Project: 3D Ant Attack For Windows

By: David Morrissey

Year: 1994

Supervisor: Michael Coughlan

Contents

Contents (Contd.)

Summary
3D Ant Attack is an adventure game that involves a rescuer (you), a rescuee, several large ants, and a city. The aim of the game is to find the rescuee in the city, and lead her to safety, taking care to avoid the vicious ants.

Designing the game to suit Windows was a task that involved a lot of trial and error. In order for the game to be useful, it had to have the look and feel of a real-time game. Designing the game was no longer enough. Designing the game to not be annoyingly slow became the top priority. This necessitation prompted constant improvement in the design of all the elements in the game. The result is a game which is tailorable to run reasonably fast on a low resolution display device. As playability was an objective, other features came into consideration, such as allowing the layout of the city to change. As the game is in 3D, your character can get hidden behind a wall or building, so the ability to change your point of view is included.

The movement of characters was another major part of this project. The humans had to act like humans, and the ants had to move like ants. They also has to respond to the user in a predictable manner. Different methods were used to achieve this with varying degrees of success.

All in all, the project involved many different tasks, ranging from the designing of the pictures that would represent the characters, to investigating the ins and outs of memory management and graphics functions under Windows. The project was written in C++, and uses many of this languages features to achieve its results.

While the game can be viewed as complete in itself in that it provides a challenge to the user, and behaves in the predictable format of a game, there are still possibilities for extending, as well as improving this game.

Chapter 1 - Introduction

Project Specification

The game "3D Ant Attack" was written in 1983 for the ZX Spectrum computer. This project involves writing a version of this game for I.B.M. compatible personal computers running Microsoft Windows.

3D Ant Attack, The Original Game

The Plot

You and your travelling companion stumble on a city that is inhabited by unusually large, human-eating ants. Being curious, your companion steps inside the city and is taken captive by the ants. She manages to free herself however, and now sits, shocked and trembling, in some dark corner of the city, awaiting your help.

The ants are vicious and hungry. They can sense your presence. They want human flesh. They don't give up the chase. You cannot outrun them.

Your only protection is a fistful of grenades and your survival instinct. Can you find your companion, and lead her out of the city in the allotted time?

Gameplay

You guide your character through a 3D city. The city is walled all the way around and has only one entrance/exit. The structures within the city, being built by insects, are crude and simple, and contain many possible hiding places, as well as many ant holes. You may climb up walls that aren't too high above your head, and you may jump down from reasonable heights in safety. The game ends if you fail to rescue your companion within the allotted time, or if you are otherwise killed (e.g. from a fall from too high). With time, more and more ants become aware of your presence, and track you down. The effect of an ant bite is a reduction in your remaining time. Stay safe from the ants by climbing onto a block, out of their reach.

3D Ant Attack, This Game

To copy the original game, the following things are needed:

1. Some method of representing the city and it's contents.
2. Some method of representing characters (humans and ants).

3. Some method of moving characters around.
4. Some method of showing the user what is happening (display issues)
5. A general format for the game (i.e. the longer you play, the shorter time you have, the more voracious the ants).

Design issues for the project are covered in Chapter 2. Movement of the characters is covered in Chapter 3.

Control Flow

The basic control flow is as follows:

1. If the user has made pressed a movement key, move the rescuer accordingly.
2. If the rescuee has been rescued, move her closer to the rescuer (if possible).
3. Move each of the ants.
4. Update the screen, including the direction indicators, and the time remaining.
5. Goto 1.

As well as this there are several other small parts, such as controlling the emergence of ants from the depths of the city, and checking to see if the rescuee has made it out of the city.

Game Format

The game starts off easy, with plenty of time, and ants that are slow to emerge. As you progress through the rounds, the ant emergence time decreases, and the time you have is reduced.

Chapter 2 - Design Issues

This chapter deals with design issues. Several issues were addressed in the design of the game. The most important of these issues is performance. The graphical nature of the game means that a lot of drawing has to be done. Unfortunately, Windows does not handle real-time drawing very well. Because of this, a lot of effort went into designing elements of the city for speed.

