VIRTUAL VIRUS
(REAL-TIME INTELLIGENT STRATEGY)

MOHAMMAD SOPIAN BIN ABDUL JAMIL

A thesis submitted in fulfillment
of the requirements for the award of the degree of
Bachelor of Computer Technology (Software Engineering)

Faculty of Computer System & Software Engineering
University College of Engineering & Technology Malaysia

OCTOBER, 2006

ABSTRACT

Computer games have grown considerably in scale and complexity since their humble beginnings in the 1960s. Modern day computer games have reached incredible levels of realism, especially in areas like graphics, physical simulation, and artificial intelligence. However, despite significant advances in software engineering, the development of computer games generally does not employ state-of-the-art software engineering practices and tools. The goal for this game is to reach the human-level AI real – time strategy. This system is near to reach AI that can be reacts with human, example evaluate their resource, enemy unit quantity, position, etc. This game has its own ability which the selected unit has their own advantages and disadvantages that can be used if the strategy is well-managed.

ABSTRAK

Permainan komputer telah meningkat dengan mendadak dan kesukaran sejak ianya mula dibina pada tahun 1960-an. Pada zaman moden ini permainan komputer telah mencapai kemajuan yang menakjubkan dari segi maya, terutama sekali dalam bahagian grafik, simulasi fizikal dan kepintaran buatan. Bagaimanapun, kehebatan pembaharuan dalam kejuruteraan perisian, pembinaannya dalam membina permainan komputer secara umum tidak mengunakan pakai seni terkini kejuruteraan perisian kebolehan dan alatan. Objektif permainan ini adalah untuk membina sebuah permainan perisian yang mengunakan strategi masa sebenar. Sistem ini juga hampir mencapai taraf kepintaran buatan yang dapat berinteraksi dengan manusia dengan cara menilai keadaan musuh contohnya sumber, bilangan musuh, kedudukan dan sebagainya. Permainan ini juga mempunyai keistimewaan tersendiri di mana unit yang digunakan ada kelebihan dan kekurangan yang dapat digunakan sekiranya bijak dalam mengunakan strategi yang tertentu.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

RTS	-	Real time strategy
AI	-	Artificial Intelligent
AIP	-	Artificial Intelligent Personality
API	-	Application Program Interface
2D	-	Two Dimension
ID	-	Information Data

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

In this chapter, an introduction to real time strategy is presented, followed by the problem statement, the objective and scope of the project.

1.1 Introduction

The overview about Virtual Virus, Real Time Strategy, type of unit and the game play. It also determines how the game can be play and what sides that human and computer take part.

1.1.1 Overview Virtual Virus (Real Time Intelligent Strategy)

Game Strategy is focusing on the ability to make deal with dynamic priorities, typically in a context of resource shortage or typically involve intricate rule systems where player must master tactics and strategies rather than fast reflexes. Strategy games may be divided into: Real-time strategy games and turn-based strategy games. This game is the real time strategy which is a type of computer strategy game which does not have "turns" like conventional turn-based strategy video or board games. The name of the game is "Virtual Virus Real – Time

Interactive Strategy". This software is about the real time strategy game. It tests the intelligent and creativity player to defense base and attack enemy territory base. There are two (2) types of sides: -

1. Virus
2. Anti virus

Player is take part as anti virus that defend the computer from being corrupted by virus. Anti virus can get the resource from defend the computer by destroyed the virus. In additional to win the game, anti virus can update the database, allow them to upgrade and get more powerful to protect the data from being corrupted. Upon meeting the virus and anti virus begin fight each other until one of sides is totally extinction. The player can command their unit by issued via mouse-selected icons by selecting a unit and give an order with their creativity to play until win.

For the game play, the player (human) control the anti-virus unit and the computer or AI control the virus. In this game, anti-virus has a base which contains a main building. The main building can build an anti-virus unit. If the main base destroys, the game is over by player defeated. Virus is created in the certain place and time to attack the base of the anti-virus. Victory can be archive if the player destroys the source of where the virus created.

1.2 Problem Statement

In the past strategy game, the problem was been found. Some of the games have the problem that makes their game not very interactive, this is because:

1. Most of game strategy does not have good mini map.
2. Many strategy games their GUI is not user friendly.
3. Most of the strategy game, the Artificial Intelligent is bad in decision making.

1.3 Objectives

1. To develop 2d interactive games that shows the role of anti-virus and virus.
2. To apply some simple AI in the game.
3. To create the engine to running the RTS game.

