Singleton<T>::s_ptSingleton = NULL;
```

When looking at this class you notice the static s_ptSingleton member. This variable ensures that there can only be one instance of this class at any given time. In the constructor it checks to see if this pointer already has a value. If it does then it throws an exception that should be caught by the code that tried to create the new instance of the ESCAPE_SIMULATION_CLIENT class.

The next thing that to note is that this class inherits from the `D3D_GRAPHICS and EVC classes.` `D3D_GRAPHICS` is the Direct3D engine. EVC stands for the Evacuation Client network module. The network engine is actually made up of two modules: client and server. Each has its own message handler and sets up the network properly according to their respective roles. The server counterpart to the EVC is called the EVS.

The next chunk of code that should be looked at is the public functions:

```
ESCAPE_SIMULATION_CLIENT(HINSTANCE _hInstance);
virtual ~ESCAPE_SIMULATION_CLIENT();

virtual INT Run();

HRESULT MainInit();
HRESULT MainCreate();
HRESULT MainRun();
HRESULT MainStop();
HRESULT MainSaveData();
HRESULT MainDestroy();
```

There is nothing special being done with the constructor and destructor. However, the other functions are what are used to get the application up and running. MainInit() is responsible for initializing anything that needs to be initialized. MainCreate() creates instances of all variables that require special creation. MainRun() is a wrapper that calls the Run() method. More about run() will be covered below. MainStop() is called if the application is halted for any reason. MainSaveData() is responsible for saving important data that is not saved within a simulation file by the state machine. Finally, MainDestroy() is responsible for destroying all resources and general cleanup of data before the destructor is called. Note: Several of these functions will probably be removed or combined when the beta version of this code is ready.

The Run() method is unique in that it is an overloaded function. It replaces the base Run() method found in the `CD3DApplication` which was mentioned in the second chapter. More will be mentioned about this class and particular function later. Right now is it enough to say that this class is responsible for the low level Direct3D render engine details. All of the critical methods from this class are declared virtual so they can be overloaded to fit the particular applications specifications. In our case we are overloading the Run() method to add the state machine to it. Run() is called every render loop and does the following tasks: checks for windows messages and, if there are any, handles them; advances the graphics as needed based on user input; renders the graphics; and finally, enters the correct state machine state.

Next in the class prototype is a group of variables for setting up and maintaining the Win32 windows, as well as some semaphores for avoiding data corruption. These last variables will briefly be covered in the network message handler.

After the above variable declarations, we have the beginning of the state machine code. The state machine has been set up so that each state has a handler and variables that are

unique to it. The handlers are called in the Run() method. Here are the prototypes for these handlers and the variables associated with each:

```
// STATE MACHINE VARIABLES AND METHODS
      Stack *states;

      HRESULT STATE_SetupSimulation();
      bool simulationWiped;
      bool serverWorldLoaded;
      bool fieldDllLoaded;


      HRESULT STATE_MicroToMacro();
      HRESULT STATE_MacroToMicro();
      HRESULT STATE_NetworkConnect();
      bool id_recieved;

      HRESULT STATE_LoadResorce();
      HRESULT STATE_ClearSimulation();
      HRESULT STATE_LoadSimulation();
      HRESULT STATE_SaveSimulation();
      bool FILE_LIST_RECEIVED;      // Notifies when the file list is
                                    // received
      void *fileList;               // Will contain the file list.
      std::string simFileName;
      std::string mapFileName;
      unsigned int simFileID;       // Current simulation file ID
      void              *object_creation_buffer;
      unsigned int       object_creation_bytes;
      unsigned int       object_creation_typeID;

      // This state handles file requests from the server
      HRESULT STATE_ReceiveFileFromServer();
      char DesiredFile[2048];       // File to get from the server
      char DestinationFile[2048];   // Destination location and
                                    // filename for the requested file
      bool FILE_RECEIVED;           // Tells the state machine when the
                                    // file has been recieved and is
                                    // ready to be handled
      void *received_data_msg;      // The msg received from the server
                                    // is stored here.  REMEMBER TO
                                    // CAST this to the proper struct.

      HRESULT STATE_Exiting();
      HRESULT STATE_Idle();
```

Each state has a unique identifier that is used in Run() to know which state to go into. These identifiers are defined in ESCAPE_SIMULATION_CLIENT_STATE_DEFS.h. Along with the identifiers, each state also has a set of internal states that are uniquely defined in this same file. Please refer back to this portion when reading the section about the handlers:

```
enum APPLICATION_STATES
{
```

```
      STATE_UNSIGNED,                           // State Machine has
not been initilized yet
      STATE_SETUPSIMULAITON,              // Read simulation file and
setup application accordingly
      STATE_CLEARSIMULATION,              // Clear simulation contents,
start from scratch
      STATE_LOADSIMULATION,               // Read simulation file and
setup application accordingly
      STATE_SAVESIMULATION,               // Write simulation file
based on the current settings
      STATE_MICROTOMACRO,                       // Switching between
microscopic to macroscopic models
      STATE_MACROTOMICRO,                       // Switching between
macroscopic to microscopic models
      STATE_NETWORKCONNECTION,            // Establish Network
Connection
      STATE_LOADRESORCE,                        // Load new resorce
      STATE_CREATETEMPLATE,               // Creates an object type on
the server
      STATE_CREATEOBJECT,                       // Creates an object on
the server
      STATE_RECEIVEFILEFROMSERVER,  // Receive a file from the server
and save it accordingly
      STATE_EXITING,                            // Preform whatever is
needed to exit the application cleanly
      STATE_IDLE                                // Idol State
};

enum SETUPSIMULATION_STATES
{
      STATE_SETUPSIMULATION_ENTER,                        //
Beginning state
      STATE_SETUPSIMULATION_WAITFORSERVERWIPE,        // Wait for
server response
      STATE_SETUPSIMULATION_OPENSIMFILE,                  // Open the
Simulation file
      STATE_SETUPSIMULATION_END                                 //
Setup any final stuff needed
};

enum CLEARSIMULATION_STATES
{
      CLEARSIMULATION_STATES_ENTER,                       //
Beginning state
      CLEARSIMULATION_STATES_CLEARSIMPARAMETERS,          // Clear
general simulation information
      CLEARSIMULATION_STATES_CLEARPUGLINS,                // Clear
the simulation plugins (dlls)
      CLEARSIMULATION_STATES_CLEARTEMPLATES,              // Clear
the object resources (meshes, textures)
      CLEARSIMULATION_STATES_CLEARMAP,                    // Clear
the map information (world data)
      CLEARSIMULATION_STATES_CLEAROBJECTS,                // Clear
the objects in the simulation
      CLEARSIMULATION_STATES_END                                //
Setup any final stuff needed
};
```

```
enum LOADSIMULATION_STATES
{
	STATE_LOADSIMULATION_ENTER,                                    //
Beginning state
	STATE_LOADSIMULATION_OPENSIMFILE,                      // Open the
Simulation file
	STATE_LOADSIMULATION_LOADSIMPARAMETERS,                // Load the
setup information form the Simulation file
	STATE_LOADSIMULATION_LOADPLUGINS,                      // Load the
simulation plugins (dlls)
	STATE_LOADSIMULATION_LOADTEMPLATES,                    // Load the
object resources (meshes, textures)
	STATE_LOADSIMULATION_LOADMAP,                          // Load the
map information (world data)
	STATE_LOADSIMULATION_WAITFORWORLDLOAD,                 // Waiting
for server reply
	STATE_LOADSIMULATION_LOADSTATICOBJECTS,                // Load the
static objects based off the simulation file data
	STATE_LOADSIMULATION_LOADDYNAMICOBJECTS,         // Load all
dynamic objects based off the simulation file data
	STATE_LOADSIMULATION_END                                       //
Setup any final stuff needed
};

enum SAVESIMULATION_STATES
{
	STATE_SAVESIMULATION_ENTER,                                    //
Beginning state
	STATE_SAVESIMULATION_CREATESIMFILE,                    // Open the
Simulation file
	STATE_SAVESIMULATION_SAVESIMPARAMETERS,                // Write
the setup information form the Simulation file
	STATE_SAVESIMULATION_SAVEPLUGINS,                      // Write
the simulation plugins (dlls)
	STATE_SAVESIMULATION_SAVETEMPLATES,                    // Write
the object resources (meshes, tetures)
	STATE_SAVESIMULATION_SAVEMAP,                          // Write
the map information (world data)
	STATE_SAVESIMULATION_SAVESTATICOBJECTS,                // Write
the static objects based off the current simulation data
	STATE_SAVESIMULATION_SAVEDYNAMICOBJECTS,         // Write all
dynamic objects based off the current simulation data
	STATE_SAVESIMULATION_END                                       //
Setup any final stuff needed
};

enum MICROTOMACRO_STATES
{
	STATE_MICROTOMACRO_ENTER,                                      //
Beginning state In this state send the stop simulation command
	STATE_MICROTOMACRO_WAITFORSERVERCONFERMATION,    // Before changes
can be made to the dynamic data simulation must have been stoped both
server and client side
	STATE_MICROTOMACRO_CHANGEDATA,                                 //
Changes the data from micro to macro
```

```
	STATE_MICROTOMACRO_WAITFORCHANGECONFERMATION,	// Waiting for
server confirmation that change has been made
	STATE_MICROTOMACRO_RESTARTSIMULATION,		// If the
simulation was halted then restart simulation
	STATE_MICROTOMACRO_END				//
Final State
};

enum MACROTOMICRO_STATES
{
	STATE_MACROTOMICRO_ENTER,			//
Beginning state In this state send the stop simulation command
	STATE_MACROTOMICRO_WAITFORSERVERCONFERMATION,	// Before changes
can be made to the dynamic data simulation must have been stoped both
server and client side
	STATE_MACROTOMICRO_CHANGEDATA,			//
Changes the data from macro to micro
	STATE_MACROTOMICRO_WAITFORCHANGECONFERMATION,	// Waiting for
server confirmation that change has been made
	STATE_MACROTOMICRO_RESTARTSIMULATION,		// If the
simulation was halted then restart simulation
	STATE_MACROTOMICRO_END				//
Final State
};

enum NETWORKCONNECTION_STATES
{
	STATE_NETWORKCONNECTION_ENTER,			//
Beginning state
	STATE_NETWORKCONNECTION_WAITINGFORID,		// Waiting
for the server to id
	STATE_NETWORKCONNECTION_END			//
Final State
};

enum LOADRESORCE_STATES
{
	STATE_LOADRESORCE_ENTER,			//
Beginning state
	STATE_LOADRESORCE_END				//
Final State
};

enum RECEIVEFILEFROMSERVER_STATES
{
	STATE_RECEIVEFILEFROMSERVER_ENTER,		//
Beginning state
	STATE_RECEIVEFILEFROMSERVER_WAITFORFILE,	// Wait for the
client to recieve the file from the server
	STATE_RECEIVEFILEFROMSERVER_END			//
Final State
};
```

The actual state that the state machine is in is kept track of by the state's variable. This variable is of type Stack class. The class supports the usual functionality that you would

expect from a stack, such as pushing, popping, peaking, etc. The Stack class is defined in the stack.h and .cpp files. The prototype looks like this:

```
class Stack
{
public:
	Stack(unsigned long val = 0);
	~Stack();

	void clear();
	unsigned long peak();
	void push(unsigned long val);
	unsigned long pop();
private:
	unsigned long *stack;
	unsigned long current_size;
	unsigned long current_position;
	unsigned long starting_val;
};
```

If there is any question about what these methods do, please look at the documentation included in the stack files. This stack is designed to be generic in that it could be used for more than the state machine.

The current state of the state machine is always the top state. If another state needs to be run then that state's identifier is pushed onto the stack. Once a state successfully finishes, it pops itself off the top of the stack. The idle state is always the bottom of the stack and never is popped from the stack. The stack class ensures this last functionality. It sets the initial value that is passed to it at instantiation to be its bottom-most value, and if the last value is ever popped off by accident then it restores this value to the stack.

More will be said about the state machine later.

The final variables in this class are used to gather network information for debugging purposes and will eventually be removed.

The last method in the class is:

```
	HRESULT MakeWindow();
```

This method creates the actual Win32 windows for the application. This code was pulled out of its standard place in WinMain() to allow for a more modular division of the code.

4.1.2.2 Escape Simulation Client Class .cpp

For the most part this section will show the different definitions of the methods talked about in the last section. Anything that requires more information will also be covered here.

```
ESCAPE_SIMULATION_CLIENT::ESCAPE_SIMULATION_CLIENT(HINSTANCE
_hInstance) : Singleton<ESCAPE_SIMULATION_CLIENT>(),
D3D_GRAPHICS(_hInstance)
{
	// Set all structures/classes/variables to their default values
	this->hInstance = _hInstance;
	memset(szAppName,'\0',256);
	memset(szTextOutName,'\0',256);
	x = y = CW_USEDEFAULT;
	wheight = wwidth = 800;
	memset(this->global_status_message,'\0',1024);

	this->network_rate		= 0.0f;
	this->packet_count		= 0.0f;
	this->startTime			= 0;
	this->dt			= 0;
	this->dynamic_switch_done	= false;
	this->simulating		= false;
	this->id_recieved		= false;
	this->hDynamic_data_mutex	= CreateMutex(NULL,false,NULL);
	this->states			= new Stack(STATE_IDLE);
	this->states->push(STATE_LOADSIMULATION);
	this->states->push(STATE_SETUPSIMULAITON);
	this->states->push(STATE_UNSIGNED);
	this->fileList				= NULL;
		// Will contain the file list.
	this->received_data_msg		= NULL;
	    // The msg received from the server is stored here.
REMEMBER TO CAST this to the proper struct.
}
```

Nothing too special here. Notice the instantiation and priming of the state machine.

```
	this->states			= new Stack(STATE_IDLE);
	this->states->push(STATE_LOADSIMULATION);
	this->states->push(STATE_SETUPSIMULAITON);
	this->states->push(STATE_UNSIGNED);
```

This ensures that after the application has finished initializing, the state machine will start in its setup state and then will setup and load the previous simulation if it exists.

```
ESCAPE_SIMULATION_CLIENT::~ESCAPE_SIMULATION_CLIENT()
{
	CloseHandle(hDynamic_data_mutex);
	// Final cleanup of all remaining objects
}
```

We need to close down all semaphores here.

```
HRESULT ESCAPE_SIMULATION_CLIENT::MainInit()
{
	// Load all param files.  Setup all structures/classes/variables
accordingly
	strcpy(szAppName,"Escape Simulation Client");
	strcpy(szTextOutName,"Text Out Window");
```

```
	simFileName = "default.sim";
	mapFileName = "world.vmap";
	this->simFileID   = 0;
	hr = this->MakeWindow();
	InitCommonControls();
	return hr;
}
```

Here we are initializing the Win32 stuff, calling MakeWindow() to actually make the windows and initializing common controls for the Win32 resources.

```
HRESULT ESCAPE_SIMULATION_CLIENT::MainCreate()
{
	hr = this->Create(hInstance);
	return hr;
}
```

The call to Create here calls the `CD3DApplication`'s Create method to have it create and initialize the 3D engine.

```
HRESULT ESCAPE_SIMULATION_CLIENT::MainRun()
{
	// All initilization is finished and the application is ready.
	// Start the application in full swing
	hr = this->Run();
	return hr;
}
```

At the moment just call the Run() function.

```
HRESULT ESCAPE_SIMULATION_CLIENT::MainStop()
{
	// Stop the application in preporation for closing everything up
	return S_OK;
}

HRESULT ESCAPE_SIMULATION_CLIENT::MainSaveData()
{
	// Save all important data and settings
	return S_OK;
}

HRESULT ESCAPE_SIMULATION_CLIENT::MainDestroy()
{
	// This might be needed.  We will see.
	return S_OK;
}
```

The last three functions are blank at the moment. They are mainly here for potential future expansion.

```
HRESULT ESCAPE_SIMULATION_CLIENT::MakeWindow()
{
```

```
	wndclass.style			= CS_HREDRAW | CS_VREDRAW;
	wndclass.lpfnWndProc	=
(WNDPROC)ESCAPE_SIMULATION_CLIET_WndProc;
	wndclass.cbClsExtra		= 0;
	wndclass.cbWndExtra		= 0;
	wndclass.hInstance		= hInstance;
	wndclass.hIcon			= LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_ICON));
	wndclass.hCursor		= LoadCursor(NULL, IDC_ARROW);
	wndclass.hbrBackground	= (HBRUSH)GetStockObject(WHITE_BRUSH);
	wndclass.lpszMenuName	= NULL;
	wndclass.lpszClassName	= szAppName;

	if(!RegisterClass(&wndclass))
	{
		MessageBox(NULL,TEXT("ERROR: Could not register class."),
szAppName, MB_ICONERROR);
		return 0;
	}

	hmenu = LoadMenu(hInstance,MAKEINTRESOURCE(IDR_MENU));

	m_hWndParent = CreateWindow(szAppName, TEXT("Escape Simulation
Client"),
				WS_OVERLAPPEDWINDOW,
				CW_USEDEFAULT, CW_USEDEFAULT,
				600, 600,
				NULL, hmenu, hInstance, NULL);

	//Graphics Window Creation & Registration
	wndclassTextOut.style		= CS_HREDRAW | CS_VREDRAW;
	wndclassTextOut.lpfnWndProc	=
(WNDPROC)ESCAPE_SIMULATION_CLIENT_TextOutProc;
	wndclassTextOut.cbClsExtra	= 0;
	wndclassTextOut.cbWndExtra	= 0;
	wndclassTextOut.hInstance	= hInstance;
	wndclassTextOut.hIcon		= LoadIcon(NULL,
IDI_APPLICATION);
	wndclassTextOut.hCursor		= LoadCursor(NULL, IDC_ARROW);
	wndclassTextOut.hbrBackground	= (HBRUSH)
GetStockObject(GRAY_BRUSH);
	wndclassTextOut.lpszMenuName	= NULL;
	wndclassTextOut.lpszClassName	= szTextOutName;

	if(!RegisterClass (&wndclassTextOut))
	{
		char errorMsg[256];
		memset(errorMsg,'\0',256);
		strcpy(errorMsg,"Error creating class: #");
		itoa((int)GetLastError(),&errorMsg[strlen(errorMsg)],10);
		MessageBox(NULL, TEXT (errorMsg),szTextOutName,
MB_ICONERROR);
		return 0;
	}

	// Generate Text Out Window
```

```
      hwndTextOut = CreateWindow(szTextOutName, TEXT ("TextOutput
Window"),
            WS_CHILD | WS_VISIBLE,
            CW_USEDEFAULT, CW_USEDEFAULT,
            100, CW_USEDEFAULT,
            NULL, NULL, hInstance, NULL);


