Ant Simulator:
Swarm Intelligence using Goal-based Agents & A*

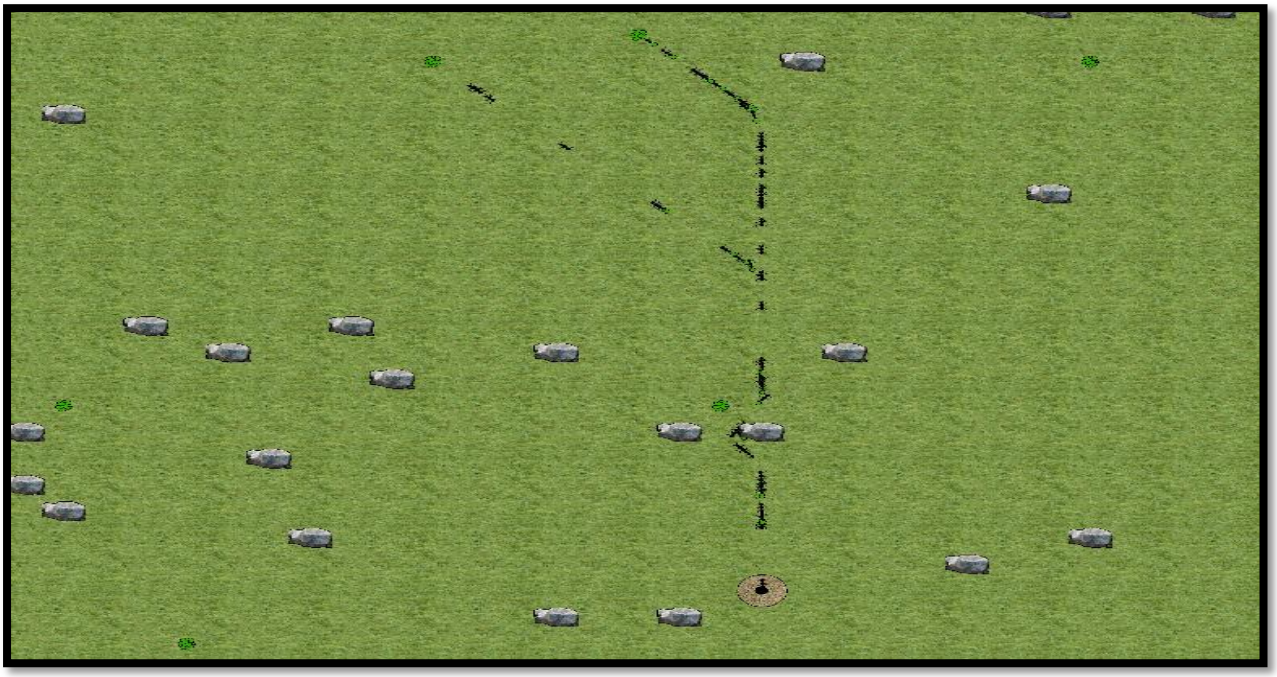
Christopher D. Canfield
CS 664: Artificial Intelligence
Boston University Metropolitan College

I. Overview

The *Canfield Ant Simulator* simulates a colony of ants, with a focus on the ants' search for food. Ants lay pheromone trails to notify other ants about food, as in the real world [7][8][9]. Over time, this gives the ants the appearance of having a higher-level coordinating intelligence that is instructing all ants according to group goals, when in reality each ant is fully in control of itself, and does not have any direct communication with other ants.

The project involved the creation of the following components:

- World Creator: random map generator
- Navigation Graph, with nodes & edges
- Ants: goal-based agents
- 7 ant goals
- A* algorithm
- GUI processing framework using the Observer pattern
- 123 unit tests + 3 test applications



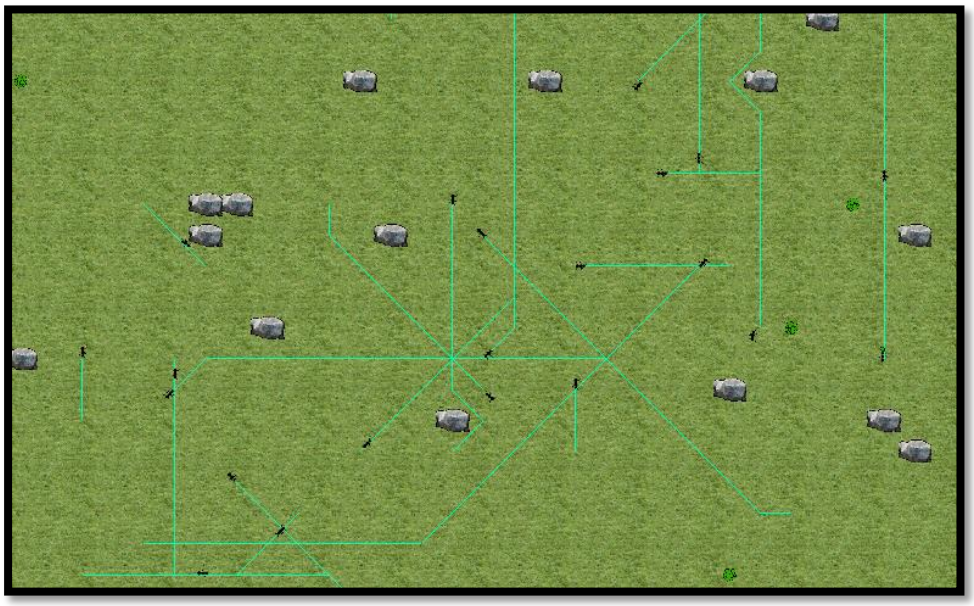
II. Artificial Intelligence Components

Ants: Goal-based agents
This is the main driver of the simulation, and was the most complex component of the simulation (particularly the implementation and testing of the ant goals). Ants and ant goals are described in detail in sections III-E and III-F.

