1. Introduction

The aim of this project is to make a real time strategy (RTS) game AI engine. A mini RTS game is developed in the project to test this AI engine. The code is made in C++ and OpenGL. In this game, players will control some combat units to destroy the enemy's base. At the same time, the opponent units are controlled by the game AI, they will try to protect their own base and attack the player's.

The AI engine in this project basically consists of three parts: An A* pathfinding algorithm, an influence map and a finite-state machine. All the AI controlled tanks will try to find the best way to the player's base and move to it. During the motion, these AI tanks can avoid obstacles, follow a leader or join a group and their velocity will be affected by different types of terrain. They will also keep tracking the opponent units position and make decisions to modify the path they have chosen in real time. The AI tanks will try to retreat when they are in danger or their base is surrounded by the opponent tanks (their life value is lower than a certain level or the enemy's influence near their base is higher than some level).

In a sense, this game is a simplified RTS game because there are not any economic individual units (sometimes called peons-the "builders" and "gatherers") in the game and there are not any resources in the game world for the AI or human player to collect and use these resources to develop their own base. However, with this AI engine, it should be easy to add these economic units and resources into the game world.

2. Previous Work

2.1 Pathfinding Algorithm

"There are many algorithms that are commonly known and used. These algorithms range from the simple to the complex. The simplest approaches are to go toward the goal until some sort of obstacle is reached then turn in another direction, and tracing around the edges of obstacles (Fig. 2.1). This is an example of a "blind-search", where the algorithm does not rely on any information about the cost of the path to the goal in selecting the next node to expand. A list of other algorithms in this "blind-search" group are the breadth-first search, the bi-directional breadth-first search, Dijkstra's algorithm, depth-first search, iterative-deepening depth-first search. There are also some algorithms that plan the whole path before moving anywhere. Best-first algorithm expands nodes based on a heuristic estimate of the cost to the goal. Nodes, which are estimated to give the best cost, are expanded first. The most commonly used algorithm is called A* (pronounced A star), which is a combination of the Dijkstra algorithm and the best-first algorithm." [1]

Figure 3.1 (picture from Anatoly Preygel's *Pathfinding: A Comparison of Algorithms* [1])

"A-Star (or A*) is a general search algorithm that is extremely competitive with other search algorithms, and yet intuitively easy to understand and simple to implement. Search algorithms are used in a wide variety of contexts, ranging from A.I. planning problems to English sentence parsing. Because of this, an effective search algorithm allows us to solve a large number of problems with greater ease.

The problems that A-Star is best used for are those that can be represented as a state space. Given a suitable problem, you represent the initial conditions of the problem with an appropriate initial state, and the goal conditions as the goal state. For each action that you can perform, generate successor states to represent the effects of the action. If you keep doing this and at some point one of the generated successor states is the goal state, then the path from the initial state to the goal state is the solution to your problem.

What A-Star does is generate and process the successor states in a certain way. Whenever it is looking for the next state to process, A-Star employs a heuristic function to try to pick the best state to process next. If the heuristic function is good, not only will A-Star find a solution quickly, but it can also find the best solution possible." [2]

2.2 Influence Map

"Influence Mapping is slowly becoming one of the most commonly used secondary AI techniques in games. Its generic structure and open-ended usage make it second only to finite-state machines(FSMs) in ease of implementation and adaptability to different game AI problems. The term influence map refers to using a simple array of data, where each element represents data about a specific world position, IMs are usually conceptually thought of as a 2D grid overlaid on the world. The resolution of this grid (and thus the number of elements in your influence array) depends on the minimum size of game space that you need to tag with information. Where your game can compromise between data size and influence accuracy will also determine this resolution. So, if you absolutely need specific information for every square inch of a large game world, your IM will have a very high resolution indeed (and take up

a lot of memory in doing so). Many games will employ multiple IMs, to either help with memory and searching costs, or to provide different levels of game space resolution to the various AI systems. Thus, a real-time strategy game might have an IM with very low resolution (say, each element is an entire game screen) that reflects the amount of each resource within The game uses this for high-level planning when building the town and determining which direction in which to expand the town. You would want to expand your base (and thus your main defenses) toward more resourceful areas to facilitate future expansion. You can also employ another IM, with a much higher resolution (each element is approximately equal to four of the smallest units standing together) that keeps track of the number of units that have been killed in each grid square, This map is used to affect the pathfinding engine so that units will not continue to take paths into areas that have a high mortality score.

In games with worlds that are heavily 3D (that is, comprising many vertical layers built on top of each other), a more complex data structure is necessary, these can be represented with correspondingly layered IMs, or by building the influence data into the navigation mesh used for pathfinding. Another technique might be to only use IMs where you need them, or local IMs. For example, a battle between forces in an RTS game might start anywhere on the map. You might want to have a heavily detailed IM during battles to coordinate forces, but not want to use the memory to have a global IM for the game world that would provide the level of information you require. Instead, you could implement a heavily detailed, but local IM. The system would detect a battle and set the coordinates of the local battle planning IM to be some distance out from the center of the battle. The center of the battle could be determined using a different, lower-resolution IM that keeps track of population data or fighting locations. Thus, the global IM is still constrained, but local information can become quite detailed."[3]

There are several kinds of influence map used in game AI. These influence maps include occupance data, ground control, pathfinding, danger signification, rough battlefield planning, simple terrain analysis, and advanced terrain analysis.

"Occupance data means tracking various populations within the game. An easy use of an IM is to keep track of the number of specific game objects within a certain area. You might want to keep track of all combat units, specialized resource locations, important quest items, or any other in-game object. Simple occupance data can be used to help with obstacle avoidance (overriding the pathfinding system with local detours around occupied terrain), give rough estimates of various game perceptions (army size, town density, the direction to the most powerups, etc.), or any other task that requires quick access to localized population data."[3]

" Ground control is finding actual influence of game ground. Although the term influence map is used in game AI as a loosely defined data structure, the phrase

historically refers to techniques derived from the field of thermodynamics (in determining heat transfer) and field analysis in general (such as electromagnetic fields). These same equations can be used in a game setting and can quickly determine which player has control over which part of the game map."[3]

" The pathfinder data may include things like passability, relative to terrain features (such as hills or cliffs) and to terrain type (land versus sea) or even which player currently controls the areas of the map you want to traverse."[3] In this project, the pathfinder data is built in the A* path finding algorithm so it is not necessary to make a pathfinder data influence map.

Danger signification is to "keep track of areas where bad things have happened over a period of time." Rough battlefield planning is similar to the ground control influence map. Simple terrain analysis "includes somewhat more mathematical determinations such as cover (how much a given position is open to attack from any given angle), visibility (in some ways, the opposite of cover but also considers lighting concerns and line-of-sight issues), and height factors (many games allow greater missile weapon range from higher ground and better visibility)."[3] Advanced terrain analysis includes determinations such as "finding food choke points in a map, or places where movement or visibility are severely restricted",[3] "determining the best way to build a town, defensive walls or other structures",[3] "determining important map areas (such as maps with severely limited resources or key strategic positions)".[3]

2.3 Finite-State Machines

" FSMs have been used broadly in the video game industry. Since ID Software released the source code to the Quake and Quake 2 projects, people have noticed that the movement, defensive, and offensive strategies of the bots were controlled by a simple FSM. ID is not the only company to take advantage of this either. The latest games like Warcraft III take advantage of complex FSM systems to control the AI. Chat dialogs where the user is prompt with choices can also be ran by FSMs.

Aside from controlling NPCs, bots, dialog, and environmental conditions within video games, FSMs also have a large role outside of the video game industry. For example, cars, airplanes, and robotics (machinery) have complex FSMs. You could even say websites have a FSM. Websites that offer menus for you to traverse other detailed sections of the website act much like a FSM does with transitions between states." [4]

" In the world of game AI programming, no single data structure has been used more than the finite-state machine (FSM) (unless you count the switch statement as a data structure, perhaps, but a switch can actually be used as a simple form of state machine). This simple yet powerful organizational tool helps the programmer to break the initial problems into more manageable subproblems and allows

programmers to implement intelligence systems with flexibility and scalability. Even if you haven't used a formal FSM class or functionality, you've probably used the principles that this structure follows because it is a basic way of thinking about software problems in general. Thus, even if your game uses a more exotic AI technique for some element of decision making, you will probably use some form of state system in your game." [3]

3. The Fundamentals of A* Pathfinding Algorithm

Patrick Lester has written a good A* pathfinding tutorial for beginners. In his article, there is a representative example about how to find the best way by using A* path finding algorithm.[5] This example is shown in figure 3.1, the grey square is the start position and the black one is the destination. There is a wall which is represented by three yellow squares between the start point and the end point. Now, if there is an AI controlled tank in the grey square and it wants to move to the black square, A* should do several things to make the tank find the best way.

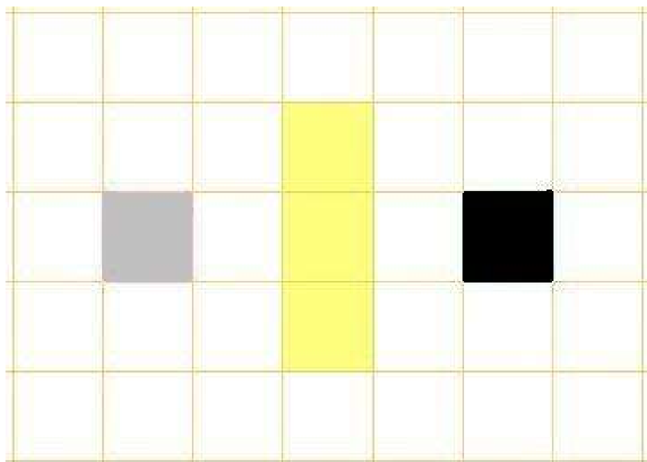

figure 3.1 (picture from Patrick Lester's *A* pathfinding for beginners [5]*)

The first step is to divide the search area into a square grid (of course it can be other shapes like triangles or hexagons). All the squares are set to walkable or unwalkable and the center points of the squares are called " nodes". Actually, a single node could be set in anywhere within one square and in the pathfinding process, after finding out the path, game units move from one node to another node until they reach the final node which is the nearest one to the destination.

There are several essential elements in the pathfinding algorithm: an open list and a closed list; F value; G value and H value. The program will check those squares and put them into open list or closed list to memorize which squares have already been checked and which ones need to be checked out. F, G and H are values used to calculate the cost of getting somewhere.

The search will start at the grey square. So, the grey square is the first one added

into the open list. Then the program will look at all the eight squares adjacent to the grey one. The unwalkable squares like the wall will be ignored and others will be put into the open list too. After that, the start square is saved as a “parent square” of the adjacent squares. This step is very important because after the search is done, the path will be saved by finding out every parent square from the end point. Then the grey square will be deleted from the open list and added on the closed list. See figure 3.2, the squares in the open list will be outlined and each of them has a pointer that points to its parent. The grey square is now in the closed list.

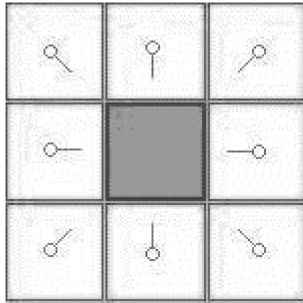

figure 3.2 (picture from Patrick Lester’s *A* pathfinding for beginners [5]*)

The next step is to calculate the F value for all the adjacent squares by using the following equation:

$$F = G + H$$

G is the movement cost from the start point to a given square on the map following the path generated by the A* algorithm. The H value is usually referred to as the heuristic because it is a guess. The way of calculating the H value greatly affects the pathfinding result. “The success of A-Star rests heavily on the heuristic function chosen. For any given problem that we may wish to apply A-Star to, there are good heuristics and bad heuristics. A good one will allow the algorithm to run quickly and find the optimal solution. A bad one may just increase the running time. Or it may be so bad that it misleads the algorithm into returning sub-optimal solutions or not find solutions at all.”[2]

To calculate the G value, first set the start square’s G value to 0. Then check the adjacent squares. If it is a vertical or horizontal square, the movement cost is 10. If it is a diagonal square, the movement cost is 14. This is just the first step and after this, to figure out a square’s G value, the G value of its parent square should be picked up and then add 10 or 14 depends on if it is a diagonal square or not.

