Chapter 1 Chasing, Evading, Intercepting, and Patterned Movement

No matter what type of game you are designing, sooner or later you will have one or more people, creatures, or things chasing after or running away from another. Your player character enters a cavern currently occupied by twenty foul orcs. As your character stands there taking in the situation, the orcs grab their swords and rush towards your character. Or perhaps you are flying a spaceship and have just entered enemy space where a dozen enemy fighter ships are located. At once, they turn their ships and fly towards your craft. Perhaps, you decide to flee and turn your ship around and attempt to fly away from the oncoming fleet. Or perhaps some enemy ground based force decides to launch a guided missile at your spaceship. Countless scenarios arise where one or more items are supposed to close or chase another, or conversely, evade the chase.

However, when a missile is launched at your ship, an interception is a better approach. That is, a course is plotted so that the missile will follow a trajectory designed to intercept your ship. Finally, any number of pre-planned patterns of movement are desirable to add realism to the game. An orc on patrol around their camp is an example. A fighter plane doing a barrel roll is another. Patterned movements are merely an extension of the basic principles involved in chasing.

Game systems fall into two broad categories, as far as chasing is concerned: **tile-based** games and **continuous environment** games. In a tile-based game, only discrete locations are allowed to be occupied, often squares or hexagons. If one considers the screen to be composed of an array of locations, often 80 columns by 24 rows in a DOS screen, then any person, creature or object is constrained to be in one of these locations. Hence, the location is designated by providing its row and column coordinates. If the game is three dimensional, then merely add a third coordinate. In a continuous environment type of game, the person, creature or object is permitted to move freely in all directions. Its location is given by x and y coordinates, relative to some universal coordinate axis. (Add a z coordinate, if three-dimensional space is used.)

The Sample Programs and Model Coding

In order to see how the various algorithms are implemented, sample C++ coding is given. The prerequisite for this course is only the beginning C++ course. Hence, all coding samples must attempt to not exceed this level of experience in programming. Once you have learned Object Oriented Programming, you will at once see much better ways to encapsulate these algorithms in your games. Also, presumably you are concurrently taking your next course in C++ programming. In later chapters, I will use somewhat more sophisticated coding than at the start.

Even here in the very first chapter of Game Programming Theory, demo programs are needed. For the level of experience in C++ programming dictated by the course prerequisites, tile-based demos are vastly easier to understand, write, and follow. This is because essentially you are dealing with nothing more complex than a two dimensional array, typically here 80 columns by 24 rows. To further ease the burden, I have included my DOS Screen class encapsulation which is thoroughly presented and discussed in the Non-graphical Games Programming text. Here, you are only expected to be a user of that Screen class and the functions are very simple to use, much like those for cin and cout.

However, continuous environments must be shown, since the algorithms are implemented differently in such situation. The only way such environments can be displayed in with a Windows application, which allows individual screen pixel access for drawing operations. Unfortunately, basic Windows programming cannot be effectively studied until you have learned well Object Oriented Programming. Again, I will present a working model for our use in the demos. I will keep the Windows programming specific coding completely separate from the algorithm coding that we are learning. That way, you can concentrate on learning the algorithms and not worry about the fancy graphics until you progress to that level in your training.

Chasing and Evading

Chasing/evading is one of the most fundamental aspects of nearly any game and is an easy entrance point into game design. Three factors govern chasing and evading.

1. The decision to chase or evade must be made. This, of course, depends upon the situation.

2. An effective algorithm must be part of the game engine to implement a chase or an evasion.

3. During the movement, obstacles must be avoided. It is unrealistic to have an orc smash through a table and a bed while chasing toward your player character.

This chapter handles only the second factor: the algorithms needed to implement either chasing or evading. Note that evading is merely going in the opposite direction from a chase. That is, if we get the logic down for handling a chase, to handle an evade, one merely negates the directions chosen.

Historically, what I call Basic Chasing is very simple to implement, but the observed results are far from satisfactory in terms of game realism as we will see.

The Basic Chasing algorithm consists of always changing the chaser's coordinates so as to decrease the distance between it and the chasee. Evading is the opposite, the chasee attempts to increase the distance between it and the chaser. The logic is very simple to implement. If we use an orc versus a player, then the algorithm is implemented in one of two ways, depending upon whether it is tile-based or continuous movement.

In a tile-based environment, it is done this way.

```
if (orcRow > playerRow)
     orcRow--;
else if (orcRow < playerRow)
     orcRow++;

if (orcCol > playerCol)
     orcCol--;
else if (orcCol < playerCol)
     orcCol++;
```

In a continuous movement system, the code is done this way.

```
if (orcX > playerX)
     orcX--;
else if (orcX < playerX)
     orcX++;

if (orcY > playerY)
     orcY--;
else if (orcY < playerY)
     orcY++;
```

If implementing evading, just reverse the increments and decrements.

```
if (orcX > playerX)
     orcX++;
else if (orcX < playerX)
     orcX--;

if (orcY > playerY)
     orcY++;
else if (orcY < playerY)
     orcY--;
```

Using this algorithm, the orc will unrelentingly chase after the player. However, as beginning game programmers, we really need to see and study the visual effects that this and other algorithms produce. One could invent some graphical game engine in which you could program these algorithms and then run them to see the effects. However, doing so implies writing vastly more complex code than can be found at the beginning level of programming. Hence, I will use a very simple model adapted from the Non-graphical Games Programming course. This model, called Screen, encapsulates the old DOS screen of 80x24. In that course, you will study how this encapsulation works. For now, we are only going to be users of this simple object oriented class, treating it similar to the way that we use either cin or cout.

The Screen Class

To use this class in your programs, copy the two files, Screen.h and Screen.cpp, from the Samples folder and paste them into your project's folder. Right click on your project and add these existing two items to your project. In your main program, include the line

```
#include "Screen.h"
```

You **must** make two project settings or face numerous compile errors. Go to Project Settings and make two changes. C++ tab, General tab, set Detect 64bit portability to "No." General tab, set Character Set to "Not Set".