Performance	Environment	Actuators	Sensors
• Is alive • Is not hungry • Colony has food	• Partially viewable (knows nav graph, but does not know food or pheromone locations) • Partially dynamic (nav graph is static, but food disappears and pheromones appear and wear out) • Navigation graph with nodes and up to 8 edges per node • Obstructions, food piles, ant hill, other ants (no direct interaction), pheromones	• Move • Pick up & drop food • Lay pheromone trail • Eat food	• View current navigation node or edge • Smell pheromones on current edge, or on all edges connected to current node

Swarm Intelligence
Ant colonies appear to be smarter than individual ants. "Ants aren't smart, ant colonies are" (Deborah M. Gordan, biologist at Stanford University) [9]. Swarm intelligence is the phenomenon of the group being smarter than the individual, despite not having a higher-level intelligent controller to instruct the individuals. See sections III-D through III-F for information about the implementation of this process.

*A**
The A* algorithm is used by the ants to create paths from the current node to a destination. See section III-G for the details on my implementation.



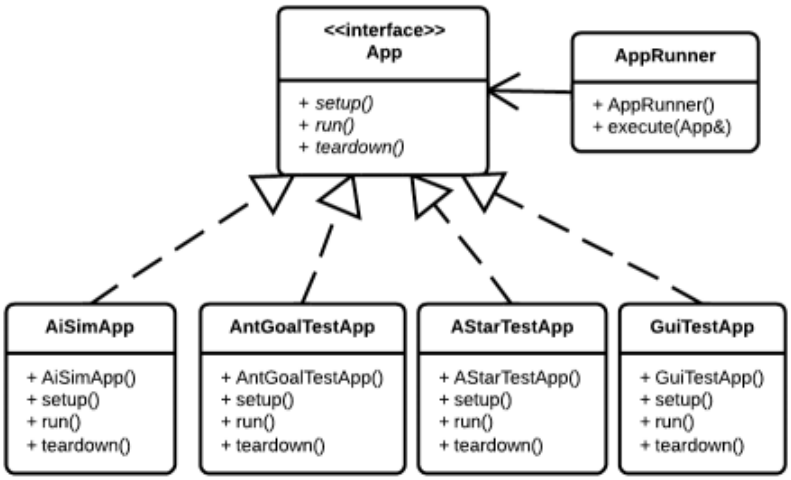
Light-blue lines are paths calculated by the A* algorithm

III. Simulation

The simulation was written in C++, and uses the following libraries:

- Simple & Fast Multimedia Library (SFML): Provides a cross-platform object-oriented abstraction layer over OpenGL and the platform's native windowing system. [10]
- Boost C++ Libraries: circular array, noncopyable [11]
- POCO C++ Libraries: UUID [12]

III-A. The App interface & related

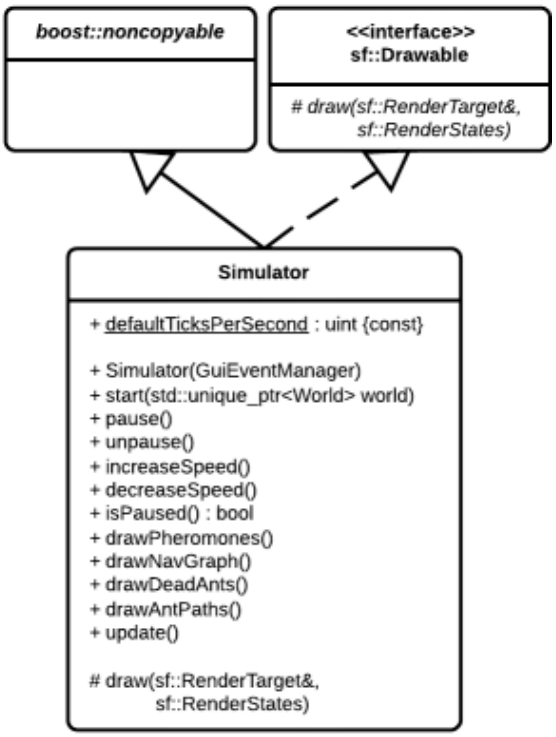


Applications inherit from the App interface. On startup, the AppRunner calls the App's setup method, and then continuously calls run() until the application returns false. At that point, the AppRunner then calls the App's teardown method. Four applications were developed for this project:

- AiSimApp: The Ant Simulator. This is the primary application.
- AntGoalTestApp
- AStarTestApp
- GuiTestApp

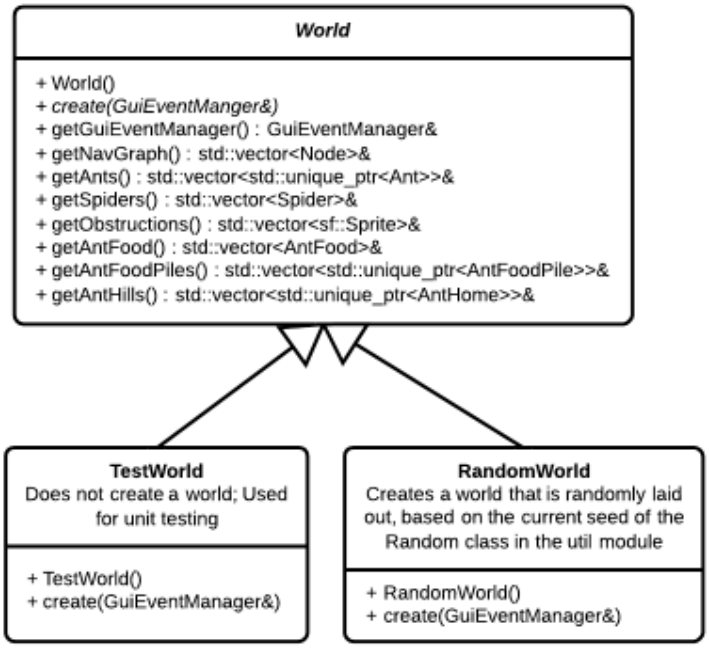
The application that is compiled into the binary is specified by `#define`ing various identifiers, which are listed in main.cpp. See the Testing & Verification section for additional information regarding the three test apps.