There are several ways such as the Manhattan method, the Eudidean method and the Max method to calculate the H value. The Manhattan method is employed in this project because it is faster than the others although it is slightly inadmissible (which means it is an overestimate of the remaining distance between the current square and the target). This method calculates the total number of squares moved vertically and horizontally to the target square from the start point. All the obstacles will be ignored. Then multiply the total number by 10 to get the final H value and add the G

value to H to get the F value.(figure 3.3)

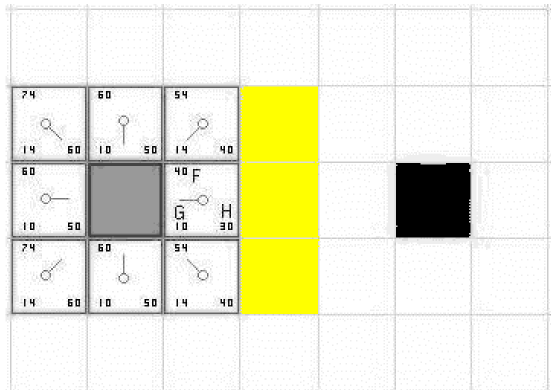

figure 3.3 (picture from Patrick Lester's *A* pathfinding for beginners [5]*)

To continue the search, check all the 8 adjacent squares of the grey one and select the one which has the lowest F value. With the selected square, first drop it from the open list and add it on the closed list. Then check the 8 squares around it, ignoring the obstacles and the squares which are already on the closed list. If the squares are not on the open list, add them on the open list and set the selected square the "parent" of the squares. Then calculate the G, H and F value for these squares which are new added to the open list. If an adjacent square is already on the open list, check to see if it is a better path to go to this adjacent square from the selected one. To do this, G value is used to make a comparison. If the game unit uses the selected square to get to the adjacent one, there will be a new G value. Compare this new G value with the old one, if the new value is higher, don't do anything. On the other hand, if the new value is lower than the old one, the adjacent square's parent square should be changed to the selected one because a lower G value means this new path will cost less than the old path. Meanwhile, the adjacent square's G and H value should be recalculated since its parent square has been changed. Thereafter, put the selected square on the closed list and search the open list to find out the square with the lowest F value, then repeat the process in this paragraph until the end square is on the closed list. (figure 3.4)

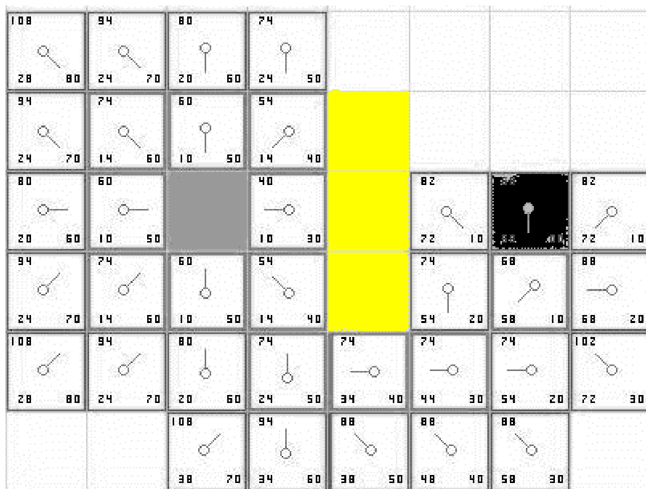

figure 3.4 (picture from Patrick Lester's *A* pathfinding for beginners* [5])

The last step is to figure out the final path. The process should be like this: Start at the end square (the black square) and move from one square to its parent until reach the start square. (figure 3.5)

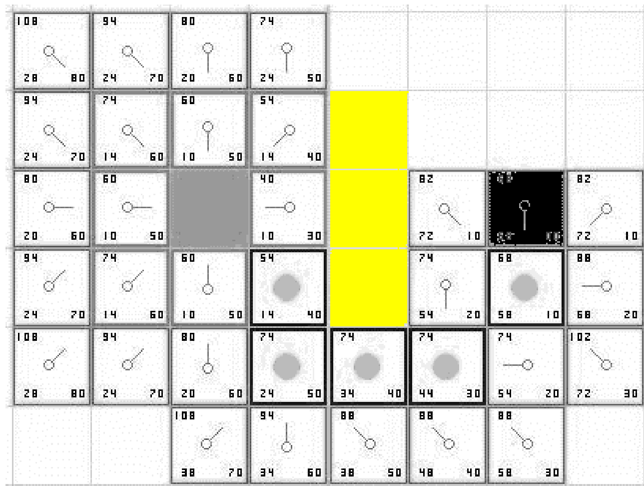

figure 3.5 (picture from Patrick Lester's *A* pathfinding for beginners [5]*)

4. The Algorithm For The Influence Map

In this project, the influence map is mainly used to collect the ground control data. "The algorithm for this is simple: First, zero out the entire map. Then, assign each grid square a value based on its team-specific control (the magnitude represents the degree of control, the sign differentiates teams; a positive value for player A's units, a negative value for player B's units, with more complex schemes for more than two players). Then go over the map again, and for each map square, add up the values of the squares surrounding it, scale by some amount (to prevent value overflow) and add that to the square's value. Repeat a few times to disperse the influence out until you achieve a stable state. Player A controls the squares that have positive values, and player B controls negatively valued squares. This technique will provide you with a way of measuring global, as well as local, control." [3]

5. Different Terrain Types

In this project, A* pathfinding algorithm is expanded by applying some different terrain types. The status of a square on the map is not only walkable or unwalkable. Apparently, in the real world, both asphalt roads and grassland are walkable but of course it is harder to walk on the grass. To simulate this kind of effect, two types of walkable terrain are made in the game. Figure 4.1 shows these two types of terrain: The square which colored in light green represents an area with low grass and the dark green square stands for the land with high grass. As a result, different movement costs should be assigned to the two squares when A* is calculating. In this program, the G value of the dark green square is double of the light green square's G cost. Accordingly, the tanks will move slower on the dark green squares.

Figure 4.2 shows how to calculate the G value when an AI tank moves from one square to another square with a different terrain type.

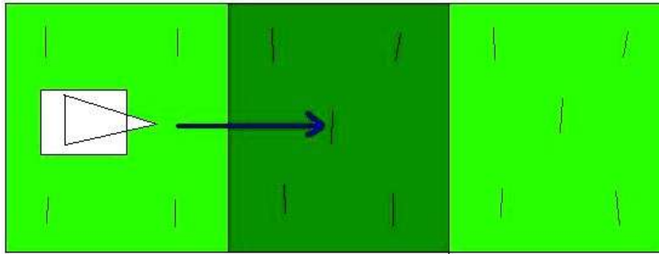

figure 4.1 Two types of terrain

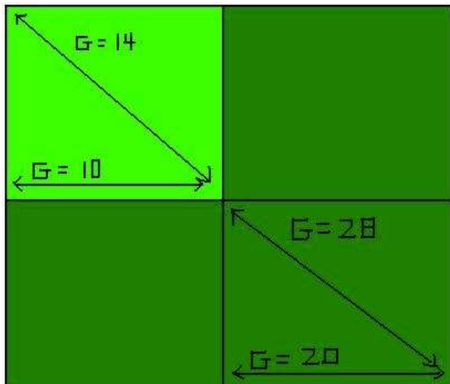

figure4.2 G cost for different terrain

This method could be improved by adding some other kinds of cost to different types of terrain. In this program, the movement cost only affects the A* pathfinding algorithm to find the best path. So, A* always tries to find out the best path which costs the least no matter what kind of terrain it is. Actually, this kind of situation should only happen in an emergency. In the real world, people usually try to find the "easiest" way and maybe this way will cost more than the others. For example, if there are two roads lead to the same place, one of them is shorter but muddy, the other one is longer but dry. People will usually choose the dry one because they don't want to make their shoes dirty. Only in an emergency, the shorter road will be chosen to save time. So, in the game world, we can add an "emergency cost" to the A* path finding algorithm. This new E (emergency) value will keep fluctuating according to the battlefield situation. Now, the equation for calculating the movement cost should be like this:

$$F = G + H + E$$

In the peacetime, the E value of a special terrain should be quite high so the AI controlled characters won't like to walk in this area. If there is an emergency, for example, the base is under attack, the E value will be turned down according to the danger level, the more dangerous the situation becomes, the lower the E value will be. At this time, all the AI characters will try to get across a difficult terrain to carry out a rescue action.

Accordingly, some kinds of "punishment" should be adopted, otherwise the E value

becomes useless because if there isn't any loss when a tank is moving on a difficult terrain, and at the same time the path can be shorter by going across this terrain. Why not just move in this area? In a real battle, the fact is like this: If a group of soldiers want to save time to get somewhere earlier and thus they decide to cross a marsh. Probably their battle effectiveness will go down because they will be demoralized by the terrible environment in the marsh. In the game, the punishment of getting across a difficult terrain could be decreasing the tank's life value or decreasing its fire power, but it is very hard to decide how much punishment should be applied.

It is hard to decide the degree of the punishment, this is one of the reasons why the E value is not adopted in this project and the punishment level setting will greatly affect the game's balance. A lot of statistic work should be done to decide a proper punishment for crossing the difficult terrain otherwise the game will be unbalanced. Another reason for not using the E value in this project is it doesn't work very well on the human controlled tanks. Sometimes the human player doesn't want his game units to get any punishment even if his base is in danger. For example, if the AI has sent out all its tanks to attack the player's base and the player just wants to abandon this base and move to another place. At this time, since the base is under attack, the human controlled tanks should think that there is an emergency and they must get across all the terrible terrains to save time. This is not what the player expects to see. A way to solve this problem is to let the player make a decision about the emergency level but it is still not a good method because it will make the game much more complicated. Players have to set up different emergency levels for different units (An emergency for the combat units might not be an emergency for the economic units). If this process is not well designed, the gamer's immersion level will be greatly decreased (especially in a RTS game because RTS games always have very quick rhythm).

6. A Combination of A* And Influence Map

Actually, A* pathfinding algorithm is more suitable for turn-based strategy game rather than real time strategy game because A* deals with discrete steps, not with continuous movement. The path is calculated and saved before the game characters start moving. Furthermore, A* will always find the best way between the start point and the end point. That means, if the start point and the destination don't change, the path will always be the same, so all the game characters will be moving on the same path and that will look really stupid. In this project, the A* pathfinding function is executed in every update loop so the game characters should be able to change their mind to modify the path during their movement. However, only this is not enough. Some kind of "motivations" must be created for the characters to change their mind. This work is done by the influence map in this project, it transfers the battle field information to the game characters and then the characters will make decisions according to these information.

A grid based map is also essential for making an influence map. The same grid map which is used in the A* pathfinding function could be employed to make the influence map. In this project, the influence of each grid square is calculated in the following way: First a certain game object is selected. Then, set up an influence range (The influence range should be decided according to the selected game unit's field of fire. It will also be affected by each opponent game unit's characteristic). Thereafter, make a circle centered at the selected game object's position and with a radius that equals to the influence range. Next, calculate the influence for each grid square:

```
If ( distance <= influenceRange)
   {
      Influence += maxInfluence * (1 - distance / influenceRange);
   }
```

In the above equation, maxInfluence is the maximum influence which is decided by the selected game unit's firepower and sometimes by the terrain type (if a tank is hiding in a forest, the maximum influence of this tank may be stored as zero until its opponent finds it). The distance value is the value of the distance from any square center to the circle's center. Finally, repeat this process for all the game units (notice that the human controlled game units' influence value should be positive and the AI units' influence should be negative).

To combine the influence map and the A* pathfinding algorithm, the equation of movement cost calculation should be modified like this:

$$F = G + H + \text{Influence}$$

Now the AI tanks will try to evade the human controlled tanks, they will keep adjusting their paths. For the AI tanks, it will cost more if they try to cross a square which is occupied by a human controlled tank. So, they will make a detour. They will also try to move together because that will make the movement cost less. All these decisions are made in real time.