In your main function, add a line to define an instance of the Screen, such as:

```
Screen s (Screen::Blue, Screen::BrightYellow);
```

You are specifying the background color and the foreground color defaults to be used. The possible colors are:

```
Screen::Black      Screen::Blue         Screen::Green
Screen::Aqua       Screen::Red          Screen::Purple
Screen::Yellow     Screen::White        Screen::Gray
Screen::BrightBlue Screen::BrightGreen  Screen::BrightAqua
Screen::BrightRed  Screen::BrightPurple Screen::BrightYellow
Screen::BrightWhite
```

Next, you can provide a title to your DOS Console Window by using the function SetTitle(), which is passed a character string to be used as the caption or title.

```
s.SetTitle ("Tile-based Basic Chasing");
```

A nice, but not necessary effect, is to draw a box around the screen in a different color scheme. This is done using the DrawBox() function. The function is passed the coordinates (row and column) of the upper left corner of the desired box and the coordinates of the bottom right corner. Additionally, it is passed the background and foreground colors to be used.

```
s.DrawBox (0, 0, 24, 79, Screen::Gray, Screen::BrightYellow);
```

Output can be done in a number of ways. One way is to display a single character at a time. This is what we need in this chapter. The function is called OutputUCharWith(). It is passed the unsigned character to be displayed, the x and y coordinates at which to display the character, and the color scheme to be used, background and foreground. For example, one could display 'F' for Frodo this way.

```
s.OutputUCharWith ('F', PlayerAtY, PlayerAtX, Screen::Blue,
                   Screen::BrightRed);
```

Note that you can pass it a normal char character, because a char can be converted to an unsigned char. It is an unsigned char because this way some of the upper ASCII characters can also be displayed properly.

At anytime, the screen can be cleared using the ClearScreen() function. It uses the installed default color scheme setup when the Screen instance was defined.

```
s.ClearScreen ();
```

At any time, the color scheme can be changed by using the SetColor() function, providing the background and foreground color desired.

```
s.SetColor (Screen::Red, Screen::Green);
```

The Screen supports a highlighting, alternate color scheme. To use it, call the SetHighlightColor() function to install the highlighting colors.

```
s.SetHighlightColor (Screen::Red, Screen::Green);
```

Normal outputting to the screen occurs at the current X-Y location, called the cursor position. The cursor position can be set at any time to any location within the 80-24 canvas using the SetCursorPosition() function, specifying the row (Y) and column (X) coordinates.

```
s.SetCursorPosition (row, col);
```

Alternately, the function GoToXY() does the same thing. Chose which version you prefer.

```
s.GoToXY (int x, int y) const;
```

To output at the current cursor position, several methods can be used. First, you can use the insertion operator to output a string, a character, or an integer.

```
s << "Tile-based Basic Chasing";
s << 'F' << 42;
```

These will be outputted at the current cursor position. The setw() and doubles are not supported, however.

Second, you can output a string at a specific location, independent of the cursor position using the OutputAt() function, specifying the string and the row and column desired.

```
s.OutputAt ("Tile-based Basic Chasing", row, col);
```

Alternatively, OutputWith() allows you to also specify the color scheme to use.

```
s.OutputWith ("Tile-based Basic Chasing", row, col,
              Screen::Black, Screen::White);
```

If you have set the highlight color scheme, then you can cause any area to be shown in the highlighted scheme by using HighlightArea(). You specify the starting row and column and the number of columns to highlight.

```
s.HighlightArea (startRow, startCol, numCols);
```

Calling the ClearHighlightArea() function returns that area back to the normal color scheme.

```
s.ClearHighlightArea (startRow, startCol, numCols);
```

Several different methods can be used to input data. Now you have access to the 101+ special keys as well, such as the arrow keys. GetAnyKey() gets any keystroke, but does not display or return it.

```
s.GetAnyKey ();
```

If you wish to accept special keystrokes, such as the up arrow, use the GetSpecialKey() function. However, just because you call it, the user might press a normal key, such as the letter

‘a’ key. Hence, the function returns the letter char or a 0. If 0 is returned, then the passed by reference unsigned character contains the special key code, the up arrow for example.

```
unsigned char specialKeyCode;
char letter;
letter = s.GetSpecialKey (specialKeyCode);
```

Normal extraction can also be used to input a char or an integer. GetString() can be used to input a string, without overwriting the array of characters. All of these input and show the characters on the screen at the current cursor position.

```
char c;
int x;
s >> c >> x;
char msg[50];
s.GetString (msg, sizeof(msg), '\n');
```

Note that the third parameter is the default delimiter for the string. It is a new line (enter key) by default. You can use a ‘\”’ if the string ends with a double quote.

Finally, you can also use cin to extract data. However, if you wish to use cin, then include a first line in your main function:

```
cin.sync_with_stdio ();
```

Finally, you can flash an area on the screen briefly using the FlashArea() function. It works similar to the HighlightArea() function.

```
s.FlashArea (startRow, startCol, numCols);
```

Sample Pgm01a illustrates the use of these functions. Run it and observer the effects on the screen.

Implementing the Basic Chase Algorithm

We can use the Screen class to show an implementation of the Basic Chase Algorithm and see it in operation. Examine the first half of Pgm01b. The scenario is Frodo steps into a room that has eight orcs in it. While he stands there looking at them, they rush towards him.