III-B. The Simulator class



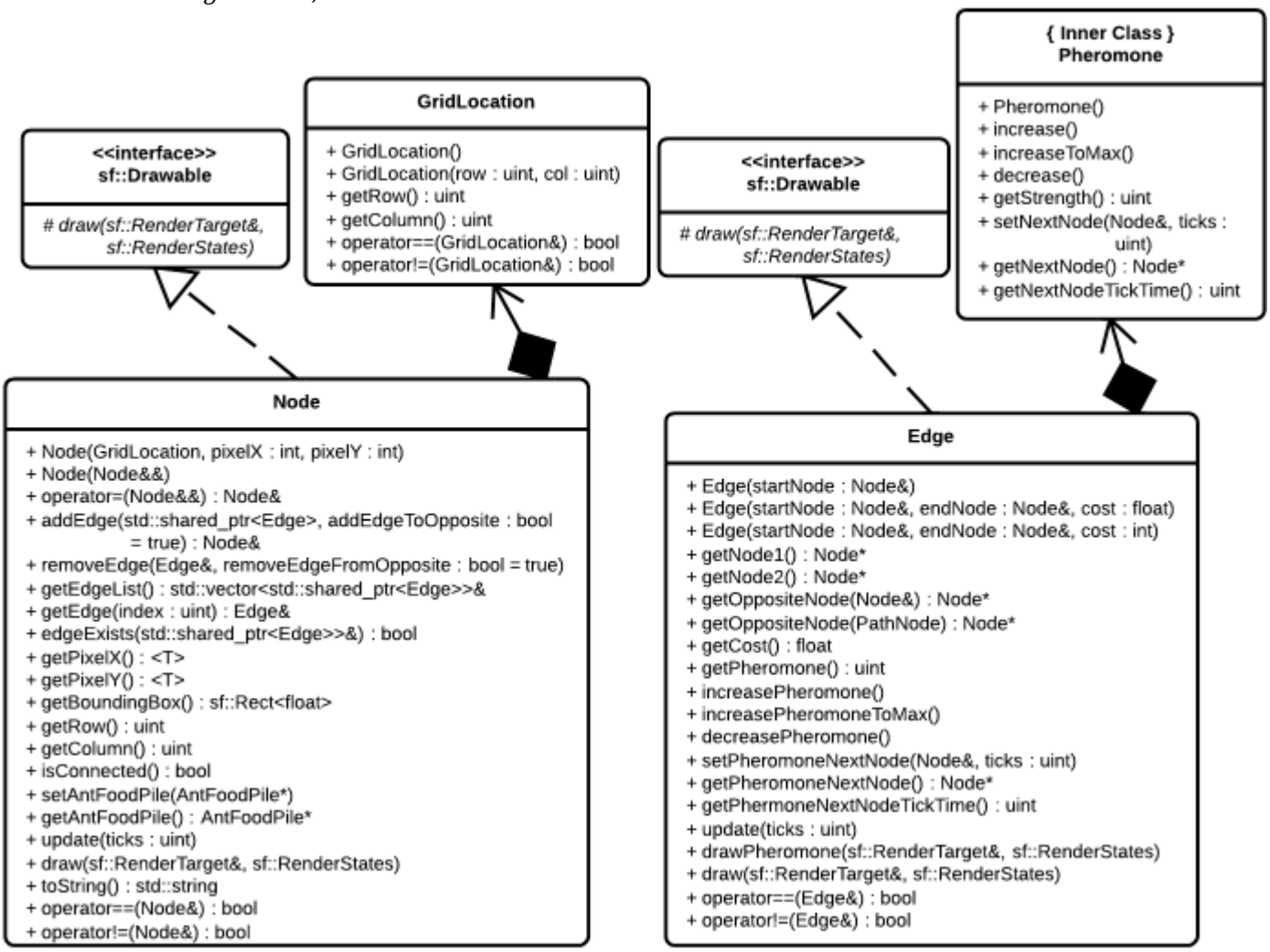
The Simulator class runs the ant simulation. Its update method is called approximately 130 times per second by the AiSimApp. The update method, in turn, calls the update methods of the various simulation entities, including ants and edges (to cause the pheromones to degrade over time). This is limited by the current speed of the simulation, which can range from 0 to 120 ticks (update calls) per second, with the default being 60 ticks per second.

III-C. The World abstract class, and subclasses

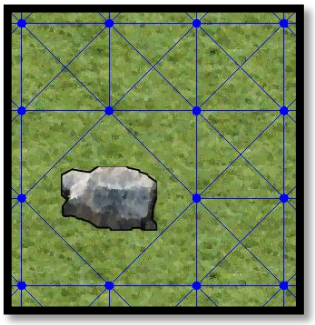


The World abstract class is the base class for world creators. I implemented two: TestWorld, which does nothing and is used for unit testing (because some constructors require a reference to a world), and RandomWorld, which creates a randomly laid out 900-node world, based on the seed provided to the Random class (in the util module).

