Artificial Intelligence in Real-time Combat Games

Braiden Kindt
CIS 203 : Artificial Intelligence
Dr.Wang

Table of Contents:

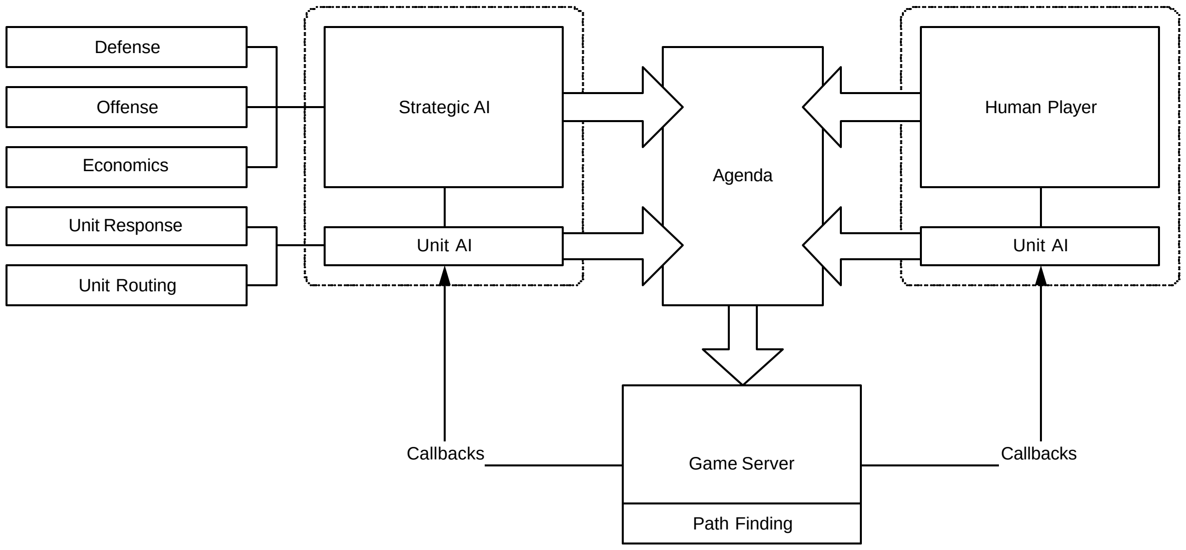
Section 1: Introduction:

Artificial Intelligence in real-time combat games has traditionally been very poor. This document will design a new artificial player. Real-time strategy offers some unique AI problems. Unlike turn-based games, the AI must meet at least soft real-time deadlines. Additionally, it must do so without full use of system resources. The first section of this document will outline my design approach. It will suggest a multi-tiered design meant to soften deadlines of more complex algorithms. The first tier will be rule base, implemented like any other approach in real-time games. The higher tier will try to apply some real AI. It is in this tier where learning will be of particular interest. What defenses are effective? How should resources be allocated? When is it appropriate to attack? These are the responsibilities of the higher tier. After establishing an understanding of the system design, later sections will detail some actual AI techniques. I hope to introduce some new ideas in those sections.

It is the goal of this document to design, and not implement, an AI player. It is sometimes difficult to assess the effectiveness of this design. Although, for the most part, failure to meet deadlines should at worst create an average AI in this type of game. I have would liked to see how this AI would perform in a real game, but time constraints made that impossible.

Section 2: A multi-tiered approach:

How can complicated search provide timely response to problems in a real-time game? It cannot. It is not acceptable for a computer to watch its demise, while busy calculating a solution that might have been effective several minutes ago. It is important that while the computer "thinks" it still responds. This is the reason for implementing a tiered AI. The computer's AI will consist of two components: "Unit AI" and "Strategic AI." In Unit AI the computer, as well as humans, can assign default behaviors of weapons, vehicles, etc. The Unit AI, based on rules established by Strategic AI, will make immediate response to an attack. Unit AI is very quick; it makes decisions on a per unit basis. It is responsible for: telling a unit to fire or run away when being attacked, placing units on patrol, assigning units to guard another, etc. How does Unit AI know whether a unit should run or attack if an enemy is in range? It does not. Strategic AI is responsible for assigning roles to units. Strategic AI provides the big picture: build more units, more defenses, attack, etc.