7. AI Characteristics

Actually, each AI tank in this game has its own influence map. In other words, the influence map data are not always the same for different AI tanks. Consequently, this will affect the A* pathfinding algorithm to find out different paths for different tanks even they start at the same position and go to the same destination. For a single AI tank, the path it determines just indicates the characteristic it has.

In this project, the AI tanks are divided into three groups. For the tanks in group one, their influence maps are as same as the original influence map. So, tanks in group one will try to keep moving together and they will evade the human player's units. Group two's influence map is modified by change all the negative influence value on the map to zero. As a result, AI tanks in group two will like to move alone. Then, change all the influence value on the map to zero. That means the AI tanks in group

three will be very dissocial and aggressive. They don't care much about what their allied tanks and the opponent are doing. Of course group two and group three are two extreme situations (extremely dissocial or extremely aggressive). To make it more realistic, two parameters can be employed in the Tank class. One of them represents the tank's aggressive level , the other one represents the tank's dissocial level. And for each AI tank, the influence value of each grid square is recalculated like this:

```
if ( influence > 0) // The current square is occupied (or influenced more) by the player
   { influence = influence – aggrressive; }
else if ( influence < 0) // The current square is occupied ( or influenced more) by the AI
   { influence = influence + dissocial; }
```

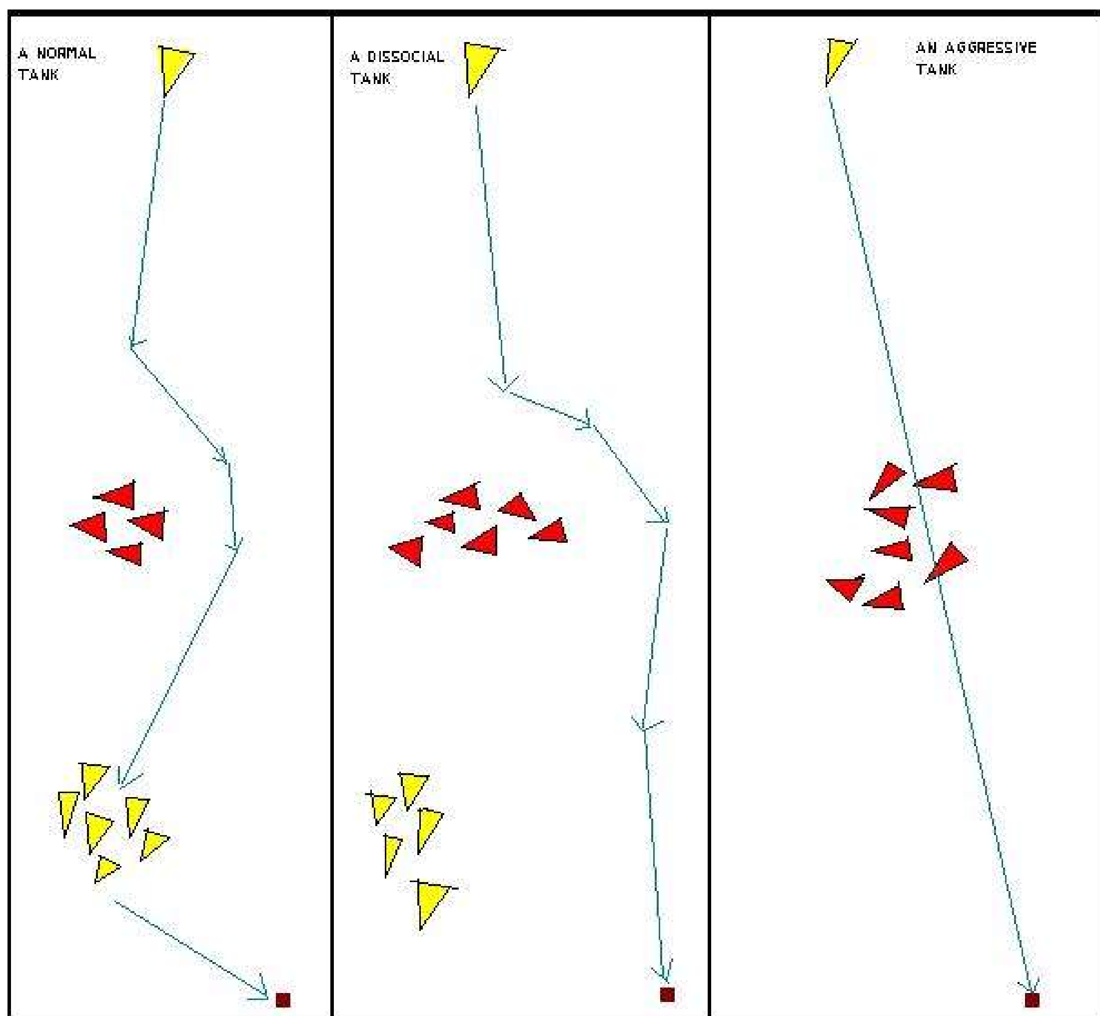

figure 7.1 Three different AI characteristics

Furthermore, when an emergency happens, different destinations could be given to different AI tanks. For example, if the base is under attack. Some AI tanks will just go back to protect their base and some tanks will go on moving to attack the human player's base. That also shows various AI characteristics. Another example is, when an AI tank encounters a human controlled tank, this AI tank might follow the player's

tank (change the destination to this human controlled tank's position) and keep attacking.

8. The Pipeline Of Decision Making

At the beginning of the game, all the AI tanks will start moving to the human player's base. Meanwhile, they keep checking the influence map to make decisions to fix their paths. They will also keep checking their own life value, their base's life value and the influence values around their base. When an AI tank is in motion, if there are any opponent tanks go into its field of fire, this AI tank will fire at the opponent tank and try to escape from the battle or follow the opponent to carry on fighting. If the AI tank is close enough to the human player's base, it will stop and attack the base. As soon as the AI tank find out that its own life value is lower that some level or its base is in danger, it will go back to its base to recover or protect the base. After the AI tank's life value is restored or its base is not in danger anymore, it will move to the enemy's base again. If the tank's life value becomes zero, it will be reborn at a certain place.

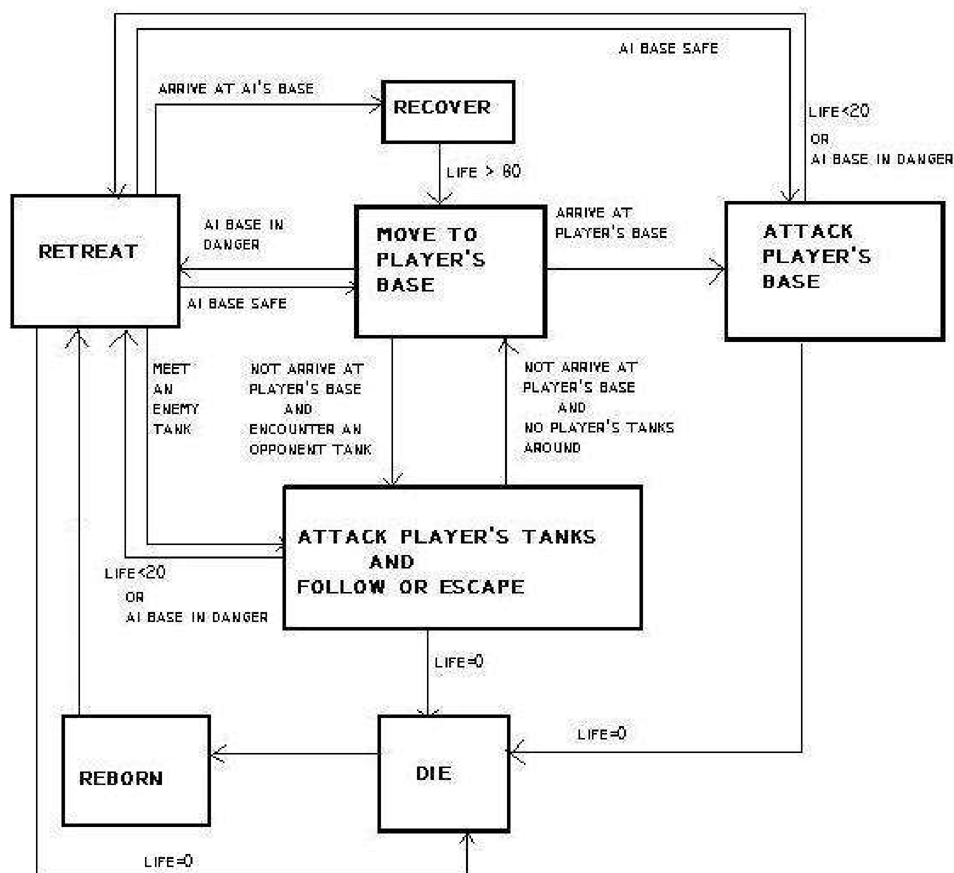

figure 8.1 The decision making pipeline

9. Conclusion

In this project, a basic RTS game AI engine is developed. The AI controlled characters are able to collect information of the terrain and the battlefield to make decisions. They have lots of steering behaviors such as avoiding obstacles, avoiding moving enemies, following a leader and joining a group. They can also keep checking the influence map to know the situation of the battlefield and make a decision like deciding whether they should carry on the current mission or go back to protect their base. The combination of the A* pathfinding algorithm and the influence map provide a flexible method to apply various characteristics to different AI units.

It is easy to add more RTS game elements based on the AI engine in this program. Currently, there is only one type of unit (combat unit) in the game. Some economic units could be added to make this game more completed. Actually, The pipeline of the economic unit's decision making is almost the same as the combat units'. Different game units just have different destination like the combat units will move to the enemy's base and the economic units will move to the resources (Of course their life value or firepower will be different too, but those parameters should be set up in the game object class rather than in the AI engine).

The game graphics is really simple in this program and it can be improved a lot by building a 3D terrain and some 3D game models (or use better 2D textures to make a 2D RTS game). The squares on the grid map can be easily reorganized according to a given 3D terrain model. Thereafter, some kind of height information should be stored in each square. However, this is not a perfect method to make a real 3D grid map. There will be some problems to add air forces into the game. To solve this problem, several layers of grid maps can be laid out vertically.

The pathfinding function in this project works very well but it is not really suitable for a high resolution grid map. There are too many calculations in the A* pathfinding algorithm. So, it is better to use the A* pathfinding algorithm as a global pathfinding algorithm and add a local pathfinding algorithm in the program. By doing this, the grid map's resolution could be kept low without any losses of game map details.

The Finite-State machine in this project is not a formal one because It is built in several functions rather than classes. A formal Finite-State machine usually consists of three classes: the FSMState class, the FSMMachine class and the FSMAIControl class. It is better to rewrite the code in the above way.

