COOPERATIVE ROBOTICS ORGANIZATION SIMULATOR
USER INTERFACE AND DISPLAY MODULE

by

SCOTT DALE GILLAN

B.S., Kansas State University, 2003

A REPORT

Submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2005

Approved By:

Major Professor
Scott A. DeLoach, Ph.D.
ABSTRACT

The Cooperative Robotics Organization Simulator (CROS) allows users to test robot organization algorithms in a controlled environment. This simulator and its accompanying Organization Model was designed and implemented by MS student, Christopher Zhong. The simulator has a small, sparsely featured display and user interface. The goal of my project is to provide a richer control and display scheme. In addition to better graphics and user control, the display engine needs to be decoupled from the simulator as much as possible. The ORG-DISPLAY uses the G Graphics Engine to display icon representations of each agent or object on a battlefield. User may start, stop, pause, and unpause the simulator from the control menu. A built-in Agent List provides the status of all agents at the click of the mouse. The refresh rate is user controlled and can be set to update every view minutes (in the case of batch jobs). A configuration file system customizes the display appearance for each user and application.
# Table of Contents

Table of Figures.............................................................................................................................. iv

Chapter 1: Introduction ................................................................................................................... 1

Chapter 2: Background.................................................................................................................... 3
  2.1: CROS ................................................................................................................................... 3
  2.2: Literature Review................................................................................................................. 4

Chapter 3: Features.......................................................................................................................... 6
  3.1: Menu ................................................................................................................................... 7
  3.2: Battle-Map ......................................................................................................................... 11
  3.3: Configuration File.............................................................................................................. 12

Chapter 4: Requirements............................................................................................................... 12

Chapter 5: Design.......................................................................................................................... 15
  5.1: The G Graphics Engine...................................................................................................... 15
  5.2: Classes................................................................................................................................ 17
    5.2.1: SimDisplay ................................................................................................................. 17
    5.2.2: AutoRunner ................................................................................................................ 19
    5.2.3: GPanel ........................................................................................................................ 20
    5.2.4: AgentGObject............................................................................................................. 20
  5.3 Objects ................................................................................................................................ 20
    5.3.1: Display Update Sequence........................................................................................... 23
    5.3.2: Highlighting Sequence................................................................................................ 24

Chapter 6: Future Additions .......................................................................................................... 26
  6.1 Agent Information via Battle-Map Mouse Interaction........................................................ 26
  6.2: Dynamic Agent Information Alteration:............................................................................ 27
  6.3: Multi-Agent Highlighting .................................................................................................. 28
  6.4: Scalable Images ................................................................................................................. 28
  6.5: Remote Display.................................................................................................................. 29

Chapter 7: Conclusions ................................................................................................................. 30

Chapter 8: References ................................................................................................................... 31

Chapter 9: Appendices .................................................................................................................. 32
  Appendix A: Display Configuration File Users Manual.......................................................... 32
Table of Figures

Figure 1: Old Display .......................................................................................................... 2
Figure 2: DataDisplay Object Diagram ............................................................................. 5
Figure 3: Standard Display View ...................................................................................... 7
Figure 4: Sleep Time Dialog ............................................................................................ 8
Figure 5: Highlighted Agent ............................................................................................. 11
Figure 6: Class Diagrams ................................................................................................. 17
Figure 7: Object Diagrams ............................................................................................... 21
Figure 8: Display Update Sequence Diagram ................................................................... 24
Figure 9: Highlight Sequence Diagram ............................................................................ 25
Chapter 1: Introduction

The Co-Operative Robotics Organization Simulator (CROS) is a high-level system designed to study different properties and methods of dynamic agent (robot) organizing. The simulator and accompanying Organization Model provides researchers an inclusive test-bed for examining reorganization algorithms and various team-based robot tasks [4]. The simulator and model was designed and maintained by Christopher Zhong, a CS Masters student.

Mr. Zhong originally included a primitive display function to give users a glimpse into the current state of simulator. Individual agents are represented as single pixel blocks on a white background. <Figure 1>. The user may pause or unpause the simulator, and speed up or slow down the simulator. All agents (regardless of type) are the same color, with non-agent objects as another color. This display was adequate for the simulator’s development, but lacked the features to make a pleasant presentation, especially to individuals unfamiliar with the technology.

My project is to design and implement a better display module. Better in terms of both extra features and customization. The display needs to be separable from the rest of simulator and extendable in an object-orientated capacity.

The CROS-DISPLAY module uses configuration scripts to customize the visual scheme of the display to the specific application. Each agent or display object may be given a unique image or polygon color to differentiate itself from the others on the battle map. The user control interface provides users with all the controls of original display, along with the ability turn off the auto-refresh. The main control interface also houses a dynamic list of all the agents in the simulator. From the list, agents can be highlighted within the battle-map, or have their current statistics displayed.
This report will discuss all the features of the CROS-DISPLAY module, along with the design considerations pertaining to each component. It will also provide a user’s manual and documentation to allow further development.
Chapter 2: Background

2.1: CROS

The Co-Operative Robotics Organization Simulator and Organizational Model provides a sophisticated framework for testing algorithms and applications of multi-agent optimization and structuring. The model defines the agent organization structure, along with the system goals, while the simulator computes the results of the experiments in pseudo real-time [4].

