Poggers - Starcraft II AI Bot

Team members: Ahzam Ahmad, Erik Ligai, Mackenzie Hauck, Thomas Chan

Motivation

Our motivation was to learn and apply the concepts of game AI programming in a real world game. With this, we hope to develop a bot that is able incorporate path finding, resource gathering, and target selection in order to play against other game AI and win.

Strategy

The team first implemented a modified tank strategy that would rush the the enemy base once 3 tanks were built along side with marines. But after many issues of having the program deadlock on the bot when building structures, it was decided that this build was not as effective as thought. It was finally decided to change the strategy so that the bot would build six barracks and produce only marines and slowly send waves of marines to attack the enemy base while

kiting then retreating. Our bot first sends one SCV to scout for an enemy position then while this is happening, another SCV builds one supply depot around the ramp and then another builds a barrack to slowly block off the ramp. To stop any enemy scouters from entering our base, the barrack trains one marine to defend the ramp area. While the marine is being trained, the command center will upgrade to an orbital command center. After the supply depot and barrack near the ramp is built, three barracks will be built simultaneously. After that is done, the remaining supply depot will be built to finally close off the wall to block the ramp. Once that is done then the last two remaining barracks will be built and all barracks will begin to start producing marines. When there are 14 marines trained, all marines will rush the enemy base and attack. When attacking the marines will group up and move as a unit to avoid having the units being killed off easily. Kiting was implemented to increase the survivability of the attacking marines which caused more disruption for the enemies base. If six units are left, they will retreat and rebase with the rest of the new marines that have been produced and then it will continue to send more and more marines to attack. With this approach it staggers the enemies building

economy, and in the end the enemy will be forced to surrender due to the fact of lack of army units and will hopefully be overwhelmed by all of the bots marine units.

Implementation

Early on in the project as the team were learning more about C++ and the Starcraft II API we mostly just used a bunch of if else statements to decide what our next move should be. For instance, we would check if we had built a barracks during every call in OnStep() which is a little wasteful when our strategy at the time only required one barracks. Further, if we ever decided we wanted to have a single barracks in the early game, and then transition to two barracks later on the logic would start to get messy with many booleans like barracks1_built = false.

Roughly halfway through the project some team members started to research better ways to implement the AI to not be so dependent on all of these checks, while other group members continued to improve our initial bot and learn the API better. We landed on using a hierarchical state machine (HSM) approach due to its isolation between states. This isolation meant that we could theoretically define all of our states up front and then each team member could work on implementing these states on their own since the logic for moving between states would already have been decided. This separation also made it easy for us to split up each state into its own file and better organize our code.

The actual implementation of the HSM was provided by the Boost Statechart library. This library works by having the user create several structs inheriting from the simple_state class for each of the states in our bot. We also need to define a StateMachine class that inherits from the state_machine class in the library. The StateMachine class is what holds all of the state structs and handles transiting to another state.

To control the state machine we send it events. Similarly to the state definitions, events are just structs that inherit from the state_chart::event class. Since events are just structs, we can also add any data associated with the event to it.

Since this is a hierarchical state machine, each state can have any number of child states. Each child state can access the data of its parent by calling a function called `context`. The base state machine class can be seen as a parent of all states, so every state can access the context of StateMachine. The Observation and Actions interface are hence stored in the state machine (as function pointers) to give all states access to them.

Example of getting the Observation interface from the state machine:

```
auto observation = context<StateMachine>().Observation;
```

We can then call `Observation()` normally, just like in the example bots.

The biggest advantage of using an HSM is that we can defer generic events (like an idle worker unit) to a parent state. The statechart library works by searching for a state that can

handle an event in its event queue starting from the current state and working its way up the parent states until one matches.

Another advantage of the HSM is that we easily add new strategies to our bot. We just need a condition to check to see if we should move to the new state. For example, we could use scouts to not only see what race the opponent is, but also see what units they have and then move to a strategy that can counter them.

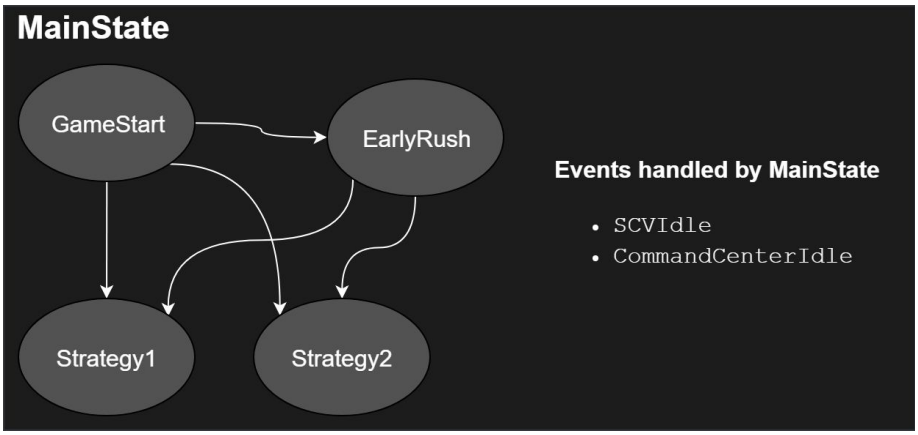
Example of the `GameStart_Refinery` definition in a header file:

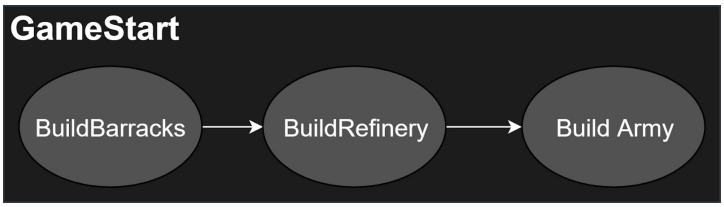
```
struct GameStart_Refinery : sc::simple_state<GameStart_Refinery, GameStart> {
	GameStart_Refinery() { cout << "GameStart_Refinery" << endl; }
	// ...
	// handler functions for events
	sc::result react(const StepEvent& event);
	sc::result react(const BuildingConstructed& event);
};
```

Example of the function call used to transit to a state from a different state:

```
return transit<GameStart_Refinery>();
```

Below are some examples of how our states were organized, and the hierarchical structure can be seen.





In the end we decided that this implementation was too difficult to use. So instead we based our code off of this concept by using the sample tutorial bot code as a base and we implemented stages in our step counter. The stage would only advance if all conditions of a stage were met. One of the implementations that was integrated was hard coded location points for certain buildings. This way we were able to build walls near the ramps to block off scouters or

enemies units. With this included. We were able to avoid any deadlock issue that had occured since everything had a certain location to build at. Since everything had certain locations to build at, building multiple units was easily implemented. The bot also has grouping of marine units implemented. With this feature this would increase the chances of winning since the units move as a pack and attack together instead of slowly building up and having less damage at the start of a battle while attacking as a pack allows for a strong amount of damage to be outputted. Paired with the implementation was kiting, kiting caused the units to move back from enemy units to avoid taking damage from melee units while being able to output constant damage to the enemies units.

Challenges

The biggest challenge that was presented was of how poorly documented this API was. There was difficulty setting up this starcraft API and it seems that the documentation for setting up in a Windows or MacOS system was currently out of date. Many hours were spent trying to figure out why the game was not compiling. When creating the bot every member had issues with setting up since half the team was using Windows and the other half MacOS. Since Windows had different setup than MacOS, separate documentation was needed. Another challenge was learning the starcraft API, the documentation online was every poor as it barely gave any documentation of the API, in order to overcome this challenge, example bots in the starcraft API were looked at to understand how to code with this API. When starting this project, none of the team members had any idea how to play the game. The game had a steep learning curve as it required us to learn a new style of gameplay which had certain build mechanics that we had to learn. We got most of our early game working with the state machine, but had lots of inconsistency with vespene gas harvesting. Sometimes the refineries would not get the correct number of SCVs to operate them, causing our state machine to deadlock. Because of this, we decided to fall back to our original architecture since it was more consistent in reaching the later stages of our strategy, and the deadline for the project was fast approaching. Looking back we should have picked a simpler architecture that was more forgiving and easier to understand and debug. We wasted a lot of time debugging state machine errors during compilation, and deadlocks that could have been better spent focusing on implementing the strategy. We still spent a fair amount of time on the state machine, hence its inclusion in this report, there were just some last minute bugs that we couldn't quite fix before the deadline that severely hampered our bots effectiveness.

Without getting the state machine code to work, changing our architecture and implementation in turn caused the team to change the strategy used in in the beginning. With this new strategy, a new set of challenges were needed. The team had to fix the issue of having a structure deadlock which would cause the stages to fail and make the bot stuck in a loop which prevented our bot from advancing and would cause us to lose against other AI. A solution to fix this was by getting hard coded points in the game which allowed the bot to avoid deadlocks and for it to have instantaneous building instead of having things build one by one. The team had to learn how to implement the kiting features alongside grouping while attacking. This posed as a challenge since it was a heavy concept to do on such a short notice.

Future Work

In the future we plan to fully incorporate the state machine architecture. This way we could implement a feature that would change the build order based on the enemies build. With this the bot could change states and start building a different build order based off of the data from the enemy team. Currently without the state machine structure implemented, features that would be added to the bot help the bot survive if the strategy did not work. The bot would be able to survive into late game and switch into a different strategy. This strategy would include building a engineering bay and upgrading the marines so they can take on much larger units late game. The strategy would have to also include counter measures to avoid enemy aircraft rush tactics. Another implementation needed is using the orbital scan to detect if the enemy is attackable. The orbital scan would help determine how many units they have and therefore determine if the bots can take down or disrupt the enemies base.