      TBBUTTON tbrButtons[6];

      tbrButtons[0].iBitmap   = 0;
      tbrButtons[0].idCommand = ID_TBPLAY;
      tbrButtons[0].fsState   = TBSTATE_ENABLED;
      tbrButtons[0].fsStyle   = TBSTYLE_BUTTON;
      tbrButtons[0].dwData    = 0L;
      tbrButtons[0].iBitmap   = 0;
      tbrButtons[0].iString   = 0;

      tbrButtons[1].iBitmap   = 1;
      tbrButtons[1].idCommand = ID_TBPAUSE;
      tbrButtons[1].fsState   = TBSTATE_ENABLED;
      tbrButtons[1].fsStyle   = TBSTYLE_BUTTON;
      tbrButtons[1].dwData    = 0L;
      tbrButtons[1].iString   = 0;

      tbrButtons[2].iBitmap   = 2;
      tbrButtons[2].idCommand = ID_TBSTOP;
      tbrButtons[2].fsState   = TBSTATE_ENABLED;
      tbrButtons[2].fsStyle   = TBSTYLE_SEP;
      tbrButtons[2].dwData    = 0L;
      tbrButtons[2].iString   = 0;

      tbrButtons[3].iBitmap   = 3;
      tbrButtons[3].idCommand = ID_TBRW;
      tbrButtons[3].fsState   = TBSTATE_ENABLED;
      tbrButtons[3].fsStyle   = TBSTYLE_BUTTON;
      tbrButtons[3].dwData    = 0L;
      tbrButtons[3].iString   = 0;

      tbrButtons[4].iBitmap   = 4;
      tbrButtons[4].idCommand = ID_TBFF;
      tbrButtons[4].fsState   = TBSTATE_ENABLED;
      tbrButtons[4].fsStyle   = TBSTYLE_BUTTON;
      tbrButtons[4].dwData    = 0L;
      tbrButtons[4].iString   = 0;

      tbrButtons[5].iBitmap   = 5;
      tbrButtons[5].idCommand = ID_TBQUIT;
      tbrButtons[5].fsState   = TBSTATE_ENABLED;
      tbrButtons[5].fsStyle   = TBSTYLE_BUTTON;
      tbrButtons[5].dwData    = 0L;
      tbrButtons[5].iString   = 0;

      hWndToolbar = CreateToolbarEx(m_hWndParent, WS_VISIBLE | WS_CHILD
| WS_BORDER | TBSTYLE_ALTDRAG,
                                    IDR_SIMTB, 6, hInstance,
                                  IDR_SIMTB,tbrButtons, 6,
```

```
                                                    16, 16, 16, 16,
sizeof(TBBUTTON));
/*********************************************************************
************************************************************
      CHILD WINDOW TEST
*********************************************************************
************************************************************/
      RECT tempRect;
      GetClientRect( m_hWndParent, &tempRect );
      this->m_hWndParent = m_hWndParent;

      wndclass.style               = CS_HREDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc    = (WNDPROC)D3DWndProc;
      wndclass.cbClsExtra          = 0;
      wndclass.cbWndExtra          = 0;
      wndclass.hInstance           = hInstance;
      wndclass.hIcon               = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_ICON));
      wndclass.hCursor        = LoadCursor(NULL, IDC_ARROW);
      wndclass.hbrBackground  = (HBRUSH)GetStockObject(WHITE_BRUSH);
      wndclass.lpszMenuName   = NULL;
      wndclass.lpszClassName  = TEXT("D3DWINDOW");

      RegisterClass(&wndclass);
      m_hWnd = CreateWindow(TEXT("D3DWINDOW"),TEXT(""),
                                                WS_CHILD | WS_VISIBLE |
WS_BORDER | WS_CLIPSIBLINGS,

      0,16,tempRect.right,tempRect.bottom - 28,
                                                m_hWndParent,(HMENU)
0,hInstance, NULL);

      ShowWindow(m_hWnd,SW_SHOW);
      UpdateWindow(m_hWnd);

/*********************************************************************
************************************************************
      END CHILD WINDOW TEST
*********************************************************************
************************************************************/
      ShowWindow(m_hWndParent,SW_SHOW);
      UpdateWindow(m_hWndParent);

      ShowWindow(hwndTextOut,SW_SHOW);
      UpdateWindow(hwndTextOut);


      //while(GetMessage(&msg,NULL,0,0))
      //{
      //    TranslateMessage(&msg);
      //    DispatchMessage(&msg);
      //}

      //return (int)msg.wParam;
      return S_OK;
}
```

Despite it's size, this function should not be too difficult to understand. If you followed the explanation of resources in Chapter 2, you will notice that this function is actually creating several windows and resources that will be used in the rest of the application. This is all basic Win32 stuff and should not come as a surprise. There is one very important thing happening here aside from the creation of these windows/resources: When the main window is created:

```
m_hWndParent = CreateWindow(szAppName, TEXT("Escape Si…
```

its handle is assigned to m_hWndParent. This variable is part of the 3D engine and the engine checks this handle to see if the 3D engine already has a window to render to. If this handle is NULL, it creates the missing window. In this application we take care of this creation ourselves.

Finally, the last method definition in the file is the Run() method. This is one of the most important functions for this application. It is responsible for running the Win32 message handler, the graphics, and the state machine. Here is the code:

```
INT ESCAPE_SIMULATION_CLIENT::Run()
{
    // Load keyboard accelerators
    HACCEL hAccel = LoadAccelerators( NULL,
MAKEINTRESOURCE(IDR_MAIN_ACCEL) );

    // Now we're ready to recieve and process Windows messages.
    bool bGotMsg;
    MSG  msg;
    msg.message = WM_NULL;
    PeekMessage( &msg, NULL, 0U, 0U, PM_NOREMOVE );

    while( WM_QUIT != msg.message  )
    {
        // Use PeekMessage() if the app is active, so we can use idle
        // time to
        // render the scene. Else, use GetMessage() to avoid eating CPU
        // time.
     if( m_bActive )
       bGotMsg = ( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) != 0 );
     else
       bGotMsg = ( GetMessage( &msg, NULL, 0U, 0U ) != 0 );

     if( bGotMsg )
     {
       // Translate and dispatch the message
       if( hAccel == NULL || m_hWnd == NULL ||
            0 == TranslateAccelerator( m_hWnd, hAccel, &msg ) )
            {
                TranslateMessage( &msg );
                DispatchMessage( &msg );
            }
        }
        else
```

```
        {
            if( m_bDeviceLost )
            {
                // Yield some CPU time to other processes
                Sleep( 100 ); // 100 milliseconds
            }
            // Render a frame during idle time (no messages are
            // waiting)
            if( m_bActive )
            {
                if( FAILED( Render3DEnvironment() ) )
                    SendMessage( m_hWnd, WM_CLOSE, 0, 0 );
            }
        }
            // State Machine Code starts here.
            WaitForSingleObject(hDynamic_data_mutex,INFINITE);
            switch(states->peak())
            {
            case STATE_UNSIGNED:                        // State Machine
has not been initilized yet
                  this->states->pop();
                  break;
            case STATE_SETUPSIMULAITON:                 // Initialize a
simulation environment
                  STATE_SetupSimulation();
                  break;
            case STATE_CLEARSIMULATION:                 // Clear
simulation data (start from scratch)
                  STATE_ClearSimulation();
                  break;
            case STATE_LOADSIMULATION:                  // Read
simulation file and setup application accordingly
                  STATE_LoadSimulation();
                  break;
            case STATE_SAVESIMULATION:                  // Save
simulation data into a sim file
                  STATE_SaveSimulation();
                  break;
            case STATE_MICROTOMACRO:                    // Switching
between microscopic to macroscopic models
                  STATE_MicroToMacro();
                  break;
            case STATE_MACROTOMICRO:                    // Switching
between macroscopic to microscopic models
                  STATE_MacroToMicro();
                  break;
            case STATE_NETWORKCONNECTION:       // Establish Network
Connection
                  STATE_NetworkConnect();
                  break;
            case STATE_LOADRESORCE:                     // Load new
resorce
                  STATE_LoadResorce();
                  break;
            case STATE_RECEIVEFILEFROMSERVER:
                  STATE_ReceiveFileFromServer();
                  break;
```

```
            case STATE_EXITING:
                  STATE_Exiting();
                  break;
            case STATE_IDLE:                              // Idle State
            default:                                            // General
state
                  STATE_Idle();
                  break;
            }
            ReleaseMutex(hDynamic_data_mutex);
    }
    if( hAccel != NULL )
        DestroyAcceleratorTable( hAccel );

    return (INT)msg.wParam;
}
```

Here, after some preliminary initialization, the method enters a loop until it is told to exit. Inside the loop it first checks for new messages. If there is one it will make sure the window and accelerators exist and are valid. If not, it fixes to problem then runs the 3D graphics engine and passes the message to the Win32 message handler.

Finally, it will wait and acquire the semaphore so the state machine can modify the data if needed. Once the semaphore is acquired it will peak at the state stack and switch on the top value on the stack, entering the state handler assigned to the given ID value.

When the while loop is told to end by receiving a WM_QUIT message it will clean up the accelerator table and return.

4.1.2.3 Escape Simulation Client States .cpp

This file contains all of the definitions of the state handlers found in the ESCAPE_SIMULATION_CLIENT class. All of the handlers essentially follow the same format and progression. In order to explain how the handlers work, one will be covered here. Other handlers will be covered in different sections where appropriate.

```
HRESULT ESCAPE_SIMULATION_CLIENT::STATE_ReceiveFileFromServer()
{
      HRESULT hr = S_OK;
      static LONG state = STATE_RECEIVEFILEFROMSERVER_ENTER;
      FILE_REQUEST load_file_request;
      fstream outfile;
      FILEIO *file_data;

      switch (state)
      {
      case STATE_RECEIVEFILEFROMSERVER_ENTER:
            this->FILE_RECEIVED = FALSE;
            SAFE_DELETE_ARRAY(this->received_data_msg);
            load_file_request.type = WORLD_DATA_REQUEST;

      memset(load_file_request.buffer,'\0',sizeof(load_file_request.buf
fer));
```

```
            strcpy(load_file_request.buffer,this->DesiredFile);
            if(S_OK != sendmsg((void *
)&load_file_request,sizeof(FILE_REQUEST)))
            {
                  hr = S_FALSE;
                  break;
            }
            state = STATE_RECEIVEFILEFROMSERVER_WAITFORFILE;
            return hr;
      case STATE_RECEIVEFILEFROMSERVER_WAITFORFILE:
            if(FILE_RECEIVED) // Wait for the client to receive the
file from the server
            {
                  state = STATE_RECEIVEFILEFROMSERVER_END;
            }
            return hr;
      case STATE_RECEIVEFILEFROMSERVER_END:
            // Ok the file is here time to save it.
            file_data = (FILEIO *)this->received_data_msg;
            outfile.clear();
            outfile.open(this->DestinationFile,ios_base::binary |
ios_base::out | ios_base::trunc);
            if(outfile.is_open())
            {
                  outfile.write(&(file_data->data),file_data->length);
                  outfile.close();
            }
            sprintf(global_status_message,"SUCCESS: %s
loaded",file_data->filename);

      SendMessage(Status_bar,WM_SETTEXT,NULL,(LPARAM)global_status_mess
age);
            // MAKE SURE TO CLEAN UP THE BUFFERED DATA SENT OVER FROM
THE SERVER
            SAFE_DELETE_ARRAY(this->received_data_msg);
            break;
      default:
            // Spit our error message and return
            this->states->pop();
            state = STATE_RECEIVEFILEFROMSERVER_ENTER;
            return hr;
      }
      this->states->pop();
      state = STATE_RECEIVEFILEFROMSERVER_ENTER;
      return hr;
}
```

This state handler manages requesting and receiving files from the server. The beginning of any handler should set up whatever variables it will need. It is important to remember that if this function does not run to completion in one entrance, any data that is needed internally from state to state must be either declared static or saved outside the method. Good examples of these two options are the following two variables:

```
      static LONG state = STATE_RECEIVEFILEFROMSERVER_ENTER;
      bool FILE_RECEIVED;
```

The “state” variable holds the handlers current internal state. This variable is defined as a static variable at the beginning of the function and initialized to STATE_RECEIVEFILEFROMSERVER_ENTER. The second variable, “FILE_RECIEVED”, is a flag for one of the internal states to check and determine when the client received the file from the server. In the beginning state of the handler STATE_RECEIVEFILEFROMSERVER_ENTER, this variable is set to FALSE. Once the handler arrives in the STATE_RECEIVEFILEFROMSERVER_WAITFORFILE state it will stay there until the FILE_RECIEVED flag is true. This flag is set to true in the network message handler when it receives a WORLD_DATA_REPLY message. Once the STATE_RECEIVEFILEFROMSERVER_WAITFORFILE sees that the flag is set to true, it transitions to the next state that handles the actual saving of the data that was transmitted from the server. Once the file is saved it pops the stack and resets the internal state to STATE_RECEIVEFILEFROMSERVER_ENTER. As a result, the next time this handler is called it will start back at the beginning.

4.1.3 Graphics Engine – D3D_GRAPHICS Class

The 3D graphics engine for this application was built using Direct3D. The main class for the graphics engine is the D3D_GRAPHICS class. This class inherits from the CD3DApplication class. This base class was developed by the Direct3D developers as a low level framework that helps speed up the development of Direct3D applications. It does this by handling some of the more tedious details of creating and maintaining a Direct3D application.

The overall structure of this base class is simple. It was already discussed in Chapter 2. Essentially, there are several virtual functions that this API uses that can be overridden to tailor it to the specific needs of the application being developed. We have already seen two of these functions being called and one of the two being overridden namely Create() and Run().

This framework only provides the base needs for a Direct3D application. It is still up to the developer to handle all the work of setting up and managing data, cameras, lights, textures, meshes, vertices, etc, and rendering everything.

For information on how to do all of the above, please refer to the first eight chapters of [7], as well as [11], [14], [15], and [16]. After covering those chapters you should be able to handle all the Direct3D material in this application.

In order to render the world correctly, the graphics data in this application is organized in a hierarchical structure, the largest tier being the entire world. The D3D_GRAPHICS class acts as this container. The world has all the graphics properties you would expect a world to have. It has floors with other rendered objects, such as people. It has lighting, referring to the Direct3D lighting. It has the cameras and views, and it also contains all the extra information and settings needed to render the world correctly.

4.1.3.1 D3D_GRAPHICS .h:

```
class D3D_GRAPHICS : public CD3DApplication
{
public:
	D3D_GRAPHICS(HINSTANCE _hInstance);
	virtual ~D3D_GRAPHICS();

	LPDIRECT3DDEVICE9 GetD3DDevice() const;
	FLOOR* GetFloor(unsigned int floorID) const;
	unsigned long getGraphicsMode();

	CD3DFont		*m_pFont;
	// FONT
```

This font is used by D3D to print out text to the render window. Unfortunately, this text is very costly to display, so it is only turned on during debug mode.

```
	// DEBUGGING
	float			fov;
	float			aspect;
	float			zn;
	float			zf;
```

These variables are all for debugging. They will eventually be removed in the final version of the code.

```
	//  Projection Matricies durring rendering to orent objects
correctly
	D3DXMATRIX matRotY,matRotX,matRotZ,matTrans,matScale,trans;
    D3DXMATRIX    m_matProj;
      // Projection Matrix
    D3DXMATRIX    m_matView;
      // View Matrix
    D3DXMATRIX    m_matWorld;
      // View Matrix
```

All the above matrices are used by the camera and for high tier rendering. Each of the individual objects, for instance, has its own D3DXMATRIX structures that it maintains.

```
	double cameraX;
	double cameraY;
	double cameraZ;
	double cameraR;
	double cameraPHI;
	double cameraTHETA;
	double cameraLookX;
	double cameraLookY;
	double cameraLookZ;

	double cameraX3P;
	double cameraY3P;
	double cameraZ3P;
	double cameraR3P;
	double cameraPHI3P;
	double cameraTHETA3P;
	double cameraLookX3P;
```

```
        double cameraLookY3P;
        double cameraLookZ3P;
```

All the above doubles are used by the application to change between first and third person views, and to move the camera in the desired view. The camera class has recently been revamped and all these values will be removed.

```
        // IF MULTIPLE VIEWPORTS WILL BE USED THEN THEY WILL GO HERE
        D3DVIEWPORT9      MainViewport;
        // Different Viewports
```

This particular application will only have a single view port. In the future, that can be changed if more are needed.

```
        // CLASS POINTERS
        TEXTURES                *textures;  // Textures being used be
                                            // various different objects
        FLOORS                  *floors;    // Floor Structures
        LIGHT                   *light;     // Lights...
        CAMERA                  *camera;    // Camera...ACTION!!!
        D3D_GRAPHICS_INPUT      *input;     // Input Decives
```

Here are the classes that make up a given world.

```
protected:
        // WINDOWS VARIABLES
        HINSTANCE hInstance;
        HWND m_hWndParent;
        unsigned long graphics_mode;
        int currentLevel;
```

These are used by several Direct3D calls, so we need to keep track of this information.

```
        // FILEIO
        HRESULT ReadWorld(std::string fileName);
        HRESULT     ReadTemplateMesh(std::string meshName, std::string
fileName);
        HRESULT ReadTemplateSprite(std::string spriteName, std::string
fileName);
        HRESULT RoomSetup();
```

These calls make up the particular file IO that is needed by the 3D engine to fill in its data structures with the world data. When the state machine reads a simulation file it will call these functions as needed. More will be said below about the Templates.

```
        // MAIN FRAIMWORK
        HRESULT OneTimeSceneInit();
        HRESULT InitDeviceObjects();
        HRESULT RestoreDeviceObjects();
        HRESULT Render();
        HRESULT FrameMove();
        HRESULT FrameMove_Keyboard();
        HRESULT FrameMove_Mouse();
        HRESULT FrameMove_Gpad();
        HRESULT InvalidateDeviceObjects();
        HRESULT DeleteDeviceObjects();
```

```
        HRESULT FinalCleanup();
```

Here are the main calls for this engine. Notice that most of the CD3DApplication virtual functions are overloaded here. Most of these methods are very easy to understand, but they will still be covered below.

```
        HRESULT ConfirmDevice (D3DCAPS9 *pCaps, DWORD dwBehavior,
        D3DFORMAT adapterFormat, D3DFORMAT backBufferFormat);

        LRESULT MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
        lParam );
```

The first method here is a helper function. The second is the WinProc function that is used by the Win32 portion of the code.

```
private:
        // VMap - IMap (static world data) HELPERS
        int         ConvertVMapToIMap(std::string vMapName);
        HRESULT ReadVMapFloor(std::ifstream& vMapFile, std::ofstream&
iMapFile);
        int         GenerateWorld(std::string iMapName);
        HRESULT ReadIMapFloor(std::ifstream& iMapFile);

        // Object - IMap (dynamic world data) HELPERS
        int         ConvertXToIMap(std::string xFileName);
        int         GenerateTemplateMesh(std::string meshName,
std::string iMapName, std::string fileExtension);
```

These last functions are all helper functions that aid in basic data file translation.

