**Preface**

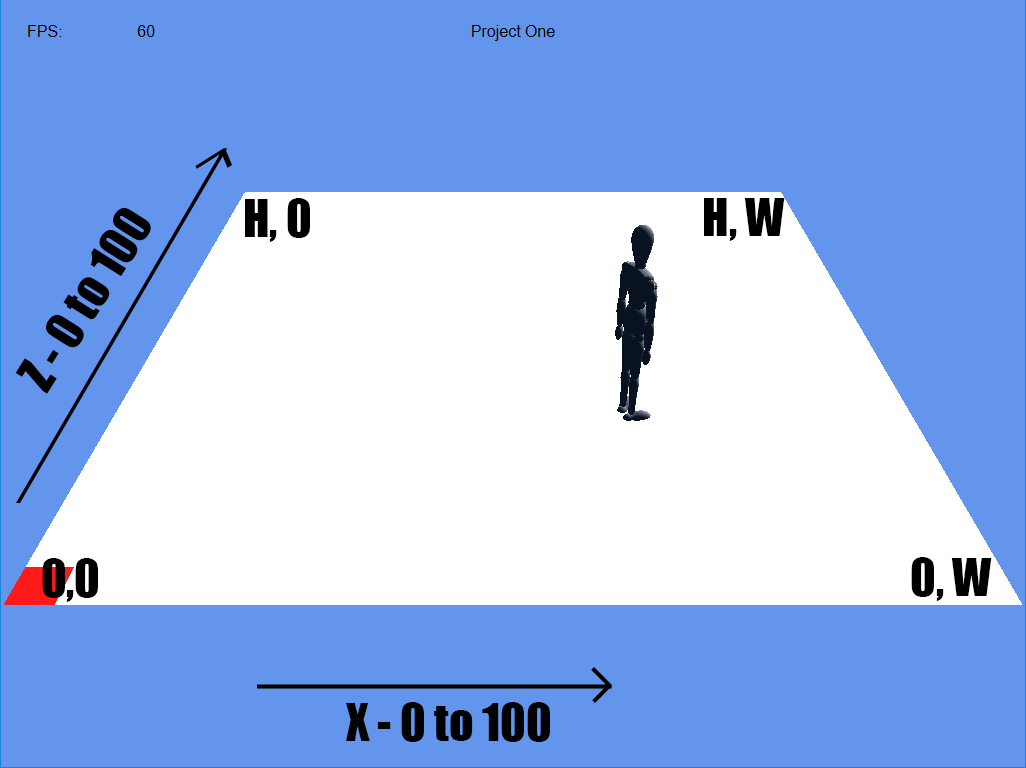
This is a brand-new framework that has only been internally tested, so it’s likely that you’ll find bugs, hopefully nothing major though. If you do run into issues you can email [the.dustin.holmes@gmail.com](mailto:the.dustin.holmes@gmail.com), and I’ll get back to you as soon as I can. Remember though, I’m not a TA, and I wrote this framework as part of an independent study. It was written in one semester, on top of GAM and all my other classes, so getting things up and running fast was a priority. So, everything may not be as fully featured as it could, because I implemented only what I needed; it’s not a full game engine.

As you work on your projects, try to take note of any questions you have regarding the framework and the projects themselves, that way we can build a FAQ over time.

**Framework Basics**

All three projects run in the same framework, in fact you can switch between them at runtime with the corresponding F key (F1 for project one, etc). In Source/Framework/Global.h you can define which project to load initially.

By default, the map area is 100 x 100 units in world space and is independent of the size of the map. So, a map may be 40 x 40 or 20 x 20 cells, but both will be 100 x 100 world units. To illustrate:



I wrote a VERY simple message broadcasting system just for decoupling a few of the systems, but I would advise against using it for anything more in each project. I’m also not a graphics programmer, so my rendering systems are pretty basic and very specific.

**Project One**

Behavior trees are still built using the tool Chi-Hao created for the previous framework but are included in a slightly different manner. The tool will generate .bht files in the BTResources folder that need to be placed in Source/Student/Project\_1/Trees. The Nodes.def and Trees.def files also need to be placed in Source/Student/Project\_1. After creating your .h and .cpp files for your nodes in their respective folders, update the NodeHeaders.h file as well. As a side note, the class name of the node needs to match the name in Nodes.def including the C\_, D\_, or L\_ prefix.

The blackboard for each agent can be populated with almost any data type, any non-usable types will trigger a static assert instead of impossible to read template errors at compile time. Values are set and retrieved using a string key, examples are in the nodes L\_CheckMouseClick and L\_MoveToMouseClick. If your behavior tree ends up calling get BEFORE set, it will trigger a break point in Debug and return a default constructed value in Release.

Just like in the example nodes, if you want your node to display its name on screen while it’s executing, be sure to add the display\_leaf\_text() call somewhere in the on\_update function.

In the previous framework, behavior trees and nodes were implemented in a very performance and memory conscious way. However, I chose to implement them in a much more straightforward and debuggable way. So, each instance of a behavior tree is separate from each other, as well as the nodes. Each node inherently knows what agent it is affecting, and the tree is ticked from the root node to the active leaf nodes every frame, so the full callstack is available.

Nodes utilize the Curiously Recurring Template Pattern (CRTP), so if you have any of your nodes inherit from each other you will need to add another layer of CRTP between them. If that doesn’t make sense or sounds complicated, just stick with inheriting from BaseNode<>, just like every example node.

**Project Two**

The starting point for project two are the Source/Student/Project\_2/ files. There are fairly detailed comments in each function that needs to be implemented, but you can also check out Source/Framework/Terrain/Terrain.h to see what functions you can call to get info about the map.

There are quite a few tests you can run, and you can cycle through them with the “Next Test” button on the left side. After the tests have been run, a text file will be created in the Output folder with a detailed breakdown of the results of each test. A summary will also be printed on the bottom of the screen. If any tests failed, the last one that failed will be recreated on screen and a message about why it failed will be at the bottom of the screen. If there were multiple failures a “Next Failed Test” button will appear so that you may cycle through them. The same information is available in the text file, and failures will be sorted to the top of the file. The diagonal, smoothing, and rubber banding tests have to be verified visually so a screenshot of each test result will be generated in the Output folder as well.

I highly recommend NOT running the speed test in Debug mode as it will take quite some time to complete. Get your algorithm working and passing all the test before you start attempting the speed test!

**Project Three**

Same as with project two, but just one file: Source/Student/Project\_3/P3\_TerrainAnalysis.cpp. Each analysis type can be toggle independently of each other, so you can have multiple running at a time. The “Ticks Per Sec” slider is how many times per second each non-static analysis is ran, and they are staggered to spread them out over a full second.