1.4 Scopes

1. This application played by one person and the opposition is computer.
2. The application simulate in 2d graphic.
3. Computer can interact with player action and the environment of the game.
4. Using the C++ language with the standard library such as msimg32.lib and winmm.lib.

1.5 Method and Technique

The project can be developed with C++ and some library needs to compile this application.

CHAPTER 2

LITERATURE REVIEW

This chapter devoted to a survey of the concept of the real-time strategy game found in literature which include definition, artificial intelligent and design in real time strategy.

2.1 Definition Real Time Strategy

Real-time strategy games ask players to collect and manage resources that include food, raw materials, and more, research technologies to improve a civilization, and ultimately control and maneuver armies to battle and take over the world. Popular real-time strategy games include: Age of Mythology, Rise of Nations, and Command & Conquer [1]. A real-time strategy (RTS) game is a type of computer strategy game which does not have turns like conventional turn-based strategy video or board games. Rather, game time progresses in real time [2].

Because of the generally faster-paced nature (and the usually shallower learning curve), RTS games have surpassed the popularity of conventional turn-based strategy computer games. In the past some traditional strategy gamers regarded RTS games as cheap imitations of turn-based games, arguing that RTS games had a tendency to devolve into clickfests, in which the player who was faster with the mouse generally won, because they could give orders to their units at a faster rate. Real-time strategy enthusiasts counter that micromanagement involves not just fast

clicking but also the ability to make sound tactical decisions under time pressure. It is noteworthy, however, that due to the games being shorter because of the faster pace of the game and absence of turn switching pauses, RTS games are far more suitable for Internet play than turn-based games; this is indubitably an important reason for their popularity. Furthermore, turn-based games are ill-suited to meet the increasing demand for realism from casual gamers and they require a greater time commitment than real-time strategy games [2].

The more recent generations of RTS games usually have features which reduce the importance of fast mousework, enabling the player to focus more on overall strategy. For example, queuing allows the player to put in an order for multiple units at a single building instead of requiring the player to return to that building to order the next unit built whenever a unit ordered earlier is completed. The ability to set waypoints allows the player to give multiple movement commands to a unit at once. Generally, most RTS games follow the same general pattern:

1. Build up base and forces (the economy).
2. Acquire more resources.
3. Attack the enemy, attempting to deprive him of resources and destroy his infrastructure [2].

2.2 Ai Using In Real Time Strategy

Real-Time-Strategy (RTS) games – such as the million- sellers Starcraft by Blizzard Entertainment and Age of Empires by Ensemble Studios – can be viewed as simplified military simulations. Several players struggle over resources scattered over a 2D terrain by setting up an economy, building armies, and guiding them into battle in real-time. RTS games offer a large variety of fundamental AI research problems, unlike other game genres studied by the AI community so far:

Resource management, which mean the players start off by gathering local resources to build up defenses and attack forces, to upgrade weaponry, and to climb up the technology tree. Proper resource management is a vital part of any successful strategy.

Decision making under uncertainty, which mean the players are not aware of the enemies' base locations and intentions. They have to gather intelligence by sending out scouts. If no information is available yet, the players must form plausible hypotheses and act accordingly.

Spatial and temporal reasoning, which mean the static and dynamic terrain analysis as well as understanding temporal relations of actions, is of utmost importance in RTS games and yet, current game AIs largely ignore these issues and fall victim to simple common-sense reasoning.

Collaboration in RTS games groups of players can join forces and intelligence. How to coordinate actions effectively by communication among the parties is a challenging research problem.

Opponent modeling, Learning is one of the biggest shortcomings of most (RTS) game AI systems is their inability to learn from experience. Human players only need a couple of games to spot opponents' weaknesses and to exploit them in upcoming games. Current machine learning approaches in this area are inadequate.

Adversarial real-time planning is in fine-grained simulations, agents cannot afford to think in terms of micro actions. Instead, abstractions have to be found which allow a machine to conduct forward searches in a manageable abstract space and to translate found solutions back. Because the environment is also dynamic, hostile, and smart adversarial real-time planning approaches need to be investigated [3].

2.3 Designs in Real-Time Strategy

Data structure and coding examples is mostly be taken from the writer project, Artifact. Artifact is an Internet-based, client-server, multi-player, persistent-world, real-time strategy game. Artifact is currently in pre-release, with full release scheduled for Fall 1999 [4].