```
};
```

4.1.3.2 D3D_GRAPHICS .cpp

```
D3D_GRAPHICS::D3D_GRAPHICS(HINSTANCE _hInstance)
{
        m_strWindowTitle  = _T("D3D_GRAPHICS TEST");
        m_d3dEnumeration.AppUsesDepthBuffer = TRUE;
        m_dwCreationWidth       = 1024;   // Width used to create window
        m_dwCreationHeight            = 800;
        //memset(m_bKey,0,256);
        this->hInstance = _hInstance;
        this->graphics_mode = 0;
        this->currentLevel      = 0;

        // These might need to be moved to the onetimesceneinit method
        m_pFont = new CD3DFont( _T("Arial"),12,D3DFONT_BOLD);
}
```

This is the D3D_GRAPHICS class' constructer. At this time we are taking care of only the minimum things needed. The rest of the initialization and setup will be taken care of below.

```
D3D_GRAPHICS::~D3D_GRAPHICS()
```

```
{
// All deleting and cleanup of objects is being done in other methods.
}
```

This method is very empty because everything is already being taken care of in other places.

```
LPDIRECT3DDEVICE9 D3D_GRAPHICS::GetD3DDevice() const
{
	return m_pd3dDevice;
}

unsigned long D3D_GRAPHICS::getGraphicsMode()
{
	return this->graphics_mode;
}

FLOOR* D3D_GRAPHICS::GetFloor(unsigned int floorID) const
{
	for(FLOOR* it = floors->bottom; it; it=it->up) if(floorID==it-
>id) return it;
	return NULL;
}
```

Simple accessors for various internal data.

```
HRESULT D3D_GRAPHICS::OneTimeSceneInit()
{
	return S_OK;
}
```

At the moment, this method is not being used. It is left here for future use.

```
HRESULT D3D_GRAPHICS::InitDeviceObjects()
{
	m_pFont->InitDeviceObjects(m_pd3dDevice);
	textures = new TEXTURES();
	camera = new CAMERA();
	light = new LIGHT();
	floors = new FLOORS(textures);
	input = new D3D_GRAPHICS_INPUT(this->hInstance, this-
>m_hWndParent);

	return S_OK;
}
```

This is where all the different devices are created and initialized.

```
HRESULT D3D_GRAPHICS::RestoreDeviceObjects()
{

	m_pFont->RestoreDeviceObjects();
	SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()-
>SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);

	// RESTOR ALL FLOORS
```

```
	floors->RestoreDeviceObjects();
	// Restore all mesh object templates
	MeshTemplate::RestoreAllDeviceObjects();

	// Set The projection Matrix
	aspect = m_d3dsdBackBuffer.Width /
(float)m_d3dsdBackBuffer.Height;
	fov    = D3DX_PI/8.0f;
	zn     = 1.0f;
	zf     = -4.0f;
	D3DXMatrixPerspectiveFovLH( &m_matProj, fov, aspect, zn, zf);
	SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()->SetTransform(
D3DTS_PROJECTION, &m_matProj);

	// Set up the view matrix.
	camera->SetTarget(0,0,0);
	camera->SetPosition(0,50,-50);


	// Set up all special Matricies that will remain static durring
all moves

	light->setPosition(0.0,100.0,0.0);
	light->setDirection(0.0,0.0,0.0);
	light->setAttenuation(0.0f,0.01f,0.0f);
	light->setDiffuse(3.0,3.0,3.0,1.0);
	light->setSpecular(1.0,1.0,1.0,1.0);
	light->setRange(200.0);
	light->setType(D3DLIGHT_POINT);
	light->setLight(1);


	return S_OK;
}
```

There are many ways that a device can get lost. One example is when the user minimizes the render window and then restores it. Before they can be rendered again all the devices need to be restored. This method restores the data related to this level of the rendering tier. Similar methods exist in each of the lower tiers.

```
HRESULT D3D_GRAPHICS::FrameMove()
{
	input->GETINPUT();

	if(input->info.KEYBOARD_VALID)
		FrameMove_Keyboard();

	if(input->info.MOUSE_VALID)
		FrameMove_Mouse();

	if(input->info.GAME_PAD_VALID)
		FrameMove_Gpad();


	// MOVE LIGHT
	D3DXMatrixTranslation(&matTrans,-3.0f,100.0f,0.0f);
```

```
	D3DXMatrixRotationY(&matRotY,(float)(m_fTime * 2.0));
	trans = matTrans * matRotY;
	light->setPosition(trans._41,trans._42,trans._43);

	return S_OK;
}
```

This method is called every time through the render loop. Its job is to first gather information from the input devices, and then make any changes to the data to be rendered based on the input or some other algorithm. For example, the GETINPUT call above updates the input data structures with the latest input data. FrameMove_Keyboard() makes changes to the camera and other things like the GRAPHICS_MODE flag based on the input from the keyboard. The // MOVE LIGHT block of code, however, does not care about any kind of input. Instead, it will rotate the global point light source that is hovering above the world based on the elapsed time.

```
HRESULT D3D_GRAPHICS::Render()
{
	char outtext[1024];		// Used for outputting text to
screen
	HRESULT result;			// Was used for debugging

	 SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()-
>Clear(0L,NULL,D3DCLEAR_TARGET |
D3DCLEAR_ZBUFFER,D3DCOLOR_XRGB(100,100,100),1.0f,0L);
	if(SUCCEEDED((result =
SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()->BeginScene())))
	{

		// DRAW ALL FLOORS

	WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).hDynamic_
data_mutex,INFINITE);
		if(this->graphics_mode & SINGLE_FLOOR_MODE)
		{
			floors->renderFloor(this->currentLevel);
		}
		else
		{
			floors->renderFloors();
		}

	ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).hDynamic_data_mu
tex);

		if(0) // RENDERING THE TEXT IS VERY COSTLY
		{
			// RENDER STATISTICS TEXT
			m_pFont->DrawText(2, 20,
D3DCOLOR_ARGB(255,255,255,0),m_strFrameStats);
			m_pFont->DrawText(2, 40,
D3DCOLOR_ARGB(255,255,255,0),m_strDeviceStats);

			memset(outtext,'\0',1024);
```

```
                  sprintf(outtext,"S1 = %x::%x  S2 = %x::%x",input-
>info.gpadState.lX,input->info.gpadState.lY,input-
>info.gpadState.lRz,input->info.gpadState.rglSlider[0]);
                  m_pFont->DrawText(2, 60,
D3DCOLOR_ARGB(255,255,255,0),outtext);

                  memset(outtext,'\0',1024);
                  sprintf(outtext,"m_fFPS = %f",this->m_fFPS);
                  m_pFont->DrawText(2, 80,
D3DCOLOR_ARGB(255,255,255,0),outtext);

                  memset(outtext,'\0',1024);
                  sprintf(outtext,"B 1-4 = %x::%x::%x::%x  B 5-8 =
%x::%x::%x::%x  B 9-12 = %x::%x::%x::%x",input-
>info.gpadState.rgbButtons[0],input-
>info.gpadState.rgbButtons[1],input-
>info.gpadState.rgbButtons[2],input->info.gpadState.rgbButtons[3],

                                          input-
>info.gpadState.rgbButtons[4],input-
>info.gpadState.rgbButtons[5],input-
>info.gpadState.rgbButtons[6],input->info.gpadState.rgbButtons[7],

                                          input-
>info.gpadState.rgbButtons[8],input-
>info.gpadState.rgbButtons[9],input-
>info.gpadState.rgbButtons[10],input->info.gpadState.rgbButtons[11]);
                  m_pFont->DrawText(2, 120,
D3DCOLOR_ARGB(255,255,255,0),outtext);

                  memset(outtext,'\0',1024);
                  sprintf(outtext,"Network Rate = %f
pps",SINGLETON(ESCAPE_SIMULATION_CLIENT).network_rate);
                  m_pFont->DrawText(2, 160,
D3DCOLOR_ARGB(255,255,255,0),outtext);
            }
            SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()-
>EndScene();
      }
      return S_OK;
}
```

Okay, this is one of the most important methods in this engine. Between Render() and FrameMove() all the rendering work is done each time through the render loop. All the above code should be fairly familiar if you have gone through the references given at the beginning of this section. The only thing that should be explained is this segment of code:

```
WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).hDynamic_data_m
utex,INFINITE);
            if(this->graphics_mode & SINGLE_FLOOR_MODE)
            {
                  floors->renderFloor(this->currentLevel);
            }
            else
            {
                  floors->renderFloors();
```

```
            }

      ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).hDynamic_data_mu
tex);
```

The WaitForSingleObject and ReleaseMutex are used to ensure that the data being rendered is not tampered with while it is being rendered. Finally, you see that depending on whether the graphics_mode variable had the SINGLE_FLOOR_MODE flag set or not, either the current floor only will be rendered or all the floors will be rendered. All the possible flags for the graphics mode and the other #defines and macros used by the graphics engine are defined in the D3D_GRAPHICS_DEFS.h file and are as follows:

```
#define SINGLE_FLOOR_MODE                       0x1
#define MICRO_DISPLAY_MODE_AVAILABLE            0x2
#define MICRO_DISPLAY_MODE_SET                  0x4
#define FIRST_PERSON_MODE                       0x8

#define MAX_WALL_HEIGHT                         2
#define MAX_R                                   1000
#define MIN_R                                   20
#define MAX_NUM_FLOORS                          256
#define MAX_NUM_OBJECTS                         256
#define MAX_NUM_OBJTEMPLATES                    32
#define DEFAULT_FLOOR_TEXTURE_ID                0
#define DEFAULT_WALL_TEXTURE_ID                 1
#define DEFAULT_WALL_HEIGHT                     2
#define FLOOR_SPACING                           10
#define     OBJECT_COLOR                        0x0000FF00  // Used in
D3D_GRAPHICS_FILEIO.cpp for the dynamic object vertex color
#define     VERTEX_TYPE                         FVFCN       // Used in
D3D_GRAPHICS_FILEIO.cpp for the dynamic object vertex creation


const DWORD FVF = (D3DFVF_XYZ | D3DFVF_TEX1);
const DWORD FVFC = (D3DFVF_XYZ | D3DFVF_DIFFUSE);
const DWORD FVFCN = (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_NORMAL);

// A structure for our custom vertex type
struct VERTEXCN
{
    float x, y, z; // The transformed position for the vertex
      float nx,ny,nz;
    DWORD color;        // The vertex color
};

// A structure for our custom vertex type
struct VERTEXC
{
    float x, y, z; // The transformed position for the vertex
    DWORD color;        // The vertex color
};

// A structure for our custom vertex type
struct VERTEX
{
```

```
    float x, y, z; // The transformed position for the vertex
    float tu, tv;  // Texture placement
};

struct FPOINT
{
      float x;
      float y;
};
```

The last thing to notice about the Render() method is that the text printing is being skipped with if(0). This is because normally this text is rarely needed and it is very costly to print. If all or a portion of the text is needed to be rendered for debugging purposes, that portion can be moved outside the if(0) statement. If(0) is not the most elegant way to do this, but since it is all debug code, it will eventually be removed from the application.

```
HRESULT D3D_GRAPHICS::InvalidateDeviceObjects()
{
      m_pFont->InvalidateDeviceObjects();
      floors->InvalidateDeviceObjects();
      // Invalidate all mesh object templates
      MeshTemplate::InvalidateAllDeviceObjects();

      return S_OK;
}
```

This method is called every time the devices need to be invalidated. This occurs whenever they are lost and RestoreDeviceObjects() needs to be called. As with RestoreDeviceObject(), FrameMove(), and render() every tier of the graphics engine has its own InvalidateDeviceObjects() method that takes care of invalidating the devices on that level.

```
HRESULT D3D_GRAPHICS::DeleteDeviceObjects()
{
      m_pFont->DeleteDeviceObjects();
      SAFE_DELETE(textures);
      SAFE_DELETE(camera);
      SAFE_DELETE(light);
      SAFE_DELETE(floors);
      SAFE_DELETE(input);
      return S_OK;
}
```

This is where all the devices are deleted. Whenever possible, the SAFE_DELETE() and SAFE_RELEASE() macros are used to handle the deleting and releasing of resources. These macros first check to make sure the object being deleted or released is not NULL, and then it deletes/releases the object and sets it to NULL. Using the macros just makes sure that these steps are always taken.

```
HRESULT D3D_GRAPHICS::FinalCleanup()
{
      SAFE_DELETE(m_pFont);
```

```
        return S_OK;
}
```

For this particular application the only thing that needs to be handled in this method is the deletion of the font being used.

```
LRESULT D3D_GRAPHICS::MsgProc( HWND hWnd, UINT uMsg, WPARAM wParam,
                                    LPARAM lParam )
{
        return CD3DApplication::MsgProc( hWnd, uMsg, wParam, lParam );
}
```

Some of what is going on in this method needs to be explained. The main WinProc() was created and assigned to the main window back in the ESCAPE_SIMULATION_CLIENT MakeWindow() method. The WinProc() callback function is found in the main_callback.h file and will be covered later. Normally in a WinProc() callback function, if a message is not handled it is passed along to the default Win32 message handler. In this application however, all messages, whether or not they are proceeded by the main WinProc() function, are passed along to this handler. Here they are given a chance to be handled and then are passed on to the CD3DApplication's MsgProc(). MsgProc() is given a final chance to handle the message before it is passed on to the default window handler. The reason for all this is that the CD3DApplication has several messages that it looks for in order to know how to handle the low level graphics details. For example, when the window is resized, the client window is changed. The low level graphics needs to know about this in order to modify the rectangle being rendered to so that it fits in the new client window space.

4.1.3.3 Graphics Templates

Before the next graphical tiers are covered, the topics of object templates should be covered. In order to pull out the messy details of graphic rendering from the DLL pluggins so the developers could focus on the algorithms, the concept of templates needed to be created. There are three types of graphics templates: mesh, point sprite, and field. Each type has all the methods and data needed to render any type of object that falls under that category. For example, a researcher may define a simulated camera to be rendered as a point sprite. Likewise, the same researcher might define a simulated light to also be a point sprite. Both objects have several characteristics that they share such as (x,y,z) floor values that define the exact position of the sprite in the world. However, they both have different textures maps, and only the light will use the ambience value. The point sprite has members for all the possible combinations of sprite usage. In both cases, camera and light, the object would create a point sprite template, but when each creates the template it would specify which fields are of interest for that particular object.

Templates are used by all objects, dynamic and static. Whereas in the case of static objects there is no need to update the state of the object by the server, in the case of dynamic objects the server will need to update different fields for different objects. For example, the camera object might be able to move, in which case it would receive an update from the server whenever its position values are changed. Similarly, the light's

ambience value might be updatable to allow the light to turn on and off or blink. So, when a template is defined it is told which values are updatable and which are not.

Templates are defined in the simulation file. When the application loads up a simulation it will read the simulation file and create whatever types are defined there. Later in the file the application will be told to create actually rendered objects based off the given templates.

Here is an example of a template:

```
// MeshTemplate class
class MeshTemplate : public Template
{
```

Notice here that the mesh template inherits from the template class. The template class contains generic methods and members that are common to all templates, for example, name, ID, and DLL file on the server that defines the object's behavior. Every template that is created must have these values defined in order to function properly. This base class also has virtual methods that are overloaded by the specific template class to fit the individual needs of the class. Finally, this base class has several static members and functions that are used to ensure that there are no duplications of object templates and for easy management of all the templates. Please refer to this base class in the appendices or the source code that accompanied this thesis for more details. There is nothing too complicated happening in the class and it should be easy to understand.

```
public:
      // Constructors/Destructors
      MeshTemplate(std::string objTypeName,
                   std::string dataFileName="",
                   std::string DLLFileName="");
      ~MeshTemplate();

      // Reporters
      OBJECT_TYPE GetType() const;
      bool Render(const D3DXMATRIX& matrix) const;

      // Mutators
      bool InvalidateDeviceObjects();
      bool RestoreDeviceObjects();
      bool SetTexture(unsigned int textureID);
      bool SetVertexBuffer(void* vBuffer, unsigned int numVertexes,
                           DWORD vertexType);
      bool SetIndexBuffer(WORD* iBuffer, unsigned int numIndexes);
```

Notice the usage of InvalidateDeviceObjects() and RestoreDeviceObject(). All the data for a specific object, except for it's unique attributes such as position, ambience, orientation, etc, are kept here in the templates. All the Direct3D devices that need to be invalidated or restored are contained here. This simplifies restoring and invalidating significantly. This way when the application needs to restore devices it just restores the templates instead of every object in the simulation. Simulation files will be covered later.

```
private:
```

```
        // variables
        DWORD m_wVertexType;       // Vertex format type
        unsigned int m_uiNumPrims; // Number of primitives
        void* m_pvVB;              // Vertex buffer (void* bcs it could
                                   // be 3 different vertex data types)
        WORD* m_pwIB;                 // Index buffer
        unsigned int m_uiNumVertexes; // Vertex buffer size
        unsigned int m_uiNumIndexes;  // Index buffer sizes

        LPDIRECT3DVERTEXBUFFER9 m_dxVB;     // DirectX vertex Buffers
        LPDIRECT3DINDEXBUFFER9  m_dxIB;     // DirectX index Buffers
        LPDIRECT3DTEXTURE9 m_pkTexture;     // Texture
};
```

The last data included in this particular template is the Direct3D device data. It should all be fairly straightforward.

4.1.3.4 FLOORS class

The next tier in the rendered data is the FLOORS class. The FLOORS class holds and manages all the floors in the application. It stores and manages this data as a doubly linked list of floors. A linked list was chosen because of the speed advantages that come with adding, deleting, and manipulating linked lists over some other data structures. When the world is rendered it always wants to step through each floor quickly, and linked lists are very fast for stepping through sequentially.

Here is the actual FLOORS class prototype:

```
class FLOORS
{
public:
        FLOORS(TEXTURES *_textures);
        ~FLOORS();

        int addFloor(const char *_name,double _lengthX,
                     double _lengthY,int level,int _texture_id);

        HRESULT addWalls(unsigned int id, VERTEX *vertex_list,
                         unsigned int num_vertices, WORD *vertex_index,
                         unsigned int num_indecies,
                         unsigned int _num_primatives, int texture_id);
        HRESULT addWalls(VERTEX *vertex_list, unsigned int num_vertices,
                         WORD *vertex_index, unsigned int num_indecies,
                         unsigned int _num_primatives, int texture_id);

        HRESULT setFloor(unsigned int id);
        HRESULT InvalidateDeviceObjects();
        HRESULT RestoreDeviceObjects();
        HRESULT renderFloor(int floor_id = 0);
        HRESULT renderFloors(unsigned int start = -1,
                             unsigned int finish = -1);
        HRESULT getDemensions(unsigned int id, double *x,double *y);

        unsigned int            numFloors;
        TEXTURES                *textures;
        FLOOR                   *bottom;
```

```
      FLOOR                   *top;
      FLOOR                   *it;
};
```

In this class we see many of the usual methods that we have already seen before. All the methods should be fairly self explanatory. It should be noted that walls are considered to be a single static object that each floor contains. This significantly speeds up rendering when you have a floor with a large number of walls. The walls are considered to be an attribute of a floor similar to the actual ground.

There is only one FLOORS class for any world. This could later be expanded to multiple instances for multiple buildings. Each FLOORS class holds all the floors of a given building. As stated above, they are organized as a linked list of FLOOR classes:

```
class FLOOR
{
public:
      FLOOR(TEXTURES *_textures, float _lengthX, float _lengthY,int
_level);
      ~FLOOR();
      char                        name[256];
      unsigned int                id;
      double                            lengthX;
      double                            lengthZ;
      int                               level;
      TEXTURES                    *textures; // This is used to
                                             // access any of the
                                             // needed textures
      LPDIRECT3DVERTEXBUFFER9 m_pVB;          // Vertex Buffers
      LPDIRECT3DINDEXBUFFER9  m_pIB;          // Index Buffers
      LPDIRECT3DTEXTURE9      texture;        // Floor Texture
      WALLS                   *walls;
      Objects                 dynamicObjects, staticObjects;
      density_field           *field;
      FLOOR                   *up;
      FLOOR                   *down;
};
```

Notice that there is no render() method or any of the other common methods that we have seen for most of the other classes up to this point. This is because the FLOORS class only holds the information for the given floor. The FLOORS class is responsible for rendering each floor it contains. This is not necessarily the best way to do this, but it works well enough. Also, you should notice that because the walls are part of the floor, they are separated from the staticObjects. All floors have walls as a single rendered object. There may be lots of other static objects like chairs, tables, etc, or none at all, but there will always be walls.

The next tier is the object tier.

4.1.3.5 Objects Class

Every floor has an instance of the Objects class in it. The Objects class, similar to the FLOORS class contains a linked list of object classes. Notice that in the floor there are actually two Objects pointers. The first points to the objects that change over the duration of the simulation, and the second to the objects that will remain the same. The reason for this division is to speed things up. The dynamic array will be constantly changing in size and contents. For example, if an object moves from floor one to floor two, it removes itself from the first floor's dynamic array and appends itself to the second.

Here is the Objects class:

```
// Objects class
class Objects
{
public:
      // Constructors/Destructor
      Objects();
      ~Objects();

      // Operators
      Objects& operator += (unsigned int incValue);
      Objects& operator -= (unsigned int decValue);
      Objects& operator ++ (int dummy);
      Objects& operator -- (int dummy);
      Objects& operator ++ ();
      Objects& operator -- ();

      // Reporters
      bool IsEmpty() const;
      Object* GetFist() const;
      Object* GetLast() const;
      Object* GetCurrent() const;

      // Mutators
      void Clear();
      bool GoToFirst();
      bool GoToLast();
      bool IncCurrent(unsigned int incValue=1);
      bool DecCurrent(unsigned int decValue=1);

      bool Append(Object* obj);
      bool Prepend(Object* obj);
      bool Insert(Object* obj);
      bool Remove();
      bool Remove(unsigned int objectID);

private:
      // Mutators
      void DeepCopy(const Objects& objects);

      // Variables
      Object*                m_pkFirst;
      Object*                m_pkCurrent;
```

```
        Object*                 m_pkLast;
};
```

Nothing in this class should be too difficult to understand. All the methods here are intended to help manage the linked list of objects that this Objects class maintains. If any of these are not clear please refer to the Object.cpp file for a explanation of the method's definition.

The really interesting things occur in the Object class itself. Here is the prototype:

```
// Object class
class Object
{
public:
        // Constructors/Destructor
        Object(unsigned int objTemplateID, bool dynamic,
                Object* prev=NULL, Object* next=NULL);
        Object(std::string objectName,
                bool dynamic, Object* prev=NULL, Object* next=NULL);
        Object(const Object& object);
        ~Object();
```

Here are your basic constructors and destructors. Notice the objTemplateID and dynamic flags that are passed in. This is the ID to the defined template type that this object falls under. Remember that the template knows how to render the object properly.

```
        // Operators
        Object& operator = (const Object& object);
        bool operator == (const Object& object) const;
        bool operator == (unsigned int objID) const;
        bool operator == (std::string objName) const;
        bool operator != (const Object& object) const;
        bool operator != (unsigned int objID) const;
        bool operator != (std::string objName) const;
```

Basic operator overloading.

```
        // Reporters
        std::string GetTypeName() const;
        OBJECT_TYPE GetObjectType() const;
        unsigned int GetID() const;
        bool IsDynamic() const;
        unsigned int GetFloorID() const;
        float GetPosX() const;
        float GetPosY() const;
        float GetPosZ() const;
        D3DXMATRIX GetTransMatrix() const;
        unsigned int GetTemplateID() const;
        std::string GetFields() const;
        Object* GetNext() const;
        Object* GetPrev() const;
        bool Render() const;
```

Basic reporters.

```
        // Mutators
        void SetNext(Object* nextObj);
        void SetPrev(Object* prevObj);
        void Detach();
        bool Update(float x, float y, float z, unsigned int floorID,
                    std::string fields);
        bool ReadRawData(void* rawData);
        bool Update(float x, float y, float z, unsigned int floorID,
                    void* fields=NULL);
```

Of these mutators the Update() and ReadRawData() methods deserve a little extra attention. The first four arguments for the Update() method are fairly obvious, but the last one is not. This argument contains the string that defines which of the possible fields for the given template will be updated by the server when it sends an update. This field is sent to the object by the server when the DLL is loaded. Part of the DLL's initialization is to determine which fields it will need to send updates about to the client.

The other mutator is the ReadRawData() method. This method is called by the network message handler when it receives an update for this specific object. Based on the fields string that is stored by the object it will interpret the raw data passed to it accordingly.

```
        // Statics
        static unsigned int GetNumObjects();
        static unsigned int GetNumStaticObjects();
        static unsigned int GetNumDynamicObjects();
        static Object** GetObjects();
        static Object* GetObject(unsigned int objID);
        static void ClearAllObjects();
```

These statics are for managing the objects in existence.

```
        // Variables
        unsigned int RAW_DATA_REQ;
```

This variable contains the number of bytes required to store all the data from the data that is to be updated by the server. This is used to speed up the actual update process as will be seen when the network message handler is covered later.

```
protected:
        // Reporters
        unsigned int GetNextObjectID() const;

        // Variables
        unsigned int        m_uiID;
        bool                m_bDynamic;
        float               m_fPosX, m_fPosY, m_fPosZ;
        unsigned int        m_uiFloorID;
        D3DXMATRIX          m_kTransformMatrix;

        unsigned int        m_uiTemplateID;

        Object*                 m_pkNext;
        Object*                 m_pkPrev;
```

All of the above should be self-explanatory. The D3DXMATRIX holds the transform matrix for this particular object. This is all the Direct3D information that an object needs to keep track of. All the rest of the information needed to render is saved in the template associated with m_uiTemplateID.

```
	static unsigned int		sm_uiNumStaticObjects;
	static unsigned int		sm_uiNumDynamicObjects;
	static Object*			sm_pkObjects[MAX_NUM_OBJECTS];
};
```

The last statics exist to help speed up the update process. It is very important for us to be able to find any object in existence as fast as possible no matter the floor when it comes time to update them. The reason for this is that there could be 1000 dynamic objects in existence, and there are no guarantees what floor they will be on. Because the objects in a given floor are stored in unsorted linked lists to make rendering faster, there is no way to find the object except by a brute force search for the object. Therefore, we must keep another data structure which is sorted and is fast to access. In this case, one of the fastest such data structures is an array. The reason linked lists were used instead of arrays for the object data on each floor is because arrays are not as fast as linked lists when it comes to modifying them, especially when it comes to adding and removing entries from the middle of an array.

However, for this array we do not need to worry about this problem. After the simulation file is loaded and has created all objects, no more objects are ever created or destroyed until the simulation is closed.

More on updating will be covered later.

The last thing that needs to be covered about objects is that the Object class is actually only a base class that is inherited by one of three specific object types: mesh, point sprite, and field. This is not the same thing as the templates. Please refer to the UML diagrams included in the appendices.

When an object is created, it is actually one of these three classes that is created. Thanks to polymorphism and inheritance everything works well. These classes are not as complex as the Object class, so only MeshObject class will be covered here:

```
// MeshObject class
class MeshObject : public Object
{
public:
	// Constructos/Destructors
	MeshObject(unsigned int objTemplateID, bool dynamic);
	MeshObject(std::string objectName, bool dynamic);
	MeshObject(const MeshObject& meshObj);
	~MeshObject();
```

Typical constructors and destructors, and in the .cpp file these call the Object class' counterparts.

```
	// Operators
	MeshObject& operator = (const MeshObject& meshObj);

	// Reporters
	D3DXMATRIX GetTransMatrix() const;

	// Mutators
	void SetTransMatrix(const D3DXMATRIX& matrix);
```

Basic mutators and accessors, nothing comlex.

```
	// Inherited
	OBJECT_TYPE GetObjectType() const;
	bool ReadRawData(void* rawData);
	bool Render() const;
	bool Update(float x, float y, float z, unsigned int floorID,
void* fields=NULL);
```

Overloaded functions to meet the MeshObject class' specifications.

```
private:
	// Variables
	D3DXMATRIX		m_kTransformMatrix;
```

The only difference between different objects of the same mesh template is the transform matrix.

4.1.3.6 Camera Class

Every world needs at least one camera. There are a lot of details that need to be taken care of to get a camera to work properly. To simplify things the CAMERA class was created. The prototype is as follows:

```
class CAMERA
{
public:
	CAMERA();
	~CAMERA();

	void SetPosition(const double x,const double y,const double z);
	void SetTarget(const double x,const double y,const double z);
	void GetPosition(double &x,double &y,double &z) const;
	void GetTarget(double &x,double &y,double &z) const;
	void RotateCamera(double h, double v);
	void RotateTarget(double h, double v);
	void MoveForward(double amount);
	void MoveBackward(double amount);
	void MoveLeft(double amount);
	void MoveRight(double amount);
	void MoveUp(double amount);
	void MoveDown(double amount);
	void SetView();
```

```
private:
      double            cameraX;
      double            cameraY;
      double            cameraZ;
      double            targetX;
      double            targetY;
      double            targetZ;

    D3DXMATRIX                m_matView;
            // View Matrix
};
```

In brief, here are each of the methods in the class and what they do:

- *SetPosition()*: Sets the position of the camera.
- *SetTarget()*: Sets the camera's target position.
- *GetPosition()*: Gets the current position of the camera in xyz.
- *GetTarget()*: Gets the current target of the camera in xyz.
- *RotateCamera()*: Rotates the camera around the target in spherical coordinates. The h value is for theta and v is for phi. The camera's magnitude from the target is kept constant during the rotation.
- *RotateTarget()*: The same as RotateCamera(), but in this case it is the target that is rotated around the camera.
- *MoveForward…MoveDown()*: As the names imply, they move the camera and target along the appropriate vectors. For example, if you use MoveForward() then it will move both the camera's and target's positions in the direction of the vector formed by the camera and the target. This vector points to the target. The magnitude of the change is given by the *amount* value which is passed in.
- *SetView()*: Sets the current camera and view being used by the graphics engine to this class' information.

At the end of each of the definitions for these methods the m_matView is reset to reflect the most current changes:

```
      D3DXMatrixLookAtLH( &m_matView,
                          &D3DXVECTOR3( (float)this->cameraX,
                                        (float)this->cameraY,
                                        (float)this->cameraZ), // from
                          &D3DXVECTOR3( (float)this->targetX,
                                        (float)this->targetY,
                                        (float)this->targetZ), // at
                          &D3DXVECTOR3( 0.0f, 1.0f, 0.0f));    // up
```

Other than this, everything in the definitions are normal trigonometric calculations which, thanks to the math library which will be covered later, are made very easy.

4.1.3.7 Light Class

There are a lot of different options when it comes to Direct3D light sources. Like the camera class, the LIGHT class hides a lot of the details involved in maintaining these light sources. Because of the shading being used, there must be at least one light source included in every world. If there is not one, then all un-textured objects, such as the density fields, will be rendered as being black.

The prototype for this class looks like this:

```
class LIGHT
{
public:
	LIGHT(DWORD _light_index = 0);
	~LIGHT();

	HRESULT setPosition(double x, double y, double z);
	HRESULT setDiffuse(double r, double g, double b, double a);
	HRESULT setAmbient(double r, double g, double b, double a);
	HRESULT setSpecular(double r, double g, double b, double a);
	HRESULT setDirection(double x, double y, double z);
	HRESULT setPhiTheta(double phi, double theta);
	HRESULT setFalloff(double falloff);
	HRESULT setRange(double r);
	HRESULT setType(D3DLIGHTTYPE type);
	HRESULT setAttenuation(float att, DWORD num = 1);
	HRESULT setAttenuation(float att0, float att1, float att2);
	HRESULT setLight(WORD options = 0);
	HRESULT setDeviceStates(WORD options = 0);
private:
	D3DLIGHT9	light;
	// LIGHT
	double			lightX;
	double			lightY;
	double			lightZ;
	DWORD		light_index;
};
```

From the names of these methods and their parameters, it should be fairly obvious what they each do. Using this class you can change any of the default settings that are used.

The definitions for each of these methods are very straightforward. The only two that need some explanation are setLight() and setDeviceStates():

```
HRESULT LIGHT::setLight(WORD options)
{

	SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()->SetLight(
this->light_index, &light );

	if(options)
		this->setDeviceStates();

	return S_OK;
```

```
}

HRESULT LIGHT::setDeviceStates(WORD options)
{
	if(options)
		this->setLight();

	SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()-
>SetRenderState( D3DRS_COLORVERTEX, TRUE );
	SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()-
>SetRenderState(D3DRS_ZENABLE, TRUE );
	SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()-
>SetRenderState(D3DRS_LIGHTING, TRUE);
	SINGLETON(ESCAPE_SIMULATION_CLIENT).GetD3DDevice()->LightEnable(
this->light_index, true);

	return S_OK;
}
```

Notice the use of SINGLETON(ESCAPE_SIMULATION_CLIENT) within both of these methods. This accesses the single ESCAPE_SIMULATION_CLIENT instance in the application. This is required here because access to the D3DDevice is needed to make the needed calls.

The SetLight() method in the LIGHT class tells the render engine to use this light when doing shading calculations. SetDeviceStates() sets the appropriate flags and settings so that the light will render things correctly. All of these settings can be looked up in [11].

4.2 Menus & UI

4.2.1 Win32 GUI

Back in the MakeWindow() method found in the ESCAPE_SIMULATION_CLIENT class the Win32 windows were setup. Along with creating the windows, the menus for this application were also setup. In Chapter 2, Win32 resources were covered. Referring to that portion of the thesis should aid in the understanding of the menus, toolbars, and dialogs. The main code for handling menu and toolbar selection is found in the WndProc() callback function in the main_callback.h and .cpp files. All the callback functions for the dialog boxes are contained in the files: dialogboxes.h and .cpp. There were only two things that had to be done out of the ordinary to get everything to work. First, within the callback functions the SINGLETON(ESCAPE_SIMULATION_CLIENT) notation needed to be used, and second, in order to allow the callback function to actually access the data refered to by the SINGLETON(ESCAPE_SIMULATION_CLIENT) tags, all the callback functions needed to be declared as friends in the ESCAPE_SIMULATION_CLIENT class prototype:

```
friend LRESULT CALLBACK D3DWndProc(HWND hwnd, UINT message, WPARAM
wParam, LPARAM lParam);

friend LRESULT CALLBACK ESCAPE_SIMULATION_CLIET_WndProc(HWND hwnd, UINT
message, WPARAM wParam, LPARAM lParam);
```

```
friend LRESULT CALLBACK ESCAPE_SIMULATION_CLIENT_TextOutProc(HWND hwnd,
UINT message, WPARAM wParam, LPARAM lParam);

friend BOOL CALLBACK AboutDialogBoxProc(HWND hDlg, UINT message, WPARAM
wParam, LPARAM lParam);

friend BOOL CALLBACK ConnectDialogBoxProc(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam);

friend BOOL CALLBACK LoadDialogBoxProc(HWND hDlg, UINT message, WPARAM
wParam, LPARAM lParam);

friend HRESULT WINAPI DirectPlayMessageHandler(PVOID pvUserContext,
DWORD dwMessageId, PVOID pMsgBuffer);
```

There are several changes that will be made to the user interface in the near future. At the moment it is still in a very basic format.

4.2.2 DirectInput

Along with the Win32 material there is DirectX input to deal with. As explained in Chapter 2, DirectInput is DirectX's interface for input. Because the input coming in from DirectInput is only used by the graphics engine, the DirectInput interface is solely contained in the D3D_GRAPHICS engine. Remember from in the D3D_GRAPHICS class prototype the following:

```
      D3D_GRAPHICS_INPUT      *input;     // Input Deceives
```

Remember, this is the class that was created to handle all the DirectInput stuff. The .cpp stuff is all fairly standard, but if you have not familiarized yourself with DirectInput it will be fairly overwhelming. Please refer to [5] on DirectInput for more information.

The D3D_GRAPHICS_IO.cpp file will not be covered. Here is the prototype for D3D_GRAPHICS_INPUT which is found in the D3D_GRAPHICS_INPUT.h:

```
class D3D_GRAPHICS_INPUT
{

      friend BOOL CALLBACK  DInput_Enum_Joysticks(LPCDIDEVICEINSTANCE
lpddi,LPVOID guid_ptr);
public:
      D3D_GRAPHICS_INPUT(HINSTANCE _hInstance,HWND _hwnd);
      ~D3D_GRAPHICS_INPUT();
      HRESULT GETINPUT();
      HRESULT ENABLE(INPUT_ID_TYPE inputTypeID = INPUT_ID_TYPE_ALL);
      HRESULT DISABLE(INPUT_ID_TYPE inputTypeID = INPUT_ID_TYPE_ALL);

      // This is a public interface for the information gathered from
the input devices.
      INPUT_INFO  info;
private:
      // STATE VARIABLES
```

```
      bool                          INPUT_SETUP;
      bool                          KEYBOARD_VALID;
      bool                          MOUSE_VALID;
      bool                          GAME_PAD_VALID;

      // GENERAL DIRECTX8 OBJECTS
      HWND                          hwnd;
      HINSTANCE                     hInstance;        // WINDOW HANDLE
      LPDIRECTINPUT8                lpdi;             // DIRECT INPUT8
OBJECT
      LPDIRECTINPUTDEVICE8    lpdikey;          // KEYBOARD INTERFACE
      LPDIRECTINPUTDEVICE8    lpdimouse;        // MOUSE INTERFACE
      LPDIRECTINPUTDEVICE8    lpdigpad;         // GAME PAD INTERFACE

      // DATA VARIABLES
      GUID              joystickGUID;     // GUID FOR GAME PAD
      static char       joyname[260];     // NAME FOR GAME PAD

      HRESULT GETKEYBOARD();
      HRESULT GETMOUSE();
      HRESULT GETGAMEPAD();
};
```

If familiar with the reference given above, most of this should be very straightforward. The only nonstandard thing found here is the INPUT_INFO info structure member. This Structure looks like this:

```
typedef struct INPUT_INFO_TAG
{
      bool                          KEYBOARD_VALID;
      bool                          MOUSE_VALID;
      bool                          GAME_PAD_VALID;
```

These bools indicate whether or not the device exists. For example, if the user forgot to plug in their gamepad before running the application then the GAME_PAD_VALID bool will be false.