The program creates an instance of the screen and gets it ready for action.

```
Screen s (Screen::Blue, Screen::BrightYellow);
s.SetTitle ("Tile-based Basic Chasing");
s.DrawBox (0, 0, 24, 79, Screen::Gray,
           Screen::BrightYellow);
```

Next, the player is defined and displayed on the screen. Note that in this sample, the player does not move at any time. Frodo remains frozen at the entrance way.

```
const int PlayerAtX = 40;
const int PlayerAtY = 2;
s.OutputUCharWith ('F', PlayerAtY, PlayerAtX, Screen::Blue,
                   Screen::BrightRed);
```

In this simulation, eight orcs are placed initially along the bottom area of the screen.

```
const int MAXORCS = 8;
const int OrcsStartX[MAXORCS] = {1, 5, 10, 20, 30,50,60,70};
const int OrcsStartY[MAXORCS] = {22, 21, 22,20,19,22,20,14};
```

The two arrays, orcAtX and orcAtY, hold the current position of orcs at each turn.

```
int orcAtX[MAXORCS];
int orcAtY[MAXORCS];
```

The initial position of all orcs is displayed with a simple loop that also installs the initial x, y location for each orc in the two arrays.

```
for (i=0; i<MAXORCS; i++) {
 s.OutputUCharWith ('O', OrcsStartY[i], OrcsStartX[i],
                    Screen::BrightYellow, Screen::BrightRed);
 orcAtX[i] = OrcsStartX[i];
 orcAtY[i] = OrcsStartY[i];
}
```

Now we are ready for the animation illustration of just how this Basic Closing Algorithm operates. The program loops through all the orcs, calculating their new positions and showing them at their new locations, over and over, until they reach Frodo's position. Different orcs will arrive at the final destination at different times, so the bool **done** controls when all of them have reached him, stopping the looping process.

```
bool done = false;
while (!done) {
 done = true;
```

```
  for (i=0; i<MAXORCS; i++) { // move each orc in turn

   // skip any orc who has arrived at the player's location
   if (orcAtX[i] == PlayerAtX && orcAtY[i] == PlayerAtY)
    continue;

   // here, at least one can still move toward player
   done = false;

   // basic closing algorithm
   if (orcAtX[i] > PlayerAtX)      // if x is > player, dec x
    orcAtX[i]--;
   else if (orcAtX[i] < PlayerAtX) // if x is < player, inc x
    orcAtX[i]++;

   if (orcAtY[i] > PlayerAtY)      // if y is > player, dec y
    orcAtY[i]--;
   else if (orcAtY[i] < PlayerAtY) // if y is < player, inc y
    orcAtY[i]++;

   // display new orc position on the screen
   s.OutputUCharWith ('O', orcAtY[i], orcAtX[i],
                      Screen::BrightYellow, Screen::BrightRed);
   Sleep (100); // pause for 100 milliseconds
  }
 }
```

I slid a new function in there, Sleep(). This function delays any further processing for the passed number of milliseconds, 100, in this case. Without this delay, the orcs “fly” toward poor Frodo.

Run the sample and observe the algorithm in operation. Here is the final screen. I pasted Frodo’s ‘F’ back onto the final image so that you can see the destination point of the eight orcs. Notice that all of them did reach the desired destination, some sooner than others.

However, examine the action once more and notice how poor the results actually are. Only for those orcs that are at a forty-five degree angle from Frodo take a path that leads straight at him. All others have a major unreal aspect to them. Those who are primarily a long Y distance from the destination point but are close with respect to the X coordinate end up quickly joining into an long vertical line, marching single file straight up to the end point. Others that have a large X coordinate disparity end up meeting along the destination row and then moving horizontally to the final location. Both of these are very undesirable movement actions, which is highly distracting to game play.

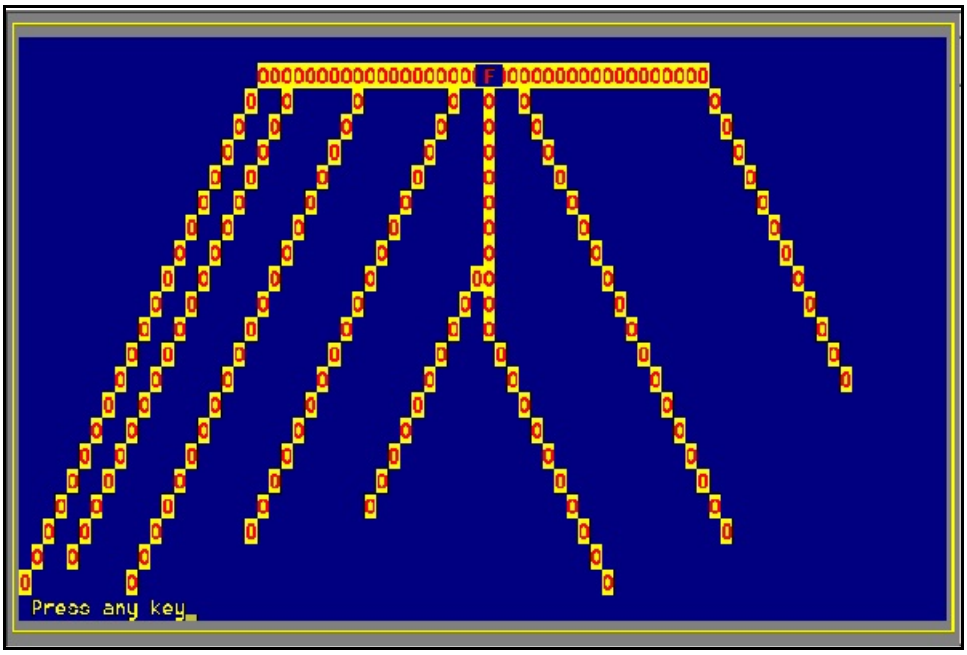

Figure 1 Basic Chasing Algorithm

The Line of Sight Algorithm

A better approach is the Line of Sight Algorithm. In a continuous movement environment, the algorithm forces the pursuer to always maintain its direction facing toward the prey. Thus, the pursuer is always traveling in a straight line toward the prey. However, if the prey is also moving, the path followed by the pursuer will not be a straight line, necessarily, but it always is moving toward the prey, wherever it moves. The results of this make for a more realistic pursuit.

The idea is to calculate based upon the pursuer's velocity how far it can travel in this unit of game time and then add that to its current position, redisplaying the pursuer at its new, hopefully closer, position. Usually, the pixel is the unit of measurement in continuous movement situation. For very smooth movement, the pursuer is moved only a pixel at a time. However, if faster action is desired, it can be moved several pixels at one time with good realism.