The generic organization structure is roughly laid out as this: a team of agents works together to accomplish a series of goals. Each agent has a number of various capabilities. The capabilities define what actions the agent may perform. These capabilities can range from communication sensors and transmitters to actuators. The software agents are generally limited to activities that physical robots might be able to perform.

To accomplish a goal, an agent is assigned to it. Each goal requires the use of certain abilities to ensure success. The required capabilities form the basis of the role structure. Each goal has a list of required roles. Each role has a list of required capabilities. An agent with all the necessary capabilities may choose to take the role. Certain agents are better at playing certain roles than others. This is where the optimization aspect plays a part. What is the best combination of agent-to-role assignments that best completes the goals? Furthermore, the goals are structured in a hierarchal goal-tree. Some goals have offspring goals. These offspring need to be accomplished first to achieve the parent goal. The relationship between the parent and children may be conjunctive or disjunctive. Certain goals may be prioritized.

The model also needs to handle dynamic re-organization of the agents. For example, let an agent become disabled. The system needs to juggle the role assignments to compensate for the lost agent. If a complete solution cannot be found, then this must be
identified. The model and simulator are used to test different solutions to the re-
organization problem.

The simulator takes the agent structure created by the model and tests it out. The
simulator itself has no knowledge of the organization structure. Each agent gets a turn to
choose its next move. The simulator performs the move and reports the result. In this
way, the simulator cannot judge the success of the organization, only the model may.

The simulator’s other task is to gather information about the current state of the
environment and give this information to the display. Every time an agent moves or
takes a significant action, the state of the agent is copied into a DataDisplay object
<Figure 2>. The DD object holds the current location of the agent and the values of all
the fields and capabilities. The simulator places the DD Object into a hashmap structure
called the Change-List. A hashmap uses a key to store and sort the various DisplayData
objects [1]. Each agent and non-agent object is given a unique identifier number when
created. All agents have an id number less than zero, while all non-agent objects have an
id greater than zero. Every time an agent makes a move, a DisplayData object is created
and inserted into Change-List with the agent’s id as the key value. If the agent already
had a DD object within the Change-list, the old object is overwritten with the new
information. With this mechanism, there is at most only one DD object per agent in the
Change-List. Thus, the Change-list represents only the most current state of the
simulator.

2.2: Literature Review

Using software to simulate robot or agent behavior is not unique to CROS. With the
rising sophistication (and cost) of modern robots, software provides an ideal and cost-
effective solution. There are many different robot simulators available to researchers.
Each of these programs usually focus on one specific domain. Some are specialized
towards a single type of robot, with the same programming interface. Others simulate
just single aspect of agent programming.
A graphical interface is the common ingredient for most robot simulators. Since the systems purposely lack a physical device, a simulator needs to some method of conveying movement and action to the user. The best solution is a visual representation of the robot and its environment, usually implemented as a graphical user interface (GUI).

RoboCode [5] is used as an AI and introductory programming test bed. Users program their individual agents and pit them each other in virtual combat. As with the CROS display, RoboCode users have no direct control of agent once the simulation has started. They are free to pause the simulator and log statistics.
RoboSim [6] is a multi-agent simulator, not unlike CROS. However, RoboSim works at a lower level of abstraction than CROS. CROS is designed specifically for reorganization algorithms and applications, while RoboSim investigates the mechanisms of robot interaction. RoboSim’s elegant GUI can display either 2D or 3D graphics.

The Kehepera Simulator [7] provides a virtual work environment for the Keherera Robot [8]. The simulator is designed to understand the same commands as the physical robot. A program developed within the simulator should behave identically outside the simulator. The simulator provides an Environment editor and a 2D GUI. Because this is a specialized simulator, there is little visual customization allowed. It is supposed to look and act exactly as physical robot.

Rossum’s Playhouse [9] uses a simple GUI to aid in robot control and navigation logic. Its applications are mostly limited to maze-mapping and other search algorithms. Unlike CROS, agent code for the simulator can often be injected into a physical robot. Its code controls the actual movements.

**Chapter 3: Features**

The CROS: DISPLAY utilizes two main windows for all features. I will call them the menu window and the battle-map window. The menu window holds most of the user control interface devices while the battle-map is graphical representation of the simulator. This section will discuss each feature in both windows, along with the customizable configuration file system.
3.1: Menu

The menu window is the main frame. Built from a traditional JFrame, the menu holds a number of user buttons that control the various functions. In the current implementation, closing the menu window terminates the entire program, including both the organization and the model.

The top three buttons, marked ‘-‘, ‘+‘, and ‘x Sleep Time’, control the simulator cool down timer, also called the ‘sleep timer’. This value is time between turns within the simulator. After each turn, the simulator waits until a timeout and then begins again. The sleep time is measured in milliseconds. Increasing the sleep value slows down the
execution time of the simulator. The starting value is 1 millisecond. A sleep time of zero makes the simulator run as fast as possible.