```
      // MOUSE INFO
      DIMOUSESTATE                  mouseState;       // MOUSE STATE
      BUTTON_CLICK_STATE            mouse_lb;
      BUTTON_CLICK_STATE            mouse_mb;
      BUTTON_CLICK_STATE            mouse_rb;
```

This section is for the mouse state information.

```
      // GAMEPAD INFO
      DIJOYSTATE                    gpadState;        // GAME PAD STATE
      float                         s1x;
      float                         s1y;
      float                         s2x;
      float                         s2y;
      BUTTON_CLICK_STATE            gpadButtons[32];
```

The game pad information is a little less obvious than the mouse state information. Most game pads have two sticks. The x and y values for the first stick are s1.x and s1.y, and for the second stick are s2.x and s2.y. DirectInput can recognize up to 32 buttons. There is no set way for them to be ordered, but for the most part, the first four buttons are the ones immediately under the user's right hand.

```
    // KEYBOARD INFO
    UCHAR            keyBuffer[256];   // BUFFER FOR KEYBOARD DATA


} INPUT_INFO, *PTR_INPUT_INFO;
```

DirectInput normally only grabs the first 256 keys. You can check to see if a given key has been processed by using the KEYPRESSED macro defined at the top of this file, and by passing this array and the DirectInput #define for the given key to the macro. For example, to check the 'k' key you would use KEYPRESSED(info.keyBuffer,DIK_K).

The info structure is updated with all the latest input state's information by calling:

```
    input->GETINPUT();
```

In the Escape Simulation client, this method is called at the beginning of the FrameMove() method in the D3D_GRAPHICS class. After the info structure is updated, there are three methods that are called that will go through the updated information and modify the cameras, lights, world, etc, accordingly. These methods are:

```
    HRESULT FrameMove_Keyboard();
    HRESULT FrameMove_Mouse();
    HRESULT FrameMove_Gpad();
```

They are all defined in the D3D_GRAPHICS_IO_PROCESS.cpp file, and are all fairly similar. The main difference is that each accesses the info data structure only for its particular data. That being the case, only a portion of one of the methods will be covered here:

```
HRESULT D3D_GRAPHICS::FrameMove_Keyboard()
{
    static z_down = false;
    static x_down = false;
    float  div_amount = 521.0f;
    div_amount = 512.0f * this->m_fFPS;
```

Basic variable initialization. The z_down and x_down variables are of particular interest. These are used to ensure that only one key press is a recognized press. This is needed because humans are relatively slow compared to computers. Even the most quickly tapped key seems to the computer to be down for several cycles. In that amount of time the computer could have updated the input information structure several times. Without a flag to let this method know that we are still on the same key press, it would think it is new input. This behavior is not necessarily always a bad thing. In the *if* statements

immediately below we do not care whether we keep running the code every time we enter this method. In fact, that is what we want.

```
if(KEYPRESSED(input->info.keyBuffer,DIK_P))//ROTATE CCW
{
	camera->RotateCamera(.01, 0);
}
if(KEYPRESSED(input->info.keyBuffer,DIK_O))//ROTATE CW
{
	camera->RotateCamera(-.01, 0);
}
if(KEYPRESSED(input->info.keyBuffer,DIK_I))//TILT DOWN
{
	camera->RotateCamera(0,.01);
}
if(KEYPRESSED(input->info.keyBuffer,DIK_J))//TILT UP
{
	camera->RotateCamera(0,-.01);
}
if(KEYPRESSED(input->info.keyBuffer,DIK_W))//MOVE CAMERA FORWARD
{
	camera->MoveForward(.1);
}
if(KEYPRESSED(input->info.keyBuffer,DIK_S))//MOVE CAMERA BACKWARD
{
	camera->MoveBackward(.1);
}
if(KEYPRESSED(input->info.keyBuffer,DIK_A))
{
	camera->MoveLeft(.1);
}
if(KEYPRESSED(input->info.keyBuffer,DIK_D))
{
	camera->MoveRight(.1);
}
```

The above *if* statements are responsible for moving the camera around based on the keyboard input

```
if(KEYPRESSED(input->info.keyBuffer,DIK_Z))// SWITCH TO SINGLE
                                           // FLOOR VIEW
{
	if(!z_down)
	{
		z_down = true;
		this->graphics_mode = this->graphics_mode ^
                                      SINGLE_FLOOR_MODE;
		MODE_UPDATE mode_change_msg;
		mode_change_msg.type = SIMULATION_MODE_UPDATE;
		mode_change_msg.mode = graphics_mode;
		SINGLETON(ESCAPE_SIMULATION_CLIENT).sendmsg(
			(void *)&mode_change_msg,sizeof(MODE_UPDATE));
	}
}
else if(KEYPRESSED(input->info.keyBuffer,DIK_X))// TOGGEL BACK
```

```
                                                    // AND FORTH FROM
                                                    // MICRO AND
                                                    // MACRO
        {
                if(!x_down)
                {
                        x_down = true;
                        if(this->graphics_mode &MICRO_DISPLAY_MODE_AVAILABLE)
                        this->graphics_mode = this->graphics_mode ^
                                                        MICRO_DISPLAY_MODE_SET;
                         MODE_UPDATE mode_change_msg;
                         mode_change_msg.type = SIMULATION_MODE_UPDATE;
                         mode_change_msg.mode = graphics_mode;
                         SINGLETON(ESCAPE_SIMULATION_CLIENT).sendmsg(
                               (void *)&mode_change_msg,sizeof(MODE_UPDATE));
                }
        }
        else
        {
                z_down = false;
                x_down = false;
        }
        return S_OK;
}
```

The last two *if* statements in the method are important. First, they show how to handle receiving one key press even when the user holds a key down. This is done by using the x_down and z_down statics. Second, they are both an example of client/server communication.

4.3 Network Engine

The end of the last section showed how the DirectInput keyboard input handler is able to toggle the graphics mode between micro and macro display modes, and single- and multi -level display mode. In both cases, the client needs to inform the server of the desired change. Neither of these changes requires the state machine to be put into a specific state to handle the communication. The reason for this is that even if there are several packets of the wrong mode that are sent, there will not be any problem with the rendering. These changes in mode only really help to reduce the number of packets being sent over the network.

The network engine uses DirectPlay to handle all the communication between the client and the server. The section in Chapter 2 on DirectPlay covered what it takes to both host and connect a client and server, and how the message handler works. Please refer to [12]for a more in-depth explanation of any method or function that is covered. The following section describes how messages are put together, sent, received, and broken down. All the major message types will eventually be covered here. To begin, the two messages above will be used as an example.

These two messages really are the same thing. In both cases the graphics mode flags have been changed and the server side mode flags need to be updated accordingly. The

duplication of the message code needs to be pulled out to reduce the code. In both cases the message code is:

```
MODE_UPDATE mode_change_msg;
mode_change_msg.type = SIMULATION_MODE_UPDATE;
mode_change_msg.mode = graphics_mode;
SINGLETON(ESCAPE_SIMULATION_CLIENT).sendmsg(
                    (void *)&mode_change_msg,sizeof(MODE_UPDATE));
```

The message being created in this case is a MODE_UPDATE message. This message was created simply to make building this message by the client, and later, extraction of the needed sent information by the server, easier.

MODE_UPDATE as well as all other message typedefs are defined in the message_prototypes.h file. In this file MODE_UPDATE is defined as:

```
typedef struct MODE_UPDATE_TAG
{
      BYTE              type;
      unsigned long     mode;
} MODE_UPDATE, *PTR_MODE_UPDATE;
```

This file is common to both the client and server components. Thus the message handler receiving a given message can cast the raw data it received after identifying the correct message by its unique type value.

In this example, the type value being used is the SIMULATION_MODE_UPDATE which is placed in the type field of the message. The type field is of type byte. This allows 256 possible unique message types. If in the future this number is too small, it is an easy change to increase this value to an unsigned int or greater. All the type values are #defined at the top of the message_prototypes.h file. For example:

```
#define ID_REQUEST                          0
#define ID_REPLY                            10
#define WORLD_LIST_REQUEST                  20
#define WORLD_LIST_REPLY                    30
#define WORLD_DATA_REQUEST                  40
#define WORLD_DATA_REPLY                    50
#define RUN_SIMULATION                      60
#define STOP_SIMULATION                     70
#define SIMULATION_WIPE                     75
#define SIMULATION_WIPE_CONFIRM             76
#define LOAD_WORLD_FILE                     77
#define LOAD_WORLD_FILE_CONFIRM             78
#define SIMULATION_MODE_UPDATE              80
#define DENSITY_FIELD_DLL_LOAD              81
#define DENSITY_FIELD_DLL_LOADED            82
#define DYNAMIC_DATA_DLL_LOAD               83
#define STATIC_DATA_OBJECT_CREATION         84
#define DYNAMIC_DATA_OBJECT_CREATION        85
#define DYNAMIC_DATA_REQUEST                86
#define DYNAMIC_DATA_UPDATE                 90
```

```
#define FIELD_DATA_UPDATE                       91
#define MICRO_TO_MACRO                          100
#define MACRO_TO_MICRO                          110
#define DYNAMIC_SWITCH_SUCCESS                  120
#define CLOSE_CONNECTION                        200
#define CLOSE_CONNECTION_CONFIRM                201
#define CLOSE_SESSION                           210
#define CLOSE_SESSION_CONFIRM                   211
#define QUIT                                    254
#define QUIT_CONFIRM                            255
```

For this message the other field in the structure is an unsigned long labeled mode. This is where the graphics mode unsigned long is placed. The server will grab this value and update its own mode accordingly.

Finally, the message that was generated needs to be sent. To do this the handler in the graphics module must have access to the network module to use the sendmsg() method. This is done by using the SINGLTON(ESCAPE_SIMULATION_CLIENT) tag.

The sendmsg() prototype defined in evacuation_client.h is:

```
HRESULT sendmsg(void *msg, DWORD length);
```

The method itself is very simple. When sending something off you take the date being sent, cast it to a void *, and include the total data width in bytes as the second parameter.

Before continuing, the internal behavior of this method needs to be addressed. Depending on whether the call is blocking or not, whether it is guaranteed or not, and how the data is treaded can make a big difference in how this method should be used.

The definition of this method in evacuation_client.cpp is:

```
HRESULT EVC::sendmsg(void *msg, DWORD length)
{
	if(connected)
	{
		dpnBuffer.pBufferData = (BYTE *)msg;
		dpnBuffer.dwBufferSize = length;
		return g_pDPClient->Send(&dpnBuffer,1,0,NULL,NULL,
					DPNSEND_GUARANTEED |
					DPNSEND_SYNC |
					DPNSEND_NOLOOPBACK);
	}
	return S_OK;
}
```

(Note: this is for the client. The server is almost identical in definition. Only the call to send() is changed to sendto() which includes a reference to the particular target client.

Of the possible flags that can be passed to this method the ones currently being sent tell send to transmit without blocking, to guarantee reception, and to make a personal copy of

the buffer being sent to ensure that the data is not prematurely freed. This last setting is part of the default behavior and adds only a little overhead.

Because the sendmsg() method makes a copy of the data to be sent, it is safe to pass it local and temporary variables.

DirectPlay will wrap the message in its own header because it has several internal message types that it sends in order to manage connections. It is possible for the developer to monitor these messages and add his/her own code. In our case we only care about a DPN_MSGID_RECEIVE message. This type of message is received whenever DirectPlay receives a message from the client or server respectively. The DirectPlay message structure associated with this type of message looks like this:

```
typedef struct {
    DWORD       dwSize;
    DPNID       dpnidSender;
    PVOID       pvPlayerContext;
    PBYTE       pReceiveData;
    DWORD       dwReceiveDataSize;
    DPNHANDLE   hBufferHandle;
} DPNMSG_RECEIVE, *PDPNMSG_RECEIVE;
```

The only two fields needed from this structure are pReceiveData and dwReceiveDataSize. The first is a pointer to the raw data that was sent and the second is the size in bytes.

For both the server and client message handlers, whenever the handler receives a DPN_MSGID_RECEIVE message it will cast the pReceiveData pointer as a (BYTE *) and then will use the first byte of raw data to switch on. This works because the first byte of all the different message structures is the message type identifier.

Returning to our example, when the handler receives the message and switches on the type value it will be caught by the SIMULATION_MODE_UPDATE case. From there the pReceiveData pointer will be cast into a MODE_UPDATE structure pointer and the data retrieved and server mode updated. Here is the code:

```
case SIMULATION_MODE_UPDATE:
      {
            MODE_UPDATE *msg = (MODE_UPDATE *)pMsg->pReceiveData;
            SINGLETON(ESCAPE_SIMULATION_SERVER).world_mode = msg->mode;
            break;
      }
```

Notice that the server also uses the SINGLETON class template, but this time the class is of type ESCAPE_SIMULATION_SERVER.

The rest of the DirectPlay details in both the client and server network components are fairly straightforward.

The rest of the message handler will be covered in a moment, but first here are the rest of the message prototypes found in the message_prototypes.h file:

```
typedef struct GENERAL_TAG
{
      BYTE  type;
} GENERAL,*PTR_GENERAL;

typedef struct GENERAL_W_SMALL_BUFFER_TAG
{
      BYTE  type;
      char  buffer[256];
} GENERAL_W_SMALL_BUFFER,*PTR_GENERAL_W_SMALL_BUFFER;

typedef struct GENERAL_W_BUFFER_TAG
{
      BYTE  type;
      char  buffer[1024];
} GENERAL_W_BUFFER,*PTR_GENERAL_W_BUFFER;

typedef struct GENERAL_W_LARGE_BUFFER_TAG
{
      BYTE  type;
      char  buffer[2048];
} GENERAL_W_LARGE_BUFFER,*PTR_GENERAL_W_LARGE_BUFFER;
```

These first typedefs define the four most basic message types. Many of the message types that follow are just one of these four with a different name, for example, the INIT message typedef immediately below:

```
typedef GENERAL_W_BUFFER INIT;

typedef struct BOOKKEEPING_TAG
{
      BYTE  type;
      LONG  length;
      char  errormessage;
} BOOKKEEPING,*PTR_BOOKKEEPING;

typedef     GENERAL_W_LARGE_BUFFER FILE_REQUEST;

typedef GENERAL_W_LARGE_BUFFER FILE_LIST;

typedef struct FILEIO_TAG
{
      BYTE  type;
      BYTE  kind;
      char  path[1024];
      char  filename[1024];
      LONG  length;
      char  data;
} FILEIO,*PTR_FILEIO;

typedef struct STATE_TAG
{
```

```
      BYTE  type;
      BYTE  state;
} STATE,*PTR_STATE;

typedef struct MODE_UPDATE_TAG
{
      BYTE              type;
      unsigned long     mode;
} MODE_UPDATE, *PTR_MODE_UPDATE;

typedef GENERAL_W_BUFFER_TAG WORLD_LOAD;

typedef struct FIELD_DLL_LOAD_TAG
{
      BYTE              type;
      char              filename[1028];
      unsigned int      bytes;
      char              buffer;
} FIELD_DLL_LOAD, *PTR_FIELD_DLL_LOAD;

typedef GENERAL_W_BUFFER_TAG DYNAMIC_DLL_LOAD;

typedef struct STATIC_OBJECT_CREATION_TAG
{
      BYTE              type;
      unsigned int      typeID;
      unsigned int      bytes;
      char              buffer;
} STATIC_OBJECT_CREATION,*PTR_STATIC_OBJECT_CREATION;

typedef struct DYNAMIC_OBJECT_CREATION_TAG
{
      BYTE              type;
      unsigned int      typeID;
      unsigned int      bytes;
      char              buffer;
} DYNAMIC_OBJECT_CREATION,*PTR_DYNAMIC_OBJECT_CREATION;

typedef struct DYNAMIC_TAG
{
      BYTE  type;
      BYTE  kind;
      int         num_objects;
      LONG  length;
      char  data;
} DYNAMIC,*PTR_DYNAMIC;

typedef struct FIELD_UPDATE_TAG
{
      BYTE  type;
      char  data;
} FIELD_UPDATE, *PTR_FIELD_UPDATE;
```

Creating a new type is as simple as adding another typedef and corresponding #define to this file.

The message handlers for the client and server are very different as far as the messages they handle. It is important to remember that all the inbound messages are handled here, but only some of the outbound ones originate here. Also, the actions of these handlers affect the state machine behavior.

4.3.1 Evacuation Client Message Handler

Starting from the switch statement, the client's network message handler is as follows:

```
switch((BYTE)(*pMsg->pReceiveData))
{
      case ID_REPLY:  // ID was received from server
      {
            WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              hDynamic_data_mutex,INFINITE);
            INIT *msg = (INIT *)pMsg->pReceiveData;
            this_ptr->server_version = msg->buffer;
            SINGLETON(ESCAPE_SIMULATION_CLIENT).id_recieved = true;
            ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              hDynamic_data_mutex);
            break;
      }
      case WORLD_LIST_REPLY: // World list was received from server
      {
            WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              hDynamic_data_mutex,INFINITE);
            FILE_LIST *world_list_reply = (FILE_LIST *)
                              pMsg->pReceiveData;
            int length = (int)strlen(world_list_reply->buffer);
            SINGLETON(ESCAPE_SIMULATION_CLIENT).fileList =
                              (void *)new char[(size_t)length + 1];
            memcpy(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        fileList,world_list_reply->buffer,length);
            ((char*)(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              fileList))[length] = '\0';
            SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              FILE_LIST_RECEIVED = true;
            ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              hDynamic_data_mutex);
            break;
      }
      case WORLD_DATA_REPLY: // World file data received from server
      {
            WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              hDynamic_data_mutex,INFINITE);
            SINGLETON(ESCAPE_SIMULATION_CLIENT).received_data_msg =
                        (void *)new char[pMsg->dwReceiveDataSize];
            memcpy(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        received_data_msg,pMsg->pReceiveData,
                        pMsg->dwReceiveDataSize);
            SINGLETON(ESCAPE_SIMULATION_CLIENT).FILE_RECEIVED = true;
            ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                              hDynamic_data_mutex);
            return hr;
      }
      case SIMULATION_WIPE_CONFIRM: // Old simulation wipe successful
```

```
{
      WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex,INFINITE);
      SINGLETON(ESCAPE_SIMULATION_CLIENT).simulationWiped = true;
      ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex);
      return hr;
}
case DENSITY_FIELD_DLL_LOADED: // DField DLL loaded successfully
{
      WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex,INFINITE);
      SINGLETON(ESCAPE_SIMULATION_CLIENT).fieldDllLoaded = true;
      ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex);
      return hr;
}
case LOAD_WORLD_FILE_CONFIRM: // World file loaded successfully
{
      WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex,INFINITE);
      SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        serverWorldLoaded = true;
      ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex);
      return hr;
}
case DYNAMIC_DATA_UPDATE: // Dynamic data update from the server
{
      WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex,INFINITE);
      DYNAMIC *tempDptr = (DYNAMIC *)pMsg->pReceiveData;
      char  *tempPtr    = &(tempDptr->data);
      int         totalRead   = 0;
      while(totalRead < tempDptr->num_objects)
      {
            if(Object::GetObject(*((int *)tempPtr)))
                  Object::GetObject(*((int *)tempPtr))
                  ->ReadRawData((void *)tempPtr);
            tempPtr += Object::GetObject(*((int *)tempPtr))
                  ->RAW_DATA_REQ;
            totalRead++;
      }
      ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex);
      return hr;
}
case FIELD_DATA_UPDATE: // Field data update from the server
{
      WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex,INFINITE);
      FIELD_UPDATE *tempFptr = (FIELD_UPDATE *)pMsg
                        ->pReceiveData;
      SINGLETON(ESCAPE_SIMULATION_CLIENT).floors
                        ->setFloor(*((unsigned int *)(&tempFptr
                        ->data)));
      SINGLETON(ESCAPE_SIMULATION_CLIENT).floors->it->field
```

