Table of Contents

Introduction

This document serves to describe my work on the MASS debugger and accompanying GUI. I have separated this writing into two main sections. The first describes my work in integrating the existing debugger, written by Hongbin Li, into the MASS library. This section will describe in detail why and how this was done, as well as the issues I encountered in doing so. I was ultimately unable to complete the integration due to these issues, and instead created my own implementation of a debugger and GUI. The second section fully details this implementation.

Section 1

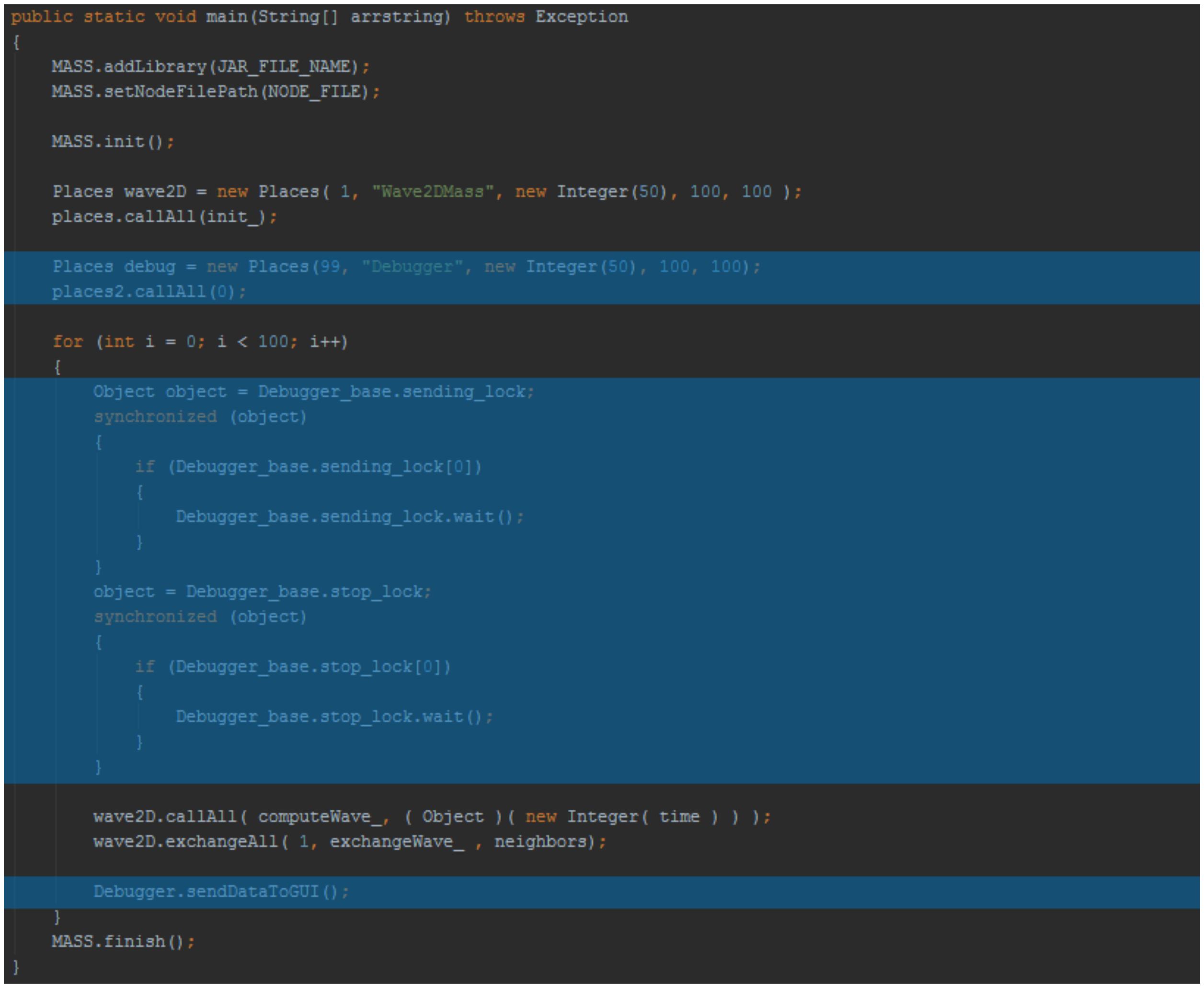
This sections describes my work in integrating Hongbin's debugger into the MASS library.

Goals

The following outlines the goals that I attempted to address.

Integration

One of the main design goals of the MASS library is to abstract the complexities of parallel programming and to minimize the user's effort in writing efficient spatial simulations. Prior to integrating the debugger into the MASS library, the user was required to have detailed knowledge of the library's internals. This was necessary so that the users program could synchronize itself with the MASS library and provide data from the users program to the MASS library. This placed much of the burden of debugging in the hands of the user. Highlighted in figure 1 is the code users would be required to add in order to use the debugger.



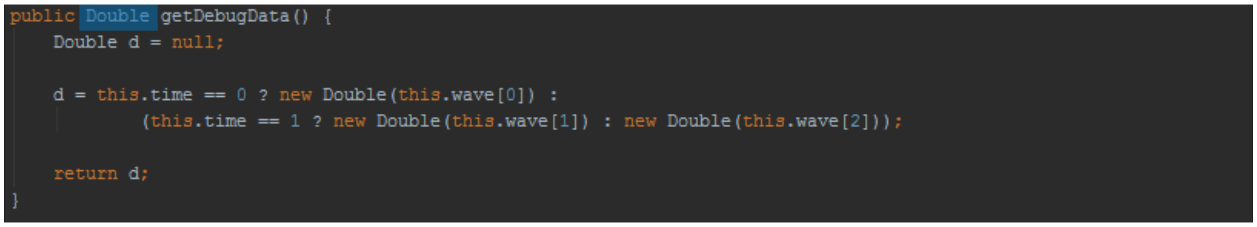
```java
public static void main(String[] arrstring) throws Exception
{
    MASS.addLibrary(JAR_FILE_NAME);
    MASS.setNodeFilePath(NODE_FILE);

    MASS.init();

    Places wave2D = new Places( 1, "Wave2DMass", new Integer(50), 100, 100 );
    places.callAll(init_);

    Places debug = new Places(99, "Debugger", new Integer(50), 100, 100);
    places2.callAll(0);

    for (int i = 0; i < 100; i++)
    {
        Object object = Debugger_base.sending_lock;
        synchronized (object)
        {
            if (Debugger_base.sending_lock[0])
            {
                Debugger_base.sending_lock.wait();
            }
        }
        object = Debugger_base.stop_lock;
        synchronized (object)
        {
            if (Debugger_base.stop_lock[0])
            {
                Debugger_base.stop_lock.wait();
            }
        }

        wave2D.callAll( computeWave_, ( Object )( new Integer( time ) ) );
        wave2D.exchangeAll( 1, exchangeWave_ , neighbors);

        Debugger.sendDataToGUI();
    }
    MASS.finish();
}
```

