# Innovations Report
# Peter Dodds

## Unit level AI library
## research and development

**Abstract**
Artificial intelligence is used in modern computer games extensively such as real-time-strategies in which each unit must behave intelligently to the players requests. In this paper and accompanying designs and code I intend to explore the methods needed to develop the unit level AI in a real-time-strategy game, and the possibility of encompassing these into a single standalone library. I explored the issues of implementing algorithms such as A* pathfinding and steering.

## Introduction

In modern computer games artificial intelligence (AI) has become one of the most crucial components of success. Doug Lombardi, of Valve Corporation said “Everyone here agrees that the great AI work done for Half-Life 1 was key to the first game’s success.” in an interview with DriverHeaven. This is true of games ranging from the blockbust titles such as Half-Life 1, but also of independent developers such a Introversion's Defcon [Morris, 2008]. This carries as far as small time developers working on small scale games. AI can often be viewed as a complex and challenging subject to break into, putting off these smaller developers. In these cases the best way to increase the potential of any game would be to offer a set of AI functions which people working in the area can use as a basis of more complex systems. What I have found through research is that there are no free stand-alone AI libraries in wide distribution.

Initially I had intended to develop a library to encompass the whole field of AI in games but with just a little research it became apparent that this was an unachievable task. Instead I set out to research and experiment with creating the basis of a unit level AI system. Unit level referring to the AI that controls individual autonomous agents within a game world. Through researching the problems involved with this area I hoped to demonstrate the feasibility of creating an flexible external AI library.

This importance of AI is no more so than in real-time-strategy (RTS) games. Even if all participants are human players, all the units have to move intelligently and understand complex orders. I plan to concentrate on the challenges RTS create for AI. In theory AI within these games can be extended as far as to make agents complex individuals with independent control over they're priorities.

In addition I have concentrated upon to problems involved around 2D AI, though this doesn't mean that these techniques would not transfer into 3D.

The eventual intentions of this project was to develop the structure for a future AI library that could be implemented and developed in future projects.

## AI systems

Real-time-strategy (RTS) games are based around war games, and the structure of command involved with military-like operations. The actions in a game are based on a single order being passed down and interpreted by the next level. At the very top is the game objective, a usually fixed case that defines the winning scenarios (eg. destroy the opposing team). This is then passed to the player, computer or human, who calculates how best to go about achieving that goal. The next level of orders are then created, these are the strategic orders (eg. build defenses, attack enemy base). This may stay within the previous manager's jurisdiction, or could be passed down to a middle manager, a commander. The order is then split down once again to simple unit instructions such as “move to this location” or “attack this unit”. When the unit, the AI agent receives this it splits it down to the final level of orders, the most basics move and attack commands.

The mechanism of order storage and processing is the same for all levels of the process. The middle managers and even to some extent the player, can be seen as AI agents, like the unit. Each has different mechanics to carry out orders, but each processes them in similar ways.

Each layer in the order chain has personal goals to try and achieve, that affect the way in which the order is carried out. Taking the example of the unit, it may have a heavily weighted goal of staying alive. This may mean during particularly intense conflict, the unit may choose to disobey orders. Whilst this particular scenario may not work, though in the Total War [Creative Assembly, 2000]

series this can often happen causing rapid changes in the battles outcome. A more likely situation may be that an initial move order is converted into a series of moves but depending on the units goals it may favor one route over another.

There are a collection of algorithms that are essential to an RTS. These are the searching and processing of movement in the game world, regular problems for games. Navigation in games is a two stage problem: firstly the pathfinding through static map elements and secondly the navigation of smaller moving objects.

Pathfinding involves the searching of waypoints within a level. These waypoints give the search algorithm an indication of how passable the terrain is and are interlinked to give a network of points to be traversed. The search algorithm can be expensive, so using the most efficient method for the structure of the game terrain can boost the AI performance. The most common search technique in games is the A* search because it will always find a path if one exists. Also using a heuristic function to calculate an estimate of the distance to the goal and node based costs enables the algorithms behavior to be honed to the behavior of an agent.

An important note is that the A* algorithm may be the most common, but is not always suited. Often a combination of search techniques are used for different problems.

> "...you're searching only small data sets, like an AI state graph, or a dozen or so path nodes in a room, you can use Dijkstra's algorithm, which is particularly effective when all the AI's in the room will path find to the same node, since the algorithm will fill out data for the whole network rather than just the start and end nodes. Sometimes even a breadthfirst search is enough."
>
> Heyes-Jones, 2007

The application of a layered pathfinder would be difficult to implement in a for-all-cases library. It would require an in depth understanding to the game scale, construction and limits.