Implementing a multi-tiered approach:

Threads will be required to implement this system. The highest priority thread, the server, will spend its time updating data structures. This amounts the physical movement of vehicles, calculation of damages, and everything else to actually run the game. The server, when it detects events, executes callbacks associated with units involved. It is these callbacks collectively that I've titled "Unit AI." In response to a signal they simply add events to the server for processing. Callbacks make their decisions very, very quickly based on data structures managed by "Strategic AI." Strategic AI runs as one or more threads. It is notified by event queues what has have taken place. It makes decisions like: whether a unit being attacked needs reinforcement, how many units should be sent in an attack, reposition of units for defense, etc.

Before introducing any AI, first an understanding of some the code is required. Particularly, how subsystems communicate. The server performs all its actions, moving, attacking, etc, from an agenda. Events are added to this agenda from both AI tiers. The server makes appropriate callbacks when necessary. The following are some of the callbacks unique to each individual unit:

```
typedef struct callback {
        int (*cb_underattack) (board_ptr gameboard, object_ptr me, object_ptr opponent);
        int (*cb_spotted)     (board_ptr gameboard, object_ptr me, object_ptr opponent);
        int (*cb_inrange)     (board_ptr gameboard, object_ptr me, object_ptr opponent);
        int (*cb_fired)       (board_ptr gameboard, object_ptr me, object_ptr opponent);
        int (*cb_failure)     (board_ptr gameboard, object_ptr me, object_ptr opponent);
        int (*cb_success)     (board_ptr gameboard, object_ptr me, object_ptr opponent);
        int (*cb_birth)       (board_ptr gameboard, object_ptr me);
        int (*cb_goal)        (board_ptr gamebaord, object_ptr me);
} callback_t, *callback_ptr;
```

These functions are responsible for two important things. They must update the agenda of server, causing a real response to event. They must decide whether or not to notify the Strategic AI. All of this must be done very quickly.

In addition to making callbacks, the server is also responsible for routing of units. Much attention will be given to this path-finding later. But, it will essentially use an A* path finding algorithm.

Strategic AI must communicate with Unit AI. It does so by modifying data structures used to make decisions in callbacks. It can also directly add events to server agenda. It can notify the Unit AI of events for which it wishes to be signaled. The strategy tier will have several threads responsible for decision-making. Three important categories of decisions exist: defensive, offensive, and economic. Later these decision processes will be discussed. Below is the structure used by callbacks, and modified by Strategic AI, to determine what immediate actions need to be taken:

```
typedef enum { patrol, standground, free, evade, work }  role_t;

typedef struct rules {
        union {
                coordinate_t rl_destination;
                unit_t target;
        } rl_attack;
        role_t  rl_role;
        rule_ptr rl_next;
} rule_t, *rule_ptr;
```

Notice that rules is a linked list, some call-backs, particularly cb_goal() will advance to the next node. This list can be a circularly linked list. If next is NULL, the current rule persists. Each rule contains a role, and possibly some movement for each vehicle. These relatively simple rules when influenced by callbacks can provide very sophisticated behavior.

A typical Response:

How does all of this work together? Suppose in a particular game the goal is to destroy some structure. This structure, like all units, has a set of callbacks associated with it. When an enemy comes within sight of the structure, a callback is executed. This particular callback notifies Strategic AI that enemies are near the base. Strategic AI knows (somehow) that it must defend that building, but takes some time to figure out. In the meantime, vehicles in the proximity have also received a callback when the enemies approached. These vehicles immediately respond. Later, when the Strategic AI learns it will lose if the building falls it can dispatch more units. It becomes clear, that one very essential role of Unit AI is to soften deadlines of the more complicate AI.

What else can Unit AI do? Suppose I wish to launch a frontal assault on an enemy base, but I don't want to engage units along the way. Attaching simple rules to the assaulting units makes this possible. I can assign an evade rule for half the path, once the units are close enough to base a new attack rule is in place. It is important to know that evade only causes a unit to run once it "sees" opponents. If units are -known- to be somewhere, it is the responsibility of Strategic engine to route around them.

Issues:

How well can this system work? Can it meet performance requirements? Occasionally failed deadlines in Strategic AI are acceptable, but there is some concern that the game could slow down processing callbacks of Unit AI. Even the case of a single unit approaching an enemy base, could result in thirty or more callbacks. Can the system be effective making fewer callbacks in situations like this? Performance concerns of this nature will be addresses later and callbacks will be reduced.