Figure 1

In addition to the code shown in figure 1, the user was required to place a file, Debugger.java, in the same directory as their main program, exposing even more aspects of the MASS library.

Debug Data Types

Another goal, pertaining to the way in which a user is able to send data from their program to the GUI, was set in order to allow users to choose the type of data sent. Initially the user was required to override 2 methods in Agents.java and Places.java that return a Double to be displayed in the GUI. This is shown in figure 2.



```
public Double getDebugData() {
    Double d = null;

    d = this.time == 0 ? new Double(this.wave[0]) :
            (this.time == 1 ? new Double(this.wave[1]) : new Double(this.wave[2]));

    return d;
}
```

Figure 2

This is a severely limited approach that assumes the user data has, or can be converted to, the Double data type. In addition to these two methods, one for Agents, and one for places, there are two more methods to be overridden that set the debug data for Agents and Places that also assume the Double data type.

User Defined Port

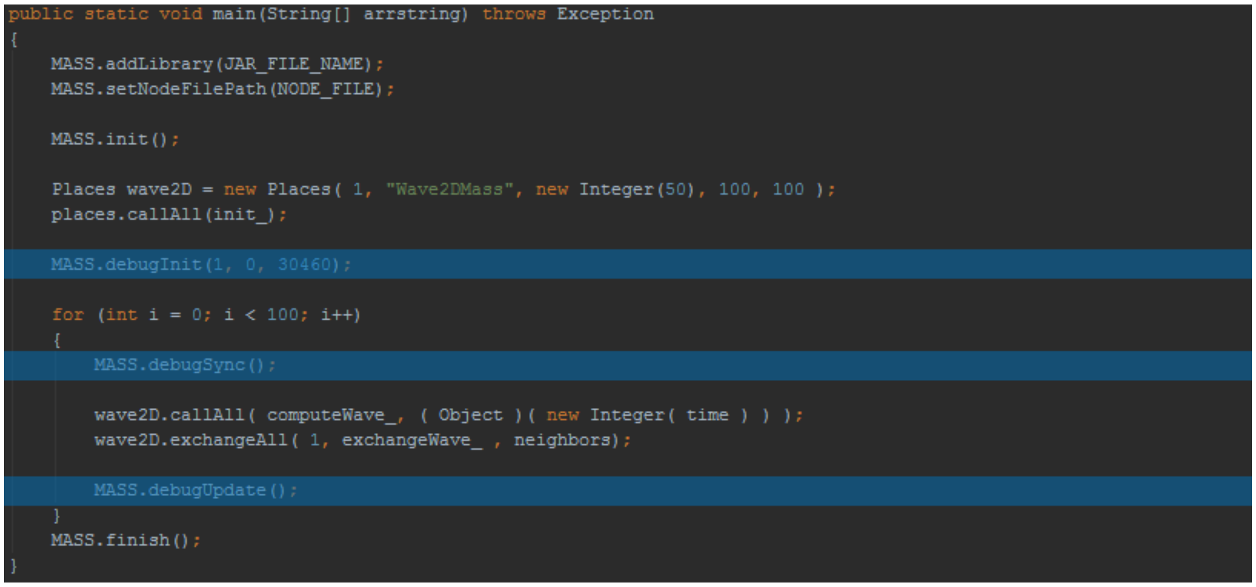
Another, more trivial goal was to allow the user to choose which port to use when connecting their MASS program to the GUI. Initially this was set in Debugger.java to a fixed port. If this port ever fails or is blocked by other use, the user must wait for this to be resolved before continuing the debugging process.

Progress

The following describes my solutions to each of the goals outlines in the Goals subsection.

Integration

The first steps were to integrate the as much of the debugging code into the MASS library. This includes the code written by users in their main method, as well as the Debugger.java that currently resides in the user's directory. My solution was to separate this code into three methods that reside in MASS.java. One to initialize the debugger, one to synchronize the users program with the GUI, and the last to send the user's data to the GUI. The resulting integration allows the users main program to be simplified from the code shown in figure 1, to the code shown in figure 3.



```
public static void main(String[] arrstring) throws Exception
{
    MASS.addLibrary(JAR_FILE_NAME);
    MASS.setNodeFilePath(NODE_FILE);

    MASS.init();

    Places wave2D = new Places( 1, "Wave2DMass", new Integer(50), 100, 100 );
    places.callAll(init_);

    MASS.debugInit(1, 0, 30460);

    for (int i = 0; i < 100; i++)
    {
        MASS.debugSync();

        wave2D.callAll( computeWave_, ( Object )( new Integer( time ) ) );
        wave2D.exchangeAll( 1, exchangeWave_ , neighbors);

        MASS.debugUpdate();
    }
    MASS.finish();
}
```

Figure 3

Highlighted in figure 3 shows the three MASS method calls required to debug MASS applications. This new implementation hides all implementation and eases the user's effort in debugging. The port number is specified in the MASS.debugInit() call, resolving the issue of allowing the user to choose the port of communication as well. The first two parameters of the MASS.debugInit() refer to the Agents and Places handles respectively. Note that in this program, there are no agents, and thus a handle of 0 is passed.