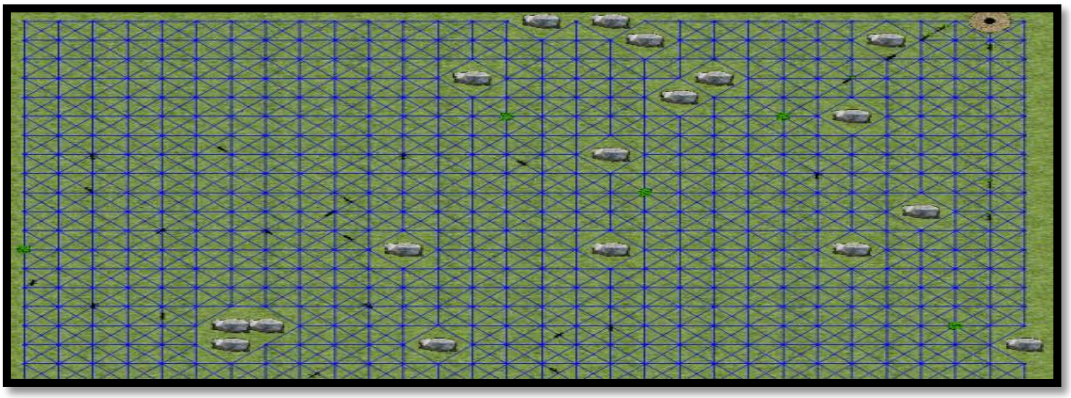
III-D. The Node & Edge classes, and related



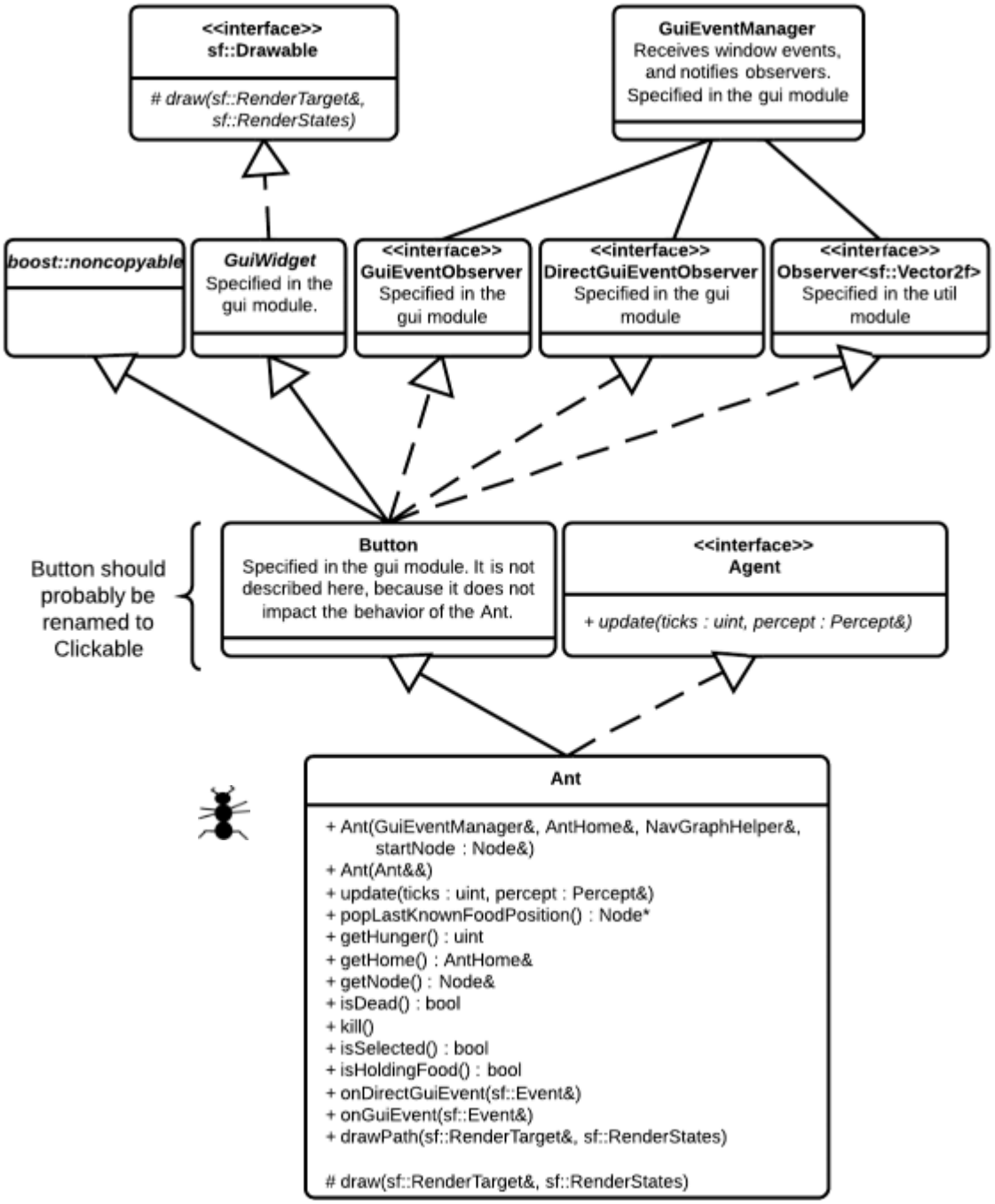
Nodes and Edges define the navigation graph, which in turn defines where ants can go, as well as where static objects are placed (ant hill, food piles, obstructions). Edges are also used to calculate A* g costs, and may contain the pheromones (if placed by an ant) that are used to give the swarm of ants the appearance of a higher-level controller.