10. References And Bibliography

References:

1. Anatoly Preygel. *Pathfinding:A Comparison of Algorithms.*
http://www.cpcug.org/user/scifair/Preygel/Preygel.html#_Toc445443578

2. Benny Tsai. *Introduction to the A-Star Algorithm.*
http://upe.acm.jhu.edu/websites/Benny_Tsai/Introduction%20to%20AStar.htm

3. Brain Schwab. *AI Game Engine Programming.* (2004). USA.
ISBN 1-58450-344-0

4. Nathaniel Meyer. *Finite State Machine Tutorial.*
http://www.generation5.org/content/2003/FSM_Tutorial.asp

5. Patrick Lester. *A* Pathfinding for Beginners.*
http://www.policyalmanac.org/games/aStarTutorial.htm

Bibliography:

1. Craig Reynolds. *Steering Behaviors For Autonomous Characters.*
http://www.red3d.com/cwr/steer/

2. Bjon Reese. *Finding a Pathfinder.*
http://home1.stofanet.dk/breese/aaai99.html

3. James Matthews. *An Introduction to Game AI.*
http://www.generation5.org/content/2000/app_game.asp

4. James Matthews. *A* for the Masses.*
http://www.generation5.org/content/2000/astar.asp

5. Stuart James Kelly. *Applying Artificial Intelligence, Search Algorithms and Neural Networks to Games.*
http://www.generation5.org/content/2003/KellyMiniPaper.asp

6. Patrick Lester. *Heuristics and A* Pathfinding.*
http://www.policyalmanac.org/games/heuristics.htm

7. Dave C. Pottinger. *Terrain Analysis in Realtime Strategy Games.*
http://www.gdconf.com/archives/2000/pottinger.doc

11. Appendices

grid.h

```
// The grid class is used to create a single square in the Whole
grid-based map
#ifndef __GRID_H__
#define __GRID_H__
#include "GraphicsLib.h"


class grid
{
 public:
  GraphicsLib::Point3 pos;// The grid position
  float step;// The square edge length
  int type;// Terrain type, type 0 represents a grassland, others are
obstacles( sea or mud )
  int f,g,h; // Tree values used in A* path finding algorithm to
calculate the movement cost
  int influence; // This variable is used to store the game units
influece on a certain area ( to make an influence map)
  int wb;// walkability, lands with high grass should be harder to go
across
  int whichlist; //There are two lists in the A* algorithm-- Closed
List and Open List. So, this variable shows which list the
//square is on. 0 means this square is not on any lists, 1 means it
is on the open list and 2 means it is on
                //the closed list

// These 3 variables store the " Parent Square" of the current
square. They are also used for A* pathfinding
  GraphicsLib::Point3 parent;
  int parentX, parentY;

  grid();// Constructor
  grid(GraphicsLib::Point3 Pos, float Step );//Constructor
  void setType(int Type);// Set terrain type
  void draw();//Draw function
  void ini();// Initialize the grid
  void setParent( GraphicsLib::Point3 Parent);// Giving the current
square its parent square's information
  void drawInfluence(); // This function is for testing the influence
map. Different influence levels are drawn
```

```
                    // in different colors
};


#endif
```

grid.cpp

```
// The grid class is used to create a single cell in the Whole grid-
based map
#include "grid.h"
#include <math.h>
using namespace GraphicsLib;
using namespace std;

//Constructor
grid::grid(GraphicsLib::Point3 Pos, float Step )
{
 pos=Pos;
 step=Step;
}

//Constructor
grid::grid()
{
 pos.set(0,0,0);
 step = 4.0;
 type =0;
 f=100000;
 g=0;
 h=0;
 influence=0;
 parent=pos;
 parentX=0;
 parentY=0;
 whichlist=0;
 wb=1;
}

//Set the grid type to create different terrain types
// Type 0 : grassland (there are two types of grassland, with low
grass and high grass)
// Type 1 : mud
```

```
// Type 2 : sea
void grid::setType(int Type)
{
  type=Type;

  if ( type == 0);
   {
    float random = RandomPosNum( 10.0 );
    // If it is a grassland with high grass, set the walkability to
2, else set the walkability to 1
    if( random > 8.0 )
      {wb = 2;}// If the walkability equals two, the speed of the tank
in this area should be slown down by 50%

    else
      {wb = 1;}
  }
}

// Draw the game units' influence in this area
// Different colors represent different danger level
// Green colored areas are safe, red colored areas are dangerous,
white colored areas are extremely dangerous
// Blue means nothing is happening in this area
// This function is just used for testing. Press "i" when game is
running to enter the influence map, "I" to exit
void grid::drawInfluence()
{
  if( influence > 0)
    {
      if( influence/400.0<=1.0 )
          {
           glPushMatrix();
         glColor3f(influence/400.0,0.0,0.0);
           pos.Translate();
           glutSolidSphere(0.3,6,3);
         glPopMatrix();}

      else
          {
           glPushMatrix();
         glColor3f(influence/90.0, influence/200.0, influence/200.0);
           pos.Translate();
           glutSolidSphere(0.3, 6, 3);
```

```
          glPopMatrix();
         }
     }

  else if ( influence == 0)
    {
      glEnable(GL_LIGHTING);
      glPushMatrix();
        glColor3f(0.5,0.5,0.7);
        pos.Translate();
        glutSolidCube(0.6);
      glPopMatrix();
      glDisable(GL_LIGHTING);
    }

  else
    {
        glPushMatrix();
        glColor3f(0.0,abs(influence)/90.0,0.0);
         pos.Translate();
         glutSolidSphere(0.3,6,3);
        glPopMatrix();
    }
}

//Draw the grid according to its type
void grid::draw()
{

  // Get the four vetices of a square
  Point3 q0, q1, q2, q3;
          q0.set(pos.x-step/2, pos.y-step/2, pos.z);
        q1.set(pos.x+step/2, pos.y-step/2, pos.z);
        q2.set(pos.x+step/2, pos.y+step/2, pos.z);
        q3.set(pos.x-step/2, pos.y+step/2, pos.z);
  // These 10 points are used to draw some grass on the grassland
  Point3 grass[10];
   grass[0].set( q0.x+step/4, q0.y+step/4, 0.01);
   grass[1].set( q0.x+step/4, q0.y+step/4+0.3, 0.01);
   grass[2].set( q0.x+3*step/4, q0.y+step/4, 0.01);
   grass[3].set( q0.x+3*step/4, q0.y+step/4+0.3, 0.01);
   grass[4].set( q0.x+step/4, q0.y+3*step/4, 0.01);
   grass[5].set( q0.x+step/4, q0.y+3*step/4+0.3, 0.01);
   grass[6].set( q0.x+3*step/4, q0.y+3*step/4, 0.01);
```

```
   grass[7].set( q0.x+3*step/4, q0.y+3*step/4+0.3, 0.01);
   grass[8].set( q0.x+2*step/4, q0.y+2*step/4, 0.01);
   grass[9].set( q0.x+2*step/4, q0.y+2*step/4+0.3, 0.01);

          if( type==0 )
          {
           glColor3f(0.3,0.3f,0.4f);
                //Draw grass
             glBegin(GL_LINES);
                grass[0].Vertex();
                grass[1].Vertex();
                grass[2].Vertex();
                grass[3].Vertex();
                grass[4].Vertex();
                grass[5].Vertex();
                grass[6].Vertex();
                grass[7].Vertex();
                grass[8].Vertex();
                grass[9].Vertex();
             glEnd();
          if( wb == 1)
          {glColor3f(0.3,0.7,0.4);} // Light green --- low grass

            else
            {glColor3f( 0.2, 0.6, 0.3);}// Dark green --- high grass

          }

          else if( type == 3) // Type 3, this type of grid is used as
an icon
          {
          glColor3f( 0.9,0.9,1.0); // White
          }

          else if( type == 1)// Type 1, mud
          {
            glColor3f( 0.9, 0.6, 0.4); // Brown
          }

          else if( type == 2)// Type 2, sea
          {
            glColor3f( 0.1,0.5,0.8); // Blue
          }
```

```
          // Now draw a square
            glBegin(GL_QUADS);
             q0.Vertex();
             q1.Vertex();
             q2.Vertex();
             q3.Vertex();
            glEnd();

           glPointSize(4.0);
           glBegin(GL_POINTS);
             pos.Vertex();
           glEnd();
}

//Initialize the grid
void grid::ini()
{
 f=100000;
 g=0;
 h=0;
 parent=pos;
 whichlist=0;
}

// Get the parent square's information. This function is used for A*
pathfinding algorithm
void grid::setParent( GraphicsLib::Point3 Parent)
{
 parent = Parent;
}
```

Tanks.h

```
// The tank class is used to create all the game units - players
controlled tanks and AI controlled tanks

#ifndef __TANKS_H__
#define __TANKS_H__
#include "Point3.h"
#include "Vector.h"
#include "grid.h"
```

```
class Tanks
{
  public:

       GraphicsLib::Point3 pos; // The tank current position in the
space
       GraphicsLib::Point3 oldpos; // The tank previous position
       GraphicsLib::Vector direction;// Velocity direction
        GraphicsLib::Point3 aimPos;// The aim position where the tank
is moving to
        GraphicsLib::Vector dirRandom;// a random direction
        GraphicsLib::Point3 dir; // for moveTo() function
        GraphicsLib::Point3 bulletPos, bulletAim;// Bullet position
and bullet target
        GraphicsLib::Vector BulletDir;// Bullet velocity direction

        float rotateAngle;// The angle used to rotate the tank
       int type; //Type 0 is hunman player's tank, type 1 is AI
controlled tank, type 2&3 are flags
        int Startx, Starty, Endx, Endy; //
       float speed, newSpeed; // Tank speed
        int beShot; // If the tank is shot
       int timeCount, timeCountRandom, bulletTimeCount; //Variables
used for A* pathfinding
        int trigger; // When trigger=1, tank shoots
        int bulletLife; // Bullet life
        int selected; // If the tank is selected by the player or not
        float life;// Tank life



        Tanks( int Type);// Constructor
        void Draw(void); // Draw function
       void move(GraphicsLib::Vector Direction); // Just move the
tank in a strait line
        void moveTo(GraphicsLib::Point3 Aim); // Tank moves to some
point
        void update(); // Update function
        void moveRandomly(int x1, int x2, int y1, int y2); // Tank
moves randomly, this function is just for testing
        void shoot( GraphicsLib::Point3 BulletAim); // Shoot at some
point
        void checkSpeed(grid cell); // Tank speed is affected by
different terrain types
```

```
        void initBullet(); // Init the bullet

};

#endif
```