Does the multi-tier'ing really offer any added performance? Can it be effective? Is it too stupid? There is a concern that initial responses will be highly predictable. If game-play is quick enough battles could end before the computer makes any decisions. Regarding the problem, there really is no good solution. Methods like search are simply too slow. Strategic AI needs to learn and respond quickly or it will perform poorly.

Section 3: Game Server:

The game server has only two responsibilities impacting AI: callback reduction and path finding. Callback reduction involves intelligent decisions on the server's part to reduce callbacks when load is too high. This is primarily a scheduling issue, so I won't discuss it any further. It is sufficient to know that some redundant callbacks, primarily cb_spotted, may not be executed (although, at least one per invading unit will be made with "best effort"). Of much more AI importance is path finding. Path finding can be as simple as heuristic search. This section will focus on how this search is implemented, and how performance can be optimized.

Memory and Learning:

It is highly likely that many units will travel from one location to another. And it is unacceptable that a path be discovered for each unit. For this reason the search must maintain a list of know paths. It is clear that when traveling from point A to point B that if a path exists from those coordinates it will be used. What about a path near point A and point B? Rather then simply selecting an ok path and find paths to its start and finish, previous paths will alter the evaluation of nodes during a search. The following is a possible heuristic function:

```
int evaluate (coordinate node, coordinate  destination) {
        /* In general, the closest node is the best */
        int rank = distance( node, destination );
        /* Rank a node higher if it is known to cross near or through destination */
        rank+= known_path( node, destination );
        return rank;
}
```

Resolution:

Do I need to search one block at a time on game board? Some games define pass-ability based on groups of blocks. Therefore a 200x200 board could be divided into blocks of 4x4, where each 4x4 block was either passable or impassable. This could increase path-finding performance by almost 4 times. The disadvantage is that it reduces the capabilities of the "world."

Real-time Path-finding:

Can I begin to route units before the entire path is known? Is this a good thing? Many games seem to move units instantly and find the paths later. With a good heuristic function, this could improve response time. With a slow search it could cause units to travel far off course. The problem is not too severe since vehicles move much slower than paths are found. But, the algorithm seems very complex.

The system will not implement a move first think later approach. It is difficult to determine when it is ok to reverse a vehicles path, and when its current location is better then reversing. It is unclear if there is an efficient way to implement a live path search. In any case, this problem is only of importance when the system has not learned paths. Very early in the game it will learn paths and perform very quickly.

Training:

If performance is unacceptable, the system can be trained. Because finding relies on past paths, it can be provided with a graph before the game even begins.

Section 4: Unit Control:

Unit control is a collection of extensible rule based functions whose behavior can be modified by Strategic AI. Very little new is introduced in this layer. Events occur, the game server executes a callback, and Unit AI makes a response based on rules. What is unique is the ability of responses to change, either by replacing a callback with a different one, or modify structures mentioned earlier.

It is a design decision that this layer handles each unit independently. There are times when roles, for example “guard,” seem to implement easiest in this layer. But since no unit talks to another, it will not be implemented here. This is a “dumb” layer, it makes can make no decisions; it only follows rules. When a unit needs guarding it makes a high priority request to Strategic AI for backup. Strategic AI “decides” what units to send and how many.

The more this layer depends on Strategic AI, the more effective it can become. This is not without disadvantage however; dependence on Strategic AI requires much more computational time. What an appropriate reliance? Can it even be quantified? I don’t know, that is precisely why this layer is extensible.

Section 5: Strategic AI:

This sections details the real AI of the game. Everything else was done to facilitate its operation. The effectiveness of the computer depends greatly on what is done here. It has been easy so far to simply claim problems will be solved by a higher level, but now there is no higher a level. Responsibilities for Strategic AI include:

- Resource Acquisition: this includes the production of energy, mining of resources, sharing with allies, etc.
- Resource Allocation: what levels of available resources should be used for defenses, offenses, increased acquisition, construction capability, etc.
- Defensive Planning: what sorts of units are required, are stationary defenses sufficient, are there defensive vulnerabilities
- Offensive Planning: when is it effective to attack, what types of units are required, is there sufficient knowledge of enemy, from what angles can I attack, etc.
- Agenda Servicing: handling of events passed by lower layers. Should I defend a unit or building, who do I attack first if base under attack, etc.