The interfaces for these classes are relatively large, but the vast majority of the methods are const and simply enable agents and other simulation components to get state.



III-E. The Ant class



Ants are goal-based AI agents. They have a desire to stay alive, by eating when they get hungry, as well as to bring as much food to their ant hill as possible. Ants lay pheromones on edges when they are carrying food, and will follow those pheromone trails to food depending on their current goal. Ants do not have prior knowledge of the location of food, but they can remember up to three food locations when food is encountered. The location of the ant hill is always known.

Ants are controlled by goals, which are described in the next section. The ant goal mechanism uses the State pattern to switch out goals and subgoals as needed.

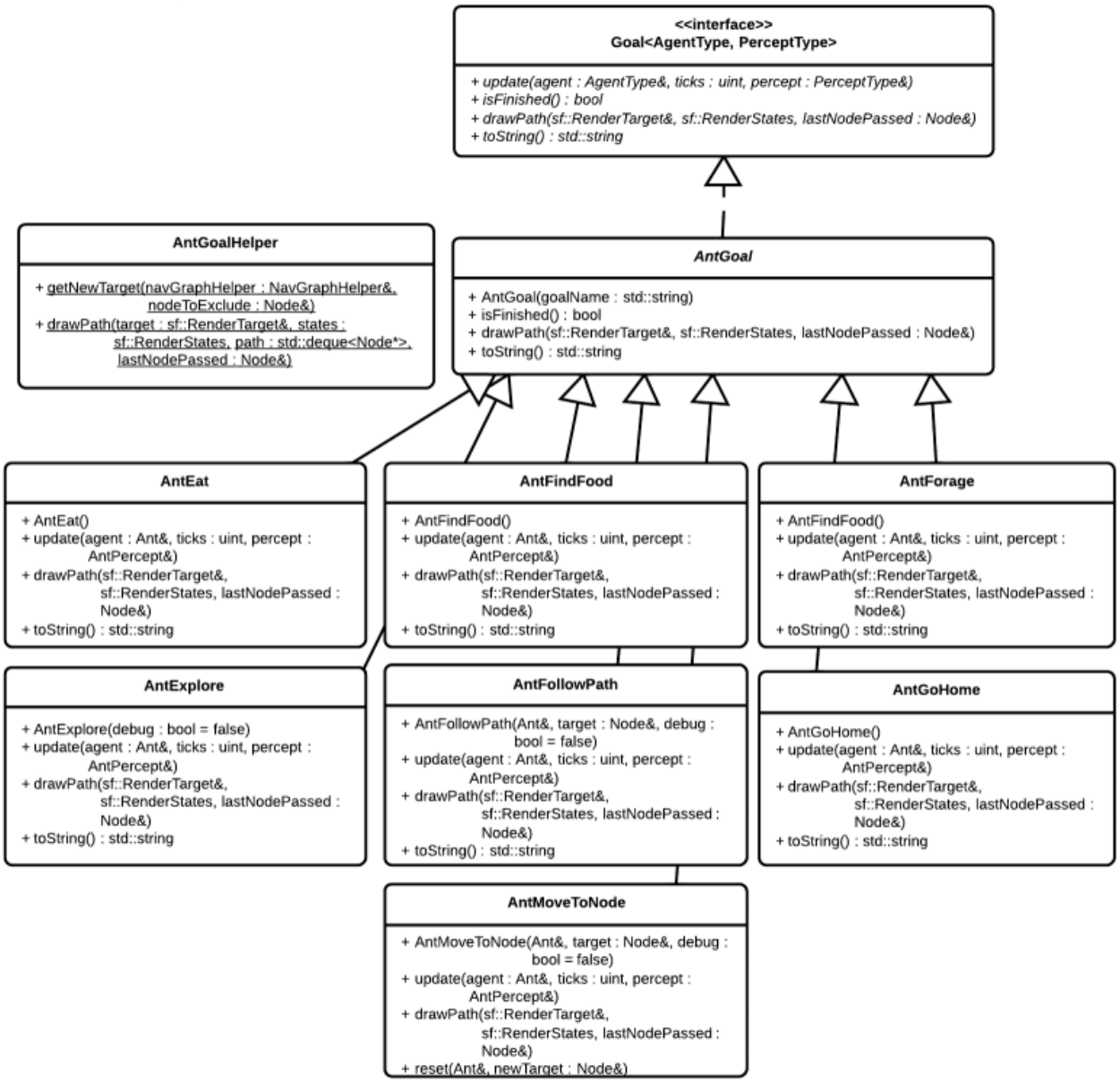
Ants use the Observer pattern to be notified of GUI events, such as when the user clicks on an ant (which causes information about that ant to be output to the console).

Pseudocode for the Ant::update method, plus related methods:

```
Ant::update(ticks)
   if (ticks > nextHungerIncreaseTime)
```

```
      increase hunger by 1
      if hunger >= maxHunger
         ant has died, so call the onDeath event
      end if
   end if

   if (goal is not finished)
      goal.update()
   end if

   else
      set new goal:
         if (ant is hungry)
            set goal to AntEat
         else
            99% of the time: set goal to AntForage
            1% of the time: set goal to AntExplore
         end if
      end set new goal
   end if
end Ant::update
```

III-F. The Goal interface, and the AntGoal classes



Goals control the ant. The current goal may be set by the ant, when the previous top-level goal has finished, or by other goals and subgoals. Ants have 7 goals and subgoals (listed below, from simplest subgoals to most complex top-level goals). Pseudocode is listed for goals that benefit from further explanation.