Debug Data Types

In an attempt to resolve the issue of the GUI only accepting a Double data type from the user, as shown in figure 2, I changed the return type to Number. Number is a java abstract class in which Double, Float, Long, Integer, Byte etc., all extend. This allows the user to choose one of many types that best fit their implementation. The GUI is then able to use reflection to display the user's data independently of the type they chose to use so long as it extends Number. Note that this will also allow a user to create their own class that extends Number in the case that none of the existing java Number subclasses will do.

Issues

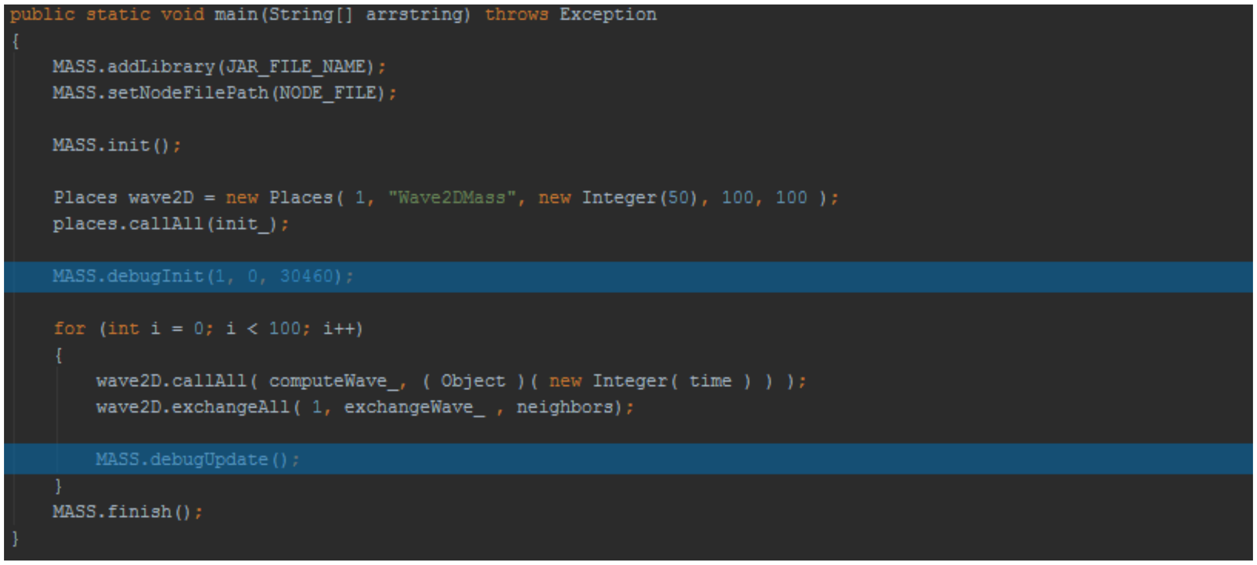
In testing the debugger with user programs in which Agents are implemented, the debugger would crash in attempting to retrieve its debug data. I attempted to resolve this issue and was unable to do so due to lack of documentation in Debugger.java and lack of communication with its author. My solution was to re-write the debugger and GUI myself which is describes in Section 2. Note that all debugging code relating to this section still exists in the MASS library.

Section 2

This section details my own implementation of debugging utilities and accompanying GUI

Enabling Debugging Features

Similar to my integration of the debugging code from figure 1 to figure 3, the newest integration is shown in figure 4.



```
public static void main(String[] arrstring) throws Exception
{
    MASS.addLibrary(JAR_FILE_NAME);
    MASS.setNodeFilePath(NODE_FILE);

    MASS.init();

    Places wave2D = new Places( 1, "Wave2DMass", new Integer(50), 100, 100 );
    places.callAll(init_);

    MASS.debugInit(1, 0, 30460);

    for (int i = 0; i < 100; i++)
    {
        wave2D.callAll( computeWave_, ( Object )( new Integer( time ) ) );
        wave2D.exchangeAll( 1, exchangeWave_ , neighbors);

        MASS.debugUpdate();
    }
    MASS.finish();
}
```

Figure 4

The only difference in MASS interface is the removal of the MASS.debugSync() method. Java synchronization is not necessary in my implementation. This allows the user to write one less line of code in order to use the debugging features of MASS. The port number is still specified by the user and the debugging data type is still specified in the overridden methods of Agents and Places.

Debug interface details

The users debugging interface consists of the two methods shown in figure 4. The details of each of each are described in the following to subsections.

debugInit()

Intuitively, this method is designed to initialize the debugging capabilities of the MASS library. First a connection is attempted to be made with the GUI. This is a blocking procedure, meaning the users program will halt until a connection is made to the GUI. This allows the user to execute their program and the GUI in any order, where as previously the user was required to start their program before execution of the GUI. After a connection is made to the GUI, data is gathered into an InitialData object that is then sent to the GUI. This initial data includes the following:

- The class names of the user defined Agent and Place.
- Booleans representing whether or not the user has overridden the get and set debug data methods.
- The data type of the overloaded get and set debug data methods in each, Agent and Place, objects.
- The number of Places and Agents.
- The dimensions of the Places grid.