Tanks.cpp

```
// The tank class is used to create all the game units - players'
tanks and AI controlled tanks

#include "Tanks.h"
#include <math.h>
#include <GL/gl.h>
#include <GL/glut.h>
#include "GraphicsLib.h"
using namespace GraphicsLib;
using namespace std;

// Constructor
Tanks::Tanks( int Type)
{
  type=Type; // Type 0 is hunman player's tank, type 1 is AI
controlled tank, type 2&3 are flags

  if( type == 0)
    {pos.set(4,4,0);}

  else
    {pos.set(76, 56, 0);}
  beShot = 0;
  aimPos = pos;
  bulletAim = pos;
  timeCount = 0;
  timeCountRandom = 0;
  dirRandom.set( RandomNum(1.0), RandomNum(1.0), 0.0);

  // A random number is used when set up the tank's speed. By doing
this, we can simulate an "old tank" effect
  // because old tanks run a bit more slowly
  if( type == 0)
    {speed = RandomPosNum(2.0)+2.0;}

  else
```

```
    {speed = RandomPosNum(2.0)+2.0;} // If we improve the AI tank's
speed, the game will be more difficult

  newSpeed = speed;
  trigger = 0;
  bulletLife = 30;
  rotateAngle=0.0;
  BulletDir.set ( 0,0,0);
  selected = 0;

  if( type == 0 || type == 1)
    {life= 100.0;}

  else
    {life = 1000.0;} // This is the flag( base)'s life value, so it
is larger
}

//Draw function, draw the tank and the bullet
void Tanks::Draw(void)
{

 Point3 a, b, c, d, e, f, g, h,i,j,k;
 a.set( 0.9, 0.0, 0.4);
 b.set( -0.8, 0.45, 0.4);
 c.set( -0.8, -0.45,0.4);
 d.set( -1.0,-0.55, 0.4);
 e.set( -1.0, 0.55, 0.4);
 f.set( 0.4, 0.55, 0.4);
 g.set( 0.4, -0.55, 0.4);
 h.set( 0.0, 0.0, 0.4);
 i.set( -0.5, 1.0, 0.4);
 j.set( 1.5, 1.0, 0.4);

 if( type==0 || type ==1)
   { k.set( -0.5 + 2.0*life/100.0, 1.0, 0.4);}

 else
   {
     k.set( -0.5 + 2.0*life/1000.0, 1.0, 1.6);
     i.set( -0.5, 1.0, 1.6);
     j.set( 1.5, 1.0, 1.6);
   }
// First draw the life bar
```

```
 glPushMatrix();
  pos.Translate();
  glLineWidth(4.0);

  if( life >1.0)
   {
    glColor3f( 0.0,1.0,0.0);
    glBegin(GL_LINES);
      i.Vertex();
      k.Vertex();
    glEnd();
   }

 glColor3f( 1.0,0.0,0.0);

 glBegin(GL_LINES);
  i.Vertex();
  j.Vertex();
 glEnd();

 glLineWidth(2.0);
glPopMatrix();

// Then draw the tanks, AI tanks and player's tanks are drawn in
different color.
if (type == 0)
{
  glPushMatrix();

    // After a tank is selected by the player, its color will change
    if( selected == 0)
      {glColor3f(1.0,1.0,0.3);}
    else
      {glColor3f(0.2,0.2,1.0);}

   pos.Translate();
   glRotatef(rotateAngle, 0.0, 0.0, 1.0);

    glBegin(GL_TRIANGLES);
      a.Vertex();
      b.Vertex();
      c.Vertex();
    glEnd();
```

```
    glBegin(GL_LINE_LOOP);
      d.Vertex();
      e.Vertex();
      f.Vertex();
      g.Vertex();
    glEnd();

    glBegin(GL_POINTS);
     h.Vertex();
    glEnd();

    if( selected == 1)
      {
        glColor3f(1.0,0.2,0.9);
        glLineWidth(20.0);

        glBegin(GL_LINE_LOOP);
         d.Vertex();
         e.Vertex();
         f.Vertex();
         g.Vertex();
        glEnd();

        glLineWidth(2.0);
      }

   glPopMatrix();

}

else if( type == 1)
{
  glPushMatrix();
   glColor3f(0.5,0.0,0.0);
   pos.Translate();
   glRotatef(rotateAngle, 0.0, 0.0, 1.0);

   glBegin(GL_TRIANGLES);
      a.Vertex();
      b.Vertex();
      c.Vertex();
   glEnd();

   glBegin(GL_LINE_LOOP);
```

```
      d.Vertex();
      e.Vertex();
      f.Vertex();
      g.Vertex();
   glEnd();

   glBegin(GL_POINTS);
     h.Vertex();
   glEnd();

  glPopMatrix();
}

else if( type == 2)
{
  glPushMatrix();
   glColor3f(1.5,0.0,0.0);
   pos.Translate();
   glutSolidSphere(1.0, 6,3);
  glPopMatrix();
}

else
{
  glPushMatrix();
   glColor3f(0.0,1.0,1.0);
   pos.Translate();
   glutSolidSphere( 1.0, 6, 3);
  glPopMatrix();
}

// Draw the sparks when a tank is shot by the enemy
if( beShot == 1)
{
 Point3 sparkPos;

 for( int i=0; i<9; i++)
  {
   sparkPos.set(pos.x+RandomNum(1.0), pos.y+RandomNum(1.0),
pos.z+RandomPosNum(1.0));
   glPushMatrix();
        glColor3f(1.0,RandomPosNum(1.0),RandomPosNum(0.3));
         sparkPos.Translate();
         glutWireSphere(RandomPosNum(1.0),9,2);
```

```
   glPopMatrix();
  }
}

// Draw the bullet
if( trigger == 1 )
{
 bulletPos.z = 0.4;
 Point3 B(0.0,0.0,0.0);
 Vector bulletDir = BulletDir;
 bulletDir.normalize();
 Point3 C = B + bulletDir;
 glPushMatrix();
       bulletPos.Translate();
       if( type == 0 )
       {
         glColor3f(1.0,1.0,0.0);
        glLineWidth(4.0);
        }
        else if( type == 1)
        {
         glColor3f(0.6, 0.1, 0.1);
        glLineWidth(4.0);
        }
       else
        {
        glColor3f(0.9, 0.9, 0.9);
        glLineWidth(8.0);
        }

        glBegin(GL_LINES);
          B.Vertex();
         C.Vertex();
        glEnd();

       glLineWidth(2.0);
  glPopMatrix();

 }
}

// Move in the given direction
void Tanks::move(GraphicsLib::Vector Direction)
{
```

```
 Direction.normalize();
 direction = 0.02*speed*Direction;
 pos += direction;
}

// Move to the given position
void Tanks::moveTo(GraphicsLib::Point3 Aim)
{
 GraphicsLib::Vector D;
 GraphicsLib::Vector iD(1.0,0.0,0.0);
 D = Aim - pos;
 D.normalize();

 if( Aim.y< pos.y)
   {rotateAngle = -1*(180.0/3.141593)* acos( D.dot(iD) );}

 else
   {rotateAngle = (180.0/3.141593)* acos( D.dot(iD) );}

 D = 0.04*newSpeed*D;
 pos += D;
}

// Move in random direction, this function is just for testing
void Tanks::moveRandomly(int x1, int x2, int y1, int y2)
{
  dirRandom.normalize();

  if(x1<pos.x && pos.x<x2 && y1< pos.y && pos.y<y2)
   {
     pos += 0.04*speed*dirRandom;
     timeCountRandom++;
   }

  else
   {
     pos.x = 8;
     pos.y = 8;
     dirRandom = -1.0*dirRandom;
     pos+= 0.01*newSpeed*dirRandom;
     timeCountRandom = (int)RandomPosNum(90);
   }

  if ( timeCountRandom == 100 )
```

```
    {
     timeCountRandom = 0;
     dirRandom.set( RandomNum(1.0), RandomNum(1.0), 0.0 );
    }
}


// Update function
void Tanks::update()
{
 // If life is lower than 2, this tank will die
 if( life < 2.0)
   {life = 0.0;}
 //Life will decrease when the tank is shot
 if ( beShot == 1)
   {
     if(life>=0.0)
       {life--;}
     timeCount++;

     if( timeCount == 10)
       {
        beShot = 0;
        timeCount= 0;
       }
   }

// Update the bullet
 if ( trigger == 1)
  {
   bulletTimeCount++;

   if( bulletTimeCount == 1 || bulletTimeCount == 0 )
     {
      Vector mistake ( RandomPosNum(0.3), RandomPosNum(0.3), 0.0);
      bulletAim.z = 0.4;
      Vector bulletDir = bulletAim - bulletPos;
      bulletDir = bulletDir + mistake;
      bulletDir. normalize();
      BulletDir = bulletDir;
      bulletPos = pos;
     }

   else
```

```
    {
      if( bulletTimeCount == bulletLife)
       {
         bulletTimeCount = 0;
         bulletPos = pos;
         BulletDir.set(0,0,0);
         trigger = 0;
       }

     else
       {
         bulletPos += 0.5 * BulletDir;
       }

    }

  }
}

//Initialize the bullet
void Tanks::initBullet()
{
      bulletTimeCount = 0;
      bulletPos = pos;
      bulletPos.z = 0.4;
      BulletDir.set(0,0,0);
      trigger = 0;
}

//Shoot function
void Tanks::shoot(  GraphicsLib::Point3 BulletAim)
{

  if( trigger == 0 )
  {
    bulletAim = BulletAim;
    trigger = 1;
  }

}

// Speed=speed/cell.wb   cell is a square of the grid based map, wb
is a variable that shows this cell's walkability
void Tanks::checkSpeed( grid cell)
```

```
{
 // Firt do collision detection between a tank and a square, if the tank is in the square, the tank's speed will be modified
 // according to the Square's walkability
 float up = cell.pos.y + cell.step/2.0;
 float down = cell.pos.y - cell.step/2.0;
 float left = cell.pos.x - cell.step/2.0;
 float right = cell.pos.x + cell.step/2.0;
 Vector D = cell.pos - pos;

 if(cell.type == 0 && cell.wb > 1 && D.length() < cell.step/2.0)
   {newSpeed = speed/cell.wb;}

 if(cell.type == 0 && cell.wb==1 && up>pos.y && down<=pos.y && left<=pos.x && right > pos.x   )
  {newSpeed = speed / cell.wb;}
}
```