Figure 4: Sleep Time Dialog

The ‘-‘ button decreases the sleep time by one unit. It is impossible to decrease below zero. The ‘+‘ button increases the sleep time by one unit. There is no maximum limit. The third button serves two purposes. First, it displays the current sleep value. The general label is ‘x Sleep Time’, where x is the current value. Every time the value changes, by the ‘-‘ or ‘+‘ buttons, the label changes to reflect the new value. The button can also set the time to any valid value. If the user wants to change the sleep time from 10 to 100, instead of clicking the ‘+‘ button 90 times, the ‘x sleep button’ allows the user to input the desired value. Clicking the button brings up a data-entry dialog. The user may then type the number into the textbox. If this value is valid, then the sleep time is
set. If the value is invalid (a negative number, or not a number), then the dialog reports the error to the user.

Below the sleep time controls is the pause/unpause button. This single button tells the environment to either pause or unpause the simulator. Pausing the simulator halts all agent activity. No turns are executed; no information in the change-list is updated. The unpause signal turns everything back on. The button’s label reflects the next command. If the simulator is paused, then the label reads ‘unpause’, and vice versa. Clicking the button switches the label.

Note: Pausing the simulator does not interfere with the display refresh rate. If the display is in constant refresh mode, then it will continue polling the environment for agent changes. Since the simulator is paused, no changes are recorded and the environment returns an empty list. The display understands that an empty change-list means that no new information has been retrieved since the last time the environment was polled and the display doesn’t need to redraw or refresh.

The next button is marked ‘unzoom’. This is a perspective recovery action. Clicking the button returns the battle-map to its original dimensions. It is sometimes possible to zoom too much, or get off centered with a zoom function. Unzooming fixes this problem. See the zooming function write-up in the battle-map feature section for more information about zooming.

The last two buttons let the user control the display’s refresh rate. The first button, marked ‘Auto’, turns on the auto-update mode. This mode lets the user decide how often to poll the environment for agent changes and display these changes. Clicking the button brings up another data entry dialog. Like the “set sleep time” dialog, a user may input a valid positive integer. The refresh rate is measured in milliseconds. After giving a valid number and pressing enter, the display will automatically poll the environment and update the battle-map every x milliseconds, if x is valid number. A lower number means a faster refresh rate, which can make the animation smoother. It makes little sense to refresh faster than the sleep time. After the auto mode has engaged, the button’s label
changes to ‘stop Auto’. Clicking this button turns off the auto refresh timer and the display will not update automatically until told to do so. The label also returns to ‘auto’.

The other button is marked ‘Snapshot’. Clicking manually forces a single display update. The simulator is polled just once and all the data is displayed on the battle-map. This function is best used when the user just wants to view the current state, without engaging the auto-update mode. The snapshot action also redisplayes the battle-map. The window holding the battle-map may be closed at any time. This doesn’t interrupt the display or the simulator, even in auto-update mode. A quick click of the snapshot button makes the battle-map window reappear, with the current state. If the auto-timer is engaged, then the display continues to update constantly.

The final structure built into the menu window is the Agent List. This JList shows the names and id numbers of all the selected agents in the simulator. From the list, the user may activate the agent highlight system, or display all current field and capability values assigned to that agent. The status bar at the bottom of the menu window indicates the number of entries in the list. The list is scrollable. Agents may be added or removed from the list via the #List config file option. See the Config File subsection or the users manual for more information.

The two primary features of the Agent List are controlled with either a single or double mouse click. A single click an agent’s name within the list highlights that agent on the battle-map. Double clicking an agent’s name produces an Agent Info Dialog. This dialog shows all the current values for that agent or non-agent object.

The highlighting system allows users to identify different display objects on the battle-map. Selecting the agent name on the Agent List visibly changes the icon associated with that agent. If the user wants to know which of the various icons on the map is ‘Agent X’, then a single click will tell them. Unselecting on the Agent List restores the original agent icon. Only 1 agent may be selected at a given time. If one is already selected and
highlighted, and the user clicks another, then the highlight icon switches to the new agent.

![Highlighted Agent](image)

**Figure 5: Highlighted Agent**

Double clicking on an entry in the Agent List brings up an Agent Info Dialog. All the current values for all the current fields and capabilities are listed in a format that is easy to read. The dialog takes focus from the menu and the menu is inactive until the info dialog is closed. The info dialog does not interrupt any of the system functions or the auto-update timer.

3.2: Battle-Map

The battle-map window holds most of the display information. It uses an overhead perspective show the 2-d movements of the agents. This map uses the traditional X & Y
axis orientation to plot the position of the agents. Each agent or non-agent object can be
given a unique icon or color on the map. The config file controls most of the display
options for this frame and will be discussed in the next section.

Besides displaying each agent, the battle-map window also contains the zooming
functions. A left click with the mouse on any location within the map zooms in slightly,
recentering itself at that location. Clicking and dragging the mouse will zoom in
completely to the area just designated. A right click zooms out. Holding either button
performs a constant zoom in either direction. To establish the original perspective, click
the unzoom button on the menu frame. The middle button on a mouse does the same.
The zooming function does not interrupt the auto-refresh. There is one small deficiency
with the zooming function. If an agent is represented as a static image, then zooming
does not rescale that image. The icon will remain the same size regardless of how far the
user zooms in or out. The Future Additions section has more information about this.