Navigation of dynamic objects, or steering, involves far more reactionary processing of the game world. Each agent needs to be able to react to changing environment, either by recalculating its path or steering, making small alterations to its movement to avoid colliding with objects. This doesn't use the waypoint system of the pathfinding algorithm, but requires information about collidable objects within the scene. The testing against collision objects is only intended to give an indication to the agent, and does not require the full library of methods associated with collision detection. Many of the techniques of avoidance are set out in Reynolds [1999]. A basic collision avoidance would be to test if anything occupies the volume created from the movement of the agent in the next *n* turns.

**Pathfinding using A* search**

**1. Theory**

The A* algorithm searches through a ordered graph network of waypoints, looking for a goal location. At each iteration the waypoints are divided into two separate groups, OPEN and CLOSED. The OPEN group consists of nodes that are candidates for examination. On the first iteration the OPEN group only contains one waypoint, the starting position. The CLOSED group contains those waypoints that have already been examined. Initially the CLOSED group is empty [Patel, 2008a].

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor): **
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

*Figure 1: Pseudo code of the A* search algorithm (Patel, 2008a)*

At the start of each iteration the best waypoint is selected from the OPEN group and this is then the current waypoint. From the current all the neighboring waypoints are processed and the cost of each is assessed, the current waypoint is saved as the neighbor's parent, and then they are placed into the OPEN group. Once all the neighbors have been processed, the current waypoint is moved from the OPEN group to the CLOSED group.

- At one extreme, if h(n) is 0, then only g(n) plays a role, and A* turns into Dijkstra's algorithm, which is guaranteed to find a shortest path.
- If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands, making it slower.
- If h(n) is exactly equal to the cost of moving from n to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.
- If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if h(n) is very high relative to g(n), then only h(n) plays a role, and A* turns into BFS.

*Figure 2: Effect of heuristic function on A* search (Patel, 2008b)*

The structure of the algorithm stays the same in most cases of it's use, what has the largest effect on the solution is the method of calculating the cost, *g( )*. This is used in A* to determine the advantage of picking one waypoint over the other from the OPEN group. Each waypoint stores this temporary

cost, and it is recalculate when each waypoint is assessed as a neighbor of the current. The calculation is specific to the game terrain, but would usually involve adding any terrain penalties and the movement cost for going between the current and the neighbor.

When the best waypoint is being got from the OPEN group, they are assessed on they're cost, but also in addition there is a heuristic function, *h( )*, which is intended to make an estimate of the distance from the waypoint to the goal point. The heuristic function gives the most control over A*'s behavior.

The heuristic function needs to be optimised for each game, and possibly to specific situations in a game.

**2. Implementation**

To get to full understand the A* algorithm and the elements needed to be give the user full flexibility I developed a pathfinding component as part of the library prototype. It is contained in AiPathfinder.(h/cpp), AiPathNode.(h/cpp) and AiPath.(h/cpp). The pathfinding algorithm inside the class AiPathfinder, and uses AiPathNodes as the representation of waypoints in game. Since the A* algorithm is very well documented I was able to implement it with ease. The main challenge was deconstructing it into parts which would be essential if a user were to want to customise the behavior of A*. It is intended that the core A* code does not change, but that the user specifies the cost functions.

Astar_GetSuccessors, Astar_CalculateCost and Astar_EstimateGoalDistance, represent the A* methods; get neighbors, calculate neighbor's cost *g( )*, and the heuristics function *h( )*. When the AiPathfinder is derived into a custom game pathfinder rather than having to reimplement the A* search, any of the methods can be replaced if needed. Otherwise they can be used in they're default setup.

The AiPathNode also contains data specific to the A* algorithm, temporary data is denoted with 'aStar_' before the name. The temporary data is usful stored in each waypoint, but for memory management purporses it is better to keep the variables to permenant or non-function specific data.

Both the AiPathfinder and the AiPathNodes have been implemented only to perform the A* search. As stated earlier, this may not be the optimal method of pathfinding for certain situations. With further development these classes and method should be expanded out into several seperate classes. If there were abstact base classes of AiPathfinder and AiPathNode with no specific search method implemented, but containing pure virtual method such as CalculatePath ( ), then the specific A* search could be confined to a derived class such as a theoretical AiPathfinder_AStar. This would implement CalculatePath and add the virtual methods associated with A*.

In addition this basic setup does not take into account many in game situations that would require more complex management of waypoints. In AiPathfinder there is only a single AiPathNode array, m_pathnodes. This means there is only a single set to test ever. Looking at the example of fog-of-war, the obscuring of areas of a RTS map because they haven't been visited or are not in any units visibity range. When deciding on a long distance path it may be preferable that some units stay inside safe areas, and so avoid areas obscured by fog-of-war. With a single list model there is no distiguishing bettween players, and so no way to label which waypoints are visible to each player.

A possible solution to labling waypoints would be to add data relating to the player inside of the AiPathNode. This would mean adding excess temporary data to a class that a game may wish to be writen to file. A better solution may be to keep groups for certain conditions, like player's visible locations, in AiPathfinder.