2.3.1 The Map

The map is probably the single most important data structure in a game of this type. The map serves as the central repository of nearly all game data and is the primary source of information used by the game logic. The various units, "fog of war", unit AI, and so on, all rely extensively on the map.

The most basic structure of the map is the map cell. A map cell is simply a single map location. The information that store in a map cell depends on the game that creating, but there are a few common elements. Such common elements include an indicator of the type of map cell (for instance, the terrain), the structure (or structure segment) built on the cell, and a pointer to the first mobile unit on the map.

```
struct world_struct
    {/* map cell */
    unsigned char reg;
    TRP_TYPE *trp;
    long ore,crops,wood,stone;      /* raw materials */
    char structure;                 /* bdg is a: 0 city, 1
facility, 2 artifact STR_* */
    BDG_TYPE bdg;                   /* pointer to union of city or
facility */
    };
```

Figure 2.1 Artifact's map cell [4]

The straight overhead map and angled isometric map have the simplest possible scenario: a two-dimensional array of map cells. The layered isometric map can also use a two-dimensional array, but there are several complexities. The biggest difference concerns adjacencies in the map [4].

Table 2.1 : Overhead and Angled Isometric Adjacency [4]

Direction	Location
North	(x,y-1)
Northeast	(x+1,y-1)
East	(x+1,y)
Southeast	(x+1,y+1)
South	(x,y+1)
Southwest	(x-1,y+1)
West	(x-1,y)
Northwest	(x-1,y-1)

2.3.2 Unit Data Structures

Once have the map structure determined, move on to unit data structures. There are 2 primary types of unit: mobile and stationary. Mobile units are the warriors, tanks, aircraft, and so on, that actually move around the map. The stationary units, on the other hand, are the buildings or building segments that do not move.

The common elements of mobile and stationary units include an identifier, a map location, an owner, and so on. These common elements may allow using object-oriented techniques and creating a common ancestor class for both kind of units, but that depends on the game and game logic needs.

Stationary units generally do not move around the map. Once a stationary unit is placed on the map, it stays there until the player removes it or an enemy unit destroys it. This can simplify how to handle stationary units. Since there can only be one stationary unit per map cell, the map cell needs only to have a pointer to such a unit. If the pointer is null, then there is no stationary unit occupying that map cell.

Mobile units, unlike stationary units, can move from one map location to another. They also have another significant difference from stationary units; there is technically no limit to how many mobile units can be in a single map location. Thus, it needs to be maintaining a list of all mobile units per map cell. Depending on the game, it's possible to limit mobile units so that they cannot stack in a single location. Or it may want to allow stacking only in locations with a special stationary unit (like a barracks). In this case, the map cell no longer needs to maintain a list, though the special stationary unit would.

These lists must keep mentioning for mobile units in the same map location or inside the same stationary unit, do not need to be incredibly complex. A simple linked list would likely work fine, though it might want to include some form of sorting to support game logic such as determining which unit in a location takes damage first or which units can leave first. To further simplify the lists, it could embed the list information that just a simple next pointer in the actual unit data structure. If the unit can simultaneously be in multiple lists, however, it may want to use an actual list container class.

It's important, however, no matter how many different lists a particular unit is in, that units be stored in a centralized data structure. A one-dimensional array works quite well for this and even allows for certain performance benefits when processing all units of a particular type. The unit's position in the array also provides a very handy identifier for that unit.

The game may have noticed that the unit data structures include the map location, and that the map cells have a list of units in those cells. This may seem

redundant, but is nearly always necessary. If the games are processing the map cell by cell, it needs a fast way to know which units are in each cell. Conversely, if the games are processing all units need to know their location without searching the entire map. This puts a mild burden on the programmer to make sure that the units always have the correct location and that the map cells they travel through are correctly updated. Example of information maintained in the unit data structure. The ID of a troop is simply its position in the single-dimensioned Troops array. The ID is stored in the troop structure even though it equals the array index because often it's necessary to know the troop's ID and only have a pointer to the troop [4].

```
struct troop_struct
    {
    int id;
    int owner_id;
    short int x,y;
    ...
    int count;              /* number of members of this troop
*/
    short int morale;       /* 0=worst, 255=best */
    short int training;
    short int experience;
    short int fatigue;      /* 255=worst, 0=best */
    ...
    TRP_TYPE *next,*prev;             /* linked list for troop
update */
    TRP_TYPE *next_here,*prev_here;  /* linked list of troops
at an x,y */
    TRP_TYPE *btl_next,*btl_prev;    /* linked list for
battles */
    ...
    };
```

Figure 2.2 Artifact's troop structure [4]

The conclusion is the map game going to be the central data structure, with the arrays of unit data structures providing the necessary details.