main.cpp

```
#include <math.h>
#include <iostream>
#include <vector>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "grid.h"
#include "GraphicsLib.h"
#include "Tanks.h"
using namespace std;
using namespace GraphicsLib;
// Map height and map width. Actually, H and W are the number of squares in X direction and Y direction
#define H 15
#define W 20
// AI controlled tank number and human cotrolled tank number, please keep these two numbers lager than 6
// because there are some test functions in the code which uses tank number 5
#define tankNum 18
#define playerNum 18
// These numbers show that in which square the two flags are and in which square the tanks will be reborn after they die
#define flag1x 1
```

```
#define flag2x 18
#define flag1y 1
#define flag2y 13
#define reborn1x 18
#define reborn1y 1
#define reborn2x 1
#define reborn2y 13
//  width  of the window
int WIDTH = 1200;
// height of the window
int HEIGHT =900 ;
// The global camera
Camera Cam;
// the values to spin the main window by in x y and z
int spinxface = 0 ;
int spinyface = 0 ;
int spinzface = 0 ;
// The values for mouse function
float
origx,origy,Origx,Origy,finx,finy,leftKeyDown=0,origz,RotateXY,Rotate
XZ=0, camMoveX, camMoveY=0;
// Draw the mouse-dragged square or not
bool drawSquare = false;
// Draw the influence map or not
bool drawInfluence = false;
// After A* pathfinding algorithm determines a path, draw the path or
not
bool DrawPath = false;
int Rotate;
// The poision where the mouse icon is
Point3 mouseTrack;
// Four vertices of the mouse-dragged square
Point3 a, b, c, d;


int Endx=3, Endy=3; // The end square's X number and Y number
int Startx=2, Starty=2; // The start square's X number and Y number
float influenceRadius = 0.0; // A tank's influence range

Tanks obj(0);// This is a game object used in A* pathfinding
algorithm to find out a path for every tank in the game
Tanks flag1(2);// The Player's flag
Tanks flag2(3);// AI's flag
Tanks **player;// Player's tanks
```

```
Tanks **tank;// AI controlled tanks

//Vector D(1,0,0);

grid aim;// An icon used to highlight the hunman cotrolled tanks'
destination
grid Map[W][H];// Two dementional array for creating the game map

//Vector g(1,1,0);
Point3 AimPos; // using this point as the next aim for astar path
finding
bool simulation = false;// Start game or stop game

Point3 eye, look;



//Create the game map by using squares to form a grid-based map
void CreateMap()
{
int x, y=0;
// Create grassland firt
for( y=0; y<H; y++)
  {
   for( x=0; x< W; x++)
     {
      Map[x][y].pos.x=x*Map[x][y].step;
      Map[x][y].pos.y=y*Map[x][y].step;
      Map[x][y].parentX=x;
      Map[x][y].parentY=y;
      Map[x][y].setType(0);
     }
   }
// Randomly generate some obstacles( sea and mud) on the map
for( int c= 0; c < H*W/(RandomPosNum(15)+2); c++)
   {
    int a = (int)RandomPosNum(W)-1;
    int b = (int)RandomPosNum(H)-1;
    Map[a][b].setType(2);
   }

for(int c= 0; c < H*W/(RandomPosNum(15)+2); c++)
  {
   int a = (int)RandomPosNum(W)-1;
```

```
   int b = (int)RandomPosNum(H)-1;

   if( Map[a][b].type!=2 )
     {
      Map[a][b].setType(1);
     }
  }
// Set up all the tanks
player = new Tanks*[playerNum];

for( int i=0; i<playerNum; i++)
   {
     player[i] = new Tanks( 0);
     player[i]->pos = Map[flag1x][flag1y].pos;
   }

tank = new Tanks*[tankNum];

for( int i=0; i< tankNum; i++)
   {
    tank[i] = new Tanks( 1);
    tank[i]->pos=Map[flag2x][flag2y].pos;
   }

// Set up other game objects
 Map[flag1x][flag1y].setType(0);// The flag shouldn't be built on an
unwalkable square so set this square to type 0
 Map[flag2x][flag2y].setType(0);

 aim.pos= Map[7][7].pos;
 aim.step = 1.0;
 aim.pos.z+=0.001;
 aim.setType(3);

 obj.pos= Map[4][3].pos;
 flag1.pos = Map[flag1x][flag1y].pos;
 flag2.pos = Map[flag2x][flag2y].pos;

// The tanks shouldn't be reborn at an unwalkable place so set type 0
 Map[reborn1x][reborn1y].setType(0);
 Map[reborn2x][reborn2y].setType(0);
}

// Initialize the game map
```

```
void initMap()
{
 for(int i=0; i<W;i++)
    {
     for( int j=0; j< H; j++)
        {
         Map[i][j].ini();
         Map[i][j].parentX=i;
         Map[i][j].parentY=j;
        }
   }
}



// CheckSquare function is a part of A* pathfinding algorithm.
// This function is used to check the 8 adjacent squares of a
selected square and calculate the movement cost
void checkSquare(int x, int y, int ex, int ey)
{
  // When the selected square is at the edge of the map
   if( x==0 || x== W-1 || y==0 || y==H-1 )
     {
      for(int i=-1;i<2;i++)
         {
          for (int j=-1;j<2;j++)
              {
// Make sure the program doesn't check the place out of the game map(
which means it is checking a square doesn't exist)
                if( x+i>=0 && y+j>=0 && x+i<=W-1 && y+j<= H-1)
               {
                 if( i==0 && j==0 )
                   {
                    Map[x+i][y+j].whichlist=2;// Add the selected
square to the closed list
                      }

               else if( (Map[x+i][y+j].whichlist==0|| Map[x+i][y+j].
whichlist ==1) && Map[x+i][y+j].type==0   )
                    {
                    // Ignore the obstacles
                 if ( (i==-1 && j==1 && Map[x][y+1].type!=0) || (i==-1
&& j==1 && Map[x-1][y].type!=0))
                     {}
                     else if ( (i==1 && j==1 && Map[x][y+1].type!=0) ||
```

```
(i==1 && j==1 && Map[x+1][y].type!=0))
                    {}
                    else if ( (i==-1 && j==-1 && Map[x-1][y].type!=0)
|| (i==-1 && j==-1 && Map[x][y-1].type!=0))
                    {}
                    else if ( (i==1 && j==-1 && Map[x+1][y].type!=0)
|| (i==1 && j==-1 && Map[x][y-1].type!=0))
                    {}
                    else
                    {
                    // If the square being checked is not on the open
list, add it to the open list and make the selected
                    // square the "parent square" of the current
square. Then calculate the current square's H value by
                    // using the Manhatton method
                   if( Map[x+i][y+j].whichlist==0 )
                     {
                           Map[x+i][y+j].whichlist=1;
                       Map[x+i][y+j].parentX=x;
                           Map[x+i][y+j].parentY=y;
                           if( x+i < ex)
                         {
                                for( int m = x+i+1; m<ex+1; m++ )
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[m-1][y+j].
wb+5*Map[m][y+j].wb;}
                              }
                          else
                              {
                                for( int m = ex+1; m<x+i+1; m++)
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[m-1][y+j].
wb+5*Map[m][y+j].wb;}
                              }
                          if( y+j < ey)
                        {
                                for( int m = y+j+1; m<ey+1; m++ )
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[x+i][m-1].
wb+5*Map[x+i][m].wb;}
                              }
                         else
                             {
                               for( int m = ey+1; m<y+j+1; m++)
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[x+i][m-1].
wb+5*Map[x+i][m].wb;}
                             }
```

```
              // Calculate the G value and F value
                // Here, the tanks influence are added to calculate
the F cost
                // By modifying the influence value, we can give the
AI tanks different characteristics
                  if(abs(i*j)==1)
                      {
                       Map[x+i][y+j].g=Map[x][y].g+7*Map[x][y].
wb+7*Map[x+i][x+j].wb;
                       Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i][y+j].
h+Map[x+i][y+j].influence;
                          }
                  else
                      {
                       Map[x+i][y+j].g=Map[x][y].g+5*Map[x][y].
wb+5*Map[x+i][x+j].wb;
                           Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i]
[y+j].h+Map[x+i][y+j].influence;
                      }
                       }
             // If the square being checked is already on the open
list, check to see if it is a better path
              // to get to the current square from the selected
square. This process is done by comparing
              // the old G value and the new G value. If the new G
value is higher, don't do anything, is it is lower,
              // recalculate the current square's movement cost and
reset its parent square to the selected one.
                      else
                       {
                      int newG;
                          if(abs(i*j)==1)
                      {
                       newG=Map[x][y].g+7*Map[x][y].wb+7*Map[x+i]
[x+j].wb;
                            if( newG < Map[x+i][y+j].g )
                              {
                               Map[x+i][y+j].g= newG;
                               Map[x+i][y+j].parentX=x;
                               Map[x+i][y+j].parentY=y;
                          Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i]
[y+j].h+Map[x+i][y+j].influence;
                              }
                          }
```

```
                    else
                    {
                       newG=Map[x][y].g+5*Map[x][y].wb+5*Map[x+i]
[x+j].wb;
                            if( newG < Map[x+i][y+j].g )
                              {
                               Map[x+i][y+j].g= newG;
                               Map[x+i][y+j].parentX=x;
                               Map[x+i][y+j].parentY=y;
                          Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i]
[y+j].h+Map[x+i][y+j].influence;
                              }
                    }
                      }
                }
             }
          }
       }
    }
 }

// When the selected square is not at the edge of the map
else
{
  for( int i=-1;i<2;i++)
     {
       for (int j=-1;j<2;j++)
           {
             if( i==0 && j==0 )
                {
                  Map[x+i][y+j].whichlist=2;
                }

             else if( (Map[x+i][y+j].whichlist==0|| Map[x+i][y+j].
whichlist ==1) && Map[x+i][y+j].type==0  )
                  {
                  if ( (i==-1 && j==1 && Map[x][y+1].type!=0) || (i==-1
&& j==1 && Map[x-1][y].type!=0))
                      {}
                      else if ( (i==1 && j==1 && Map[x][y+1].type!=0) ||
(i==1 && j==1 && Map[x+1][y].type!=0))
                      {}
                      else if ( (i==-1 && j==-1 && Map[x-1][y].type!=0)
|| (i==-1 && j==-1 && Map[x][y-1].type!=0))
```

```
                    {}
                    else if ( (i==1 && j==-1 && Map[x+1][y].type!=0)
|| (i==1 && j==-1 && Map[x][y-1].type!=0))
                    {}
                    else
                    {
                    if( Map[x+i][y+j].whichlist==0 )
                      {
                           Map[x+i][y+j].whichlist=1;
                       Map[x+i][y+j].parentX=x;
                           Map[x+i][y+j].parentY=y;

                           if( x+i < ex)
                         {
                              for( int m = x+i+1; m<ex+1; m++ )
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[m-1][y+j].
wb+5*Map[m][y+j].wb;}
                             }
                          else
                             {
                              for( int m = ex+1; m<x+i+1; m++)
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[m-1][y+j].
wb+5*Map[m][y+j].wb;}
                              }
                           if( y+j < ey)
                         {
                              for( int m = y+j+1; m<ey+1; m++ )
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[x+i][m-1].
wb+5*Map[x+i][m].wb;}
                              }
                          else
                             {
                              for( int m = ey+1; m<y+j+1; m++)
                    {Map[x+i][y+j].h=Map[x+i][y+j].h+5*Map[x+i][m-1].
wb+5*Map[x+i][m].wb;}
                             }

                     if(abs(i*j)==1)
                       {
                        Map[x+i][y+j].g=Map[x][y].g+7*Map[x][y].
wb+7*Map[x+i][x+j].wb;
                        Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i][y+j].
h+Map[x+i][y+j].influence;
                           }
```

```
                    else
                      {
                       Map[x+i][y+j].g=Map[x][y].g+5*Map[x][y].
wb+5*Map[x+i][x+j].wb;
                            Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i]
[y+j].h+Map[x+i][y+j].influence;
                      }
                       }
                      else
                      {
                    int newG;
                        if(abs(i*j)==1)
                     {
                       newG=Map[x][y].g+7*Map[x][y].wb+7*Map[x+i]
[x+j].wb;
                            if( newG < Map[x+i][y+j].g )
                              {
                               Map[x+i][y+j].g= newG;
                               Map[x+i][y+j].parentX=x;
                               Map[x+i][y+j].parentY=y;
                          Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i]
[y+j].h+Map[x+i][y+j].influence;