In the future some optimisation of the A* algorithm can be performed, primarily with the management of memory. This can be improved in some cases such as choosing different methods of storing data in OPEN and CLOSED groups, along with the organization of neighbors. These issues are discussed at length in Patel (2008a), notes on Set representation. The implementation of different data structures depend predominantly upon the operations carried out on the groups and the frequency of the operations.

**Steering and Collision Avoidance**

Once an order has been received by an agent it must be broken down into mechanical operations that the AI understands. Taking the example of a path generated by the pathfinder, the agent receives a list of waypoints to navigate. First the list is translated into a series of move commands, then these moves are executed by the agents steering. The steering component manages the input to the locomotion elements of a agent, which would be unique to an agents construction. This is the level of processing where agent receives direct information about the world, and acts upon it to primarily avoid collisions, but also to pursue and flee from other agents.

When placed within a hierarchy of motion behavior, pathfinding falls into the first layer, Action Selection. Additionally the agents broken down move commands are also in this top layer. The actually ongoing execution of the movement by the agent, and the processing corrections to the action where necessary, lies within the middle layer of motion behavior. Finally as stated in Figure 3, the final layer is the processing of the required movement into steps or animation cycles.

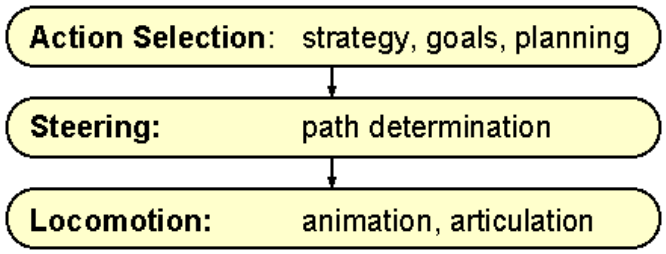



*Figure 3 : A hierarchy of motion behaviors (Reynolds, 1999)*

Reynolds [1999] develops many of the methods of steering for autonomous agents. These agents are based upon a simple vehicle model based around position, velocity and orientation. It is controlled by apply vectors forces to the agent, altering it's direction an speed. Though all of the agent behaviors are of interest to developing an all round AI agent library, of particular interest are the arrival, obstacle avoidance, separation, and unaligned collision avoidance behaviors.

The most basic is arrival and its related behavior, seek. Arrival, which can be used as the basic mechanic behind the move command, involves traveling to a single point and stopping without overshooting. Seek, which arrival is derived from, is the attraction towards a point. For the moving along a path, with or without sets of move commands, Reynolds [1999] has a path following behavior. This keeps the direction of the agent close to the line, allowing a more natural movement such as cutting corners.

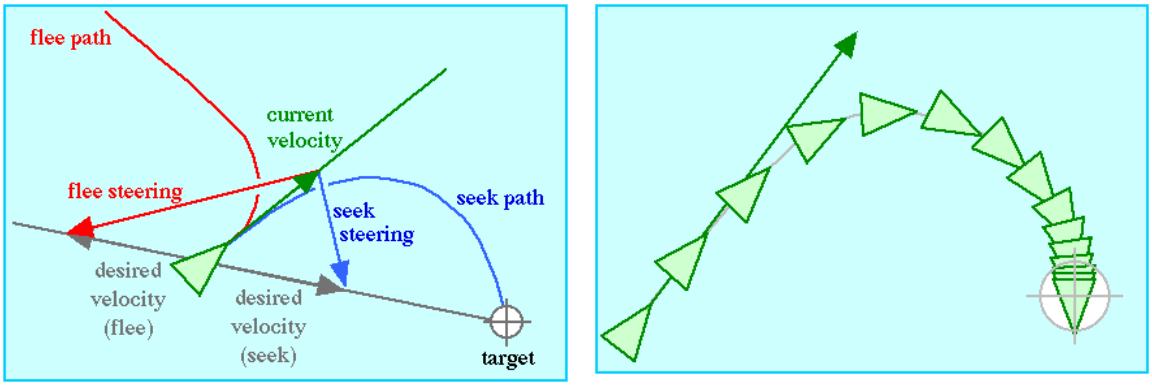



*Figure 4 : Seek/flee and arrival behaviors (Reynolds, 1999)*

Separation is part of a more general set of behaviors to do with flocking. It is intended to keep groups of agents from getting to close and risking a collision. If neighboring agents are within a certain radius they are used to calculate a repulsive force. This repulsion is in the opposite direction of the average position of the neighboring agents. This collision avoidance technique does not require any information on any of the agents collision objects. Note that none of the collision avoidance behaviors actually require any form of actual collision detection, and so can predominantly be estimations.