The initial data that is sent can be expanded by adding data members to the InitialData class. Once the GUI received this InitialData, it displays the user relevant data such as Agent/Place names and sizes. The other data is saved and accessed when needed, such as the user defined data types, which are needed when sending user injected data back to the Agents and Places.

debugUpdate()

This method is used to transfer data between the users program and the GUI. The type of data sent is dependent upon the user's action in the GUI. For example if the user presses the "next" button, a packet of data pertaining to current Agents and Places state is sent to the GUI to be displayed in the visualization grid. A complete description of each type of transfer than can be completed by calling the debugUpdate() method is as follows:

1. As stated above, if the user has pressed the "next" button or the "play" button, data is gathered pertaining to the current state of Agents and Places into an UpdatePackage object. This object contains the following member fields which may be extended by adding members to this class:
    - An array of PlaceData objects, each of which contain:
        a) This places data, as defined by the user overridden getDebugData().
        b) The index in which this place exists in the array of Places held by the MASS library.
        c) A Boolean representing whether this place contains agents.
        d) An array of AgentData objects, each of which contain:
            i. This agents data, as defined by the user overridden getDebugData().
            ii. This agent's index corresponding to the index of the place in which it resides.
            iii. This agent's id number as assigned by the MASS library.
2. If the user injects Place or Agent data using the GUI, an AgentData or PlaceData object is sent back to the MASS library. The contents of each are described above. This data is then set using the Agents/Places setDebugData() method.
3. When the user chooses to end the connection between their program and the GUI or when the user closes the application a Disconnection object is sent from the GUI to MASS in order to do clean up (closing sockets and streams etc.). This is not yet implemented at this time. When it is a complete description of this object and its contents will be shown here.

The following is a diagram showing the above inheritance hierarchy:

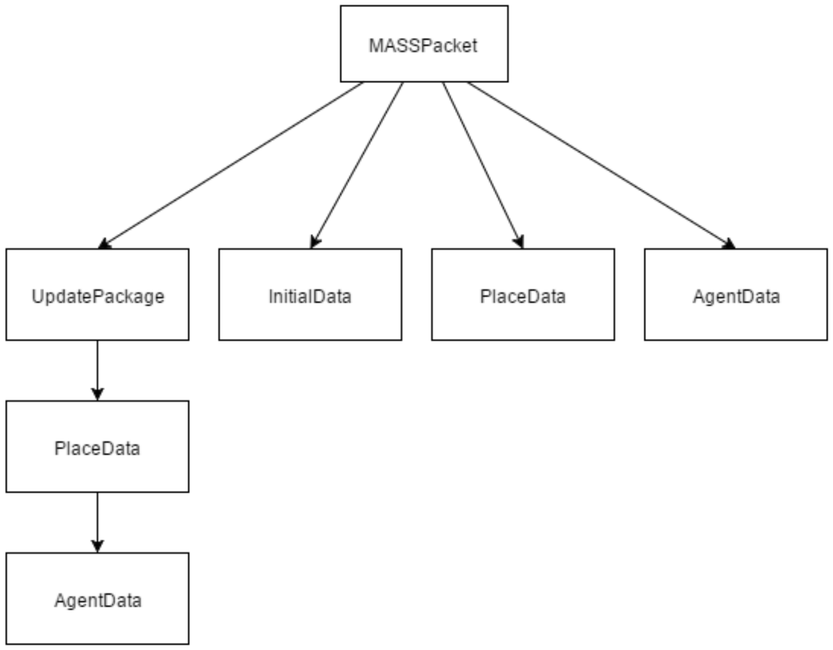
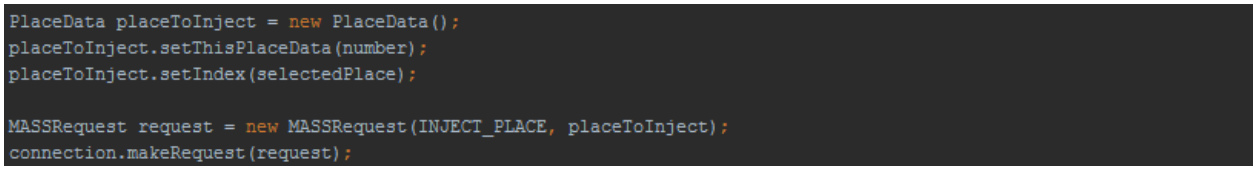


Figure 5

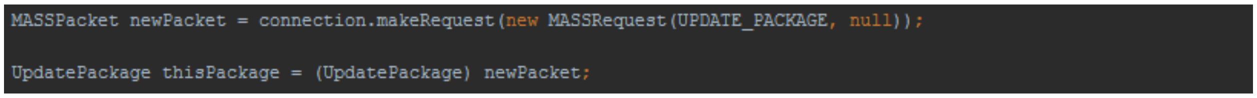
MASSPacket is the top level abstract class in this hierarchy and exists as a wrapper class so that all subclasses may be sent via java Socket to and from the users program and the GUI. MASSPacket also ensures that all subclasses are serializable. Figures 6 and 7 show an example of two different types of data transfer between the users program and GUI that utilizes this inheritance hierarchy.



```
PlaceData placeToInject = new PlaceData();
placeToInject.setThisPlaceData(number);
placeToInject.setIndex(selectedPlace);

MASSRequest request = new MASSRequest(INJECT_PLACE, placeToInject);
connection.makeRequest(request);
```

Figure 6

Figure 6 shows the data transfer that occurs when the user has used the GUI to inject new Place data into the users program using the “injectPlace” button. First a PlaceData object is constructed with the new data to be injected and the index in which this place exists within the Places array held by the MASS library. A request to send the data to MASS is made with the Connection class’ makeRequest() method. This method takes a MASSRequest object containing an enum representing the type of request, and the PlaceData object created earlier. Once this code is executed, the appropriate data from the GUI has been sent the MASS library where it interprets the data based on the type of request.



```
MASSPacket newPacket = connection.makeRequest(new MASSRequest(UPDATE_PACKAGE, null));

UpdatePackage thisPackage = (UpdatePackage) newPacket;
```

Figure 7

Figure 7 shows another type of data transfer that occurs when a user has pressed the “play” or “next” buttons. When this happens, the user is requesting and update regarding the latest state of all Places and Agent objects. This request is done in a similar fashion as shown in figure 6. A request is made via the Connection class' makeRequest() method that is passed a MASSRequest object containing an enum representing that we are requesting an UpdatePackage from the MASS library. The null value indicates to MASS that the GUI is not attempting to send data *to* MASS, but instead is requesting data *from* MASS. In this case the connection object returns the data that was requested as a MASSPacket that can then be cast into an UpdatePackage. The updates can now be displayed appropriately to the user.