1. AntMoveToNode: Instructs an ant to move one node to an adjacent node. This is the simplest subgoal, and is used by many of the other goals.

2. AntFollowPath: Takes a target node, generates a path using A*, and moves the ant along the path. Uses AntMoveToNode to move between individual nodes.

3. AntFindFood: Instructs the ant to attempt to find food by doing the following:
   - Check the locations of the three most recently found food piles, starting with the most recent location
   - If that does not work, select a series of random nodes in an attempt to find food or pheromones
   - If at any time pheromones are encountered, follow that path

   Uses the AntMoveToNode subgoal.

   Goal pseudocode:

```
AntFindFood
   // Ensures that the ant switches goals after a while when hungry. Otherwise,
   // it could end up starving while foraging for food, instead of searching
   // for food to feed itself.
   maxSearchAttempts = 20

   if (food was found or the number of search attempts exceeds maxSearchAttempts)
      finished = true
      return
   end if

   if (subgoal is finished)
      if (the ant's current node has food)
         foodFound = true
      else
         if (ant is hungry)
            increase number of search attempts by 1
         end if

         set new subgoal:
            if (ant's current path is not empty)
               if (last edge had pheromones)
                     set path to follow the pheromones
               else
                     follow existing path
               end if
               set subgoal to AntMoveToNode

            // The ant holds the 3 most recently found food positions in a
            // circular array (organized as a stack, so most recently encountered
            // can be popped off).
            else if (ant's last known food position is not empty)
               if (last edge had pheromones)
                     set path to follow the pheromones
               else
                     set path to last known food position
               end if
               set subgoal to AntMoveToNode

         // Ant does not have an existing target, and ant does not know where any
         // food is located, so explore a random path in search of food & pheromones.
         else
```

```
            if (last edge had pheromones)
               set path to follow the pheromones
            else
               set path to random target
            end if
            set subgoal to AntMoveToNode
         end if
      end if
   else
      subgoal.update()
   end if
end AntFindFood
```

4. AntGoHome: Instructs the ant to return to the ant hill. Uses the AntFollowPath subgoal.

5. AntExplore: Instructs the ant to explore random paths, noting the location of any food it finds. Pheromones are not followed when this goal is being pursued. The purpose of this goal, which is selected 1% of the time when a new top-level goal is needed and the ant is not hungry, is to allow a small number of ants to periodically explore for additional food sources, before the current food source runs out. This goal uses the AntMoveToNode subgoal.

   Goal pseudocode:

```
AntExplore
   if (subgoal is finished)
      if (path is empty)
         target = random target node
         path = A*(currentNode, target)
      end if

      if (path is empty)
         finished = true
      else
         target = path.pop()
         set subgoal to AntMoveToNode
      end if

   else
      subgoal.update()
   end if
end AntExplore
```

6. AntForage: Instructs the ant to search for food, and bring the food back to the ant hill if food is found. A pheromone trail is released while the ant is holding food. This goal is selected 99% of the time when a new top-level goal is needed, if the ant is not hungry. Uses the AntFindFood subgoal.

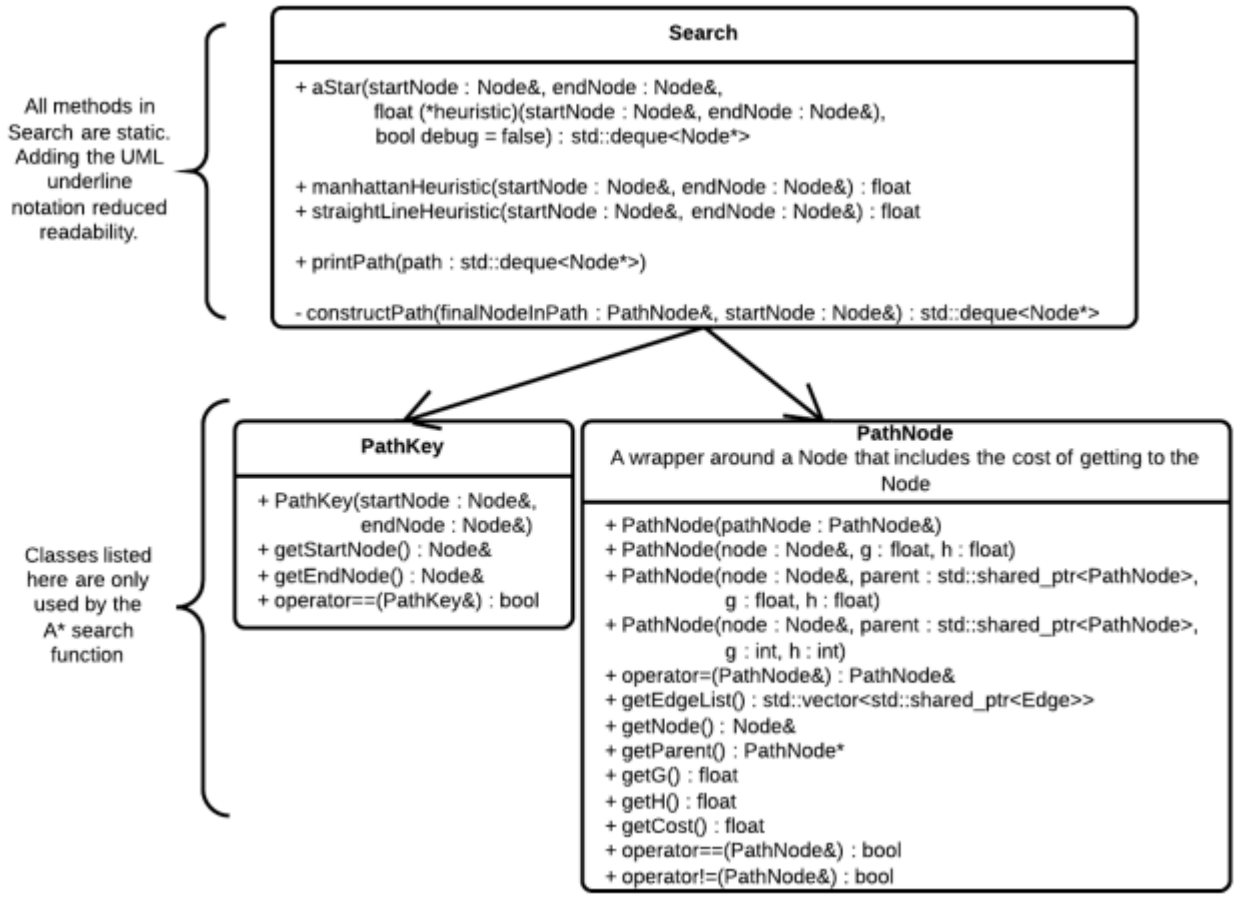
   Goal pseudocode:

```
AntForage
   if (subgoal is not finished)
      subgoal.update()

   else  // The subgoal has finished.
      if (subgoal is AntFindFood)
         if (ant's current node has food)
            take food from food node
            // A circular array of the 3 most recently found food piles.
            ant.lastKnownFoodPosition.push(foodPile)
         end if
         set subgoal to AntGoHome

      else if (subgoal is AntGoHome) // Ant has reached its home.
         if (ant is holding food)
            add one food to ant hill
```

```
            ant is no longer holding food
         end if
         finished = true
      end if
   end if
end AntForage
```