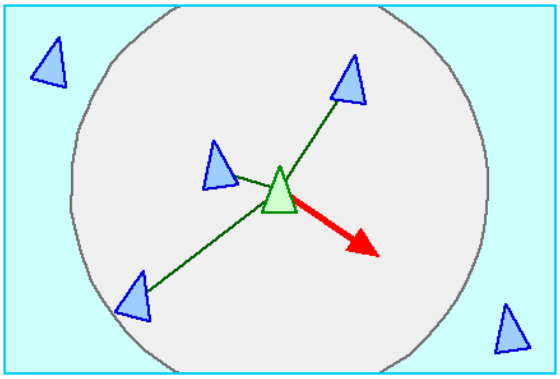
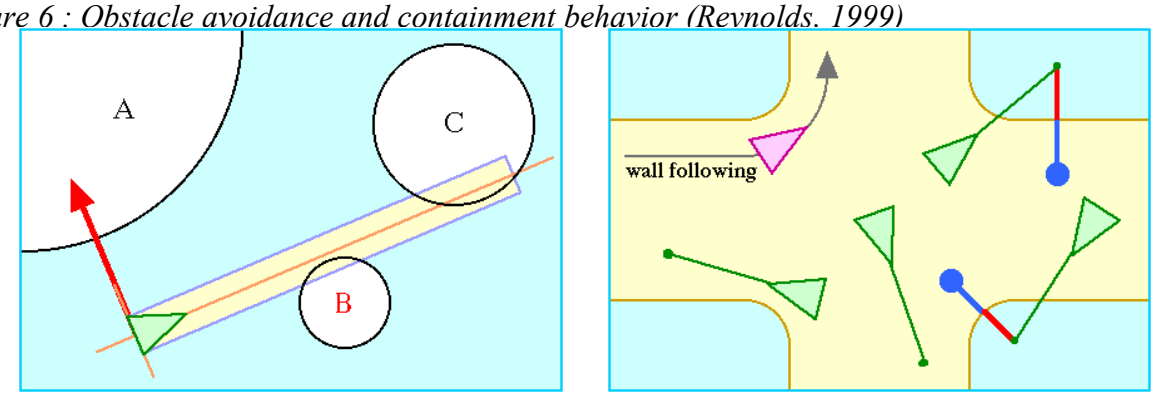


*Figure 5 : Separation behavior (Reynolds, 1999)*

Obstacle avoidance involves the detection of collisions with objects and agents in the game world. this concentrates on static objects that an agent can collide with. It is tested by testing if a volume or plane created by the future motion of the agent. If it crosses a collision object, then a force is applied to correct this. A simpler version demonstrated by Reynolds [2004] using the containment steering behavior. In this case rather than testing a volume, the agents test points within the it's future path.

*Figure 6 : Obstacle avoidance and containment behavior (Reynolds, 1999)*



**Collision Estimation**

With both obstacle avoidance and unaligned collision avoidance behaviors there is a need to test if the agent is about to run into a collision object. In the case of an external library it is not possible to read how the game has stored collidable objects, so an interpretation of the game world must be stored. The prototype stores this data as a AiBlock (a *block*ing-object), which is a base class containing the information needed to do bounding box calculations. This has then been derived into two separate collision objects, AiBlock_Circle and AiBlock_Polygon. AiBlock_Circle as the name suggests is a circle defined by a radius and position. AiBlock_Polygon stores a series of points that define a convex polygon, convex due to make collision tests simpler.

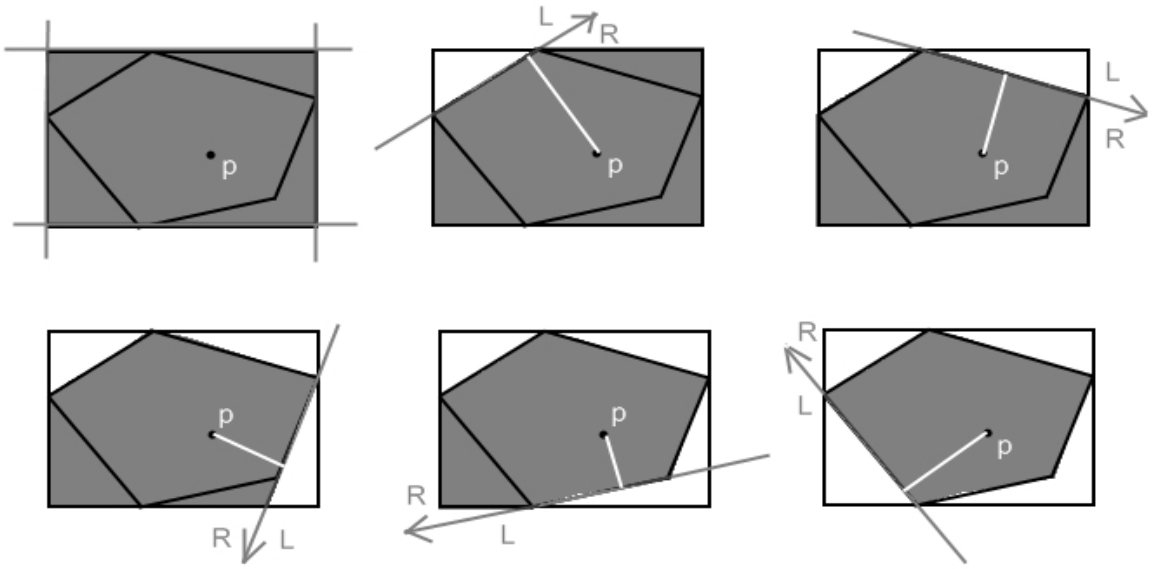



*Figure 7: steps to test a point-inside a polygon. First using bounding box, then testing which side the point lies of each face.*