However, there are other considerations with which to deal. Suppose that the pursuer is a guided missile heading for a collision with a ship. What happens if the ship is faster than the missile? The missile may never actually be able to catch up with the ship! In such a scenario, interception course plotting would be a better choice, which we will discuss later on in this chapter.

When we take the line of sight approach to a tile-based game, a new constraint is placed upon the pursuer's movement. It can only move in eight possible directions: up, down, left, right, and the four diagonals into the adjacent tiles. Please note that diagonal movement seems to be faster than up-down, left-right movement. This is because the length of a diagonal across a tile is the sqrt(2) times larger, or about 1.4 times larger. An additional constraint occurs because a tile often is fairly large, showing terrain and similar features, which require a number of pixels to render. Hence, a tile size might be twenty pixels or more square. Thus, to avoid ragged, jerky movement, movement must be constrained to one tile at a time.

This problem of line of sight movement in a tile-based environment has a good solution, one that forces the pursuer to take a straight line toward the prey. That solution is called the Bresenham algorithm. In the early days of crude CGA monitors, it was used by graphics engines to draw a line between two points on the screen.

The key to the algorithm lies in determining which axis requires the most tile movement. Then, forcing movement long that axis more so than along the shorter axis. Specifically, it guarantees that the movement will never traverse two tiles in a row along the shorter axis, rather forcing multiple tiles moved along the lengthier axis before taking one along the shorter axis.

The Bresenham algorithm is given the positions of the pursuer and prey. It then calculates the best straight line of sight path from the pursuer to the prey, filling an array with each move that must be made. Obviously, if the prey should move before the pursuer reaches the prey, the entire process must be redone, based upon the new, current positions of both. Figure 1.2 shows the eight orc's line of sight path to Frodo using the Bresenham algorithm, as shown in the second half of Pgm01b.

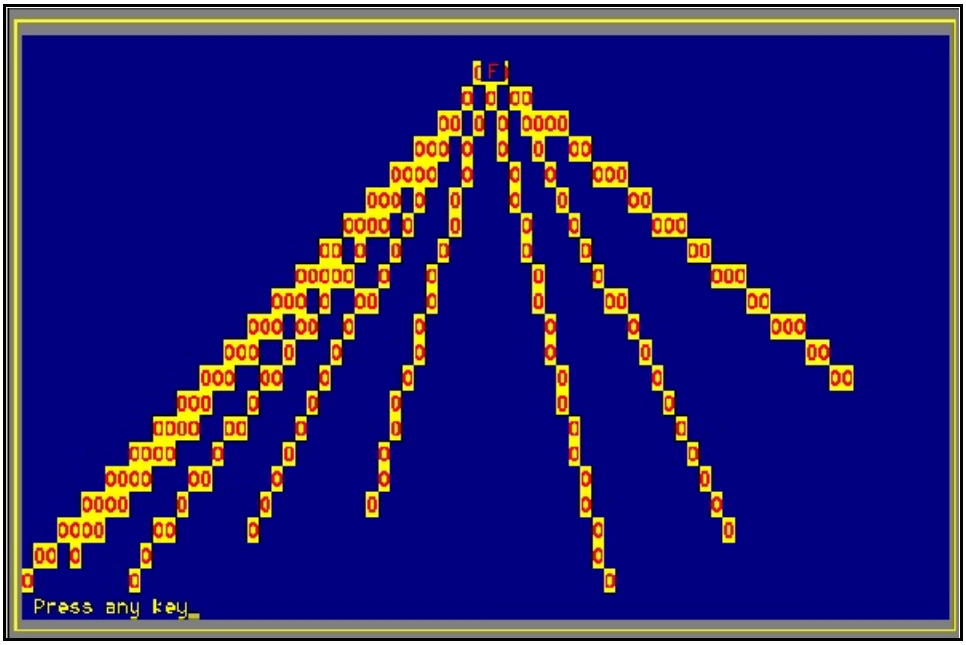

Figure 2 Bresenham Algorithm for Line of Sight Tile-based Chasing

The main program coding is relatively simple. A pair of arrays sufficiently large enough to hold all of the moves for an orc are defined.

```
const int MOVES = 200;
int orcPathX[MAXORCS][MOVES];
int orcPathY[MAXORCS][MOVES];
```

Next, the program loops through each orc in the array, calling Bresenham() to calculate its path.

```
for (i=0; i<MAXORCS; i++) {
 Bresenham (orcPathX[i], orcPathY[i], MOVES, OrcsStartX[i],
            OrcsStartY[i], PlayerAtX, PlayerAtY);
}
```

Finally, one by one, the path taken by each orc is shown on the screen.

```
for (i=0; i<MAXORCS; i++) {
 for (int j=0; j<MOVES; j++) {
  // if an orc has aleady arrived, skip that one
  if (orcPathX[i][j] == -1 || orcPathY[i][j] == -1)
   break;
  s.OutputUCharWith ('O', orcPathY[i][j], orcPathX[i][j],
                     Screen::BrightYellow, Screen::BrightRed);
  Sleep (100); // delay 100 milliseconds
 }
}
```

The actual coding for the line of sight is contained within the Bresenham() function. The starting point for the orc and player are passed to the function along with two arrays to be filled with the successive movements, assuming that the player has not moved during the entire time. It the player moves, the function must be recalled to calculate the new line of sight path.

```
void Bresenham (int orcPathX[], int orcPathY[], int max,
                int orcAtX, int orcAtY,
                int playerAtX, int playerAtY) {
 // nextX and nextY hold the orc's next position
 int nextX = orcAtX;
 int nextY = orcAtY;

 // deltaX and deltaY hold the difference between the locations
 int deltaX = playerAtX - orcAtX;
 int deltaY = playerAtY - orcAtY;

 // stepX and stepY hold how much to inc or dec each time
 int stepX = deltaX < 0 ? -1 : 1;
 int stepY = deltaY < 0 ? -1 : 1;

 // clear out the answer arrays, using invalid screen coord: -1
 for (int i=0; i<max; i++) {
  orcPathX[i] = -1;
```

```
  orcPathY[i] = -1;
 }
```