7. AntEat: Instructs the ant to find food for consumption. The ant first goes to the ant hill, using the AntGoHome goal. If no food is available there, it uses the AntFindFood subgoal to search for food in the three last known food locations. If no food is found in those locations, the ant will randomly search locations for food and pheromones.

*III-G. A**



My A* method takes a heuristic function pointer, which allows the heuristic to be changed at runtime if needed. Both Manhattan and Straight-line heuristics were implemented, because originally I was only going to allow four degrees of movement, which implies using the Manhattan heuristic. I later decided to increase movement to 8 degrees.

Pseudocode for heuristic functions:

```
Manhattan Heuristic
   rowDifference = abs(startRow - endRow)
   columnDifference = abs(startColumn - endColumn)
   return (rowDifference + columnDifference)

Straight Line Heuristic
   rowSquared = (startRow - endRow)^2
   columnSquare = (startColumn - endColumn)^2
   return squareRoot(rowSquared + columnSquared)
```

Pseudocode for A* and constructPath private method, which it uses:

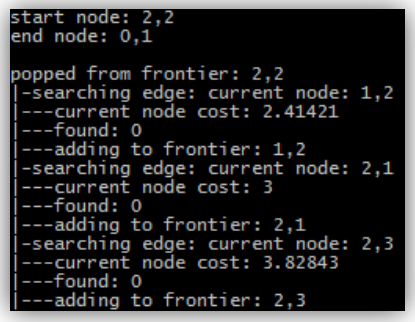
```
A* (startNode, endNode, heuristicFunction)
   if (path from startNode to endNode previously found)
      return path
   end if

   Declare Variables:
      frontier : Priority Queue
      searched : Set

   frontier.push(startNode)

   while (frontier is not empty)
      lowestCost = frontier.pop()

      if (lowestCost == endNode)
         path = constructPath()     // See below.
         add path to hash map of previously found paths
         return path
      end if

      for each edge in lowestCost.edges
         if (edge is not in searched)
            edge.oppositeNode.h = heuristicFunction(edge.oppositeNode, endNode)
            edge.oppositeNode.g = edge.cost + lowestCost.g
            edge.oppositeNode.parent = lowestCost

            frontier.push(edge.oppositeNode)
            searched.add(edge.oppositeNode)
         end if
      next edge
   loop

   // If no path could be calculated, return an empty path.
   return empty path
end A*


constructPath(finalNodeInPath, startNode)
   Declare Variables:
      path : Deque
      currentNode = finalNodeInPath

   while (currentNode exists and currentNode is not startNode)
      path.push(currentNode)
      currentNode = currentNode.parent
   loop
end constructPath
```

I currently cache the results of the A* calculation in a hash table, because of the O(1) search time. However, it would be interesting to measure whether this actually improves performance, due to the loss of locality of reference. Regardless, with the current implementation, the simulator spends between 0.8% and 2.4% of its time in the A* method (including methods used by A*), according to the performance profiler, so the A* search is not a bottleneck. The range of times is due to differences in world maps: the more food there is, and the easier that food is to find, the fewer calculations need to be done by A*, since following pheromones does not require an A* search.

IV. Testing & Verification

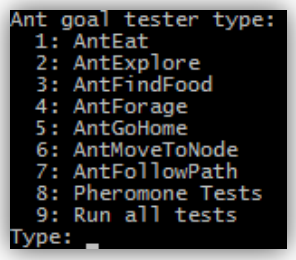
Because of the large scope of the project, testing was highly integrated into the development process from the start. This gave me the confidence to refactor as needed, and resulted in a smooth final debugging process.