3.3: Configuration File

The display module uses a custom-built configuration file reader to create the visual
scheme of the presentation. The config file is given to the system as a command line
argument. With this file, the user can change any of the colors, or any of the agent icons,
without recompiling the system. The config file also controls the various entries in the
Agent List. It can also suppress certain objects from being displayed if so desired.

Each application that uses the simulator should have its own config file. The users can
choose which features best represent the data. A full list of config file features and a
sample file is found in the user’s manual, in the appendix.

Chapter 4: Requirements

While the project lacks any rigorously defined system specifications, there are a number
of high-level goals that were required. Among the wish list of features are the three
requirements that drove this design. They are: 1) Make it customizable and extendable per each application; 2) Keep the display decoupled from the simulator; 3) Implement the ability to turn off the graphics, or to periodically take a snapshot of the environment.

The simulator will run experiments on many different examples. There are many students designing unique applications. Each of these applications has different visual requirements. Hard coding the various aspects of a visual scheme is inefficient and requires the users to maintain knowledge of the internal implementation of the module. Hiding implementation is the goal of all object-orientated programming. Thus, the design is required to provide a mechanism for easy customization that lets the users remain unaware of how the graphics are displayed.

The module uses the configuration file system to meet this requirement. This config file controls nearly all-significant visual choices. Everything from the specific icons to the background color can be altered without changing any of the source code. The system does not need to be recompiled, only restarted. The config system presents an additional level of abstraction to the user. The various applications can be packaged with a custom-built config file that best represents the environment.

If an application needs a more advanced feature, like additional user control over specific agents, the object-orientated design of the module allows new features to be plugged in with relative ease. The hierarchal nature of both the module design and G Library provides many opportunities to add new functionality. This is an advantage of both Java and object-orientated programming in general.

In an effort to shore up both security and efficiency of the simulator, it was required that the display uses the fewest number of system-hooks as possible. Ideally, all communication between the simulator and the display should run through a single node within the programming. If done correctly, the simulator and display are ‘decoupled’ from each other. This provides the following benefits. 1) Since the simulator and display do not actively share any mutable data structures, there is no danger of synchronization
errors. The display periodically polls the simulator for any new information. It does not have access to any real system data. 2) Decoupling allows the display the potential to be split off from the common java virtual machine and placed on a remote computer. The simulator may then have full access to the computer’s resources without sacrificing cycles to the display.

The simulator’s environment class provides the single link between itself and the display. The display may only pause or unpause the simulator, or set the time per ‘game-turn’. It may also receive a current snapshot of states of all the agents in the simulator. At no point can the display alter any data within the simulator. The agent info is packaged into DisplayData variables. These values do not contain any pointers to the actual agent fields and methods. All data within the simulator remains private and protected. The display has no deep-level hooks into the simulator and only one shallow connection. As long as this bridge is maintained, the two modules can evolve separately.

Since there is but a single connection point and this connection only uses serializable data structures and signals, the display and simulator can be further separated with a network protocol. This protocol has not been designed or implemented, but is not that great of a challenge to do so. There is more discussion of this topic in the Future Additions section of this report.

The final requirement the drove the design involves establishing a periodical refresh rate. For long batch jobs, it is not necessary that the display refresh every 10 milliseconds, or even every minute. The user must be able to refresh whenever convenient. Turning off constant refresh reduces the load on the CPU.

The display module meets this requirement with the snapshot feature and the autotimer class. The snapshot feature does exactly that. It tells the system to get the current state of simulator and display just that information. The DisplayData data type and the environment’s change-list allow the display to present the current state of all the agents without having to know anything about the previous states. Whenever the user wants to
see what is going on within the battle-map, a single click of the button shows all the
detail. The battle-map window can be closed without interrupting the simulator or the
display. Taking a snapshot will redisplay the window.

The AutoRunner class provides a constant refresh rate. The user may set this rate to any
length of time. The display will only poll the simulator at the leading edge of this timer.
The faster the rate, the smoother the graphics look, but at the expense of computational
resources.

Chapter 5: Design

This section describes the internal implementation of the module, including a description

5.1: The G Graphics Engine

The Java 2D library provides a litany of graphical tools to build a good two-dimensional
display. However, to prevent an unnecessary ‘wheel reinvention’, I choose to use a pre-
built graphics package, the G Graphics Library from Geosoft. This free package “is a
generic graphics library built on top of Java 2D in order to make scene graph oriented 2D
graphics available to client applications in a high level, easy to use way”[2]. Using the
library made development of low-level graphics technologies unnecessary. This section
will describe the hierarchal nature of the G Library.

The top-level component is the GWindow object. This object is added to a generic
JFrame and holds the rest of the G components [3]. As with most components in Java,
multiple GWindows can be placed within a single JFrame. The GWindow is also the
holder of any GIInteraction objects. These interaction objects define the connection
between the Frame and client application. The zooming function is a type of
GIInteraction and is handled by the GWindow.
Within every GWindow is a GScene. The API for this class indicates that is the link between the GWindow and the rest of graphic objects [3]. Furthermore, the GScene establishes the view-port. The view-port is the viewable dimensional space. When the user alters the default battle-map dimensions in the config file, this view-port reflects those values. If the client is executed on different devices with different display capabilities or dimensions, the GScene also holds transformation objects to fit the graphics properly. This particular feature is not used in the module, but is easily implemented.