Notice that –1 is not allowed for a row-col location on the screen. Hence, a –1 can indicate an empty, unused element in the array.

```
 // install the starting location in the answer arrays
 orcPathX[0] = orcAtX;
 orcPathY[0] = orcAtY;

 // double the total difference between locations
 // and take the absolute value to remove negative signs
 deltaX = abs (deltaX * 2);
 deltaY = abs (deltaY * 2);

 // set the current step to 1, [0] holds the original location
 int currentStep = 1;
 int fraction;

 // two cases: is the x length greater than the y length?
 if (deltaX > deltaY) {
  // x > y, so constantly move x but control when to move y
  // fraction controls times to move in y
  fraction = deltaY * 2 - deltaX;


  // now fill in all steps the orc must take to reach player
  while (nextX != playerAtX && currentStep < max) {
   if (fraction >= 0) { // fraction is still > 0, must move in y
    nextY += stepY;     // add another y step to total y move
    fraction -= deltaX; // remove one column amount
   }
   // here, we have moved enough in y to justify moving in x
   nextX += stepX;
   // now add in another y move for next iteration
   fraction += deltaY;
   // store this move in the answer array
   orcPathX[currentStep] = nextX;
   orcPathY[currentStep++] = nextY;
  }
 }

 // here, y is greater than the x length
 else {
  // fraction controls times to move in x for each y move
  fraction = deltaX * 2 - deltaY;
  // fill in all steps the orc takes to get to the player
  while (nextY != playerAtY && currentStep < max) {
   if (fraction >= 0) { // fraction > 0, so must move in x
```

```
    nextX += stepX;     // add another x to total move
    fraction -= deltaY; // remove one row amount
   }
   // here we have moved enough in x to justify moving in y
   nextY += stepY;
   // add in the x move to fraction for next iteration
   fraction += deltaX;
   // store this move in the answer array
   orcPathX[currentStep] = nextX;
   orcPathY[currentStep++] = nextY;
  }
 }
}
```

The Bresenham algorithm is the standard for plotting a line of sight path in a tile-based game. What about implementing line of sight in a continuous environment?

The Complexity of Line of Sight in a Continuous Environment

Let's examine a two-dimensional typical situation. Suppose that we have a pair of space ships or bi-planes, one the attacker and the other the defender, where the attacker is pursuing the defender, hoping to engage it in combat. Now we are dealing with vectors that represent the current motion of each ship.

Remember that motion has two components, speed and direction. That is the definition of a vector quantity, a magnitude and a direction. When one wishes to deal with closing in a line of sight method in a continuous environment, he or she must deal with vectors and various coordinate systems. Let's see why.

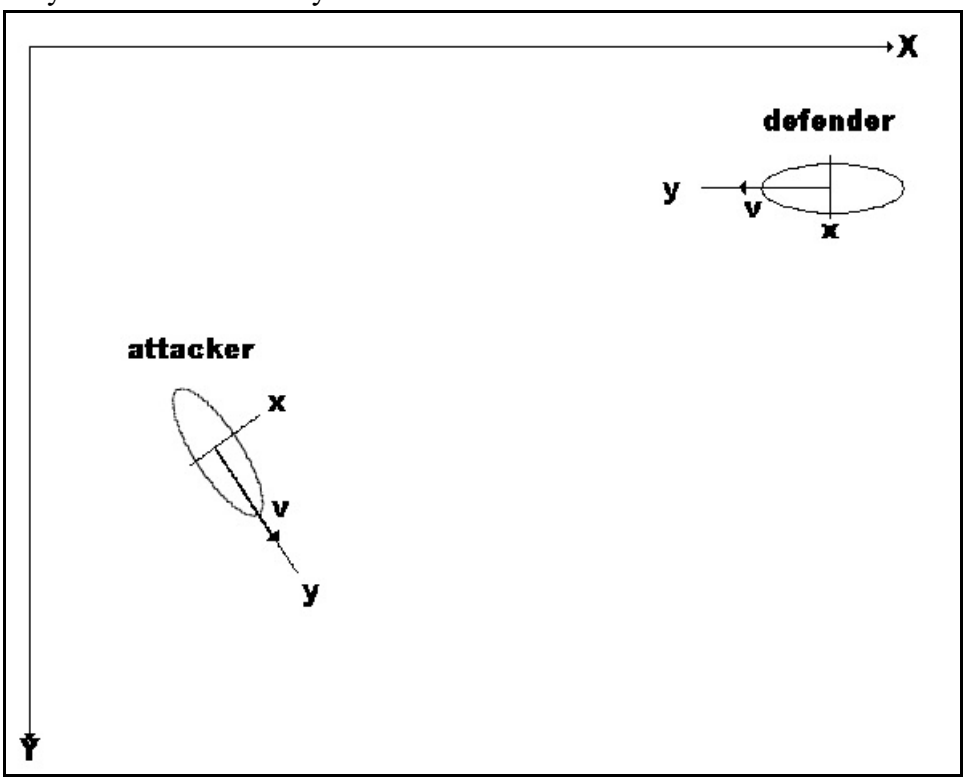

Figure 3 Attacker-defender Initial Situation

In Figure 3, the world X-Y coordinate system is shown along with the local x-y coordinate systems of both the attacker and defender. The attacker is currently traveling in one direction at some speed, v, while the defender is traveling in another direction at some other speed, v. That is, imagine you are the driver of the attacker vehicle. From your point of view, your local positive y-axis is straight ahead of you and your speed is in the forward direction you are facing. Now, having spotted the defender, you need to turn your craft so that your craft is facing the defender's craft and your speed is now directed toward the defender's craft. Using the

available mechanical means, such as bow thrusters, steering wheel if this is a car, or whatever means is applicable, you must turn your vehicle to face the defender.