Implementation Approach:

Several of the above mentioned jobs could be done using low priority threads. Solutions to some responsibilities emerge naturally while solving others. For this reason, I should be able to fulfill all responsibilities with only two threads: offense and defense. These two threads will run along with agenda servicing. They will be able to request the production of units, development of defenses, etc. Of course they will contend with each other for resources, and agenda servicing will deal with that. The actual design of each of these threads will be discussed later.

Agenda Servicing:

Agenda servicing responds to events in the system. Sources for these events come from two places: the worker threads just mentioned and Unit AI. Agenda servicing handles the complex decisions of Unit AI. Since a response has already happened in lower levels, this servicing can take some time to analyze a situation before responding.

Servicing Unit Events:

Much of the analysis done at this level is still rule based. For example, dispatching the units most effective against an attacker. But, this level has the time to perform a search. It can evaluate the effectiveness of a response through simulation. Obviously it cannot

evaluate many alternatives, but perhaps a few. It can learn through reward an punishment the best responses to recognizable patterns. There are many patterns the computer can learn to recognize including: similar sorts of units, geographic positioning, etc. With this sort of AI it should be possible for the computer to learn, for example, that if a player is attacking with infantry and tanks, it is more effective to destroy tanks first.
Why not just establish a bunch or rules instead of learning? Perhaps for a single game it would be possible, but I have tried to develop for a very generic game. And it was a major goal of this design to be able to learn new "rules." It is very easy to beat a computer player if its response is entirely predictable.

Servicing Thread Events:

Events from the Offense/Defense threads will have already been planned. The server's responsibility is to decide which events, either from offense or defense should be serviced. Resources can be limited, and it is difficult to know where to allocate them. Resources should be allocated in opposition to other players. If a human player is devoting 90% of resources to offense, clearly the computer should dedicate a good portion to defense. If I was developing a "cheating AI," I could always look. But, I am not. With appropriate recon, the server might know whether to give defense or offense priority. This is not always the case. A better solution is to learn a player's habits, and to allocate resources appropriately. If I know a player is slow to develop offense, I want to learn to attack early. This is one case where learning, and particularly, learning, for each player who uses the computer, can greatly outperform rule based actions.

Other Responsibilities:

Up to this point everything has been event driven. That is the computer only takes action to events. Of course the computer cannot just sit idle when not being attacked. It is the responsibility of the agenda server to keep the computer busy. The computer should always be collecting resources; it should typically always be building units and buildings.

Offense Thread:

This process is responsible for the planning of attacks. This includes the actual strategy as well as information collecting. Its primary purpose is the simulate attacks. Based on simulations the computer can request more units, do recon, or attack. This thread, as well as defense, can communicate its intent through data-structures.

How does searching work? Attacks are actually simulated. It was an attractive alternative to simply evaluates some probabilities, but too much is overlooked. Simulations must be accurate enough to reflect the benefits of any strategy I'd like the computer to employ.

Can I really search? There is a concern that with so many variables the search tree could be enormous. Many factors contribute to an effective attack including: path(s) taken, direction(s) of attack, composition of assaulting force(s), wave(s) of attack. Clearly I cannot build an entire tree. Some factors will have to be ignored. Only unit composition and direction will be considered in the search. This means the computer must decided how to divide its forces and what targets they should attack. This is still a very complicated procedure.

How can learning increase performance? There is little recognizable to the computer with regard to direction of attack. But it should be fairly simple for the computer to learn what units are effective against an enemy. This could be as simple as learning a player's preference. Or as complicated as remembering that attacks are more effective when assisted from the air. Significant training will be needed before offense is intelligent enough.

Defense Thread:

It should be possible to reuse significant portions Offense code to assist in Defense. But, I cannot afford another computationally intensive process. For that reason I'll borrow a technique used in other games. Defense will request no movable units. After all, a force in direct opposition to the opponent is already being developed by defense. Instead Defense will concern itself with stationary defense include buildings and guarding units. Borrowed from other strategy games (Ages of Empire), the computer will remember where it was previously attacked and fortify those locations. Again, significant training should allow the computer to put defense in the correct locations.

Conclusion:

When I started this design, I had a goal to design a better AI. I am unsure if I have succeeded. The more time I spent, the more problems I found. Initially I was certain the only reason there hadn’t been a good AI was because it was more marketable to focus on graphics. But, it was truly difficult to design an AI that could meet all of my expectations. I am confident that I could develop this system as designed, but am uncertain if it would be able to meet my performance expectations. Contrary to my complaints, I still believe this system can perform better then many games I’ve played.