```
                        ->ReadRawData((void *)&tempFptr->data);
            ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex);
            return hr;
      }
      case DYNAMIC_SWITCH_SUCCESS: // OLD MESSAGE
      {
            WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex,INFINITE);
            SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        dynamic_switch_done = true;
            ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex);
            return hr;
      }
      case STOP_SIMULATION: // Stop the smulation
      {
            WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex,INFINITE);
            SINGLETON(ESCAPE_SIMULATION_CLIENT).simulating = false;
            ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                        hDynamic_data_mutex);
            return hr;
      }
      default:
      {
            break;
      }
}
```

There are two messages above that should be covered in a little more depth: the DYNAMIC_DATA_UPDATE and the FIELD_DATA_UPDATE. The first is sent whenever there has been a time step on the server side, and a portion of the dynamic data has been changed. (More will be said later about time steps.) This message contains the information needed to update the client's graphics to match the world's data. The second is a similar idea. Until now nothing has been said about fields, so here is a brief explanation: One way to represent the placement of objects in the world is by creating a density field. Depending on how small you want to make the steps for the two axes you average all the objects in a given cell on the grid and give it a value equal to the density. The Escape Simulator creates a density field for every floor that is created. The simulation file is given an opportunity to change the algorithm used in modifying these fields as well as the dimensions.

Depending on whether the current view is macro or micro, either the field will be visible or the individual objects will. Each is updated slightly differently in the client, and therefore there are two update messages that have been created.

Common to both is the acquiring of the semaphore. This ensures that the data can be changed without causing problems elsewhere. At the end of the case handler the semaphore is released to allow the state machine and the rendering engine to continue.
In order to update the dynamic data properly both the client and the server need to have the same numbering for each of the objects in existence. This ensures a proper update

because as a simulation is loaded each object is created on both sides at the same time and in the same order. Obviously it is desirable to make the update as fast and small as possible. The question is *how*?

It might be nice to have some kind of preset order where both the client and the server have all the objects in the same order. This would allow the packet containing the update to not need any object identification data. This would also mean that you always have to update all the objects, regardless of whether or not they were actually changed.

We also have to consider the fact that the objects are stored as linked lists and are constantly changing, so we cannot use a preset order to get rid of transmitting the object identifiers. That being the case, it would still be nice to improve the means of looking up the different identifiers in order for that part to be as fast as possible.

As far as access speeds for data structures, nothing is as fast as an array. Due to the fact that none of the objects are destroyed after being created, and that both the server and the client create the objects in the same order, we can create an array of object pointers that we fill as we create the objects. This array has no other purpose than to speed up the updating process by giving the client a fast way to find the objects by ID, the ID being the index into the array that points to the correct object.

Finally, in order to reduce the number of bytes being transferred, and the speed with which they are encoded, when the server loads up a DLL pluggin for an object template type, it asks the DLL which fields will be updated. This information is then saved in the client templates. The data has a predefined order in which it must be updated. This order will be discussed more in the section on DLLs below. Since the client and server know what data is being updated and the order in which the data will be stored, there is no need for extra information to be transmitted.

Part of the Object base class client side is a method ReadRawData() that takes in a void * to the raw update data sent from the server. This pointer is cast properly and the data is withdrawn by the object based on the flags that were given it at creation.

Server side, each pluggin is required to have a function Dump() which is the counterpart to the client's ReadRawData().

After all the discussion above, it is time to dissect the DYNAMIC_DATA_UPDATE state:

```
case DYNAMIC_DATA_UPDATE: // Dynamic data update from the server
{
      WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                          hDynamic_data_mutex,INFINITE);
```

This is where the state grabs the semaphore.

```
      DYNAMIC *tempDptr = (DYNAMIC *)pMsg->pReceiveData;
      char  *tempPtr    = &(tempDptr->data);
```

```
int         totalRead   = 0;
```

The first of these lines is the cast of the raw data received by DirectPlay into the proper message type. The next pointer will be used to increment through the actual object update data. Finally, totalRead is used to ensure that the number of objects read matches the total number of objects being updated in this message.

```
while(totalRead < tempDptr->num_objects)
{
```

Loop until we get all the objects.

```
if(Object::GetObject(*((int *)tempPtr)))
      Object::GetObject(*((int *)tempPtr))
      ->ReadRawData((void *)tempPtr);
```

If there is an object in the object array at the index stored in the first four bytes of tempPtr, then have that object read in the data intended for it.

```
tempPtr += Object::GetObject(*((int *)tempPtr))
      ->RAW_DATA_REQ;
totalRead++;
```

To understand the first if these two lines one must remember that part of every object template created is a RAW_DATA_REQ member that holds the byte requirement for the object. Once the object at the given index has extracted its update data, tempPtr needs to be advanced to the beginning of the next object's data. After the pointer is adjusted, we add one to the totalRead count.

```
}
ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                  hDynamic_data_mutex);
```

Finish up by releasing the semaphore.

```
      return hr;
}
```

Here is the FIELD_DATA_UPDATE case handler. Note that there are few major differences with DYNAMIC_DATA_UPDATE. The only significant one is that there is only one thing being updated here, and this is because there is only one field per floor.

```
case FIELD_DATA_UPDATE: // Field data update from the server
{
      WaitForSingleObject(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                  hDynamic_data_mutex,INFINITE);
      FIELD_UPDATE *tempFptr = (FIELD_UPDATE *)pMsg
                  ->pReceiveData;
```

Gets the semaphore and casts to the FIELD_UPDATE message type.

```
SINGLETON(ESCAPE_SIMULATION_CLIENT).floors
                  ->setFloor(*((unsigned int *)(&tempFptr
```

```
                                    ->data)));
```

Gets the floor's linked list's iterator pointing to the correct floor.

```
            SINGLETON(ESCAPE_SIMULATION_CLIENT).floors->it->field
                                    ->ReadRawData((void *)&tempFptr->data);
            ReleaseMutex(SINGLETON(ESCAPE_SIMULATION_CLIENT).
                                    hDynamic_data_mutex);
```

Updates the field data and releases the semaphore.

```
            return hr;
      }
```

4.3.2 Evacuation Server Message Handler

After everything that has been covered on the client's Message Handler, there is not much that differs on the server side. Though the messages are different, the format and structure are identical. Some of the server side messages will be covered later in the section that is directly related to them, such as the DENSITY_FIELD_DLL_LOAD message. Here is the message switch statement code:

```
switch((BYTE)(*pMsg->pReceiveData))
{
      case ID_REQUEST:  // ID
      {
            INIT id_responce;
            id_responce.type = ID_REPLY;
            strcpy(id_responce.buffer,"ESS 1.0");
            this_ptr->sendmsg((void*)&id_responce,sizeof(id_responce));
            break;
      }
      case WORLD_LIST_REQUEST:      // World List request
      {
            HANDLE dirHandle;
            WIN32_FIND_DATA file_info;
            string fileList;
            FILE_LIST fileListMsg;
            dirHandle = FindFirstFile("../media/vmaps/*",&file_info);
            parser = file_info.cFileName;
            fileList = "";
            if(parser.find(VMAP_END) != parser.npos)
            {
                  fileList += file_info.cFileName;
            }
            while(FindNextFile(dirHandle,&file_info))
            {
                  parser = file_info.cFileName;
                  if(parser.find(VMAP_END) != parser.npos)
                  {
                        fileList += " ";
                        fileList += file_info.cFileName;
                  }
            }
            FindClose(dirHandle);
```

```
        fileListMsg.type = 3;
        memset(fileListMsg.buffer,'\0',sizeof(fileListMsg.buffer));
        strcpy(fileListMsg.buffer,fileList.c_str());
        this_ptr->sendmsg((void*)&fileListMsg,sizeof(fileListMsg));
        break;
}
case WORLD_DATA_REQUEST:
{
        // SETUP THE NESSESARY VARIABLES
        LONG length,bufferlength;
        FILEIO *world_data_reply;
        FILE_REQUEST *world_data_request =
                    (FILE_REQUEST *)pMsg->pReceiveData;
        fstream infile;

        // OPEN THE REQUESTED FILE
        parser = VMAP_LOCATION;
        parser = parser + world_data_request->buffer;
        infile.clear();
        infile.open(parser.c_str(),ios_base::binary |
                                   ios_base::in );
        if(infile.is_open())
        {
                // GET THE FILE LENGTH
                infile.seekg(0,ios_base::end);
                length = infile.tellg();
                infile.seekg(0,ios_base::beg);
                bufferlength = length + sizeof(FILEIO) - 1;
                // WRITE DATA INTO THE RESPONCE STRUCT
                char *buffer = new char[bufferlength];
                memset(buffer,'\0',bufferlength);
                world_data_reply = (FILEIO *)buffer;
                world_data_reply->type = WORLD_DATA_REPLY;
                world_data_reply->length = length;
                strcpy(world_data_reply->filename,
                                        world_data_request->buffer);
                strcpy(world_data_reply->path,VMAP_LOCATION);
                world_data_reply->kind = VMAP;
                infile.read(&(world_data_reply->data),length);
                infile.close();
                // SEND IT AWAY :)
                this_ptr->sendmsg((void*)world_data_reply,
                                     bufferlength);
                // CLEANUP TIME
                delete [] buffer;
                world_data_reply = NULL;
        }
        break;
}
case SIMULATION_WIPE:
{
        GENERAL wipe_confirmation;
        wipe_confirmation.type = SIMULATION_WIPE_CONFIRM;
        if(SINGLETON(ESCAPE_SIMULATION_SERVER).floors)
        {
                delete SINGLETON(ESCAPE_SIMULATION_SERVER).floors;
                SINGLETON(ESCAPE_SIMULATION_SERVER).floors =
```

```
                              new Floors();
        }
        if(SINGLETON(ESCAPE_SIMULATION_SERVER).field_dll)
        {
              delete SINGLETON(ESCAPE_SIMULATION_SERVER).field_dll;
              SINGLETON(ESCAPE_SIMULATION_SERVER).field_dll =
                              new Dlls();
        }
        if(SINGLETON(ESCAPE_SIMULATION_SERVER).dlls)
        {
              delete SINGLETON(ESCAPE_SIMULATION_SERVER).dlls;
              SINGLETON(ESCAPE_SIMULATION_SERVER).dlls =
                              new Dlls();
        }
        this_ptr->sendmsg((void*)&wipe_confirmation,
                        sizeof(GENERAL));
        }
        break;
}
case SIMULATION_MODE_UPDATE:
{
        MODE_UPDATE *msg = (MODE_UPDATE *)pMsg->pReceiveData;
        SINGLETON(ESCAPE_SIMULATION_SERVER).world_mode = msg->mode;
        break;
}
case LOAD_WORLD_FILE:
{
        GENERAL world_load_confirm_msg;
        world_load_confirm_msg.type = LOAD_WORLD_FILE_CONFIRM;
        WORLD_LOAD *msg = (WORLD_LOAD *)pMsg->pReceiveData;
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                        ReadVMapFile(msg->buffer);
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                        sendmsg((void*)&world_load_confirm_msg,
                        sizeof(GENERAL));
        break;
}
case DENSITY_FIELD_DLL_LOAD:
{
        void *buffer;
        FIELD_DLL_LOAD *msg = (FIELD_DLL_LOAD *)pMsg->pReceiveData;
        if(SINGLETON(ESCAPE_SIMULATION_SERVER).field_dll)
        {
              SINGLETON(ESCAPE_SIMULATION_SERVER).
                              floors->DestroyFields();
              delete SINGLETON(ESCAPE_SIMULATION_SERVER).field_dll;
              SINGLETON(ESCAPE_SIMULATION_SERVER).field_dll =
                              new Dlls();
        }
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                        field_dll->ImportDLL((char *)msg-
                    >filename);
        buffer = (void *)new char[msg->bytes];
        memcpy(buffer,&msg->buffer,msg->bytes);
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                        floors->AddFields(buffer);
        GENERAL density_field_msg;
```

```
        density_field_msg.type = DENSITY_FIELD_DLL_LOADED;
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                          sendmsg((void*)&density_field_msg,
                          sizeof(GENERAL));
        delete [] buffer;
        break;
        }
case DYNAMIC_DATA_DLL_LOAD:
{
        DYNAMIC_DLL_LOAD *msg = (DYNAMIC_DLL_LOAD *)
                          pMsg->pReceiveData;
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                          dlls->ImportDLL((char *)msg->buffer);
        break;
}
case DYNAMIC_DATA_OBJECT_CREATION:
{
        DYNAMIC_OBJECT_CREATION *msg =
                          (DYNAMIC_OBJECT_CREATION *)pMsg
                          ->pReceiveData;
        SINGLETON(ESCAPE_SIMULATION_SERVER).floors
                          ->addObject(msg->typeID,(void *)&msg
                          ->buffer);
        break;
}
case RUN_SIMULATION:
{
        SINGLETON(ESCAPE_SIMULATION_SERVER).simulate = TRUE;
        break;
}
case STOP_SIMULATION:
{
        GENERAL stop_sim;
        SINGLETON(ESCAPE_SIMULATION_SERVER).simulate = FALSE;
        stop_sim.type = STOP_SIMULATION;
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                          sendmsg((void*)&stop_sim,
                          sizeof(GENERAL));
        break;
}
case CLOSE_CONNECTION:
{
        GENERAL close_confirm_msg;
        close_confirm_msg.type = CLOSE_CONNECTION_CONFIRM;
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                    sendmsg((void*)&close_confirm_msg,
                    sizeof(GENERAL));
        break;
}
case CLOSE_SESSION:
{
        GENERAL close_confirm_msg;
        close_confirm_msg.type = CLOSE_SESSION_CONFIRM;
        SINGLETON(ESCAPE_SIMULATION_SERVER).
                    sendmsg((void*)&close_confirm_msg,
                    sizeof(GENERAL));
        break;
```

```
        }
        case QUIT:
        {
                GENERAL quit_confirm_msg;
                quit_confirm_msg.type = QUIT_CONFIRM;
                SINGLETON(ESCAPE_SIMULATION_SERVER).
                            sendmsg((void*)&quit_confirm_msg,
                            sizeof(GENERAL));
                SINGLETON(ESCAPE_SIMULATION_SERVER).quitFlag = true;
                break;
        }
        default:
        {
                break;
        }
}
```

4.4 Simulation File

The actual simulation file format is actually a new addition to the Escape Simulator. It is not fully tested nor integrated at the moment. However, it has more or less been defined and is being used by the simulator. This section will cover the current format for the file and how it is integrating and will integrate with the Escape Simulation Suite.

4.4.1 Simulation File Format

This file is intended to be the main file that will define a given simulation. Within this file such things as object templates, DLL pluggins, world maps, settings, and the position and types of objects in a simulation are identified and defined.

The file itself is divided into three parts. The first is for general simulation parameters, the second for templates definitions and other world-specific things, and the last is for the actual object creation and placement. One thing to note is that this file is a fixed format binary file. This saves space and makes it simpler to read and write out. Also, there are more files than this one that make up a simulation. The other files will be covered at the end of this section.

4.4.1.1 General Simulation Parameters

- *SHORT simID*: This short is the simulation ID value. This is a very important number. It uniquely identifies and ties this file to the actual recorded simulations. A portion of this number changes anytime the simulation data is modified. In order to playback a recorded simulation the correct simulation file must be loaded up. Several runs of the same simulation file can be recorded and played back later. This allows the developer to play with different simulation parameters, such as step size, to get less or more detail from a given simulation trial. Things that would cause the simulation ID number to change are: changes to object templates, the addition of new objects, or other such changes to the world. These types of changes makes it impossible for the previously saved recordings to be played back under the

new settings. If the old recordings are desired then the old simulation file must be loaded.

- *SHORT graphicsmode*: This is the actual value of the graphics mode flags as of the end of the last session. Changes to this flag do not result in a change to the simID number.
- *SERVER IP,PORT,SESSION#*: These are the values to the last server that this simulation was run on.. When this file is first created these can be given a value. Otherwise, localhost and port 54321 will be assumed. If the server that this address-port-pair points to is not available when the simulation is started up, then the user will be given the option to specify a different address and port. The session ID is used to allow the client and server to be temporarily disconnected and then reconnected later. An example of when this might be useful is if a researcher were to set up a large simulation with a big world with many objects, and a very fine time step. Such a simulation may take the server awhile to simulate a decent time span. In this case, the researcher would setup the simulation, start the server simulating, and then disconnect. Later, perhaps in a few hours or days, he/she would reconnect to the server and check on the progress. The session ID would allow the client to reconnect to the last session it left running on the server. In this case, the server would not reload any of the objects or data. Instead, it would inform the client of the information needed to setup properly. Once the client was setup then the server would start sending updates on the data again. This functionality has not yet been implemented, but is intended to be in the near future.
- *WORD's number of types, static objects, dynamic objects, and pluggins*: Here we tell the simulation file reader how many of the following things to expect. This ensures that it reads the rest of the file correctly.

4.4.1.2 Template and World Definitions

- *BYTE PLUGIN, "filename"*: All the pluggins that the simulation needs in order to load up properly are given here. The first byte is a pluggin ID that the simulator will keep track of. For each pluggin there will also be a null terminated character array which tells the server where to go to load up the DLL. These DLLs, are not the same as the ones that are used by the object templates. These are to enhance the simulator's abilities. For example, a new math library could be written with special functions that the object pluggins will need to use. This is where that type of pluggin will be placed.
- *BYTE type,"typename","filename","DLL",ULONG fields*: The first number identifies which of the three template types this definition will fall under. Next is a null terminated character string that is used by the user in making modifications to objects. For example, after setting up the simulation, the user might decide to change which DLL is being used by a given object template. The internal identification numbers for the created templates would have little or no meaning for the user, so this identification is given to the user and will help him/her find the templates he/she needs. The filename null terminated string identifies either the .imap file needed by the mesh template type, the .bmp file needed by the point sprite template type, or the .fdat file needed by a

field template format. These files tell the template how to render properly. Finally, the DLL null terminated string identifies the pluggin that will be used server side by this particular template. This pluggin will define the algorithm that will be used every time-step to update an object's data; it will also define how an object is to be updated. When the simulation file reads in this field it requests that the server load up the specified DLL. When the simulation file is created, it will query the DLL for the field's flags value. This field tells the client what fields will be updated on update messages to objects of that template. Note: Like the pluggins above, there can be multiple sets of these fields—one set per defined template.

- *"map name"*: This is the actual floor plan that the client will display and the server will use in its calculations. The map name field is a null terminated string. It actually identifies two files at once. The first is the "map name".imap file. This file contains the world map data in a format that has been optimized for the graphics engine. For example, there is only one wall object in this file. Since walls are static in this world they have been grouped together into one indexed vertex list. This significantly speeds up the load-up time of the world and the overall rendering. Such an optimization does not work for the server side because, in order to do collision detection and other such calculations, the server must know about each wall individually. Therefore, the server uses a "map name".vmap file. This file has all the information that the server needs to be able to do all the simulation calculations about the world.
- *Double posX, posY, posZ, targetX, targetY, targetZ*: This is used to save the last camera position and target values. Changes to these values do not result in a change of the simID value.

4.4.1.3 Object Creation and Placement

In this section, the actual objects in the world are defined. Usually, the static objects will be defined first and the dynamic ones second. Each object will have the following fields:

- *WORD typeID*: The first field tells which template the object will be an instance of. Note that this ID is assigned to the templates as they are created. Any program that generates this file needs to take this into account and keep track of these IDs, and when it writes out this part, it needs to put the corresponding value here.
- *FLOAT posX,posY,posZ*: This is the x,y,z position of the object in 3D space.
- *WORD floorID*: This assigns the particular object to a specific floor.
- *STARTING VALUES fields*: Depending on what fields are specified in the template fields field, this section will vary in size. Basically, all the fields flags set in the field need to be given starting values, whether the object is static or not.

4.4.2 Current and Future Integration

As stated above, at the moment the Simulation file format is defined and should not change drastically in the future. As far as implementation and integration go, the

simulation file is read by the state machine's load simulation state. This state handler has been written and the majority of its functionality is there and working. There are some fields, like the client/server session field, that are not being used. This is because the actual code that manages saving and restoring a session has not been implemented. At the moment, the complete implementation and integration of the Simulation file with the Escape Simulator is about 50-60% complete.

This is one area that is on top of the list of things to get done, and should be finished in the next month or two.

4.4.3 Secondary File formats

There are four other file formats that have been defined and are used by the simulation file. The simulation file itself has a .sim file extension. The other four files have .imap, .vmap, .bmp, and .fdat extensions.

4.4.3.1 .imap File

Like the .sim file, the .imap file is a fixed format binary file. It is used by both client side world definitions and object mesh definitions. At the beginning of the file there is an identifier that tells whether the file contains a world or an object mesh. When the format is a world, there can be multiple floors that can be defined. When the format is for an object mesh, there is only one floor defined. Here are the fields, listed in order starting with the world format first and followed by the object format.

Floor Mesh:

- *Version number*: This number tells the .imap file reader the version the file was written in. This is to ensure that the files are read correctly.
- *Number of Floors*: The number of floors that will be defined below. Each of the fields that follow will be repeated for each floor.
- *Name*: Human readable null terminated string that is used whenever the human wants to reference this floor.
- *Size*: The width and length of the floor, in that order.
- *Level*: The level the floor is on. This allows for multiple floors on the same level.
- *Number of Vertices*: The number of vertices that will immediately follow this value.
- *<vertex-1x> <vertex-1y> <vertex-1z> <vertex-1textcoorx> <vertex-1textcoory>*: Each of the floor vertices will have this format.
- *Number of Indices*: The number of indices that will immediately follow this value.
- *<indext1-1> <indext1-2> <indext1-3>*: Each triplet defines a different triangle face on the overall mesh.

Object Mesh:

- *Version number*: This number serves the same purpose as the corresponding field in the floor mesh.
- *Vertex type*: This is an important number because there are many possible types of Direct3D vertices formats that can be put together to fit the programmer's needs. This number is created by ORing the desired vertex

field flags together to tell Direct3D and the .imap reader which fields to expect.

- *Number of Vertices*: The number of vertices to expect immediately following this value.
- *FORMAT BASED ON VERTEX TYPE FLAGS*: This is similar to the formatting above, but which fields and the order of the fields will be based on which flags are set in the Vertex type field.
- *Number of Indices*: The number of indices that will immediately follow this value.
- <*indext1-1*> <*indext1-2*> <*index1-3*>: Same format and interpretation as the world format.

4.4.3.2 .vmap File

This file format is text based, though it will probably be moved to a binary format when ESPaint is finished. It defines the world information required by the server to run properly. The format is fixed in the sense that the tags and values given must be included and given in the order in which they appear. The tags are very self explanatory so the format will be shown here without explanation.