                              }
                          }
                  else
                    {
                     newG=Map[x][y].g+5*Map[x][y].wb+5*Map[x+i][x+j].
wb;
                           if( newG < Map[x+i][y+j].g )
                             {
                              Map[x+i][y+j].g= newG;
                              Map[x+i][y+j].parentX=x;
                              Map[x+i][y+j].parentY=y;
                         Map[x+i][y+j].f=Map[x+i][y+j].g+Map[x+i]
[y+j].h+Map[x+i][y+j].influence;
                             }
                    }

                      }
                 }
              }
           }
         }
      }
```

```
   }




// A* pathfinding algorithm
void astar( int startx, int starty, int endx, int endy)
{
 int sx=startx, sy=starty, ex=endx, ey=endy;// Get the start position
and the end position
 initMap();// Initialize the map
 // Make sure the start square and the end square are both walkable
 if( Map[startx][starty].type==0 &&Map[endx][endy].type==0)
   {
     for(int h=0; h<W*H; h++)
       {
         int fval1=100000, fval2=0;
         if( Map[ex][ey].whichlist!=2)
           {
             // 1.put start square on the openlist
            Map[sx][sy].whichlist=1;
            // 2.check the adjacent squares
            checkSquare( sx, sy, ex, ey);
            // 3.put start square on closedlist
            Map[sx][sy].whichlist=2;
            // 4. choose the square with the lowest f value from the
adjacent square
            for(int i=0; i<W; i++)
              {
               for (int j=0; j<H; j++)
                  {
                   if ( Map[i][j].whichlist==1 && Map[i][j].
type==0 )
                     {
                      fval2= Map[i][j].f;
                      if ( fval2<=fval1)
                       {
                         sx= i;
                         sy= j;
                         fval1=fval2;
                       }
                         }
                     }
```

```
                }
            }
        }
 // If the path doesn't exist, initialise the map
   if ( Map[endx][endy].whichlist!=2)
      {
        initMap();
      }

  }
}



// The drawPath function is just for testing the A* pathfinding
// Press '9' to draw the path

void drawPath()
{
 int a=obj.Endx, b=obj.Endy, c= obj.Endx, d= obj.Endy;
 Point3 height( 0.0, 0.0, 0.15);
 for( int i=0; i<H*W; i++)
    {
     c= Map[a][b].parentX;
     d= Map[a][b].parentY;
     glPushMatrix();
     height.Translate();
     glColor3f(1.0,1.0,0.0);
     glLineWidth(2.0);
     glBegin(GL_LINES);
     Map[a][b].pos.Vertex();
     Map[c][d].pos.Vertex();
     glEnd();
     glPopMatrix();
     a=c;
     b=d;
   }
}

// Transfer a point position in 3D space to a square position on the
game map
void getEndPoint()
{
// Here, we reverse the start point and the end point because to
determine the final path, we need to go backwards from
```

```
// the end point to the start point. So, just reverse them to make it
easier
 for ( int i=0; i< W; i++)
     {
      for( int j=0; j<H; j++)
         {
            if( Map[i][j].pos.x-Map[i][j].step/2.0<=aim.pos.x && Map[i]
[j].pos.x+Map[i][j].step/2.0 > aim.pos.x && Map[i][j].pos.y-Map[i]
[j].step/2.0<=aim.pos.y && Map[i][j].pos.y+Map[i][j].step/2.0 >
aim.pos.y)
             {
              obj.Startx= i;
              obj.Starty= j;
            }
         }
     }

 for ( int i=0; i< W; i++)
     {
      for( int j=0; j<H; j++)
         {
            if( Map[i][j].pos.x-Map[i][j].step/2.0<=obj.pos.x && Map[i]
[j].pos.x+Map[i][j].step/2.0 > obj.pos.x && Map[i][j].pos.y-Map[i]
[j].step/2.0<=obj.pos.y && Map[i][j].pos.y+Map[i][j].step/2.0 >
obj.pos.y)
             {
             obj.Endx=i;
               obj.Endy=j;
             }

        }
    }



}

// DrawParent function is also used to test the A* pathfinding
algorithm
// In this function, some pointers are drawn to point out a certain
square's parent square
void drawParent()
{
    glPushMatrix();
    Point3 up( 0.0, 0.0, 0.1);
```

```
     up.Translate();
    Vector direction(0,0,0);
    Point3 end(0,0,0);
   for ( int i=0; i< W; i++)
     {
      for (int j=0; j< H; j++)
          {
           if( Map[i][j].whichlist==2 || Map[i][j].whichlist==1)
             {
              direction = Map[ Map[i][j].parentX][Map[i][j].parentY].
pos-Map[i][j].pos;
              end =Map[i][j].pos + 0.5*direction;

              glColor3f( 1.0,1.0,1.0);
              glBegin(GL_LINES);

                 Map[i][j].pos.Vertex();

              end.Vertex();
                 glEnd();
              glBegin(GL_POINTS);
              Map[i][j].pos.Vertex();
              glEnd();
            }
        }
    }
  glPopMatrix();
}

// This function is used to give the A* a start square position and a
end square position
void path()
{

 AimPos = Map[ Map[obj.Endx][obj.Endy].parentX][ Map[obj.Endx]
[obj.Endy].parentY].pos;
 obj.dir = AimPos;

}

// Check the terrain type of the area where a tank is moving and
modify the tank's speed according to the terrain type
void checkSpeed()
{
```

```
  for ( int i = 0; i<W; i++)
      {
       for ( int j = 0; j<H; j++)
          {
           for( int t=0; t<tankNum; t++)
              {
                if( tank[t]->life>0.0)
                   {
                    tank[t]->checkSpeed(Map[i][j]);
                   }
              }
          for( int t=0; t<playerNum; t++)
             {
              if( player[t]->life>0.0)
                {
                 player[t]->checkSpeed(Map[i][j]);
                }
             }
         }
      }
}

// Find out a target for a tank to shoot at
void findTarget()
{

 for ( int i = 0; i< tankNum; i++)
     {
      tank[i]->update();
      if(tank [i] ->trigger == 0)
        {

         Vector Dflag = tank[i]->pos-flag1.pos;
         if(Dflag.length()<10.0)
           {
            tank[i]->shoot(flag1.pos);
            flag1.shoot(tank[i]->pos);
           }
         else
           {
             for( int j=0; j< playerNum; j++)
                {
                 Vector D = player[j]->pos - tank[i]->pos;
                 if( D.length()<=15.0)
```

```
                {
                 tank[i]->shoot(player[j]->pos);
                 j = playerNum;
                }
            }

        }
    }
   }

for ( int i = 0; i< playerNum; i++)
    {
     player[i]->update();
     if( player[i]->trigger == 0)
       {
        Vector Dflag = player[i]->pos-flag2.pos;
        if(Dflag.length()<10.0)
          {
           player[i]->shoot(flag2.pos);
           flag2.shoot( player[i]->pos);
          }
        else
         {
          for( int j=0; j< tankNum; j++)
             {
              Vector D = player[i]->pos - tank[j]->pos;
              if( D.length()<15.5)
                {
                 player[i]->shoot(tank[j]->pos);
                 j = tankNum;
                }

             }

         }
      }
    }
}

// Do collision detection between tanks and bullets
void checkBulletCollision()
{
   Vector DisToFlag1, DisToFlag2;
   for( int i=0; i<tankNum; i++)
```

```
  {
    DisToFlag1 = tank[i]->pos - flag1.bulletPos;
    if( DisToFlag1.length() < 1.0)
      {
        tank[i]->beShot = 1;
        flag1.initBullet();
      }

   if( tank[i]->life >0.0)
     {
      for (int j = 0; j<playerNum; j++)
         {
           Point3 b = player[j]->bulletPos;
           b.z = 0.4;
           Vector D = tank[i]->pos - b;

           if( D.length()< 1.0)
             {
              tank [i] -> beShot = 1;
              player[j]->initBullet();
                }
         }
   }

 else
  {
   delete tank[i];
   tank[i] = new Tanks( 1);
   tank[i]->pos = Map[reborn2x][reborn2y].pos;
   tank[i]->life=10.0;
   tank[i]->initBullet();
  }

if( flag1.life > 0.0)
  {
   Vector Dflag = tank[i]->bulletPos - flag1.pos;
   if( Dflag.length()< 1.5)
     {
       flag1.beShot = 1;
      tank[i]->initBullet();
       }
 }
else
  {simulation = false;}
```

```
  }

for( int i=0; i<playerNum; i++)
   {
    DisToFlag2 = player[i]->pos - flag2.bulletPos;
    if( DisToFlag2.length() < 1.0)
      {
       player[i]->beShot = 1;
       flag2.initBullet();
      }

    if(flag2.life>0.0)
      {
       Vector Dflag = player[i]->bulletPos - flag2.pos;
       if( Dflag.length()< 1.5)
         {
           flag2.beShot = 1;
         player[i]->initBullet();
          }
      }
    else
     {simulation = false;}
 if( player[i]->life> 0.0)
   {
     for (int j = 0; j<tankNum; j++)
        {
          Point3 b = tank[j]->bulletPos;
          b.z = 0.4;
          Vector D = player[i]->pos - b;
          if( D.length()< 1.0)
            {
             player[i] -> beShot = 1;
             tank[j]->initBullet();
               }
        }
  }
 else
  {
    delete player[i];
    player[i] = new Tanks( 0);
    player[i]->initBullet();
    player[i]->life=10.0;
    player[i]->pos = Map[reborn1x][reborn1y].pos;
```

```
    }
  }

}

// If an AI cotrolled tank finds out its life is lower than some
level, it would like to ge back to its base and recover
// Tanks' life will increase gradually when they are near their own
base
  void tankRecoverNearBase()
  {
    for( int i=0; i< tankNum; i++)
       {
        Vector D = tank[i]->pos - flag2.pos;
        if(D.length()<10.0 && tank[i]->life< 80.0)
          {
           tank[i]->life+=0.02;
          }
       }

    for( int i=0; i<playerNum; i++)
      {
        Vector D = player[i]->pos - flag1.pos;
        if( D.length() <10.0 && player[i]->life < 80.0)
        {player[i]->life += 0.02;}
      }

    if( flag1.life < 700.0 )
    {flag1.life += 0.1;}

    if ( flag2.life < 700.0 )
    {flag2.life += 0.1;}

  }


  void initInfluenceMap();
  void influenceMap();
  void reverseIM();

// Actually, this function is a Finite-State machine for the tanks
// Here, the tanks will move to their enemy's base and try to destroy
it. The tanks will keep checking the influence map
// to get information of the battle field.
```

```
// By modifying the influence map, different characteristics can be
applied to the AI tanks, some of them would like to
// move together with other tanks, some of them will move alone, some
of them will try to avoid a battle and some of
// them will ignore the enemy's influence( go directly into the
enemies)
void Path()
{
  initInfluenceMap();
  checkSpeed();

// Hunman controlled tanks
  for( int i=0; i< playerNum; i++)
     {
      if( player[i]->life> 0.0)
        {
         obj.pos = player[i]->pos;
         if( player[i]->selected == 2)
           {
             getEndPoint();
             player[i]->aimPos = aim.pos;
           }
         else
           {
             aim.pos = player[i]->aimPos;
             getEndPoint();
             aim.pos.set( Origx, Origy, 0.1);
           }

        astar(obj.Startx, obj.Starty, obj.Endx, obj.Endy);
        path();

        player[i]->dir = obj.dir;
        // check collisions between tanks
        int collision = 0;
        for( int j = 0; j< playerNum; j++ )
           {
            if( j != i)
              {
               Vector playerDis = player[i]->dir - player[j]->pos;
               if(playerDis.length() <0.5)
               {collision = 1;}
               }
           }
```

```
 // check distance between unit and aim
Vector distance = player[i]->aimPos - player[i]->pos;
if( distance.length() > 2.0 && collision == 0 )
  {
   // if no collision and not reach the aim yet, go on moving
   player[i]->moveTo(player[i]->dir);
  }
 }
}


influenceMap();

// AI controlled tanks
  for( int i=0; i< tankNum; i++)
  {

     if( i == tankNum/2 )
     {
        for( int x=0; x<W; x++)
         {
        for ( int y= 0; y<H; y++)
          {
          // These AI tanks will ignore other AI tanks influence,
they want to move alone, but they will try to
            // avoid enemies
            if( Map[x][y].influence <20.0)
              {Map[x][y].influence = 0;}
          }
         }
    }




/* The next 12 lines of code will affect the drawPath function
because the path drawn is the last AI tank's path.
 The next 12 lines will make the last AI tank a "careless" tank, that
means it won't notice its enemy's influence and
 it won't change its path once it has decided a destination. If you
want to observe how the AI tanks change their
 ideas according to the human player's action, just get rid of the
following 12 lines( line 818 - 829).*/
     if( i == 4*tankNum/5 )
       {
```

```
        for( int x=0; x<W; x++)
             {
             for ( int y= 0; y<H; y++)
                    {
                   // these tanks will ignore their enemies influence
                   if( Map[x][y].influence >0.0)
                         {Map[x][y].influence = 0;}
                    }
             }
      }

 // AI tanks make decisions about where they should go and what they
should do,
 // to attack the enemy's base or to protect their own base
 // wether they should retreat or not
    if( tank[i]->life >0.0)
     {
      obj.pos = tank[i]->pos;
      if( Map[flag2x][flag2y].influence > 0 && flag2.life<100.0 )
        {
         if(Map[flag2x][flag2y].influence > 200 )
          {
           aim.pos = Map[flag2x][flag2y].pos;