To estimate the collision detections I started by for each collision object setting up a test to see if a point is within the collision shape. For bounding-boxes and circles this is a simple test to see if either it is within minimum and maximum limits, and if the distance of the point is within a radius respectively. Polygon point-inside detection is more complex and it is for this that the shape must be convex. Once the point has been testing against the bounding-box, each face tested. The position of the point is tested in comparison to the face. If there is an inconsistency, it is to the right of all faces except one, the point is located outside of the polygon.

**Prototype Implementation**

**1. Structure**

The prototype, called BUG, is structured around the AiManager, this is a singleton class and so once called can be accessed from anyway in a game. It is intended to act as a hub from which the game is able to access AiPathfinder and the agent, AiAgent.

The prototype is intended to experiment with the design and implementation of a flexible AI agent library. Taking what has been found researching pathfinding and steering I initially started by developing methods of storing AI data which would interface with an unknown game system.

The major issues are that the AI system needs a central hub from which data is processed and

edited, but most, if not all the data will already be in some form stored within the game. A possibility would be to copy all the data from the game database, and store it within the AI structures locally. This method is unadvised firstly because of the amount of data the would need to be duplicated. In just the case of waypoints, which may just be based upon a low resolution polygon map of the game world, there could be several thousand points duplicated. In addition to the memory usage the data would have to be updated with any changes, causing excess set operations.

The solution I adopted when implementing the agent class is to use an abstract class to track the data. With this solution the user can choose how to manage the data and as to whether to store it as part of the game world data, or as a secondary AI data structure.

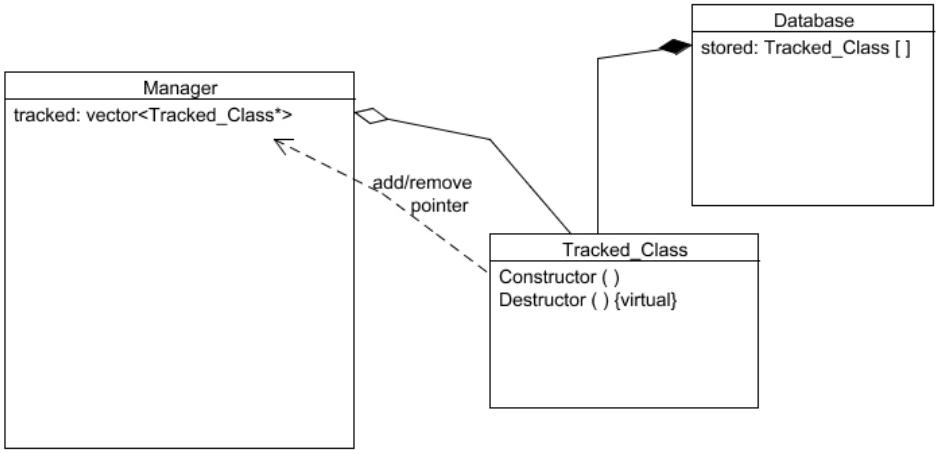



*Figure 8 : UML diagram demonstrating a tracked class.*

**2. Tracked Class**

These classes are stored by the game, but have to be linked back to the AiManager so the data can be read and edited. These have been implemented by creating abstract classes to represent AI components that the game must derive from to use. When the abstract class is called the constructor notifies the AI library by adding itself to a protected array inside AiManager. Then when the derived class is deleted or goes out of scope it is erased from the same array. So in the case of AiAgents, all instances of an agents available from the central AiManager class, enabling testing a single agent against all of its neighboring agents for separation behavior discussed in Steering and Collision Avoidance.

This method of tracking is only used in the prototype to organise the AiAgents, but would be effective at storing AiBlocks and possibly, in keep with a uniform method of data management, also suitable for AiPathNodes.

This system is not perfect, specifically because it tracks all agents in memory irrespective of whether they are suppose to be used in game calculations. Possibly the best solution to this would not be to alter the method of tracking, but rather to add variables to the abstract classes, such as AiAgent, that distinguish whether it should be considered in use.