Substance of the City

What is the city made of ? Looking from above, the city could be described as a 2D array of columns. A column can contain anything from zero to a controllable limit of blocks. The pre-requisites of the city are as follows:

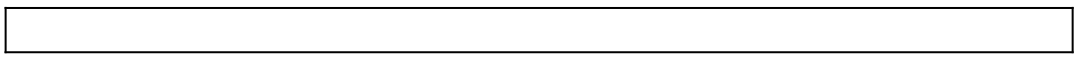
1. It should be capable of holding a large quantity of information persistently.
2. Stored information should be changeable.
3. Storage and retrieval of information within the city should be reasonably fast.

The first attempt at designing the city was using bytes as columns. The bits within each byte were on to signify the presence of a block, or off to signify no block. Nothing within the byte itself told of the presence of a rescuer or ant. This information was stored separately. Therefore, when it came to the drawing of a column, a check had to be made to see if a character existed in each location. This was clearly undesirable, as the number of ants can be large.

For the purposes of drawing, it would be more efficient to have all the information needed to draw a scene stored within the scene. The second attempt used two bytes with this in mind. The first byte contained the column information, while the second contained information regarding characters that were currently in that column. However, it is possible that there could be more than one character in a column (e.g. the rescuer standing on a block under which there is an ant). As well as that, the number of blocks in a column is limited to less than eight (the number of bits in a byte). This approach did not cater satisfactorily for these.

The third attempt is the approach that is used in the game. It involves using one byte per block. The value within the block determines what is to be drawn. Everything that needs to be drawn within a scene is contained in the scene. Also, there is no limit to the height of a column.

The following diagram depicts the flexibility of this system:



Here, the numbers indicate the value that is stored in the relevant place in the column in question. In drawing this column, the value "20" is a direct instruction to draw the feet of the rescuer. When the rescuer moves, the values "20" and "21" are wiped, and return to zero. The great advantage here is that each unit within a column could hold one of one hundred and twenty eight different types of character. This means the possibility of a great many different types of blocks, characters, and ants.

There are disadvantages to this system over the other two:

1. Memory cost is high.
2. Functions that operate on the contents of the city as a whole (such as rotating, initialising) take significantly longer.

Memory Cost

To store a city that has two hundred columns across and down, each column being eight units high, memory required is calculated as follows:

200 * 200 * 8 bytes
= 320, 000 bytes
= 320 Kilobytes

Length of Time

Many times more data is being manipulated, so the time taken is longer. Also, the information cannot be stored in the program's default data segment, so the address of each block takes longer to calculate.

Layout of the City

One of the considerations for the layout of the city is that with time, you will get to know the buildings. One way around this would be to get WinAnt to design random buildings each time the game is played. However, it would be difficult to guarantee that a place is accessible if that was done. Another possibility is to allow someone to design buildings that could be used in the game. If there was a large selection of these buildings, and they were placed at random, then it would be more difficult to learn about them.

That is the approach that is used. The city is surrounded by a wall, and inside this wall, the city is divided into equally sized areas. Into each of these areas, a sector can be loaded at random. Each of the sectors is pre-defined, having buildings, hiding places for the rescuee, and ant-holes for the ants. Each time a new game starts, or after a rescuee has been rescued, the city re-initialises itself.
This allows for a fair amount of newness to each city, directly proportional to the

number of sectors that are defined.

Animation

Animation involves placing pictures in front of the viewer in rapid succession. If the screen can be updated quickly, smooth animation is possible. The trick for this game is to create the pictures as quickly as possible. Because this game uses 3 dimensions, a lot of what is drawn is then completely or partly covered so the time spent drawing the covered part is wasted. Drawing a scene involves starting off with the background, and adding columns one by one. Drawing directly to the screen would result in a very blinky display, as the scene would be drawn, then deleted, and then redrawn. Therefore, the drawing of the screen has to be a two step process. Firstly, all the assembly and drawing of a scene is carried on with behind the scenes, in memory. Then, the resulting bitmap is copied to the screen, on top of the last frame. The two steps involved make the update operation even more time costly.

Drawing a Scene

Drawing a scene is the trickiest part of the project. The problem is that you have a city data structure, full of information and you want to draw only the relevant scene.