The next level down is the GObjects. This class represents the graphical objects. They are added to the GScene. Each GObject has a draw method that is called whenever the frame chooses to refresh or update. Geosoft recommends extending this class and overloading the draw functionality [2]. The display module does exactly that with the AgentGObject class. For more specific discussion of this class, see the next section of this report.

At the lowest level is the GSegment. Each viewable shape or polygon is packaged into its own GSegment. The segment has a specific geometric location and geometrical shape. This defines where the segment is located on the map and what it looks like. There are many built in shapes to choose from, plus the ability to define new shapes. Users can select various rectangles, circles, and star shapes. The dimensions of each shape are defined in the GSegment. There is exactly one segment for each viewable agent or non-agent object. All these segments are added to the parent GObject and stored in a collection for future access. Changing the x & y coordinates within a segment moves that ‘piece’ to its new location.

To alter the look and feel of each GSegment, a GStyle object is applied. The GStyle holds the various appearance properties of any graphics object. They can be applied any GScene, GObject, or GSegment. The GStyle sets the color and line width. An entire static image can be substituted for a GStyle.
5.2: Classes

The core components of the display module are the four main classes, as shown in figure x. This section will describe each class and the interaction of objects within them.

Figure 6: Class Diagrams

5.2.1: SimDisplay

The SimDisplay class is the starting point for the module. It houses the menu frame and substantiates the other classes. The config file reader is also located in this class. The other main function of the SimDisplay class is to communicate with the simulator. The
SimDisplay keeps a private Environment pointer to either send pause/unpause commands or get the change list. No other class may get access to this pointer.

The SimDisplay constructor starts the config file reader and uses the information gathered to build both the menu and battle-map frames. Its last action set the simulator into run mode. The simulator and display are now executing correctly.

The config file reader parses through the user provided file and stores the information within several data structures. If necessary information is missing from file, the class provides default values. Any errors in the file are identified and reported to the user. In some instances, a default value is substituted for a file error. The best example is the color mapping function. Within the config file, users may designate the background or the agent icons as one of ten different colors. If a user selects a color not recognized by the system, or misspells the name of the intended color, then a default color is used. The user is told through console messages of the problem.

The menu is a typical Java frame, with components from the swing library. All the buttons use a common action-performed method kept in the SimDisplay. A single method that handles all user interaction has less synchronization issues than separate methods/classes for each action listener. The method identifies the source of the event, and executes the proper instructions.

The simulator gives each agent or non-object agent a unique id number and the SimDisplay uses this number to move data around. The first instance of this is within the Agent List of the menu frame. Each entry is the id number and the name of the agent composed as a single string. JLists are unable to keep any values other than strings [1]. To retrieve an agent’s information from a JList entry, the front part of the string is cut off and analyzed. This sub-string is the id number and uniquely identifies the agent. This method allows multiple agents and non-agent objects to have the same name, but still be separable within the JList.
As described earlier, the auto-update feature retrieves agent data and refreshes both the battle-map and the Agent List. Ideally, the SimDisplay would use a timer to wait for the necessary time and then update the frames. However, forcing the SimDisplay to wait locks its thread and prevents anything else to happen. The battle-map, the menu frame, and other classes sue the same thread to prevent synchronization issues. Locking the thread for a wait time prevent any user input to be handled. If the auto-display timer controls the thread, then it is impossible to turn the feature off. All event handling occurs after the timer halts, and it does not halt until told so.

The solution requires a new thread dedicated to the auto-update timer.

5.2.2: AutoRunner

The AutoRunner class uses a separate thread to prevent system lock. The class’s only functionality is to periodically call SimDisplay’s public update method. When the user clicks the ‘Auto’ button and gives a valid value, the SimDisplay creates an AutoRunner object with the sleep value as a constructor parameter. The new class instance starts a new thread, and executes a continuous loop. The loop forces the thread to sleep for the prescribed time, and then instructs the SimDisplay to update. The loop then restarts. The resulting periodic update achieves the auto-update function.

The AutoRunner’s continuous loop is controlled by a private Boolean variable stored within SimDisplay and retrieved with a public ‘get method’. When the user engages the auto-update, the variable is assigned a true value. The continuous loop checks the status of this variable each iteration. When the variable is true, the loop completes a single iteration. The sleep timer is started and the update occurs after the timeout. When the Boolean is assigned the false value, the continuous loop halts and terminates the auto-update. The Boolean value is only set to false when the user chooses to stop the auto-update and reclicks the ‘auto’ button.
Normally, some synchronization issues might occur with the design. Since the variable’s value is written by one thread and read by another, then a problem might exist if both threads try to access the variable at the same time. However, the Boolean has been declared as volatile. This requires the compiler to fetch fresh copies instead of caching the value. Boolean assignment in Java is atomic. These two properties prevent any syncing errors [1].

5.2.3: GPanel

The GPanel class is created by the SimDisplay and marks the beginning of the battle-map objects. The GPanel extends Java’s JFrame. Its primary purpose is to create the GWindow and GScene objects and substantiate AgentGObject class. After constructing the objects and setting them within the JFrame, the GPanel just passes messages from the SimDisplay to the rest of G components.