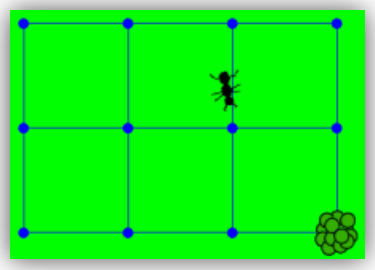
123 unit tests were written in Visual Studio, using the Microsoft Unit Testing Framework, and another 23 JUnit tests were written in eclipse to further test the Java implementation of my A* algorithm, which I used for the Wumpus World project. The JUnit tests uncovered issues with my original A* algorithm that my C++ unit tests hadn't uncovered, because the layout of the navigation graph used by the Wumpus World is more complex than the layout of either my hand-created test graphs or generated graphs.



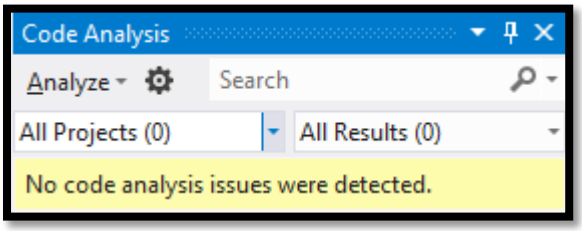
In addition to the unit tests, I implemented three test applications:

- GUI Tester: Tests gui implementation & related aspects, including various user input events, observer pattern implementation, drawing items to screen, etc.
- A* Tester: Text-based output of paths generated by A*. This allowed for a deeper level of debugging information to be available to me than the unit tests provided.
- Ant Goal Tester: Seven test agents were implemented specifically to test each individual goal. This was invaluable in debugging the goals, and in verifying that I had implemented my design properly.





As an additional check, I ran my code through Visual Studio's static code analysis tool periodically, though this uncovered very few issues.



Throughout the semester, I developed the project on two PCs with different setups. This helped ensure that the code is compilable on other PCs. To verify this, once the project was complete I installed Visual Studio 2012 Express on my wife's Windows 7 laptop, which had never had a compiler installed on it. The code compiled successfully without changes.

Finally, I had friends and family test the binaries to ensure that it ran on various Windows setups (see Appendix A). In addition to uncovering errors, I wanted to be sure that no installations would be needed, such as the Visual C++ Runtime. This testing passed without any problems.

V. Appendix A: Running the Simulator

A Windows binary is included with this project. There are no special requirements or additional installations needed to run the application, though the Canfield Ant Simulator folder structure (/res, plus the three included dll's) should not be changed. The application was successfully tested on the following operating systems:

- Windows 8.1
- Windows 8
- Windows 7
- Windows Vista

Simulation controls:

Space	Pause/Unpause
+	Increase simulation speed
-	Decrease simulation speed
1	Show/hide navigation graph
2	Show/hide pheromone strength
3	Show/hide ant paths
4	Show/hide dead ants
5	Show/hide background
Mouse wheel	Zoom in/out
Arrow keys	Move screen up/down/left/right (only when zoomed)
?	Display these commands in the console
Escape	Exit

The "Canfield: Ant Simulator" window must be active for these commands to work. You may need to click on the window's title bar to activate it.

VI. Appendix B: Compiling the Code

The C++ source code, plus compiled libraries, are included with this project. Additionally, a Visual Studio 2012 solution file is included, which should be used when compiling, since all settings are properly configured. The project was compiled successfully on three different machines using the following:

- Visual Studio 2012 Ultimate (Windows 8.1)
- Visual Studio 2012 Premium (Windows 8.1)
- Visual Studio 2012 Express, fresh installation (Windows 7)

Earlier versions of Visual Studio will not be able to compile the project, because C++11 features were used.

In theory, the code was written to be cross-platform. However, this was not tested, and all libraries (SFML, POCO, Boost) will need to be recompiled on the target OS. Similarly, other Windows compilers may work, if they support C++11, but the libraries will need to be recompiled.

VII. References

[1] S. Russell & P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd edition. Upper Saddle River, NJ; 2010.

[2] M. Bukland, *Programming Game AI by Example*. Sudbury, MA; 2005.

[3] B.C. Bridger, Chris S. Groskopf, "Fundamentals of Artificial Intelligence in Game Development," Proceedings of the 38th annual on Southeast regional conference, April 7, 2000.

[4] Christoph Salge , Christian Lipski , Tobias Mahlmann , Brigitte Mathiak, "Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games," Proceedings of the 2008 ACM SIGGRAPH symposium on Video games, August 09-10, 2008, Los Angeles, California.

[5] T. S. Ray, "An Approach to the Synthesis of Life," *Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen (eds.), Addison-Wesley, Reading, MA, pp. 371-408.

[6] Amit Patel, *Amit's A* Pages* [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/

[7] A. Dussutour, S. C. Nicolis, G. Shephard, M. Beekman and D. J. T. Sumpter, "The role of multiple pheromones in food recruitment by ants," *The Journal of Experimental Biology*. Aug 2009.

[8] M. Mizunami, N. Yamagata and H. Nishino, "Alarm Pheromone Processing in the Ant Brain: An Evolutionary Perspective," *Frontiers in Behavioral Neuroscience*. vol. 4, article 28, June 2010.

[9] Peter Miller, "The Genius of Swarms," *National Geographic* [Online]. Available: http://ngm.nationalgeographic.com/2007/07/swarms/miller-text

[10] Simple & Fast Multimedia Library: http://www.sfml-dev.org/

[11] Boost C++ Libraries: http://www.boost.org/

[12] POCO C++ Libraries: http://pocoproject.org/

[13] Paul Bourke, "Texture Library" [Online]. Available: http://paulbourke.net/texture_colour/