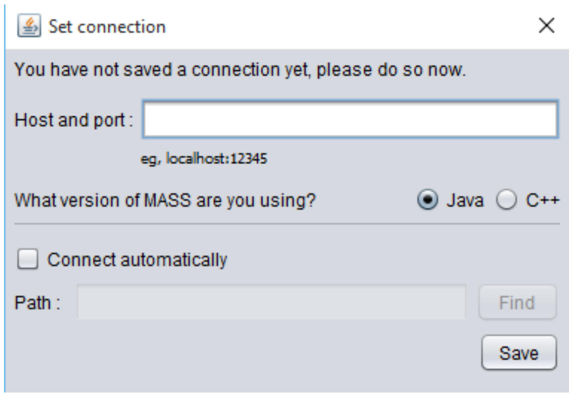
This hierarchy was designed to allow many types of requests of data transfer. This hierarchy can be extended to allow for any conceivable types of transfers between the GUI and MASS in a polymorphic manner. Details of the Connection class will be described later in this section.

The GUI

This section will be divided into two sections. The first describes the interface and the second will discuss the implementation details.

The Interface

The first time a user opens the GUI a form will appear that asks the user to enter host and port information corresponding to the host that their MASS application is running on and the port of their choice. The form also asks whether the user is using the Java or the C++ implementation of MASS, as well as an option to auto-connect the GUI to their MASS application. These last two options are not implemented at this time. The form is shown in figure 8.


Set connection

You have not saved a connection yet, please do so now.

Host and port :

eg, localhost:12345

What version of MASS are you using? [✓ Java] C++

Connect automatically

Path :

Figure 8

Once the user enters the host and port and clicks the save button, the main GUI form opens. Host and port information is saved from here on out and the user will not have to enter it again unless they wish to change it.

At this point the GUI informs the user that there is not yet a connection between the GUI and the user's program. This is shown in figure 9. The connection state will be displayed here. At this point all buttons and field are disabled except for the "connect" button.

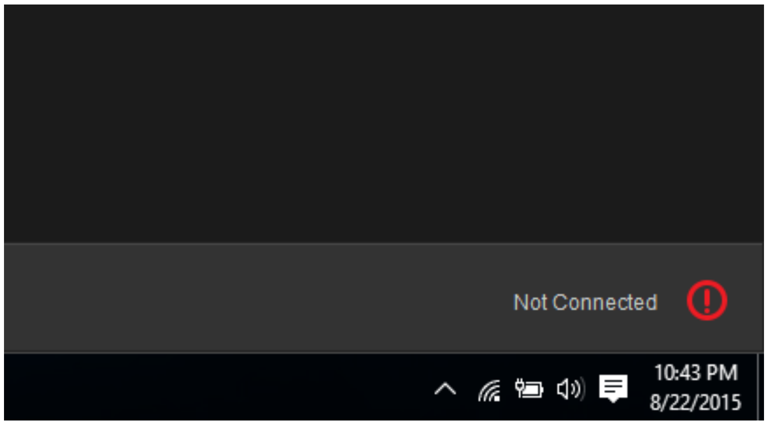


Figure 9

Once the connect button has been clicked, a connection will be made to the users program. If successful the displayed connection state will change from figure 9 to figure 10 or an error message will be displayed in this space.

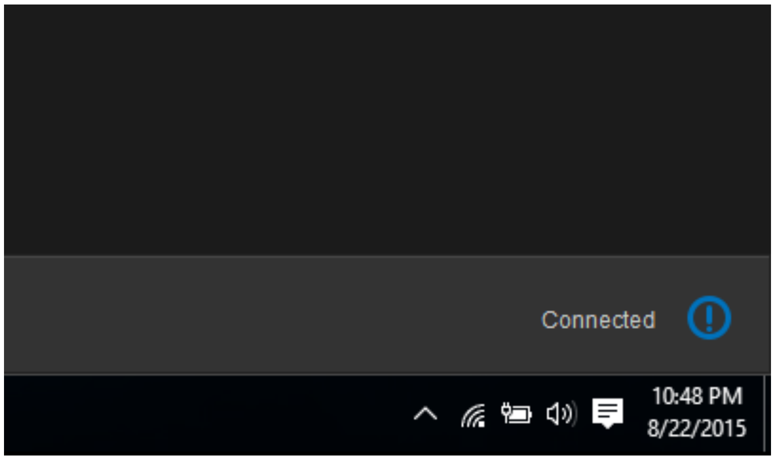


Figure 10

The names and sizes of Agents and Places will be displayed to the user as well. From here the user has a couple options. The user may click the play button or the next button shown in figure 11.