The Scene

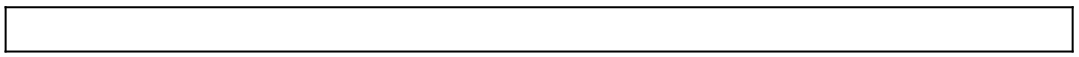
The scene is the area immediately surrounding the rescuer. When the rescuer moves, the scene changes. As ants move, they venture in and out of the scene. To draw the scene, a few observations are necessary:

1. The rescuer is always in the centre of the screen.
2. There are always the same number of columns to be drawn.

These observations tell us that:

1. It is possible to calculate the column to start drawing from by looking at the rescuer's position. This is the starting position.
2. The area in the city to be drawn will always be the same relative to this starting position.

The following diagram represents the columns that are drawn relative to the starting position:



The topmost shaded block above is the starting position. The parts of squares shaded black are outside the edge of the screen. Drawing a scene involves drawing the furthest away columns first, and overlaying these with the nearer columns until the scene is completed.

Drawing a Column

There were many attempts to find the fastest way to draw a column. However, due to the shape of blocks, and the way in which Windows stored bitmaps (only rectangular shapes) all approaches would have this in common: In order to draw shapes that aren't rectangular, you firstly need to place a mask of the same shape on the screen.

The first attempt was to draw complete columns onto the screen in one operation. Each column had a mask as well, so there was a lot of drawing. Consequently, it was slow. Also, special provision had to be made for columns that contained the characters.

The second attempt drew each block individually. Again, each block had a mask. This was an improvement, as time was not wasted drawing where there wasn't a block (The column approach "drew" spaces as well as blocks), but further improvements were possible.

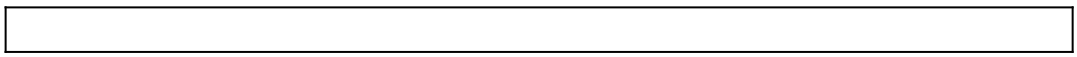
The final attempt reduces the drawing to a minimum. Instead of drawing each block, each column is examined and broken up into groups of similar things (e.g. the column that reads bottom to top "block, block, space, space, block, block, block, space" is broken up into "draw two blocks, draw two spaces, draw three blocks, draw a space" instead of "draw a block, draw a block, draw a space, draw a space, draw a block, draw a block, draw a block, draw a space."). Then for each group of blocks, a mask is drawn for only the very bottom of the bottommost block (the portion that can't be represented using a rectangular bitmap), and the relevant block portion is drawn on top. Then for each block, the rectangular portion is drawn (no mask). Then for the topmost portion of the top block, a mask is drawn, followed by the top part of the block.
(see Appendix B for more). This attempt also give the option of turning off and on the colour. When colour is off, there is only one colour plane in the frame, so drawing is fast. When colour is used, there are many colour planes, and the same drawing has to be copied to each frame. This makes the whole operation significantly slower.

Views

In the original game, you could view the scene from different angles. The reason for this was that the characters often became obscured behind blocks. To bring the characters back into view, all you had to do was change the viewing position.

For this project, the first attempt was to design a viewing system where the same scene could be viewed from four different positions. This meant, however, that there was a lot of replication of similar code. The algorithm for drawing the city had to exist in four different versions. This was awkward, and effectively meant that any change or improvement in one version had to be copied in a different form to three other versions.

The next attempt involved taking three viewing versions of the four away, leaving just one. The viewing from different angles can also be achieved by rotating the contents of the city around. This seems like a lot of work, but is in fact more suitable for design than maintaining four different views. As well as rotating the contents of the city, the characters have to change position and direction.
The difference between the two viewing systems is this: With the first system, it is like having four different cameras, each facing in a different direction from the others. Switching views is like switching cameras. With the second version, it is like having one camera fixed permanently somewhere, e.g. a mountain top. Switching views effectively means that the whole contents of the city rotate around a fixed point (the centre) of the city, so what you see from the mountain changes. The concept is illustrated in the following diagram:



Using the second system above imposed limitations on the design of the city for minimum wastage of memory, namely that the distance from the top of the city to the bottom should be the same as the distance from left to right. By doing this, the rotated city could fit into the same memory address space as the original one.