An additional issue if this were to be applied to the AiPathNode would be the need to make more differentiation between players. Returning to the fog-of-war problem discussed in Pathfinding using A* search, Implementation; how does the pathfinding algorithm determine for each player which waypoints are visible. Since this is a very common demand of a RTS game it is worth considering if

a AI Agent library should implement methods of dealing with fog-of-war. Once again the best solution would be to implement groups that hold pointers to the nodes visible to each player. Note that this would not be necessary for agents since they are unique to a player, and blocks have no need to be associated to a player.

**3. Orders**

The key requirement of a AI agent is that it must be able to process orders given to it. These orders inherently cannot be defined by the library since they are game specific. The agent therefore also has to be able to process any order the game requires it to. These orders are stored as classes derived from AiOrder, examples can be seen in AiMove and the further derived AiMove_Goto.

Inside the agent there are stored queues of orders being sent to it. These queues are able to be reorganized, have priority orders inserted in front of less important orders, but only remove orders from the head of the queue. This queue stored data as pointers to AiOrders, meaning it could refer to any derived order passes to it and since AiOrder has a type ID that can be used to determine the interface theoretical it wasn't a problem. When looking at the transfer of data from the game, which was giving the order, to the agent, then there was a problem with how to communicate what was contained in the unknown derived class.

The solution I came upon was to keep the orders internal to the agent. When a new order is requested it is allocated within AiAgent, the pointer is stored in the queue. As it has been allocated the memory will not need to be copied and can be managed internally. Specific data can then be input via a returned pointer. When the being processed the type ID is tested, and if it can be processed it will. This all requires that the game implements customised versions of the new order function and the process order function.

Initially AiAgent had been designed with a single order queue, but this became clearly unworkable. Looking at the simple case of a agent being ordered to fire at another agent. First the agent must move towards the target then fire at it, but these action do not want to be performed in sequence. A viewer would expect the agent to fire upon the target as soon as it was in range but also to continue to get closer to the target to improve any possible range bonuses. These are a set of orders that are processed each turn simultaneously. In the prototype design there are two queues to hold movement and actions, along with a state flag which can be used to effect how it .

Only two order queues maybe sufficient for most game agents but it could be restrictive in some cases. Ideally the game should be able to implement several order queues and since they have a specific data management design it would be good if broken down into duplicable elements. The game could have an expanded number of queues and even dynamically change the number.

**4. Problems and Improvements**

Within the prototype there are several issues that have only come to light under critical analysis. These have prominently arose from the implementation of improved data management, such as the tracked class AiAgent, after an initial component has been developed, the AiPathfinder and AiPathNode classes.

One of the central concerns about the prototype is this inconsistency between data storage. In the case of AiPathNodes the it would be very advantageous to change the waypoint to be stored similarly to AiAgent. In they're current data structure doesn't allow them to be derived, and so customised. If the user were to have to store them, and they be tracked in AiManager, then they could expand the waypoints to suit the game structure.

Another issue, looks again to the AiPathNodes and the specification of data to calculate appropriate paths. If we consider the case of a car and a hovercraft, these two agents require different weighting on waypoints. With the example of a hill and a river; the car will be penalised the most over a river, but the hovercraft will have little to no penalty on the river but may suffer more on the hill. This demonstrates the effects of agent specific waypoint cost.

In the current implementation agents are not considered in pathfinding, and so it is not possible to weigh the costs differently for each. A method considered would be to move the entire pathfinding calculations into the AiAgent class. This would localise the the calculation, allowing for access to the agent type and the player, if that were to be stored in the agent.

The last concerns are based predominantly upon the detailed organisation of data within the library. The amount of data stored within many of the classes is not consistent with the usage of the class. In the cases of the AiPathNode which is a data class, it stores basic location and cost data, in the prototype though it contains temporary data which is only used by one operation. This issue is not sever but should be streamlined if the library were to be released. Secondly there is occasionally holes within data management structures. By this I mean that in some cases where data is not intended to be accessible to change or a pointer is not intended to be available to be deleted. It is important that these are cleared up.