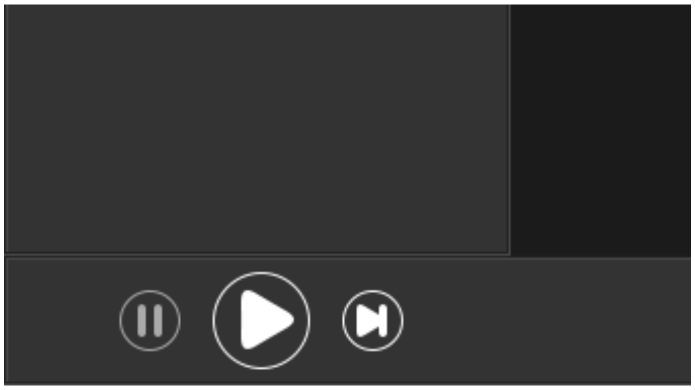
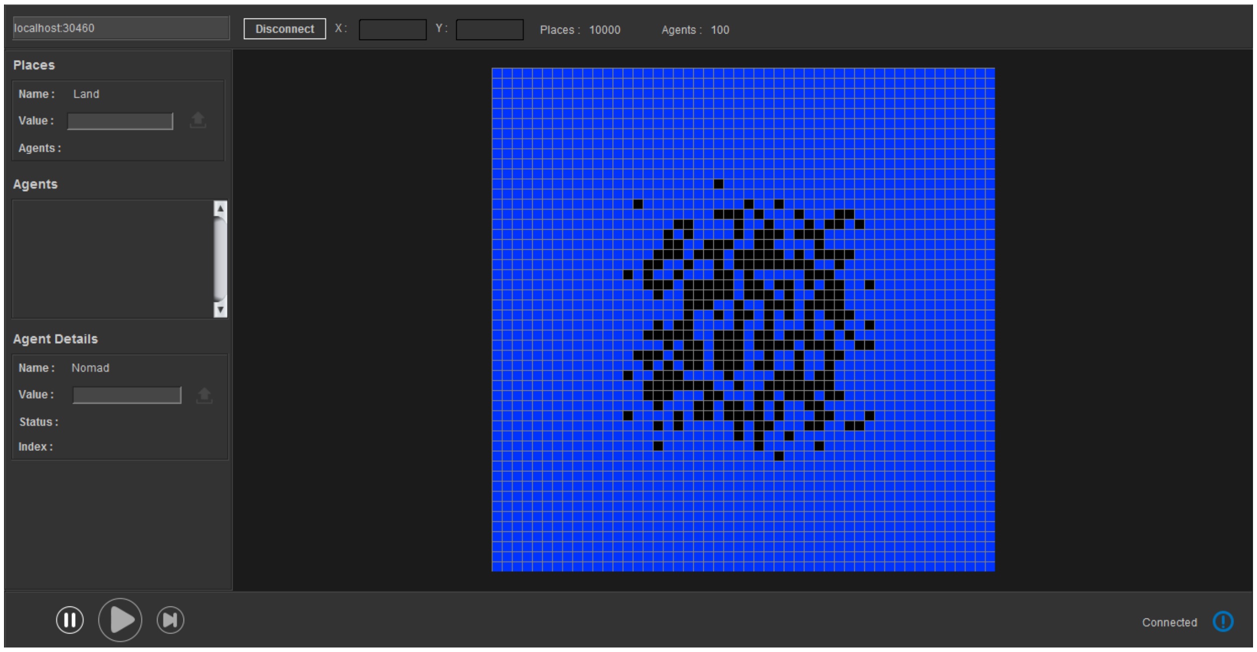


Figure 11

The play button runs through the users entire simulation loop with 1.5 seconds between each iteration. The user may pause the simulation at any time during this loop by clicking the “pause” button. The “next” button on the other hand, will run just one iteration of the simulation and immediately pause it so the user may iterate at their own pace. Figure 12 shows the GUI’s state after the user has pressed either the “play” or the “pause” buttons.



localhost:30460

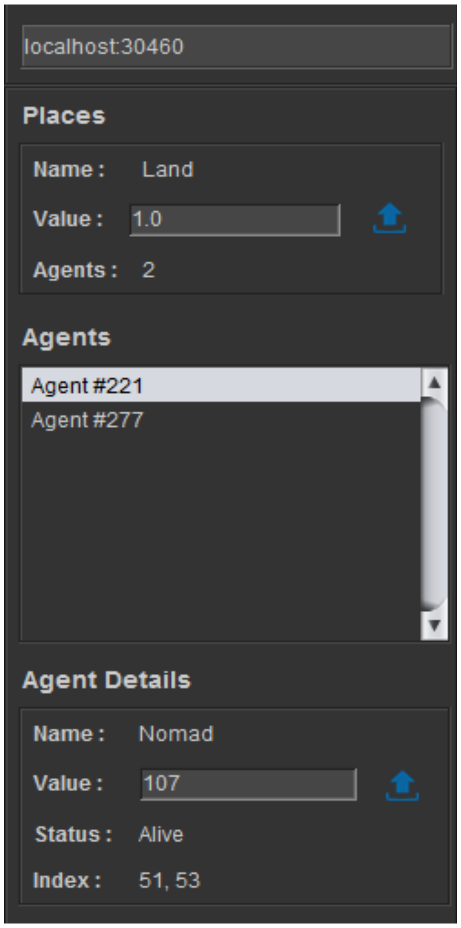
Places : 10000 Agents : 100

Name : Land
Value :
Agents :

Name : Nomad
Value :
Status :
Index :

Figure 12

The grid generated in the center of the screen is based on the debug data of all Places and Agents modulus 20. Meaning there are 21 colors that can be displayed, each of which representing a difference in the state of Place objects. Agents are always displayed black. If a user wishes to view this exact data instead of differentiating based on the color, they may click on any of the grid spaces. This will display all Place/Agent data residing on this square on the left side panel as shown in figure 13.


localhost:30460

Places

Name : Land

Value : 1.0

Agents : 2

Agents

Agent #221

Agent #277

Agent Details

Name : Nomad

Value : 107

Status : Alive

Index : 51, 53

Figure 13

If Agents exist on the selected Place, each will be displayed in a list. The user may click on each agent in the list to view its specific data in the Agents section beneath the list. The blue “inject” buttons will become enabled only if the user has overloaded the Place and Agent setDebugData() methods. When enabled the user may change the current Place/Agent data, click the “inject” button, and send that data to their MASS program. The data will be injected and can be viewed in the GUI grid on the next iteration of the simulation. An example of this can be seen in figure 14 in which a new wave has been injected into the Wave2D simulation underneath the original wave. The colors represent varying wave heights.