5.2.4: AgentGObject

As mentioned earlier, the AgentGObject class is an extension of a GObject. This class creates all the GS segments, sets their values, and draws them on the screen. The bulk of the update method is housed within. AgentGObject overloads GObject’s draw method, requiring that each redraw call gathers updated agent information. The highlight system is implemented within this class. See the highlight sequence diagram and corresponding section for more information.

5.3 Objects

Figure 7 is an object diagram of the display module. It shows the hierarchal nature of the design. The SimDisplay object creates the GPanel, which creates the GWindow, GScene, and the AgentGObject. Each object is embedded in the previous. The SimDisplay also creates the four hashmap and hashset object at the top of the diagram. These objects hold
most of the data collected from the config file. All four of these objects are created by the SimDisplay and used by the AgentGObject for its GSegments. Since all classes access these data structures exist on the same thread, synchronization is not an issue.

Figure 7: Object Diagrams
A hashmap is an implementation of Java’s Map interface. An entry in a hashmap has two entities, a key and a data value. Each key is associated with only one value. Every key value must be unique to produce a proper mapping. When an entry with a duplicate key is inserted into the structure, the existing data value is overwritten with the new. The key must be used to retrieve the data value [1].

The Icons hashmap tells which agents get which static image icons. The name of the agent is the unique key value. Multiple agent names may have the same image. The mapping does not use the agent’s id number as the key because the config file has no method of predicting the id number each agent. Also, multiple agents or non-agent objects might have the same name, but different id numbers. If they have the same name, they should receive the same icon.

The Colors hashmap works on the same principle. Objects listed in this structure will be represented by a small rectangle of a particular color on the battle-map. The config file reader maps an agent name to a Color object. If an agent name is in both the Colors hashmap and Icons hashmap, then the icon get priority.

When the Agent GObject class creates an agent’s GSegment, these two hashmaps are polled. If agent’s name is within one of them, the corresponding value is extracted and applied to the segment’s GStyle. If it’s an image, a GImage is created and added to the GStyle. If it’s to be a color, then a square geometric figure with the correct color is added instead.

The forbid object is not a hashmap, but a hashset. A hashset contains at most one copy of an object, but does not use a key [1]. Agent names listed in this structure will be suppressed from the battle-map. A GSegment will not be created for it, nor can it exist in the Agent List.

The Agents hashmap uses the unique id number assigned to every agent and non-agent object as the key value. When an agent is updated and retrieved from Change-List, its
DataDisplay object is inserted into the hashmap. Since the id number never changes, it overwrites the old data.

Most of the interactions between the classes are fairly trivial. The SimDisplay receives a command from the user via the menu frame. This command is sent to the appropriate G component via the GPanel, or to the Environment. However, there are two interesting interaction sequences that need to be explained.

5.3.1: Display Update Sequence

The display refresh sequence provides the most interaction between the objects. Figure x is the sequence diagram. Either the autotimer or the snapshot button triggers an event. The SimDisplay queries the Environment for the Change List, which it returns. From the data in the Change List, the menu’s Agent List is updated. It then calls the update method within the GPanel. The GPanel sends out both a redraw signal, and a refresh signal. Redraw() is a built-in method of a GScene. The actual mechanics of this function are unknown to me. As per true object orientated programming, Geosoft is hiding the implementation. From the outside, it looks as if the GPanel directly calls AgentGObject’s draw method [3]. This is how it is depicted on the sequence diagram.

The AgentGObject has a back pointer to the SimDisplay for the purpose of receiving the Change-List. For each DataDisplay object in the Change-List, a GSegment needs to be updated. The AgentGObject stores all the GSegments in a hashmap called agentList, with an agent’s id number as the key. Each agent has exactly one GSegment. Using the id number stored in the DataDisplay object from the Change-List, the correct GSegment is recovered. The geographic data from the DataDisplay object is extracted and implanted into the GSegment. This occurs for every agent or non-agent object.

After all the updates are performed, the AgentGObject executes a refresh and the battle-map is updated. All the method calls return and the module awaits the next display update signal, or any other command.
5.3.2: Highlighting Sequence

Agent highlighting is the other complex interaction. The highlight icon can only be assigned to one agent at a time. If an agent is selected in the Agent List, then the highlight effect needs to be removed from the previously selected agent and reapplied anew. To borrow a concept from networking protocols, the system must identify which agent has the highlight token. As the token is passed around, the system must restore all the previous possessors.

Figure 8: Display Update Sequence Diagram
When an agent entry in the Agent List is selected with a single click from the mouse driver, the SimDisplay identifies the id number. This number is stored as the owner of the highlight token. If the selected agent is deselected, then the stored number is set to zero. The value zero is reserved by the simulator and is never assigned as a unique agent number. When the token value is zero, then the system understands that the highlight effect must be removed from all agents.