Immediately a number of physical problems arise. First, we are dealing with a two-dimensional solid body or rigid body situation. The actual turning is done around the center of gravity of the object, if this is a flying machine. Second, turning is not an instantaneous action unless the vehicle is traveling very slow indeed. Imagine driving your car and you desire to turn left. If you are traveling at one mile per hour, you can turn a very tight turn almost at once. However, if you are traveling at sixty miles per hour, the turn takes place over a considerable distance, compared to the slow speed turn, or the car overturns. The vehicle's momentum must be considered when making a turn. Third, the actual turning is accomplished by providing a varying acceleration to the vehicle so that it ends up facing the desired direction. In the case of a flying ship, thrusters would provide the steering accelerations along the positive and negative local x-axis. However, the amount of change applied varies over time. When entering a turn, you begin with a large motion of the steering wheel and then gradually lessen is as you come out of the turn. Fourth, each vehicle has a maximum speed of which it is capable of traveling.

In short, a lot of additional complexity arises very quickly. A study of physics is needed to totally describe the problem. During the semester, we will be examining all of the component parts of this problem in detail. By the end of our study, you should be able to deal with the complete situation posed by games such as these. However, at this point, let us only concentrate on how to deal with the line of sight chasing, leaving the remainder for later chapters.

So what do we know about the problem of line of sight? We know the two vectors of motion and the positions within the global game coordinates, the upper case X and Y axes in Figure 3 above. What do we need to know so that we can steer toward the defender? We need to know whether to turn our ship left or right, ignoring the unlikely possibility that we are already heading straight for it. If we subtract the two position vectors which locate the ships in global game coordinates, we can find its relative position to us in global coordinates. However, our viewpoint is not global coordinates, our thrusters or turning mechanism are related to our own local x-y axes, that is, our local x axis is perpendicular to our forward direction. Do we fire the port or starboard turning mechanisms? Do we turn the steering wheel to the left or right? Thus, we must convert the global coordinates of the resulting relative position back into our own local coordinates.

Finally, if we then normalize that resultant vector such that its magnitude is unity, we have the vector pointing directly from us to the defender. We do not need the y component, only the x. If the x component is 0, we are heading straight to the attacker. If the x component is positive, we must turn to our right or starboard. If the x component is negative, we must turn to port or left. Simple enough, if we review how to subtract vectors and how to convert from local to global coordinates.

Vectors

Figure 4 shows how we can store a vector in three-dimensional space. By using three-dimensional space, we can easily handle both two and three dimensional problems in vectors. If you are using only two dimensions, simply let the z-axis component be zero.

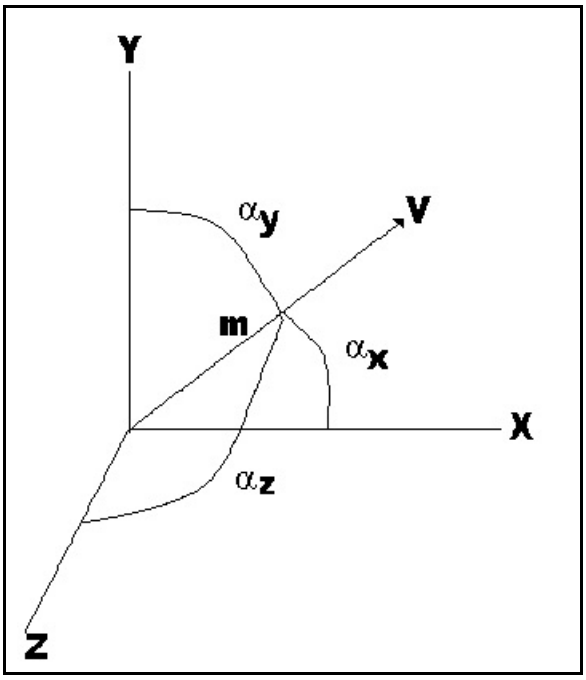

Figure 4 A Three-dimensional Vector

The vector **V** has its magnitude (total length from the origin point) and its direction stored as three components, its x, y, and z axis components. The magnitude or length of the vector is notated |**V**|. The projection of **V** onto each of the axes is given by

Vx = |**V**| cos (αx)
Vy = |**V**| cos (αy)
Vz = |**V**| cos (αz)

where the magnitude is given by

$$|\mathbf{V}| = \sqrt{x^2 + y^2 + z^2}$$

Along with the Screen class, I have also provided in Pgm01a a simple Vector class to aid our programming samples. The Vector class was discussed in depth in chapter 0. In the problem at hand, determining line of sight so that course corrections can be made, the resultant vector of the difference between the pursuer and prey needs to be normalized. That is, it needs to be scaled so that the magnitude is 1. Then, the x component can be used by the pursuer to make course corrections so that it continues to head towards the prey.

Conversion from Global Coordinates to Local Coordinates

From the viewpoint of the driver of the attacker ship, the positive y axis is directly in front of the ship. The local origin point is the center of gravity of the object, which will be covered later on in 2d solid body theory. For now, think of it as being the "center" of the attacking ship. The local x axis extends to the attacker's left and right. Think of the distinction between local and global coordinates as our ancestors once did here on earth. When the earth was considered the center of the universe, everything was measured from the earth as the origin point, local coordinates. It is, after all, how we view things. Yet the earth goes around the sun, and the sun goes around the galaxy. Thus, to study the large scale movements of earth, a more encompassing coordinate system is used, the global coordinates.

In games, we store the location of objects in the global coordinate system, such as an object's position, as a vector in two or three dimensional space. Yet, when it comes to operations, such as orienting or firing missiles at a target, local coordinates are used because it is more convenient and relates to the actual viewpoint of the craft. Figure 5 shows the geometry of the conversion situation.