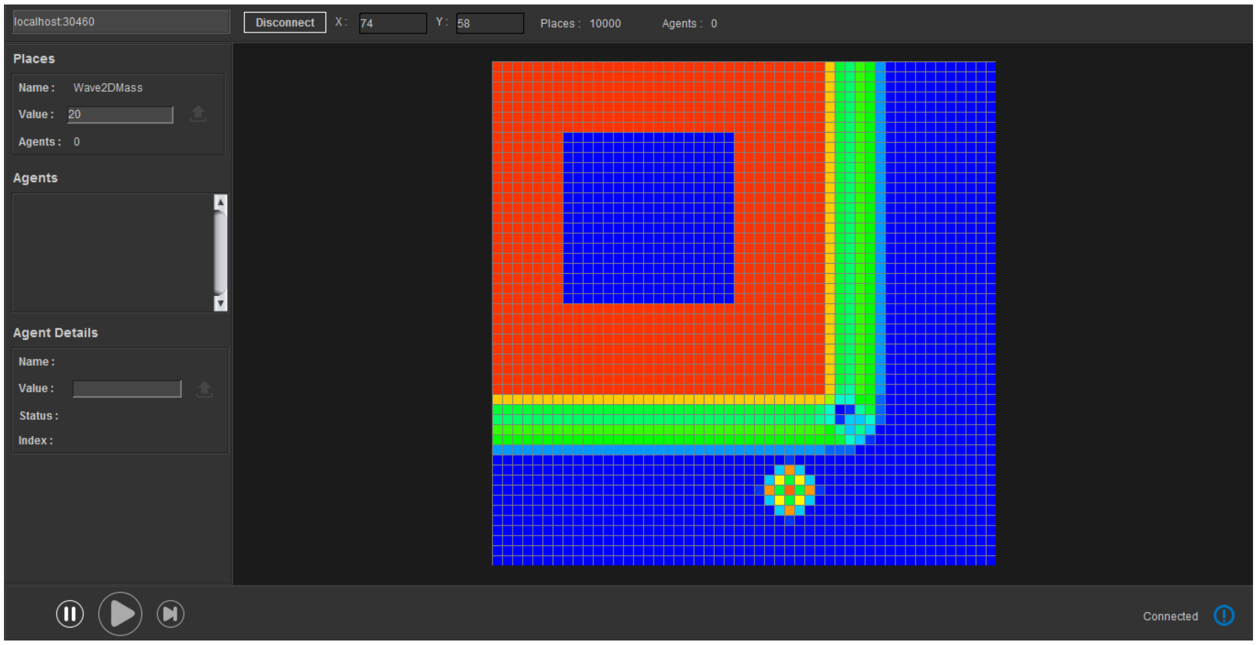


Figure 14

Once the user is done debugging their simulation they may click the "disconnect" button which will close the connection. This is not yet implemented. The user may then close the application or create another connection to a MASS program.

Implementation Details

The GUI was created using the NetBeans GUI Editor. It is recommended that NetBeans is the IDE used to perform future additions/edits to avoid incompatibilities between IDE's.

Connection Class/Subclasses

As shown above in figures 6 and 7, the Connection class is used to make data transfers between the users program and the GUI using the makeRequest() method. The Connection class also contains utilities for validating hosts and ports, and opening and closing Socket connections. Connection is an abstract class with two subclasses, one for connecting to a MASS java (MASSJavaConnection.java), and one for connecting to MASS C++ (MASSCppConnection.java). Connections and data transfers must be implemented differently depending on which version of MASS the user is using. Because of these differences, Connection contains abstract methods connect(), makeRequest(), and endConnection() to be implemented by each subclass. A diagram of this relationship is shown in figure 15.

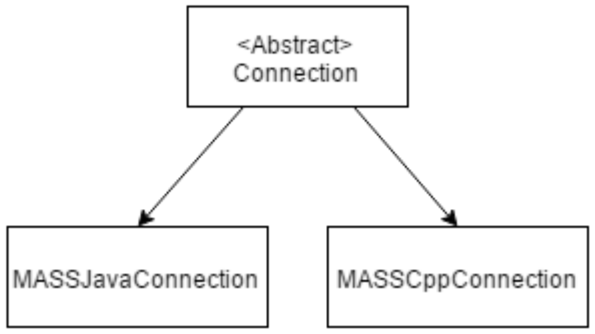
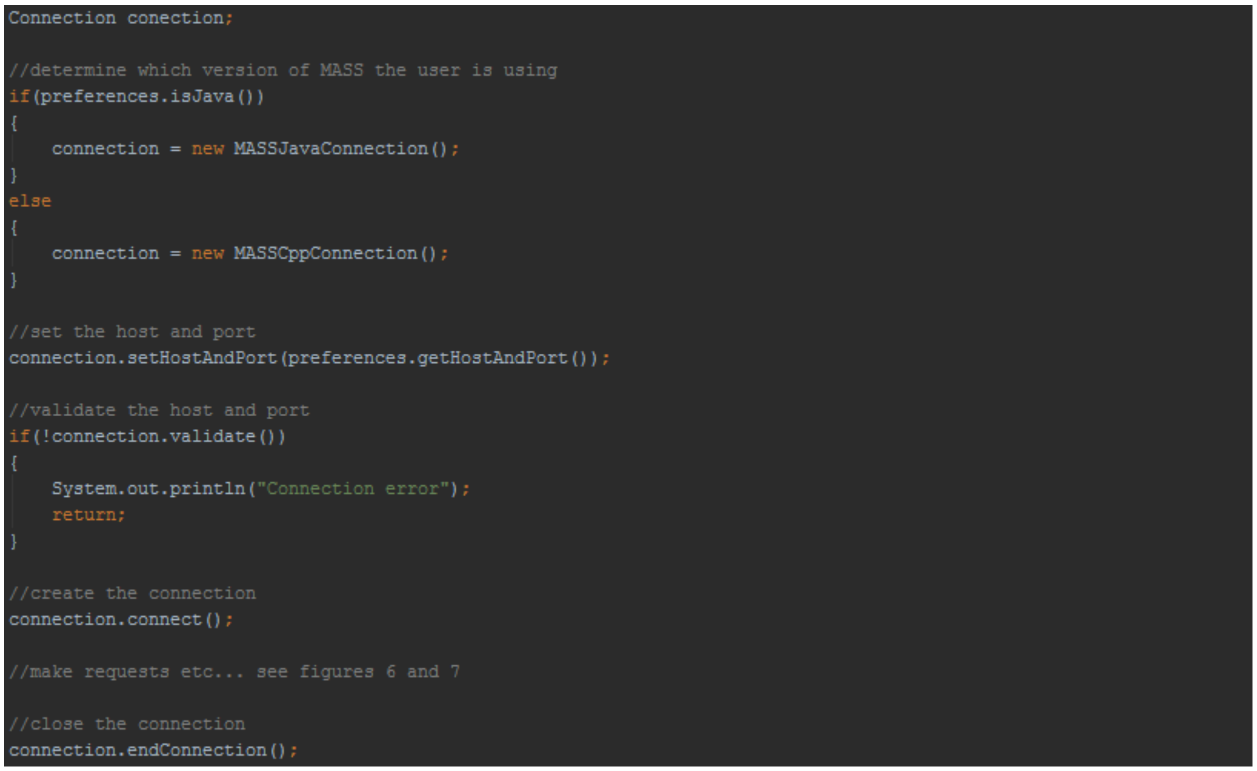