Chapter 3 - The Characters

Operation of Characters

This chapter deals with the interaction between, and the operation and movement of the three main types of characters in the game, i.e. the rescuer, the rescuee, and the ants.

Interaction Between Characters

The main character is the rescuer. This is the character that is largely controlled by you. It is also the only character that is directly controllable.

When the rescuee has been located by the rescuer, she will attempt to follow in his footsteps.

As soon as the rescuer and rescuee have moved, each of the ants get a chance to move. Their movement is also largely dependant on where the rescuer is located, however, they also use randomness to make them look antlike.

The Rescuer

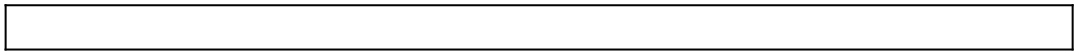
Controlling the Rescuer

Most of the rescuers movements are controlled by you. You have the following options for controlling the rescuer:

1. Walk left.
2. Walk right.
3. Walk up.
4. Walk down.
5. Jump left.
6. Jump right.
7. Jump up.
8. Jump down.
9. Jump.

Typically, these instructions to move are given by you by pressing a key. Your instruction will remain in effect until the key is released. This means, for instance, that the rescuer will continue to walk left until you release the walk left key. Sometimes however, as in the case of the rescuer jumping from a height, the rescuer may continue to make his own movements well after the key has been released. For example, when the rescuer jumps, he must also land. This is purely

to give a human-like appearance to the character. During the landing and recovery part, you have no control over him. The following diagram gives examples of where control is lost and regained by you in the movement of the rescuer:



Sometimes, you may press and release a key, and nothing happens. This is because of the way the program polls the keyboard to see if the keys are currently being pressed. Thus, a key must be currently be down when it checks. The alternative to this was to wait for the Windows-generated message to accompanies each key press. The problem with this is that the keyboard delay interferes with the movement of the rescuer. If you were to press the key and hold it down, the rescuer would move, then stop for a significant time before continuing movement.

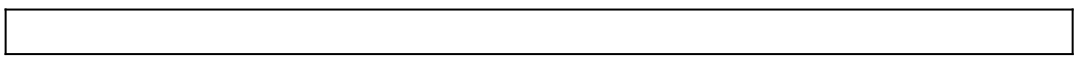
The rescuer will not move into solid objects. He will not jump if there is a block above his head. Also, his maximum jumping height is one block.

The Rescuee

The rescuee only moves when she has been located by the rescuer. Then, she achieves movement by looking at two things:

1. The location of the rescuer.
2. The direction in which the rescuer is facing.

This following diagram demonstrates the route that the rescuee will take in order to follow the rescuer.



The rescuee follows the above paths when the rescuer is on the ground. The third and fourth examples above enable the rescuer to move freely about without the rescuee getting in the way, but do not occur when the rescuer is on a building. If she did follow them, she would fall off.

As the rescuee tries to reach the location of the rescuer by moving in a straight line, if there is a building in the way, she will become lodged behind it. In this case it is necessary to move the rescuer so the rescuee is free to move.

There are limitations to the rescuee's movement, however. For example, if the rescuer moves into a dead end, he may become trapped in by the rescuee. This is because she faithfully follows him, unable to detect that she is in the way. The only option is to restart the game.

The Ants

The pre-requisites of the ants movement are:

1. They must move in an ant-like fashion.
2. They must pose a threat to the rescuer and rescuee.

The first pre-requisite suggests moving the ants at random. The second suggests that they should follow the characters. The ideal solution consists of a mixture of these.

Ant movement is operated as follows. Firstly, a walking distance is selected at random. Walking distances may be from zero to a controllable limit (e.g. four). Next, a random number is generated to decide whether to move in a random direction, or to attempt to get closer to the rescuer. The probability of each of these can also be controlled. If the movement is at random, then the ant will try to go straight ahead, turn left or turn right (equal probability). If the movement is towards the rescuer, then another random number is selected to determine which direction to approach the rescuer from. The next time the ants get to move, one of two things happens.

1. If the walking distance remaining is zero, the process restarts.
2. If the walking distance remaining is greater than zero, then the ant attempts to move in it's current direction. If it happens at this stage to run into one of the characters, it bites. If a block is in it's path, then the walking distance is reduced to zero. Otherwise, it moves and the walking distance is reduced by one.