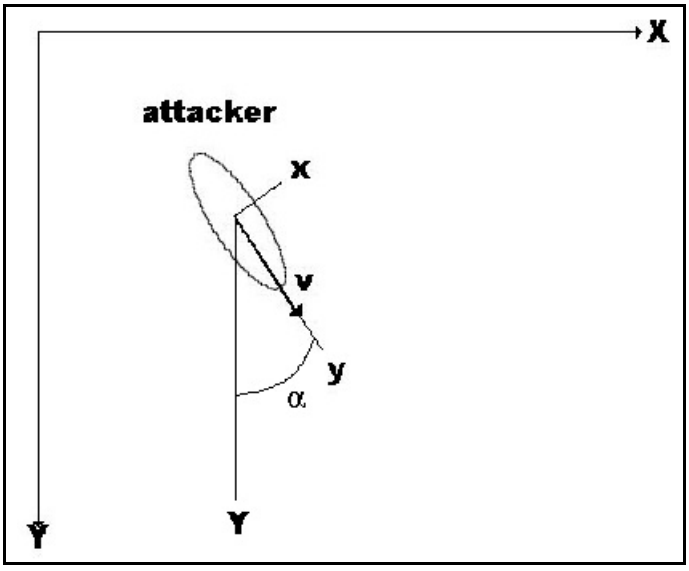

Figure 5 Local Coordinates from Global Coordinates

To project point X,Y of the attacker onto the local x,y system of the attacker, we must also know the angle alpha, or the orientation of the craft with respect to the Y global axis. From geometry, the formula to convert from global to local coordinates is given by the following.

$x = X \cos \alpha + Y \sin \alpha$

$y = -X \sin \alpha + Y \cos \alpha$

Of course, the angle in degrees must be converted to radians for the trig functions. The Vector class implements this conversion in the GlobalToLocalCooords() function. It was coded this way.

```
Vector GlobalToLocalCoords_2D (double angle, const Vector& u) {
```

```
    double    x, y;
    x = u.x * cos (DegreesToRadians (angle)) +
        u.y * sin (DegreesToRadians (angle));
    y = -u.x * sin (DegreesToRadians (angle)) +
        u.y * cos (DegreesToRadians (angle));
    return Vector (x, y, 0);
}
```

However, a complexity arises immediately concerning that angle alpha, in the figure above. Normal trig functions assume the positive Y is up; here it is **down**. If we wish to show the ship at a 45 degree angle, using the above coding, the ship will be pointing upwards and to the north west (to the left) not down and to the southeast as expected. To get the above function to work properly, when we want an angle of 45 degrees to the left of down, we would enter -135 degrees. This is just too confusing. Looking at the screen, we don't think of Y as being inverted, positive Y being downward. We naturally think of up as the positive direction. Thus, we would like to say that the angle is 135 degrees from the up position and to the right, to yield the ship oriented to the southeast. Let's call this the **orientation** angle. An orientation angle of 0 degrees points straight up. 180 degrees points straight down. 90 degrees points due east or to the right. If we wish to pass the orientation angle to the function, then in the function, let's simply negate the angle.

```
Vector GlobalToLocalCoords_2D (double angle, const Vector& u) {
    double    x, y;
    x = u.x * cos (DegreesToRadians (-angle)) +
        u.y * sin (DegreesToRadians (-angle));
    y = -u.x * sin (DegreesToRadians (-angle)) +
        u.y * cos (DegreesToRadians (-angle));
    return Vector (x, y, 0);
}
```

Again, this Vector class is part of the Pgm01a sample program as well as many other samples covered later on in the text.

Line of Sight in a Continuous Environment

Armed with vectors and coordinate conversion, we can now tackle the problem of line of sight closing upon the defender in a continuous environment. The objective: keep the defender right in front of the attacker as the attacker moves toward the defender. All that is known are the two vector locations (the attacker and defender) and the defender's orientation angle from the Y global axis. Further, assume that the attacker has some means of steering its vessel or ship by means of a function called TurnShip().

First, calculate the relative position vector between the attacker and the defender, which also gives the line of sight from the attacker to the defender. This is done by subtracting the defender's position vector from the attacker's position vector. However, this resultant vector is in global coordinates. In order to control the attacker's steering, it must be converted into local coordinates as viewed from the driver's seat, so that the craft can know whether to turn right or left or not at all. After the vector is converted to local coordinates, examine the x value. If its x value is positive, turn left or port. If negative, turn right or to the starboard. If zero, no turning is required.

The coding of the function is then very simple, assuming that we know how the attacker and defender are defined. They could be structures or object oriented classes. Let's keep it simple for now and assume that both the attacker and defender are instances of a structure called SHIP, whose partial definition is as follows.

```
struct SHIP {
 Vector position;
 double orientation;
 . . .
};
```

Further, assume that we also have defined an enum to indicate the type of turning required to keep the attacker moving toward the defender.

```
enum TurnType {None, Left, Right};
```

Our line of sight function then is passed two instances of the structure.

```
void LineOfSightChase (struct SHIP& attacker,
                       const struct SHIP& defender) {
 Vector lineOfSight = defender.position - attacker.position;
 Vector local = GlobalToLocalCoords_2D (-attacker.orientation,
                                        lineOfSight);
 local.Normalize ();
 TurnType turn = None;
 if (local.x < EPS)
  turn = Left;
 else if (local.x > EPS)
  turn = Right;
 if (turn != None)
```

```
  Steer (attacker, turn);
}
```

Again, remember that calculations involving floating point are subject to small roundoff errors. Hence, do not check against 0., but use the error precision value of EPS, which is 0.000000000000001.

What will be the overall appearance of our algorithm? If the defender does not move, the attacker will follow a straight line to the defender. If the defender is moving and varying its position, the defender's path will be curved, continuously adjusting its orientation as it attempts to close the distance. The algorithm is relentless. Again, it is likely that eventually the attacker will end up behind the defender, following it and closing the distance, hopefully.

Ah, but this raises an immediate question. What happens if the velocity of the attacker is much larger than the defender's? It is very likely that the attacker will overshoot the defender's position and be forced to circle around and come at it again. This is not very realistic. If the game simulation has a way for the attacker to slow down, then you can add in another check. If the distance between the attacker and defender is less than a predetermined amount, begin to slow down or if the distance is too great, increase the defender's speed. To find the distance between the two simply subtract their position vectors and take the magnitude of that result.

Finally, if evading is required, then reverse the turn variable settings, go left when closing suggests going right.