```
VMap File Format - Floor data:
.
.
.
OBJECT

HEADER
TYPE <type>
NAME <name>
WIDTH <width>
HEIGHT <height>
NUMWALLS <number walls>
ENDHEADER

WALL
START <x> <y>
FINISH <x> <y>
ENDWALL

.
.
.

WALL
START <x> <y>
FINISH <x> <y>
ENDWALL

ENDOBJECT
.
.
.
```

4.4.3.3 .bmp File

This is nothing more than a normal windows bitmap file. Point sprites only require a position and a single bitmap to render correctly. Please refer to [14] for an explanation of how to implement sprites.

4.4.3.4 .fdat File

This file holds the raw data needed to initialize a density field. Not all simulations will have moving dynamic objects, but every simulation will have at least one density field per floor. The format for this file is fixed binary, and is as follows:

- *ULONG row*: Number of rows
- *ULONG col*: Number of columns
- *ULONG data…*: Based on the dimensions given in the first two fields, there will be row*col fields here. Each will be an unsigned long in size and will represent one entry in the density field.

4.4.4 ESPaint

Creating the simulation files and secondary files from scratch is a tedious task, especially now that two of the three file formats are fixed binary. What is really needed by the researchers is a nice GUI based program that will allow them to draw the world in a manner similar to Microsoft Paint. Along with drawing the world, this application would allow them to define object templates and graphically add objects of the defined templates to the world. Finally, the program would generate the .sim and world .imap and .vmap files. (The object .imap files will also be generated, but by pluggins that will interprate .x and other well know mesh file formats which can be generated by many free 3D drawing programs.)

This application is currently being development, and for the best information on its details, the reader should refer to documentation that is being created specifically for it. The information that follows is only high level.

This application will act as a permanent pluggin to the Escape Simulation Suite. It will be invoked whenever the user selects a new simulation from the main file menu.

When invoked, ESPaint will start by gathering information on: what DLLs are already installed in the server pluggin directory, what .vmap/.imap pairs are available (this is if the user would like to start with and modify a floor plan that is already created), and what object mesh .imap, point sprite .bmp, and field .fdat files exist and are available to be used. All this information will serve to flesh out selection menus. There will also be an option to allow the user to import and install new media and pluggins.

After the user has setup the simulation to their liking, it will generate the .sim and other files needed to run the simulation.

Finally, after a simulation has been created it can later be edited by ESPaint. Note: editing a simulation with ESPaint will change the simID number and will make this new .sim file invalid with any currently saved simulation runs.

All in all this will be a VERY useful tool for the researcher.

4.5 World Engine

In its structure and design, the world engine is very similar to the client's graphics engine. Both have a hierarchical structure that starts with the WORLD/D3D_GRAPHICS class and then moves down to the FLOORS class -> FLOOR class -> Objects class -> Object class. Both the Floors and Objects classes are linked lists of Floor and Object classes respectively. If you look at the UML diagrams for both the WORLD and D3D_GRAPHICS classes they will look almost identical. The main differences are as follows:

- *No Direct3D*: This is an obvious one, but the implication might not be immediately apparent. First off, this means that there are no object templates, no template types, and no mesh/point sprite/field object classes. There is just one Object class that is identical to the D3D_GRAPHIC version, except that all the Direct3D code is removed. Instead of having render() and ReadRawData() methods, this class has step() and dump() methods.
- *Object DLL hook code*: Instead of the graphics engine's templates, the world engine relies on pluggin DLLs to give the developer the ability to change the runtime behavior of the objects and fields in the engine. In fact, without any object pluggins the world engine cannot run. There is no hard coded object or field behavior. The world was designed to allow the researcher to be able to manipulate almost everything through the use of pluggins.

The details for the DLL pluggins will not be given here, but will be saved for the Object Pluggins section below. The topic is large enough to merit a section of its own.

The structure for this engine is in place and working. There may still be some minor changes in order to fix unexpected problems that may arise from the further development of the different pluggins.

4.6 Math Engine

The math engine is one of the most important components of the world engine. The basic engine is available to both the client and the server. The static portion of the engine provides solutions to such common fields of mathematics as geometry, trigonometry, and basic linear algebra. It was designed with speed and efficiency in mind. Some examples of its use include the input handlers for the graphics engine use, the library to calculate the change to the camera based on the users input, and the collision detection algorithm used by the object pluggins in the world engine. For a more in-depth description of the basic math engine and its API please see the documentation specific to this component

Along with the basic math engine that will be included with every compilation of the Escape Simulation Suite, it will be possible to extend the math engine with extra pluggins. These pluggins will be loaded in with the server when a simulation file is read in and requests the DLL. This aspect will allow future researchers to be able to add commonly used algorithms and functions to the default list of available calls.

The exact protocol for extending the math library with DLLs is a topic that is currently under development and should be decided upon within the next two to three months.

4.7 Object Pluggins

One of the most powerful aspects of the Escape Simulation Suite is the fact that all the world objects and fields can be completely redefined to suit a researcher's needs. In order to be able to get this kind of functionality the server uses dynamically linked libraries.

DLL generation was already covered to a decent degree in Chapter 2. Please refer to that section if something does not seem clear in the text that follows. In this section, the actual DLL pluggin hooks and code used by the world engine for the Escape Simulator will be covered. The goal is to help the researchers easily write their own pluggins by providing a simple example skeleton that they can follow and add their own code to..

4.7.1 Setting up the pluggin project

As stated many times in this thesis, the assumption has been made that the reader is using Visual Studio.net 2003 to build and compile their code. If some other IDE and/or compiler is used then the responsibility of setting up the project correctly, compiling, and linking falls on their shoulders.

Start by following the direction in the DLLs section of Chapter 2 on creating a Windows console DLL project. After the project is created, the following libraries must be linked in:

dplayx.lib dxguid.lib comctl32.lib winmm.lib odbc32.lib odbccp32.lib

Also, the latest version of ess.lib needs to be added to the linker input. In the future, this file will be changed to a DLL to reduce the size of the pluggins and the amount of repeated code that is dynamically linked in.

The latest version of the ess.lib file is required so that the pluggins will be compatible with the most current version of the Escape Simulator.

Depending on whether you have the entire Escape Simulation source tree or just the Escape Simulation Pluggin SDK, you need to make sure that the path to the Escape Simulation Server .h files is in your include path. It should be either ./ess/include for the entire application source and ./espSDK/include for the pluggins SDK. Finally, in your main pluggin file please add #include "ess.h". Doing all this will ensure that the correct .h files are included to allow your pluggin to function properly.

4.7.2 Coding the Pluggin

Add to the blank project two files. The first will be your DLL's .h file and the second will be the corresponding .cpp. For the sake of this tutorial please call these files test_person.h & .cpp.

Add to the .h the following code:

```
#ifndef TEST_PERSON_H
#define TEST_PERSON_H

#include "ess.h"

class Person : public Object
{
public:
	Person(void *buffer);
	~Person();

	virtual void step();
	virtual void dump(void *data);
	virtual unsigned long optionalDataFieldFlags();
	virtual void SetESSptr(void *ptr);

	void move(float _x,float _z,float _level,float _rad);

	ESCAPE_SIMULATION_SERVER *pESS;
};

#endif
```

Nothing here should be unexpected. The only thing unusual is the ESCAPE_SIMULATION_SERVER *pESS member. In order for the objects you develop to see and communicate with all other objects, they at least need to have access to the object linked lists in the world. Please note, however, that in this example we are using Object as the base class. If the developer wanted to write a DLL for a density field instead, almost everything would be the same as it is found in this example. The only differences are that the class would have to inherit from the Density_field class first, and that the requirements for the dump() function are relaxed a bit. The developer only has to give the out_data_requirement member a value. The optional_field_flags is not needed. For an example of this type of pluggin please refer to default_density_field.h and .cpp. These make up the default density field pluggin that is part of the Escape Simulation Server Application, and are included in the main application solution as well as the Escape Simulation Pluggin SDK samples.

Now add this code to the .cpp:

```
#include "test_person.h"

struct in_data          // This struct is used to case the input buffer
to the proper data type;
{
	unsigned int            typeID;
	unsigned int            id;
	float                   x;
	float                   z;
	float                   level;
	float                   rad;
};
```

```
struct out_data
{
      unsigned int            id;
      float                   x;
      float                   z;
      float                   level;
      float                   rad;
};

Person::Person(void *buffer)  // uint _type,uint objectID,float
_x,float _z,float _level,float _rad
{
      in_data *pd = (in_data *)buffer;
      id          = pd->id;
      out_data_requirement    = sizeof(out_data);
      optional_field_flags    = OFF_ID | OFF_X | OFF_Z |
                                OFF_LEVEL | OFF_RAD;
      x = pd->x;              // X location.
      Z = pd->z;              // Z location.
      level = pd->level;      // Which level is the Person on
      faceX = -1;             // The X value for the vector for the
                              // forward position
      faceZ = -1;             // The Y value for the vector for the
                              // forward position
      rad   = pd->rad;        // This angle is how far from the z-axsis
                              // the forward face of the Person is.
                              // This is used by the graphics engine
                              // for quicker rendering
      vX    = -1;             // The X velocity component
      vX    = -1;             // The Y velocity component
      aX    = -1;             // The X acceleration component
      aZ    = -1;             // The Y acceleration component
      typeID = pd->typeID;    // Type of Person.  This will allow the
                              // rendering engine to know how to render
                              // this Person
      next  = NULL;           // Next Person
      prev  = NULL;           // Previous Person
}

Person::~Person()
{
}

void Person::move(float _x,float _z,float _level,float _rad)
{
      x           = _x;
      z           = _z;
      level       = _level;
      rad         = _rad;
}

void Person::step()
{
      // Here is where all the AI calls are to be made

}
```

```
void Person::dump(void *data)
{
	out_data *od = (out_data *)data;
	od->id		= id;
	od->x		= (float)x;
	od->z		= (float)z;
	od->level	= (float)level;
	od->rad		= (float)rad;
}

void Person::SetESSptr(void *ptr)
{
	pESS = (ESCAPE_SIMULATION_SERVER *)ptr;
}
```

Two of the above method definitions are of major significance. The first is the step() method. This method is the workhorse of the algorithm. You can create whatever helper functions for this function that you need, but this is the method that will be called every time step, without fail. This is where you would put any kind of AI or other algorithm to model this object's behavior.

The second method is the dump() method. This is where the data that can change for this object is packaged up and sent to the server. The casting done in this method definition to the out_data structure type that is defined at the top of the file just serves to make life easier. When you write your own pluggin, you would modify the out_data structure to include the fields you want to send to the client. In order to get dump() to work correctly, there are three things that must be done besides writing the method definition. First, the out_data_requirement member of the Object base class must be given a value equal to the sizeof(out_data). It is recommended that the user leave this line alone in the constructor and only change the structure. Second, the option_field_flags member of the Object base class must be given the correct flags. A full list of the possible flags can be found in world_defs.h. In this case, we are sending the ID of the object, x, z, level, and rad values. These flags should all be ORed together and the result given to this member. When the simulation is created, your pluggin will be asked for this value in order to fill out the fields field in the simulation file for the object template your DLL is assigned to. Third, the order of the variables in the out_data structure is important. There is a set order that the data must be in. The variables must be in the same order as the corresponding flags in the world_defs.h file. For instance, the #define OFF_X has the value of 0x2 and comes before, and the #OFF_RAD has the value of 0x20 and comes after, so when the data is outputted, the x value must come before the rad value, whether the structure is used or not.

There are only three more files needed to get the pluggin ready. Create another .h and .cpp pair. These will always be the same for every pluggin you write, with the exception that the .cpp will need to include whatever the pluggin .h is (in this case it is test_person.h,), and the name of the object class will need to be changed in the .cpp. For this example please call these files dllLoader.h and c.cpp.

Again do not change these functions in the .h in any way. Doing so will make your pluggin useless.

Please add the following to dllLoader.h:

```
#ifndef DLLLOADER_H
#define DLLLOADER_H


#define EXPORT __declspec(dllexport)

typedef void *(*VoidPtrVoidPtr)(void *buffer);
typedef void      (*VoidVoidPtr)(void *ptr);

EXPORT void *Create(void *buffer);
EXPORT void Destroy(void *ptr);


#endif
```

and this to dllLoader.cpp:

```
#include "dllLoader.h"
#include "test_person.h"

void *Create(void *buffer)
{
      Person *person = new Person(buffer);
      return (void *)person;
}

void        Destroy(void *ptr)
{
      Person *person = (Person *)ptr;
      delete person;
}
```

Notice that our test_person.h file is included at the top of this file and the use of the Person class. When you write your own pluggins you will change these to be your own.

The last thing needed is to create a .def file for the DLLs. How to correctly do this and add it to your project has already been covered in Chapter 2. Please refer to that section for specifics. For this example, please call the file dllLoader.def and give it the following lines:

```
LIBRARY test_person

EXPORTS
      Create      @1
      Destroy     @2
```

There you have it. Compile this and add it to the simulation file and your pluggin should work. In order to make it do something specific, you will have to add some code to the step() function.

4.7.3 Object Pluggin Hooks

Now here is a quick look at the server side code that makes all the pluggins work.

The following Dlls class has the responsibility for keeping track of all the DLLs loaded for a given world. The prototype is found in the dll.h file, and the definition in the .cpp of the same name. Here is the prototype:

```
#define MAX_NUM_DLLS    256

class Dlls
{
public:
      Dlls();
      ~Dlls();

      int ImportDLL(char *dllName);
      void ReleaseDLL(unsigned int index);

      void *CreateObject(unsigned int index,void *buffer);
      void DestroyObject(unsigned int index,void *ptr);

      int NumLoadedDlls();

protected:
private:

      HMODULE                 dlls[MAX_NUM_DLLS];
      VoidPtrVoidPtr    createPtrs[MAX_NUM_DLLS];
      VoidVoidPtr       destroyPtrs[MAX_NUM_DLLS];
      int                     loadedDlls;
};
```

Note that there can only be 256 DLLs per Dlls class. This is because there are currently only 256 templates allowed to be defined within the client. This is fairly straightforward. Notice that each Dlls class is assigned an index. When it is created, this index is associated with a given template. When that template is used to create an object its ID is used to index the appropriate CreateObject() method. Later, when the same object is being destroyed, the corresponding DestroyObject() method is used to destroy the object.

Here is the code for the .cpp:

```
Dlls::Dlls()
{
      int i;
      loadedDlls = 0;
      for(i = 0;i < MAX_NUM_DLLS; i++)
      {
            this->dlls[i] = NULL;
```

```
            this->createPtrs[i] = NULL;
            this->destroyPtrs[i] = NULL;
      }
}