Figure 9: Highlight Sequence Diagram

Figure 9 shows a typical highlight selection sequence. This sequence is assuming that no agent has the token. The highlight effect is currently off. The SimDisplay sends the id number of the selected agent to the GPanel, which sends it to the AgentGObject. Using the id number, the AgentGObject finds the proper GSegment. Since the highlight token previously did not belong to any agent, there is nothing to restore. The icon assigned to the GSegment is stored away in a local variable. The highlight image is placed with the
GSegment. To finish the sequence, the AgentGObject refreshes the display and the highlight effect appears on the battle-map.

Chapter 6: Future Additions

6.1 Agent Information via Battle-Map Mouse Interaction

Goal: Allow the user to ‘right-click’ on an agent icon within the battlefield to summon the Agent-Info dialog.

Obstacles: In the current implementation, each agent object exists in only one grid location (read pixel) within the map-model. Only this one set of coordinates is linked to the object. The G engine draws an expanded image centered at this spot. However, this is just a paint job to the area. Unless the mouse selects this one specific pixel, G will not be able to identify which object is drawn there. The user’s task of finding this one pixel is difficult.

Solutions: A limited search for the nearest agent to the geographic location selected may identify the user’s choice. Location data is kept in both the GSegment collection of the AgentGObject class and the agent list within the SimDisplay class. Either one of these classes can perform the search. The actual search algorithm runs in linear time, with each agent’s distance to chosen spot identified and compared. The smallest distance should identify the correct agent. However, if two or more agents are the same distance from the chosen pixel, then there is an ambiguity.

What would need to be changed: The scene GScene must implement the GInteraction interface to allow mouse interaction with the battlefield. This is a single method. The method would identify the coordinates selected by the user and can send the information to the search method. Once the search method has identified the agent’s id number,
SimDisplay’s displayData() function will create the dialog and display all the appropriate data.

Reasons for omission: Instead of selecting via the battlefield, the agent list on the menu provides an easier method of choosing agents to explore. Coupled with the highlighting feature, this mouse interaction isn’t necessary. With all user interaction limited to the SimDisplay class, there is no issue with synchronization of user action events.

6.2: Dynamic Agent Information Alteration:

Goal: Allow the user to change the values in the various capability fields within the Agent-Info dialog. Any changes made by the user will migrate to the simulator via the environment.

Obstacles: The simulator currently does not accept any information from the display. The system requirement for keeping the simulator and the display relatively decoupled prevents the display from making changes to the agents on its own accord. The Environment does not pass pointers to the agents themselves, but instead constructs snapshots of each agent’s progress and sends these snapshots to the display. The display could change the snapshots in any way, but there is no mechanism for them to be reincorporated back into the simulator.

Solutions: A public method within the Environment that accepts DataDisplay objects solves this issue. This method must check to see if the changes within the object are compatible with the intended agent. If passable, any alterations must migrate to the agent threads. All this must be done within the simulator.

What would need to be changed: Besides the simulator changes described above, the Agent-Info dialog needs a user input mechanism. Each capability and each field within must have some kind of text box that accepts user input. Agents have a variable number of fields, so the dialog needs to build the correct number of text boxes.
6.3: Multi-Agent Highlighting

Goal: Allow the user to select multiple agents in the Agent List and have each of these agents display the highlight effect within the battlefield.

Obstacles: To create the highlight effect, the selected agent’s GSegment-GStyle is swapped out with the highlight GStyle. The system maintains a pointer to the old style, ready to be swapped back in. If multiple agents may be selected, each of the old styles must be kept safe. The current system is designed for just a single selected agent. A complete reworking is necessary for multiple agent highlighting.

Solutions: Keep all the styles in a collection. When an agent is selected, get the style from the collection, and change the icon to the highlight icon. A new deselect method is required to tell the GSegment to restore the old style.

What would need to be changed: The Agent List must allow multiple selections and send each id number to the highlighting system. It must also keep track of which entries in the list have been selected and when it detects a change, send the deselected id number to the highlighting system. The system itself needs additional methods and data structures to store all the extra GSegment and GStyles.

Reason for Omission: Limiting the highlighting to just a single agent works just fine. There is little necessity for multi-selection. This idea was suggested in a meeting.

6.4: Scalable Images

Goal: When a zoom operation occurs in the battlefield, the icons and images will scale properly. Currently, only geometric icons built by the G engine scale. Static images and icon do not. This is really only an issue when the battlefield is zoomed out.
Obstacles: The G engine provides no functionality for scalable images. Any scaling must be handled by the application. If there is a java implementation of image scaling, it won’t be compatible with the G engine, so it must be custom built. Also, the configuration file reader does not identify multiple images/icons for the same agent.

Solutions: Instead of actually trying to scale the images within the system, the easiest solution is to provide pre-scaled images. Three sizes for each icon should be enough. When the zoom operation reaches a particular size ratio, the alternate sizes are loaded into the GSegments.

What would need to be changed: The configuration file reader needs to be able to identify multiple image files for each icon. Unless the images are all a standard size, the user must also identify the ratios when the switch out occurs. The zoom operation must identify when to make the switch and tell each GSegment to load in the new image.

Reason for Omission: Scalable images are not necessary to understand what is going on in the battlefield, nor do they provide any additional information. Only when the battlefield is zoomed out to a high distance do the images begin to overlap. If the user really needs scalable images, the default geometric icons do provide scaling.