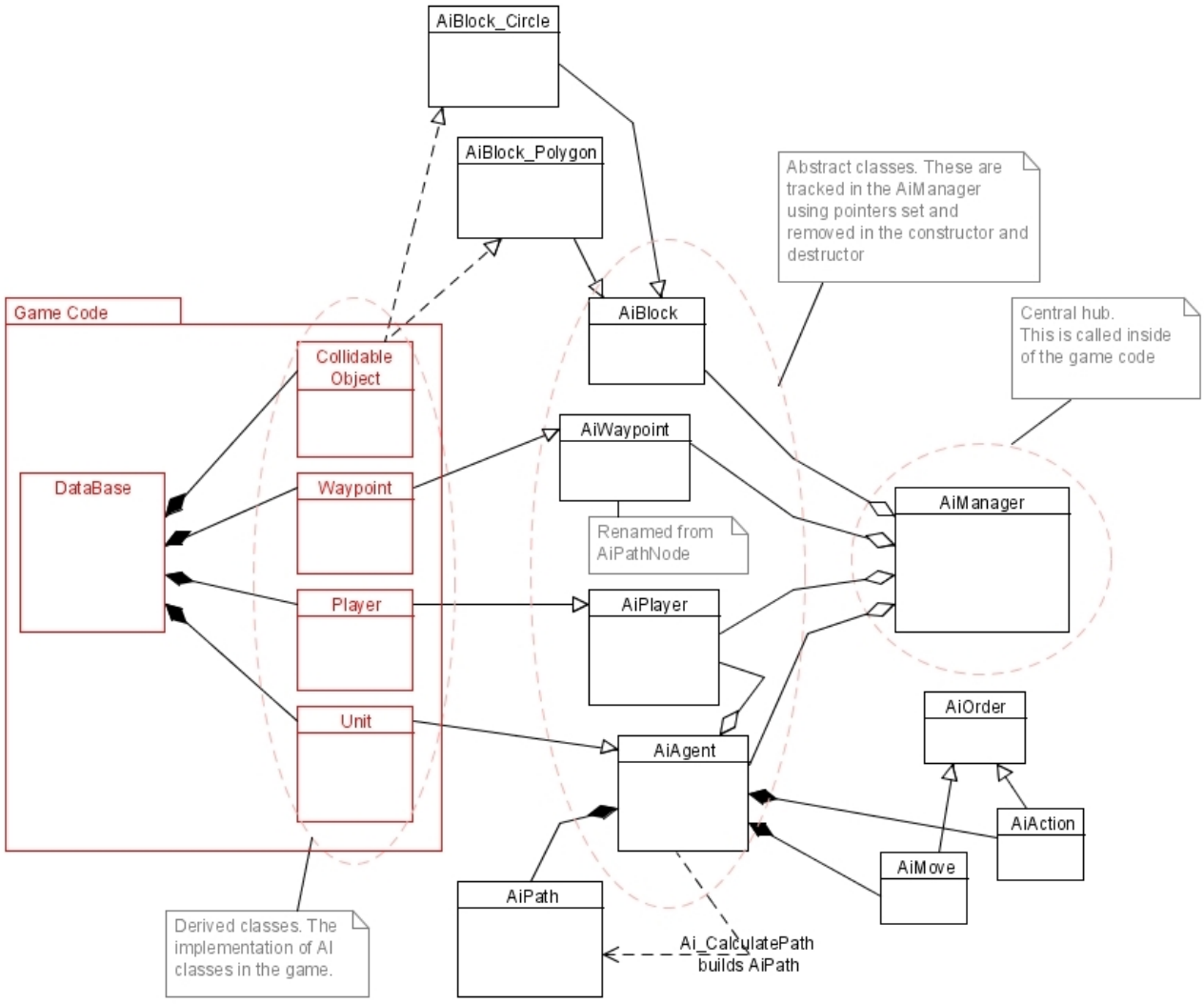



*Figure 9 : Simplified final design of library, See Appendix for full page diagram*

**Future Design**

The structure of the prototype is largely a combination of initial planning and experimental implementation of algorithms and data structure. Due to this it has ended up being a collection of method in an unorganised systems structure.

The final design in Figure 9 shows many of the improvements suggested earlier (Note that the AiManager maybe part of the game database, but is not illustrated in that form). One key difference is the organisation of data into a central hub with AI object classes treated in the same way. In addition to this the pathfinding functions have been moved into the agent, and the path itself is stored within the agent. It is not a perfect configuration, some ideas of improvement would include introducing the possibility of a group class which would could path find for a set of agents.

**Conclusion**

Though the initial of creating a working AI library was not accomplished, I thoroughly explored the areas of pathfinding using the A* search algorithm, steering and object avoidance developed in Reynolds [1999], and the architectural design of a AI agent. I developed prototype classes and worked on system designs for a first implementation of the real-time-strategy (RTS) game library.

The area of unit level AI for a RTS game is possible to encompass in a single library but for advanced systems this would probably be a hindrance. The AI for any game has to be honed to get the optimal gameplay out. This means that for anything that requires beyond the most basic control will find restrictions. Though this does not mean that a standalone library is not feasible, just that anyone interested in advanced uses would be better to look at the algorithms incorporated within the library.

The methods researched and developed in this project will go on to be used in my future projects and games.

**References**


DriverHeaven, Interview with Valves Doug Lombardi Available from: http://www.driverheaven.net/dhinterviews/doug/ [Accessed 13 March 2008]

Morris, M (2008) Introversion Blog, Rise of the machines. Available from: http://forums.introversion.co.uk/introversion/viewtopic.php?t=1216 [Accessed 13 March 2008]

Reynolds, C. W. (1999) Steering Behaviors For Autonomous Characters, *in the proceedings of Game Developers Conference 1999 held in San Jose, California. Miller Freeman Game Group, San Francisco, California.* Pages 763-782.