2.4 Developing game using C++

This topic is describing how to develop this application using the C++. This includes drawing the graphic object, transparent the graphic and to tracking the mouse clicked.

2.4.1 Sprucing Up the Bitmap Class

Bitmaps definitely are square graphical objects [5]. Transparency is that can identify a color as the transparent color, which is then used to indicate parts of a bitmap that are transparent. When the bitmap is drawn, pixels of the transparent color aren't drawn, and the background shows through.

From a graphics creation perspective, bitmaps can be created with transparency by selecting a color that isn't used in the graphics, such as hot purple, which is also known as magenta. Use magenta to fill areas on the bitmaps that need to appear transparent. It's then up to the revamped game engine to make sure that these transparent regions don't get drawn with the rest of the bitmap.

The trick making bitmap transparency work in the game engine is to expand the existing Bitmap::Draw () method so that it supports transparency. This is accomplished by adding two new arguments:

1. bTrans—A Boolean value that indicates whether the bitmap should be drawn with transparency.
2. crTransColor—The transparent color of the bitmap.

It's important to try making changes to the game engine that doesn't cause problems with programs that already written. Therefore, rather than add these two arguments to the Draw () method and require them of all bitmaps, it's much better to add and provide default values:

```
void Draw(HDC hDC, int x, int y, BOOL bTrans = FALSE,
COLORREF crTransColor = RGB(255, 0, 255));
```

Figure 2.3 A function to transparent bitmap [5]

The TransparentBlt () function is part of the Win32 API, but it requires the inclusion of a special library called msimg32.lib in order for the games to compile properly. This is a standard library that should be included with the compiler, but need to make sure that it is linked in with any programs that use the TransparentBlt() function [5].

2.4.2 Tracking the Mouse

The Win32 API includes a series of mouse messages that are used to convey mouse events, similar to how keyboard messages convey keyboard events [5]. The following are the mouse messages used to notify Windows programs of mouse events:

1. WM_MOUSEMOVE—Any mouse movement
2. WM_LBUTTONDOWN—Left mouse button pressed
3. WM_LBUTTONUP—Left mouse button released
4. WM_RBUTTONDOWN—Right mouse button pressed
5. WM_RBUTTONUP—Right mouse button released
6. WM_MBUTTONDOWN—Middle mouse button pressed
7. WM_MBUTTONUP—Middle mouse button released

The first mouse message, WM_MOUSEMOVE, lets to know whenever the mouse has been moved. The remaining messages relay mouse button clicks for the left, right, and middle buttons, respectively. A mouse button click consists of a button press followed by a button release. Implement a mouse dragging feature by

keeping track of when a mouse button is pressed and released and watching for mouse movement in between.

The mouse cursor position is provided with all the previously mentioned mouse messages. It's packed into the lParam argument that gets sent to the GameEngine::HandleEvent () method. The following is the prototype for this method is like below.

LRESULT GameEngine::HandleEvent(HWND hWindow, UINT msg, WPARAM wParam, LPARAM lParam);

The wParam and lParam arguments are sent along with every Windows message and contain message-specific information. In the case of the mouse messages, lParam contains the XY position of the mouse cursor packed into its low and high words. The following is an example of a code snippet that extracts the mouse position from the lParam argument in a WM_MOUSEMOVE message handler:

```
case WM_MOUSEMOVE:
WORD x = LOWORD(lParam);
WORD y = HIWORD(lParam);
return 0;
```

Figure 2.4 Mouse move handler [5]

The wParam argument for the mouse messages includes information about the mouse button states, as well as some keyboard information. More specifically, wParam lets to know if any of the three mouse buttons are down, as well as whether the Shift or Control keys on the keyboard are being pressed. The following are the constants used with the mouse messages to interpret the value of the wParam argument:

1. MK_LBUTTON—Left mouse button is down.
2. MK_RBUTTON—Right mouse button is down.
3. MK_MBUTTON—Middle mouse button is down.
4. MK_SHIFT—Shift key is down.
5. MK_CONTROL—Control key is down [5].