The appearance of the ant is created by using two different bitmaps for the ant for each of the four directions. Alternate bitmaps are used each time the ant moves, animating it's movement.

Chapter 4 - Improvements

Further Expansion of the project

The City

One of the main design considerations for this project is the creation of the buildings in the city. There is an element of randomness in the appearance of the city, but it is only to the level of arrangement of the sectors. At present, the contents of the sectors are hard coded within the game. However, it would be desirable to have the contents of the sectors in a separate file. If the game needed sector information, it could then go to the file in question. Different files could hold different types of buildings. There would be no limit on the number of sectors held within the file. Therefore, if the city measured four sectors across by four down, and the file contained one hundred sectors, then the game would load any sixteen sectors that it chooses. A separate program could exist purely for the creation of these sector files, in which you could see the sectors being created, again in 3D. The position of ant holes and hiding places could also be controlled.

The Humans

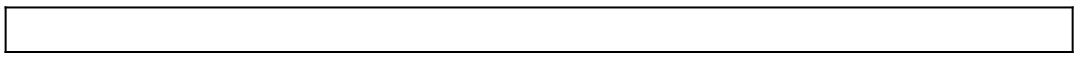
Again different humans with different abilities could exist. One character may have strong resistance to ants, but may not be very good at jumping. These characters could be held in a separate file and loaded into the game by the user at run time.

The Ants

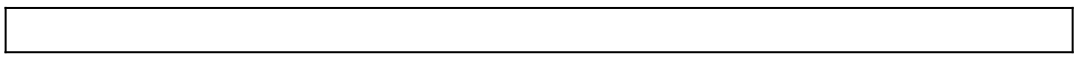
At presents, ants form a colony. There is only one colony and one type of ant. Different colonies could be designed whose ants look different and have different characteristics. These ants would then inhabit the same city. They could fight or even help each other in the pursuit of the humans. Their movement patterns could also differ.

Appendix A - Drawing the Screen

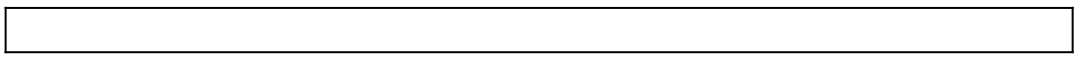
The following is a selection of screen shots showing how a column is drawn:



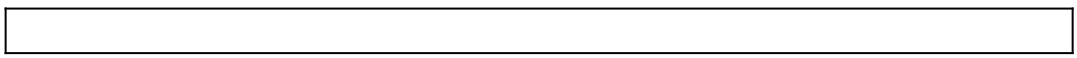
1. This shows the scene before any drawing is done.



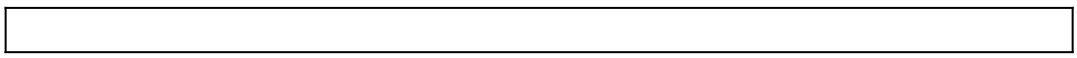
2. The first mask is drawn.



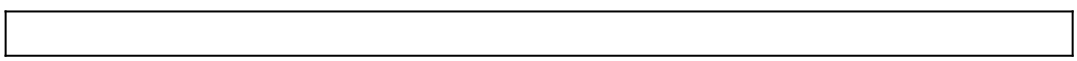
3. The lower portion of the bottom block is drawn.



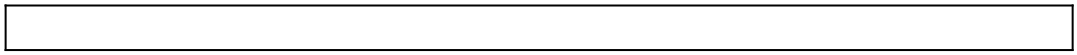
4. The body of the first block is drawn (no mask).



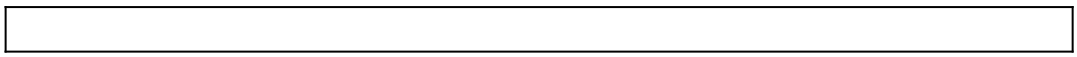
5. The body of the second block is drawn (no mask).



6. The body of the top block is drawn (no mask).



7. The mask is drawn for the top part of the top block.



8. The top part of the top block is drawn. This is the completed column.