Dlls::~Dlls()
{
      int i;
      for(i = 0;i < MAX_NUM_DLLS; i++)
      {
            Dlls::ReleaseDLL(i);
            this->createPtrs[i] = NULL;
            this->destroyPtrs[i] = NULL;
      }
}

int Dlls::ImportDLL(char *dllName)
{
      if(loadedDlls == MAX_NUM_DLLS)
      {
            // No more dlls can be loaded
            return -1;
      }
      dlls[loadedDlls] = LoadLibrary(dllName);
      if(dlls[loadedDlls])
      {
            createPtrs[loadedDlls] =
(VoidPtrVoidPtr)GetProcAddress(dlls[loadedDlls],"Create");
            destroyPtrs[loadedDlls] =
(VoidVoidPtr)GetProcAddress(dlls[loadedDlls],"Destroy");

            if(!createPtrs[loadedDlls] || !destroyPtrs[loadedDlls])
            {
                  // Invalid Library
                  FreeLibrary(dlls[loadedDlls]);
                  return -1;
            }
            loadedDlls++;
      }
      else
      {
            // Bad dll name
            return -1;
      }
      // Dll loaded correctly
      return (loadedDlls - 1);
}

void Dlls::ReleaseDLL(unsigned int index)
{
      if(dlls[index])
      {
            FreeLibrary(dlls[index]);
            dlls[index] = NULL;
      }
      loadedDlls--;
}
```

```
void *Dlls::CreateObject(unsigned int index,void *buffer)
{
	if(createPtrs[index])
		return (*createPtrs[index])(buffer); // Call the correct
creation function
	return NULL;
}

void Dlls::DestroyObject(unsigned int index,void *ptr)
{
	if(destroyPtrs[index])
		(*destroyPtrs[index])(ptr); // Call the correct destroy
function
}

int Dlls::NumLoadedDlls()
{
	return loadedDlls;
}
```

If any of this code looks strange please refer to the Chapter 2 section on DLLs.

The CreateObject() returns a pointer to the class in your pluggin that inherited from the base class. The "Object" part of the name is somewhat misleading. Remember that the prototypes for Create and Destroy in dllLoader.h return and take void * instead of Object *. This means that any base class can be used here as long as dllLoader cast it properly, the pluggin class inherits from it, and the method that calls Dll::CreateObject() and Dll::DestroyObject() casts the pointers to the correct values. In the future, the name of these two methods may be changed to reflect this point. From the pluggin developer's standpoint, as long as he/she has the latest .lib file and .h files they will not notice any difference.

CHAPTER 5: Conclusion

After all is said and done the real question is, “Did it work?”

5.1 Achievement of Goals

The goal of this thesis was to create a tool to allow researchers who are developing dynamic evacuation systems to test and run experiments on their hypotheses and ideas. This goal was met.

The Escape Simulation Suite is in its Alpha stage, meaning that it is ready for researchers to start writing pluggins and testing it.

To reiterate, the main goal of the Escape Simulation Suite is not to solve the evacuation problem, but rather to provide a tool that researchers can use to test their theories quickly. This tool must be both flexible and easy to use. That being the case…is it?

The use of pluggins has allowed the application to be very adaptable. Adding a new object to a simulation is simply matter of adding a new pluggins with desired behavior defined in it. Changing the behavior of an existing object just requires changing the objects pluggin and recompiling. The pluggins themselves have been designed to be easy to write. By using the pluggin SDK a researcher follows three steps:

1. Add algorithm to the step method
2. Add the correct flags to the optional fields flag variable
3. Compile

Along with the pluggin capability which achieves the flexibility goal, there are many other goals that this application has met. They are as follows:

- The application is made up of a client and server that allow for a clean division of labor between two computers.
- The client has a powerful, fast, and extendable graphic engine.
- The client has a clean overall design that will allow for easy modifications and improvements in the future as well as good performance at runtime.
- The server manages the world information in an extendable data structure that will allow for easy improvement in the future, as well as quick execution and management at runtime.
- Both the client and server communicate through a reliable and quick interface to help improve throughput and bandwidth.
- The application as a whole is Alpha ready as of the submission of this thesis.

5.2 Application’s overall performance

At the time of this writing, the application is in its alpha version. The following is the current state of the different components:

- *Graphics Engine*: The 3D engine is working well. It can display meshes and other graphics well, and it can load up files and keep track of the world from a graphical point of view. The 3rd person view is working fine and the 1st person view is in the works.
- *Network Engine*: The network engine can make a connection and send data properly. That being said, DirectPlay was not necessarily the correct choice as far as API's. In the future, it would be interesting to swap this component with one using Windows sockets.
- *Menus & UI*: There are several changes that will be made to the user interface in the near future. At the moment it is still in a very basic format.
- *IO*: All the basic IO features are functioning correctly.
- *Simulation File*: The simulation file has been defined and is being integrated with the application.
- *World Engine*: The structure for this engine is in place and working. There may still be some minor changes made to allow for internal helper functions that the researcher will be able to use to get commonly executed tasks done. Eventually, there will be further pluggin support for more than just the objects and fields.
- *Math Engine*: This engine is in a stable and functional state offering most of the basic geometric and trigonometric functions, as well as some basic linear algebraic functions for working with matrices and vectors. In the future a calculus and differential equations portion along with DLL pluggin support will be added.
- *Pluggins*: The DLL hooks have been added to the application and pluggins can be written and used for the objects and fields.

Here are a few screen shots of the running application:

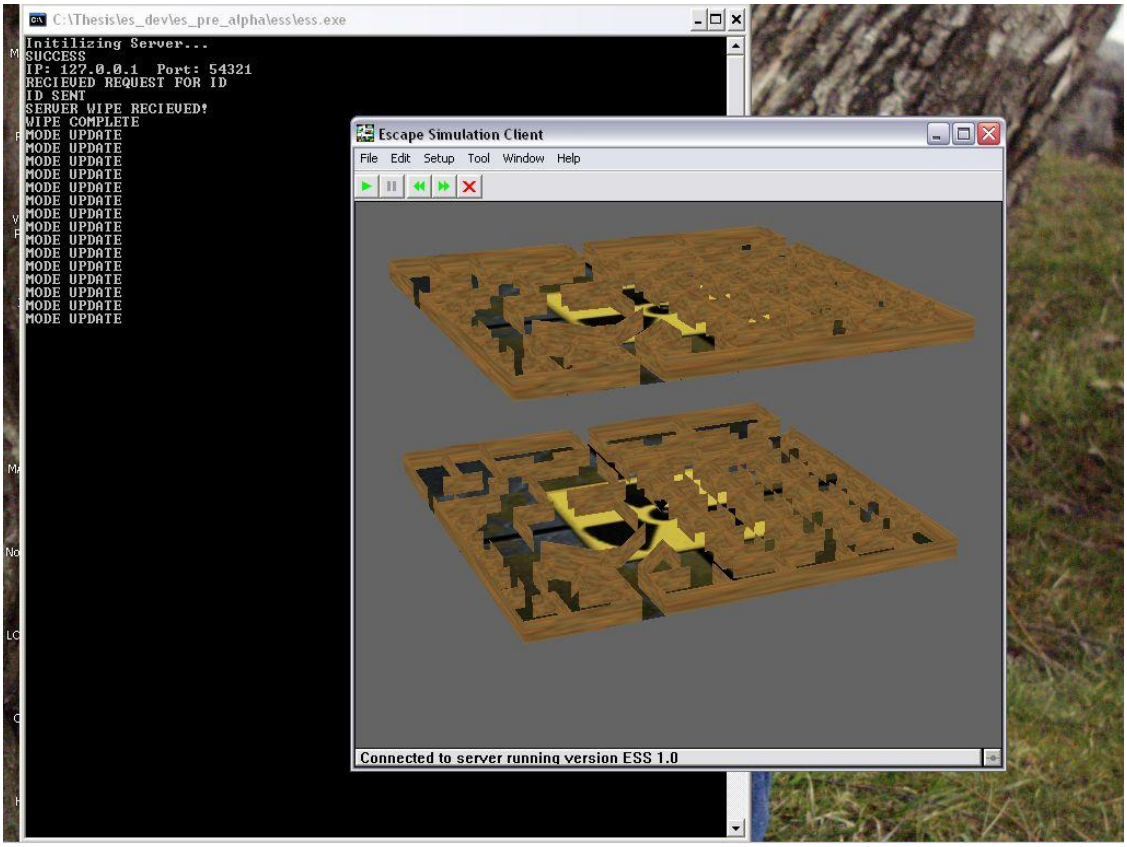

Figure 22 This is a screenshot of the Escape Simulation client and server running. In this case, a three story building (only two floors visible) was loaded up and a single agent object was added to the middle. The current rendering is set to microscopic.

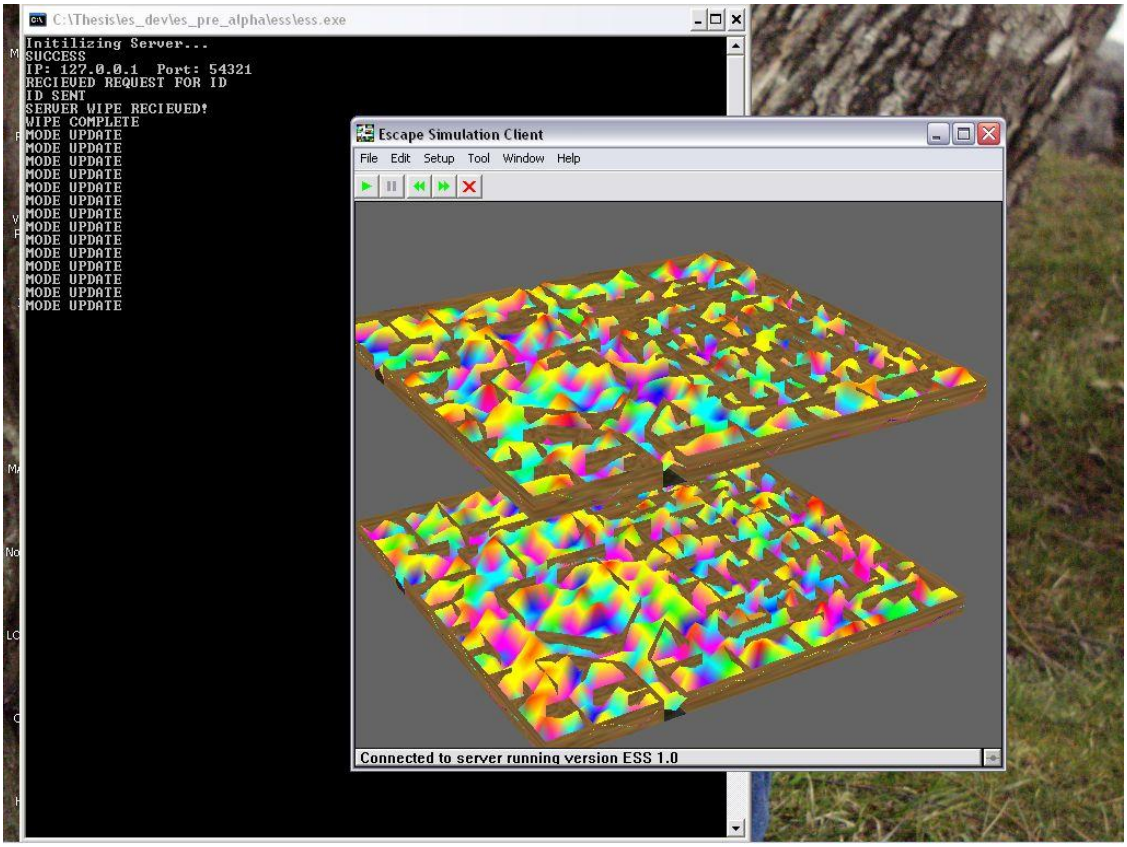

Figure 23 This is the Escape simulator running the same setup as in the last screen shot, but this time it generated random density fields for each of the floors. Also, the rendering is set to macroscopic

There is still a lot to be done. Even when it is to a first run production level it will have possible areas for improvement.

For the near future, the most important improvements, in no particular order, are:

- *Simulation Recording & Playback*: This is a key feature that must be in the final version.
- *Simulation file generator*: In order to generate the simulation files, some kind of GUI wizard would be VERY helpful.
- *World Generator*: It would be very nice to have an application that allowed researchers to easily generate worlds in a nice graphical environment.
- *Wider Range of Pluggin Support*: Adding support for math engine, world, and general application pluggins.

5.3 Where to go from here?

There will be, of course, researchers that will be developing pluggins with their own objects and specifications from here on out. It is the hope of the author that in the future there will be other graduate students that will have the desire to expand and improve the

main application. Again, as stated above, there is much room for improvement. However, for the most part, it looks like the simulator will be a success and a useful tool for future researchers.

CHAPTER 6: APPENDIX

6.1 UML and Secondary Components/ APIs Documents

The actual structural UML documents were created with Microsoft Visio and are rather large. These documents are included in the thesis_UML.zip file that was submitted along with this thesis. Also, there are several APIs and/or components that have been developed by the undergraduate researchers mentioned in the Acknowledgements section. There have been separate documents created describing the specifics of these APIs or components.

These secondary documents, the UML documents for the Escape Simulation Suite, the thesis, and all the code are included on the Escape Simulation Suite homepage. http://www.ee.vt.edu/~pushkin/evacuation/

The files found in the thesis_UML.zip file are:

- *Escape Simulation Client.vsd*: UML document of the Escape Simulation Client. This document only contains a structural diagram of the application.
- *Escape Simulation Server.vsd*: UML document of the Escape Simulation Server. Like the Client's document, this only contains a structural diagramming for the application.

The secondary documents found on the Escape Simulation Suite website are:

- *Math Engine API.doc*: Document explaining the Math library's API, including examples. This document was written and maintained by Greg Dean.
- *Simulation and Data file IO API.doc*: Document explaining the file parsers and file formats for all the Simulation and Secondary files used by the Escape Simulator. This document was written and maintained by Ryan Wilson.
- *ESPaint.doc*: Document describing how to use ESPaint. This document was written and maintained by ESPaint's author Greg Dean.

6.2 Sample Code

Due to the size of the Escape Simulation Suite, and the number of sample applications that make up the research of this thesis, the actual source code cannot be included in this document. Instead, all the source code for each application mentioned in the preceding chapters, including the full source code for the Escape Simulation Suite, is included in the zip file: thesis_full_source_code.zip. Each application is stored in its own individual directory and includes the solution and project files needed to compile and run the applications.

One important note: In order to run any of the applications that use DirectX, you will need to have the latest version of the DirectX runtime libraries installed on your computer. And, in order to build any of the same projects you will need the entire DirectX SDK installed.

The included applications in the zip file are as follows:

- *Project Ant*: Basic Win32 project that contains three actual applications. These were intended to test the basic Win32 GDI drawing.
- *Pinball*: This application is similar to the Poles and zeros application in the math that is being done. However, in this example each particle is repulsed by all other particles.
- *Poles and Zeros*: This was the next evolution after Pinball. The particles themselves have no effect on each other, but the user can place poles and zeros which either attract or repel the particles.
- *Fish Pond with Threads*: This is the Win32 based fish pond simulator mentioned above. Along with being written using Win32 GDI calls, it is also an example of a multithreaded application.
- *Children Timers, Buttons, and Things*: Though this application is not mentioned above, it is a fun little application that creates a large 2D bank of buttons and uses message calls to update the buttons' statuses. This application was a test of Win32 parent-child window communication. The end result is that the buttons toggle themselves in a way that makes them appear to be counting in binary.
- *Space Flight*: This application was written completely with 2D DirectDraw calls. This was the first attempt made at DirectX programming, and for the most part, it worked out well, especially for only about four days of programming.
- *Simulator First Run*: This application was written using a combination of Win32 and DirectDraw. It was the first hybrid application written. It also uses an algorithm similar to those that will be used by researchers in the actual Escape Simulator for macroscopic modeling. A detailed explanation of this algorithm is given in the "Initial Modeling and Controlling of Evacuation Grid.doc" file included in this project directory.
- *TCP Chat*: This is a simple chat application that was written using the TCP/IP library developed in the early stages of this project.
- *DirectPlay Chat*: This was written to be identical to TCP chat, except that it uses DirectPlay to get the job done.
- *Resources In Use*: This is the sample tutorial on Win32 resources that was used in the above chapters.
- *Escape Simulation Pluggins SDK*: This is the first generation of the SDK that will be given to the researchers for developing their own pluggins for the Escape Simulator.
- *Escape Simulator*: This is the full application's current state as of April 1, 2005. The application's state is just barely pre-alpha. It is within about a week or two of being ready for researchers to actually start coding and testing pluggins. That being said, this version of the source code has already become outdated, and will continue to at a rapid pace over the rest of the semester. This is due to the constant work being done to improve the core application and its peripherals. For the latest version of the source code please see the Escape Simulation website included in this document. It is available in a single zip download and also as a CVS repository.

CHAPTER 7: References

[1] S. Gwynne, E. R. Galea, M. Owen, P. J. Lawrence, L. Filippidis "A review of the methodologies used in the computer simulation of evacuation from the built environment," *Building and Environment (34)*, pp. 741-749, 1999

[2] Brian W. Kernighan and Dennis M. Ritchie "The C Programming Language," *2nd edition*, Prentice-Hall,1988

[3] Deitel and Deitel "How to Program C++," *3rd edition*, Prentice-Hall, 2001

[4] Charles Petzold "Programming Windows: The definitive guide to the Win32 API" *5th edition*, Microsoft Press, 1999

[5] André LaMothe "Tricks of the Windows Game Programming Gurus," *2nd edition*, Sams Publishing, 2002

[6] André LaMothe "Tricks of the 3D Game Programming Gurus: Advanced 3D Graphics and Rasterization," Sams Publishing, 2003

[7] Wolfgang F. Engel "Beginning Direct3D Game Programming," *2nd edition*, Premier Press, 2003

[8] Douglas E. Comer and David L. Stevens "Internetworking with TCP/IP: Client-Server Programming and Applications Windows Sockets Version," *3rd vol.*, Prentice-Hall, 1997

[9] Microsoft "MSDN," *Windows API*, Microsoft, 2003

[10] Microsoft "MSDN," *DirectX*, Microsoft, 2003

[11] Microsoft "MSDN," *Direct3D*, Microsoft, 2003

[12] Microsoft "MSDN," *DirectPlay*, Microsoft, 2003

[13] Microsoft "MSDN," *Windows Sockets*, Microsoft, 2003

[14] Microsoft "MSDN," *Point Sprits*, Microsoft, 2003

[15] Microsoft "MSDN," *Vertex Buffers*, Microsoft, 2003

[16] Microsoft "MSDN," *Index Buffers*, Microsoft, 2003

[17] Microsoft "MSDN," *Resources*, Microsoft, 2003

[18] Microsoft "MSDN," *dllexport, dlimport*, Microsoft, 2003

[19] Microsoft "MSDN," *Dll Export and Import Functions*, Microsoft, 2003

[20] Microsoft "MSDN," Microsoft, 2003

[21] Thomas Y. Merrell, Sadeq J. Al-Nasur, and Nirvana Mehan "Initial Modeling and Controlling of Evacuation Grid", 2004