Figure 15

Figure 16 illustrates how the Connection class is used to make connections and perform validation of hosts and ports.



```
Connection conection;

//determine which version of MASS the user is using
if(preferences.isJava())
{
    connection = new MASSJavaConnection();
}
else
{
    connection = new MASSCppConnection();
}

//set the host and port
connection.setHostAndPort(preferences.getHostAndPort());

//validate the host and port
if(!connection.validate())
{
    System.out.println("Connection error");
    return;
}

//create the connection
connection.connect();

//make requests etc... see figures 6 and 7

//close the connection
connection.endConnection();
```

Figure 16

User Preferences

From figure 16 you can see that we can tell whether the user prefers MASS java or MASS C++, and what their preferred host and port is from the preferences object. This object is an instance of the Preferences class and exists persistently, even when the GUI closes. The object is first created when the user first runs the GUI and enters their preferences in the form presented in figure 8. The Preferences class was created to avoid the user re-entering their preferences each time the GUI is executed. The following details all members of the Preferences class:

- URI uri - this member holds host and port information. It is used primarily for its validation utilities so that I do not have to do so on my own.

- boolean autoConnect – (to be implemented) contains whether or not the user would like to connect automatically to the host and port upon execution of the GUI.
- boolean java – (to be implemented) if the user is using MASS java or MASS C++ so that the appropriate Connection subclass can be instantiated.
- String programDirectory – (to be implemented) the directory of the users MASS program to be used in conjunction with autoConnect in order to start the users application automatically.

The Preferences class is sure to be extended as GUI development continues. The preferences object is serialized to a file and saved in the Preferences package in the Debugger directory. Each time the GUI is opened, preferences are deserialized into a preferences object and preferences are loaded appropriately. The user has the ability to adjust their host and port preferences using the "hostField" in the GUI shown in figure 17.

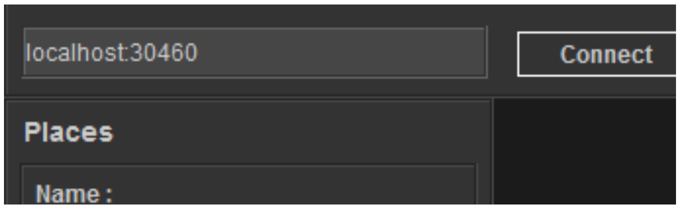


Figure 17

In the future the user will be able to click the connection image shown in figures 9 and 10, a form will pop up in which the user has full editing ability of all preferences.

The Visualization Grid

The grid can be seen in figures 12 and 14. One of the main problems with the original GUI grid was its extremely slow refresh rate. Each rectangle in the grid could be seen to be drawn one at a time. In a grid of 100 by 100 the grid took several seconds to completely display one iteration of data. My solution was to create the custom class GridPanel.java that extends JPanel. The result is instant refresh rates for each iteration. When the GridPanel is instantiated, a number of CellPanels are added to it corresponding to the number of Place objects specified by the user's MASS program. CellPanel.java is another custom class that extends JPanel that has listeners attached to detect user clicks. Each CellPanel also has listeners to detect which cell is being hovered in order to display the corresponding Place object's index in MASS's Places array. The display of this index is shown in figure 18.

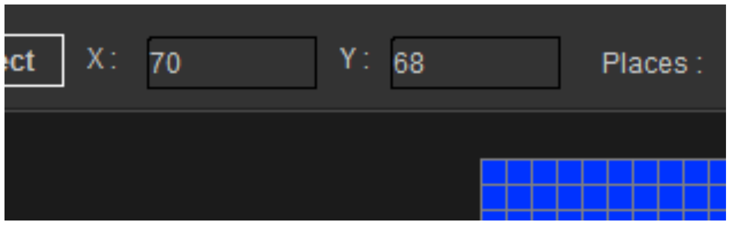


Figure 18

Work to be completed

During the quarter I spent the first 7 weeks working on Hongbin Li's debugger. The last week was spent on my own implementation. Given that I only had one week to work on this, there are a few features that are left incomplete. The following subsections detail each feature/function to be completed.

Grid Resizing

Resizing the visualization grid was a known issue in Hongbin Li's GUI implementation as well. The issue arises when the user has roughly more than 10,000 Places in their simulation, which is very common. After this point the GUI has only a few choices, each with severe drawbacks. One solution is to maximize GridPanel as much as possible and minimize each CellPanel to fit within the GridPanel. The problem with this solution is that one each CellPanel become less than one by one pixels, the grid becomes useless to the user as action listener accuracy fails at this point. Ideally, the user would be able to zoom in and drag the grid around, similar to the features of Google Maps. The issue with implementing these features are the limitations of the Java Swing library. Although possible, my research indicates this task would be difficult and time consuming. In the future I would like to switch from Swing to JavaFX. JavaFX is java's newest graphics library that has built in utilities for implementing zoom and drag features. JavaFX would also allow for 3D grids, allowing debugging capabilities for 3D MASS Places, which is currently not supported.