Reynolds, C. W. (2004) Containment steering behavior. Available from: http://www.red3d.com/cwr/steer/Containment.html [Accessed 13 March 2008]

Nareyek, A (2004) AI in Computer Game, Availible at : http://doi.acm.org/10.1145/971564.971593

Patel, A. (2008) Amit's Game Programming Information, Implementation notes. Available from: http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html [Accessed 12 March 2008]

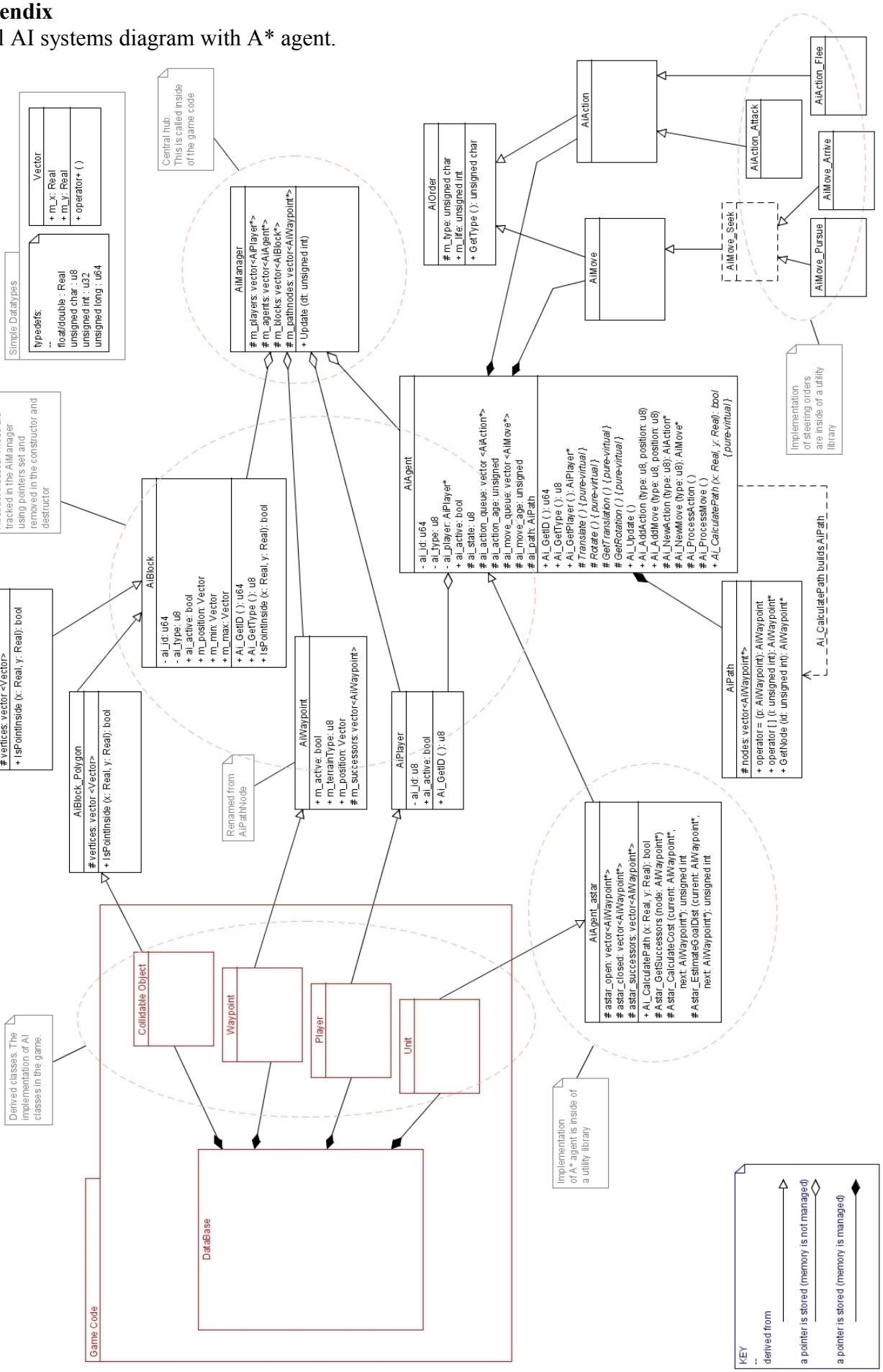
Patel, A. (2008) Amit's Game Programming Information, Heuristics. Available from: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html [Accessed 12 March 2008]

Heyes-Jones, J (2007) Avoiding Ten Common Game AI Mistakes. Available from: http://justinsboringpage.blogspot.com/2007/10/avoiding-ten-common-game-ai-mistakes.html [Accessed 12 March 2008]

Creative Assembly (2000) Game. Shogun: Total War.

**Appendix**

Final AI systems diagram with A* agent.



Earlier system designs