6.5: Remote Display

Goal: Allow the display module to be run within a different java virtual machine, and even on a separate system from the rest of the simulator and model. The display should be able to remotely connect to an already simulator and get the current state of all the agents. All the functionality between the display and the simulator should remain in place.

Obstacles: There is not much keeping this from happening. The display module was designed with as few system hooks as possible. Connection between the display and
simulator is limited to a single connection point, through the simulator’s environment class. This connection point can run through a java network interface with little effort.

Solution: A protocol between the simulator and the display needs to be designed. The display needs to be able to send the pause and unpause commands to the environment, while the simulator needs to send the change-list ArrayList back to the display. Fortunately, the pause and unpause commands can be repackaged as simple signals, and the change-list is serializable.

What would need to be changed: ideally, neither the Environment or SimDisplay classes require any alteration. An additional new class that reads messages from the network and calls the appropriate methods within Environment can handle connection to the Simulator. The display module also has similar connection class. Both the Environment and SimDisplay believe that they are talking directly to each other, but are actually communicating through a network protocol.

Reason for Omission: It was deemed unnecessary to make the display networkable at this early project status. None of the various applications are mature enough to warrant remote presentation. But the potential functionality is in place when that day comes.

**Chapter 7: Conclusions**

The CROS: DISPLAY module provides users with a flexible and customizable presentation environment. Since the configuration file controls nearly all visual aspects of the module, demonstrations of various multi-agent organization applications are more impressive. It is easier to understand the agent behavior with on-demand information. For as long as researchers use the simulator, the display module will be a benefit.
Chapter 8: References

   <http://java.sun.com/j2se/1.5.0/docs/api/>

   <http://geosoft.no/graphics/>

   <http://geosoft.no/graphics/G/docs/index.html>

[4] MACR Website (find later)


[6] RoboSim, Kansas State University, CIS department. 2004
   <http://robosim.user.cis.ksu.edu/tiki-index.php>

   <http://diwww.epfl.ch/lami/team/michel/khep-sim/>

   <http://www.k-team.com/robots/khepera/>

[9] Lucas, G.W., Rossum’s Playhouse, 1999
   <http://rossum.sourceforge.net/sim.html>
Chapter 9: Appendices

Appendix A: Display Configuration File Users Manual

The configuration file for the Organization Display allows the user to substantially alter to display characteristics for their application. The file itself is a simple text file with different fields. Each field controls a different element of the appearance. None of the fields are mandatory. A config file may only use a few of the possible options. In each instance, a default configuration is loaded instead. The ordering of the fields is not set. The reader will parse through the file sequentially. When a properly formatted field is read, that option in the display is set.

All fields heading start with the pound (#) symbol. The header name concludes with a colon. If the field allow for multiple values, then the field listing must be terminated with a #end symbol. Fields that accept only one value do require a terminating symbol. Each field header, value listing, and terminating symbol must be on its own line.

Any other text in the file is ignored. However, this extra text must not exist between the field header and terminating symbol.

Header Names and Descriptions:

#Title: Sets the title of battlefield. Accepts just one value.

#Background: Set the background color. Accepts just one value;

#Dimension: Sets the x & y dimensions. Place both values on the same line, with a space between them.

#Forbid: Listing of all object and agent names to be suppressed from the battlefield. Accepts multiple values. Use the same name as given in the application UML file. Generic names don’t work. If there are 3 SearchAgents and all are to be suppressed, then all 3 need to be included in the forbidden list.

#Icons: Maps the agents to the chosen icon or image. For each agent, put its name and then the name of the image (with path if necessary) on the same line, separated with a space. This accepts multiple values.

#Colors: Instead of using images, the display can give an agent a colored box as an icon. List both the name of the agent and the color desired. Accepts multiple values.

#List: A listing of all the agents and/or objects to be displayed in the Agent List. Each name listed will exist in the Agent List (embedded into the main control frame). Accepts multiple values.
Colors

The configuration file accepts a large range of color names. These names are case sensitive and all names are lower case. Here is a list of all the currently supported colors.

black white blue red yellow gray magenta orange pink cyan dark_gray

Dark Gray is the default color

Sample File

/*Configuration file for Simple Search and Rescue
Ordering of command fields is arbitrary.
Fields that have multiple entries require a #end
Most fields aren't necessary. Default values will be used instead.
*/

//Title of Application
#Title:
Simple Search and Rescue

//Desired Background color
#Background:
black

//Listing of objects to be ignored in display
#Forbid:
WallObject
#end

//Desired geometrical dimensions to be displayed
#Dimension:
450 450

//Listing of objects to be given icons and the names of those
//icons
#Icons:
SearchAgent1 robocode.jpg
RescueAgent1 robocode.jpg
Victim1 icon8.gif
Victim3 icon8.gif
Victim4 icon8.gif
#end

//Listing of object to be given colors and name of those colors.
//color names must be lower case. See SimDisplay.java for all
//available
#Colors:
Victim2 blue
Victim5 yellow
#end

//Listing of objects to be displayed in the agent JList.
//If listed, then can be high-lited and summon state information
#List:
SearchAgent1
RescueAgent1
Victim4
Victim5
#end