           tank[i]->aimPos = Map[flag2x][flag2y].pos;
           getEndPoint();
           aim.pos.set( Origx, Origy, 0.1);
          }
        else
          {
           if  ( i > tankNum/3)
              {
               aim.pos = Map[flag2x][flag2y].pos;
               tank[i]->aimPos = Map[flag2x][flag2y].pos;
               getEndPoint();
               aim.pos.set( Origx, Origy, 0.1);
              }
          else
             {
              aim.pos = Map[flag1x][flag1y].pos;
              tank[i]->aimPos = Map[flag1x][flag1y].pos;
              getEndPoint();
              aim.pos.set( Origx, Origy, 0.1);
             }
         }
```

```
        }

    else if( i >= 1000.0/flag2.life-1.0 && tank[i]->life >= 70.0 )
      {
         if( i >=2 && i <=5)
           {
             // This is a test of making some aggressive AI tanks
( AI tank number 2 to AI tank number 5)
             // they will follow a human controlled tank( player tank
number 3) and keep attacking it
           aim.pos = player[3]->pos;
           tank[i]->aimPos = player[3]->pos;
           getEndPoint();
           aim.pos.set( Origx, Origy, 0.1);
           }
         else
           {
            aim.pos = Map[flag1x][flag1y].pos;
          tank[i]->aimPos = Map[flag1x][flag1y].pos;
          getEndPoint();
          aim.pos.set( Origx, Origy, 0.1);
         }
     }
   else if( i >=1000.0/flag2.life-1.0 && tank[i]->life <= 20.0)
     {
       aim.pos = Map[flag2x][flag2y].pos;
       tank[i]->aimPos = Map[flag2x][flag2y].pos;
       getEndPoint();
       aim.pos.set( Origx, Origy, 0.1);
     }
   else if  ( i<= 1000.0/flag2.life-1.0 )
    {
      aim.pos = Map[flag2x][flag2y].pos;
      tank[i]->aimPos = Map[flag2x][flag2y].pos;
      getEndPoint();
      aim.pos.set( Origx, Origy, 0.1);
    }

   else
    {
     Vector D = tank[i]->pos - flag2.pos;
     if( D.length() < 10.0)
       {
        aim.pos = Map[flag2x][flag2y].pos;
```

```
        tank[i]->aimPos = Map[flag2x][flag2y].pos;
        getEndPoint();
        aim.pos.set( Origx, Origy, 0.1);
       }
     else
      {
       if( i >=2 && i <=5)
           {
             // This is a test of making some aggressive AI tanks
( AI tank number 2 to AI tank number 5)
             // they will follow a human controlled tank( player tank
number 3) and keep attacking it
           aim.pos = player[3]->pos;
           tank[i]->aimPos = player[3]->pos;
           getEndPoint();
           aim.pos.set( Origx, Origy, 0.1);
           }
        else
        {
        aim.pos = Map[flag1x][flag1y].pos;
        tank[i]->aimPos = Map[flag1x][flag1y].pos;
        getEndPoint();
        aim.pos.set( Origx, Origy, 0.1);
       }
      }
    }

 }




    astar(obj.Startx, obj.Starty, obj.Endx, obj.Endy);
    path();
    tank[i]->dir = obj.dir;
    Vector distance = tank[i]->aimPos - tank[i]->pos;
    if( distance.length() > 2.0  )
    {tank[i]->moveTo(tank[i]->dir);}

   }
// The reason why we call the influenceMap function again is to draw
the original influen map
// It is just used for testing and it will slow down the program
running speed a little bit
  influenceMap();
```

```
}


// draw function
void display()
{
  // clear the current buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  // save the current transformation matrix
  // turn on the lights
  glPushMatrix();

    glRotated    ( (GLdouble) spinxface, 1.0, 0.0, 0.0 ) ;
    glRotated    ( (GLdouble) spinyface, 0.0, 1.0, 0.0 ) ;
    glRotated    ( (GLdouble) spinzface, 0.0, 0.0, 1.0 ) ;
// draw the mouse-dragged square
 if( drawSquare == true )
   {
    glColor3f(0.9,1.0,1.0);
    glLineWidth(1.0);
        glBegin(GL_LINE_LOOP);
            a.Vertex();
            b.Vertex();
            c.Vertex();
            d.Vertex();
            glEnd();
    glLineWidth(2.0);
   }
// draw the game map
 for( int y=0; y<H; y++)
    {
     for( int x=0; x<W; x++)
        {
         Map[x][y].draw();
         if( drawInfluence == true)
           {
            Map[x][y].drawInfluence();
           }
        }
    }

aim.draw();
```

```
for( int i=0; i<tankNum; i++)
   { tank[i]->Draw(); }

for( int i=0; i<playerNum; i++)
   { player[i]->Draw(); }

if( DrawPath == true)
   {
    drawParent();
    drawPath();
   }

 flag1.Draw();
 flag2.Draw();

 glPopMatrix();

 glutSwapBuffers();

}

// initialise the influence map
void initInfluenceMap( )
{
 for( int y=0; y<H; y++)
    {
     for( int x=0; x<W; x++)
        {
         Map[x][y].influence = 0;

        }
    }
}

// This function is just for testing, AI tanks will move randomly
void tankMove()
{
  for ( int i = 0; i< tankNum; i++)
  {
    tank[i]->moveRandomly(0, (W*(int)Map[1][1].step-1), 0, (H*(int)
Map[1][1].step-1));
  }

}
```

```
// Calculate the influence
void influenceMap()
{

 influenceRadius = 12.0;
 for( int y=0; y<H; y++)
   {
    for( int x=0; x<W; x++)
       {
        Vector distance(0,0,0);
        float iv ;

        for( int i=0; i < playerNum; i++)
           {
            distance = player[i]->pos - Map[x][y].pos;
            iv = 60 *player[i]->life*influenceRadius/(distance.length
()*100.0);
            if( 0.0<=distance.length()&&distance.length()
<=influenceRadius )
              {Map[x][y].influence += (int)iv;}

           }

       for( int i=0; i < tankNum; i++)
          {
           distance = tank[i]->pos - Map[x][y].pos;
           iv = 60 * influenceRadius *tank[i]->life/(distance.length()
*100.0);
           if( 0.0<=distance.length()&&distance.length()
<=influenceRadius )
             {Map[x][y].influence -= (int)iv;}
          }
      }
   }

}

// If we want to use the influence map for the human controlled tanks
too, we have to reverse it first because if
// the influence is positive for AI tanks, it should be negative for
human players.
void reverseIM()
{
```

```
  for( int i = 0; i<W; i++)
     {
      for( int j = 0; j<H; j++)
         {
          Map[i][j].influence = -1*Map[i][j].influence;
         }
     }

}

// initialise the game
void initGame()
{
  for ( int i = 0; i< tankNum; i++)
      {
       delete tank[i];
       tank[i] = new Tanks( 1);
       tank[i]->pos = Map[flag2x][flag2y].pos;
       tank[i]->initBullet();
      }
 for ( int i = 0; i< playerNum; i++)
     {
      delete player[i];
      player[i] = new Tanks( 0);
      player[i]->pos = Map[flag1x][flag1y].pos;
      player[i]->initBullet();
     }
 flag1.life = 1000.0;
 flag2.life = 1000.0;
 aim.pos = Map[7][7].pos;
}


float timeElapsed = 0;
                                                                        //
int Time,timeprev=0;
void Update(int i)
{
        Time=glutGet(GLUT_ELAPSED_TIME);
        timeprev = Time;

if(simulation==true)
{
 flag1.update();
```

```
 flag2.update();
 findTarget();
 tankRecoverNearBase();
 checkBulletCollision();
 Path();
}


glutTimerFunc(10,Update,0);
glutPostRedisplay();

}

/*! InitGL set's up the opengl drawing state */
void InitGL()
{
  glClearColor(0.0f, 0.1f, 0.0f, 1.0f);
  glShadeModel(GL_SMOOTH);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHT0);
  GLfloat AmbColour[]={0.2,0.2,0.2};
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT,AmbColour);

  Point3 Eye(38.0f,28.0f,72.5f);
  Point3 Look(38.0f,28.0f,0.0f);
  eye = Eye;
  look = Look;
  Vector Up(0.0f,1.0f,0.0f); //Y == UP
  Cam.set(Eye,Look,Up);
  Cam.setShape(45,640.0/480.0,0.5,150);

  glEnable(GL_COLOR_MATERIAL);
  glColorMaterial(GL_FRONT,GL_AMBIENT_AND_DIFFUSE);
}



void KeyPressed(unsigned char ch, int x, int y)
{
  switch (ch)
  {
                // ESC key to exit program
                   case 27 : exit(0); break;
```

```
                case 'i' : drawInfluence=true; break;
                   case 'I' : drawInfluence = false; break;
                case ' ' : initGame(); simulation = true; break;
                   case 'p' : simulation = false; break;
                   case 'a' : for(int i = 0; i<playerNum; i++)
                              {player[i]->selected = 1;}; break;
                   case 'h' : Cam.slide(camMoveX,camMoveY,0.0);
camMoveX=0;camMoveY=0; break;
                   case 's' : flag1.life = 1000.0; break;
                   case '7' : glPolygonMode
(GL_FRONT_AND_BACK,GL_LINE);DrawPath = true; break;
                   case '8' : glPolygonMode
(GL_FRONT_AND_BACK,GL_FILL);DrawPath = false; break;
                   case '9' : DrawPath = true; break;
                   case 't' : for(int i = 0; i<playerNum; i++)
                              {player[i]->life = 100.0;}; break;
                   case 'm' : if( simulation == false )
                              {CreateMap();}; break;

                   case '1' : for( int i=0; i<playerNum; i++)
                              { if( i< playerNum/3)
                                 {player[i]->selected = 1;}
                                else
                                 {player[i]->selected = 0;}};break;
                   case '2' : for( int i=0; i<playerNum; i++)
                              { if( i< 2*playerNum/3&& i>=
playerNum/3)
                                 {player[i]->selected = 1;}
                                else
                                 {player[i]->selected = 0;}};break;
                   case '3' : for( int i=0; i<playerNum; i++)
                              { if( i< playerNum && i>=2*playerNum/3)
                                 {player[i]->selected = 1;}
                                else
                                 {player[i]->selected = 0;}};break;

  }

 glutPostRedisplay();
}




// camera controls
```

```
void SpecialKeyPressed(int key, int x, int y)
{
   switch (key)
   {
   // slide the camera left right up and down
   case GLUT_KEY_LEFT   :
                Cam.slide(2.5,0.0,0.0);
                camMoveX-=2.5;
                cout<<camMoveX<<endl;
        break;
   case GLUT_KEY_UP     :
        Cam.slide(0.0,-2.5,0.0);
        camMoveY+=2.5;
        break;
        case GLUT_KEY_RIGHT     :
                Cam.slide(-2.5,0.0,0.0);
                camMoveX+=2.5;
   break;
        case GLUT_KEY_DOWN      :
        Cam.slide(0.0,2.5,0.0);
        camMoveY-=2.5;
        break;
   //slide the camera along the z axis
        case GLUT_KEY_PAGE_UP   :
        Cam.slide(0.0,0.0,2.5);
                break;
        case GLUT_KEY_PAGE_DOWN :
        Cam.slide(0.0,0.0,-2.5);
                break;




   }
 //redisplay the scene
glutPostRedisplay();
}




void motion ( int x, int y )
{
 if (leftKeyDown==1) {

       drawSquare = true;
```

```
          finx = x/15.0-Map[1][1].step/2.0-camMoveX;
          finy = ((float)HEIGHT-y)/15.0-Map[1][1].step/2.0-camMoveY;
          a.set( origx, origy, 0.3);
          b.set( finx, origy, 0.3);
          c.set( finx, finy, 0.3);
          d.set( origx, finy, 0.3);




  }
}

/* The button callback is called when a mouse key is pressed */

static void Button(int button, int down, int x, int y)
{

  switch(button)
  {
  case GLUT_LEFT_BUTTON:
          if (down == GLUT_DOWN)
       {
                 leftKeyDown = 1;
                 origx = x/15.0-Map[1][1].step/2.0-camMoveX;
                 origy = ((float)HEIGHT-y)/15.0-Map[1][1].step/2.0-
camMoveY;


       }

          if( down == GLUT_UP)
          {
            finx = x/15.0-Map[1][1].step/2.0-camMoveX;
            finy = ((float)HEIGHT-y)/15.0-Map[1][1].step/2.0-camMoveY;

            for ( int i= 0; i<playerNum; i++)
            {
            if ( abs(player[i]->pos.x-origx)+abs(player[i]->pos.x-
finx) == abs(origx - finx) &&  abs(player[i]->pos.y-origy)+abs(player
[i]->pos.y-finy) == abs(origy - finy) )
              {
             player[i]->selected = 1;
              }
```

```
            else
            { player[i]->selected = 0;}
          }
        leftKeyDown = 0;
          drawSquare = false;
      }
break;
case GLUT_RIGHT_BUTTON:
    if (down == GLUT_DOWN)
      {

              for ( int i =0; i< playerNum; i++)
                {
               if(player[i]->selected == 1)
                   {player[i]->selected = 2;}

                }

                Origx = x/15.0-Map[1][1].step/2.0-camMoveX;
                Origy = ((float)HEIGHT-y)/15.0-Map[1][1].step/2.0-
camMoveY;
                //cout<< aim.pos.x<< "and" << aim.pos.y<<endl;
                mouseTrack.set(Origx, Origy, 0.0);
                aim.pos.set( Origx, Origy, 0.1);
              for(int i=0; i<playerNum; i++)
                {
               if( player[i]->selected == 2)
                  {
                 player[i]->bulletAim.set( Origx, Origy, 0.4);

                  }
                }
                //cout<< origx<< " and" <<origy<<endl;
                //Rotate = 1;
      }
break;

}
}


// application main loop
int main(int argc, char **argv)
{
```

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DEPTH | GLUT_RGBA | GLUT_DOUBLE |GLUT_ALPHA
|GLUT_ACCUM);
glutInitWindowSize(WIDTH, HEIGHT);
glutCreateWindow("AI");
   //load the glut callbacks
glutDisplayFunc(display);
 //glutIdleFunc(Update);

  glutTimerFunc(10,Update,0);

glutMouseFunc(Button);
glutMotionFunc(motion);
glutKeyboardFunc(KeyPressed);
glutSpecialFunc(SpecialKeyPressed);
//initialise opengl
InitGL();
CreateMap();

glutMainLoop();

return 1;
}


//end of file
```