Drawbacks to this algorithm become apparent. Having the attacker always heading directly toward the defender is not necessarily either the shortest path in distance or time. Usually, the attacker ends up behind the defender, closing in on it. If the attacker is much faster than the defender, the attacker often overshoots and has to circle around and come back. If the attacker is much slower than the defender, it may never actually catch the attacker. This last is highly unreasonable. Often by taking a different course, a slower vehicle can intercept a faster one. Similarly, some guided missiles have a limited range. Plotting an interception can yield a hit where as simply following it and closing may miss because too great a distance traveled is required.

Sample program Pgm01d is a Windows application that illustrates these principles. Essentially, Snoopy is flying the yellow biplane and is being chased by the Red Baron. The Action menu item Basic Chase, implements the coding we have just seen. Figure 6 shows the action in progress.

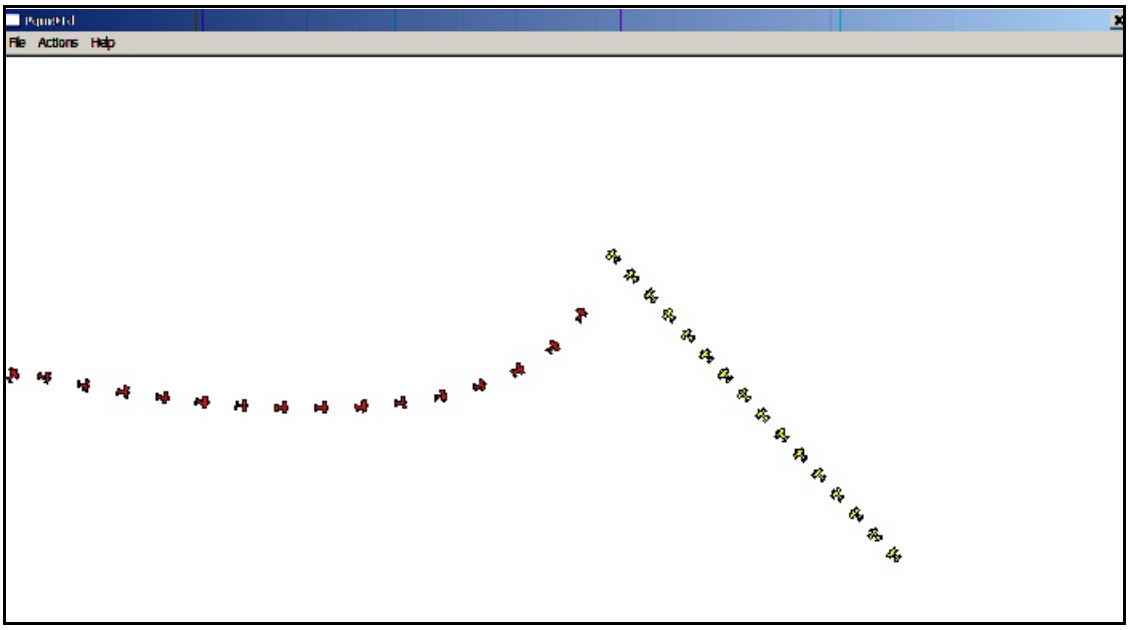

Figure 6 Chasing in a Continuous Environment

Run the demo and watch what happens when the red plane gets behind or ahead of the yellow one. Realistically, one needs to determine if this is the case and then revert back to a Basic Closing method for a short time doing a small circle or such before resuming the Line of Sight method. Doing so yields a better angle of approach to the target.

Intercepting

Intercepting an opponent requires knowing one additional property of the opponent, its velocity vector. The idea is to predict where the defender will be at some future time and then move the attacker to that position so as to arrive when the defender arrives there. Please note that the interception point is not the shortest distance to the attacker, because that is ignoring the relative speeds of the two ships. If the attacker has a shorter distance to travel to that location and moves much faster than the defender, the attacker will arrive at that location way ahead of the defender and must sit there waiting for the defender to arrive, very unrealistic.

To intercept, one calculates based upon the attacker's and defender's position and speed where they will intersect, and steer the defender to that location. Obviously, that location will have to be continuously updated as the attacker alters course or changes speed. The basic equation involved is

distance = velocity * time

or

time = distance to travel / velocity

We can calculate the closing velocity by subtracting the two velocities. By subtracting the two

distance vectors, we can get the distance to close.

$\mathbf{V}_{closing} = \mathbf{V}_{defender} - \mathbf{V}_{attacker}$

$\mathbf{D}_{closing} = \mathbf{D}_{defender} - \mathbf{D}_{attacker}$

Thus the time to close is

$T_{closing} = |\mathbf{D}_{closing}| / |\mathbf{V}_{closing}|$

that is the magnitude of the closing distance divided by the magnitude of the closing velocity.

Finally, knowing the time to close, you can then calculate where the defender will be at that time in order to steer towards that location.

$\mathbf{D}_{predicted} = \mathbf{D}_{defender} + \mathbf{V}_{defender} * T_{closing}$

The changes from LineOfSightChase to do an interception are highlighted in bold face.

```
void LineOfSightIntercept (struct SHIP& attacker,
                           const struct SHIP& defender) {
 Vector vClosing = defender.velocity - attacker.velocity;
 Vector dClosing = defender.position - attacker.position;
 double tClosing = dClosing.Magnitude() / vClosing.Magnitude();
 Vector predicted = defender.position +
                    defender.velocity * tClosing;
 Vector lineOfSight = predicted - attacker.position;
 Vector local = GlobalToLocalCoords_2D (-attacker.orientation,
                                        lineOfSight);
 local.Normalize ();
 TurnType turn = None;
 if (local.x < EPS)
  turn = Left;
 else if (local.x > EPS)
  turn = Right;
 if (turn != None)
  Steer (attacker, turn);
}
```

Rerun Pgm01d and chose the Action menu item Intercept. Figure 7 shows the interception point as a small circle, which is continually updated as needed. Also watch what happens when the attacker gets in front or behind the target.

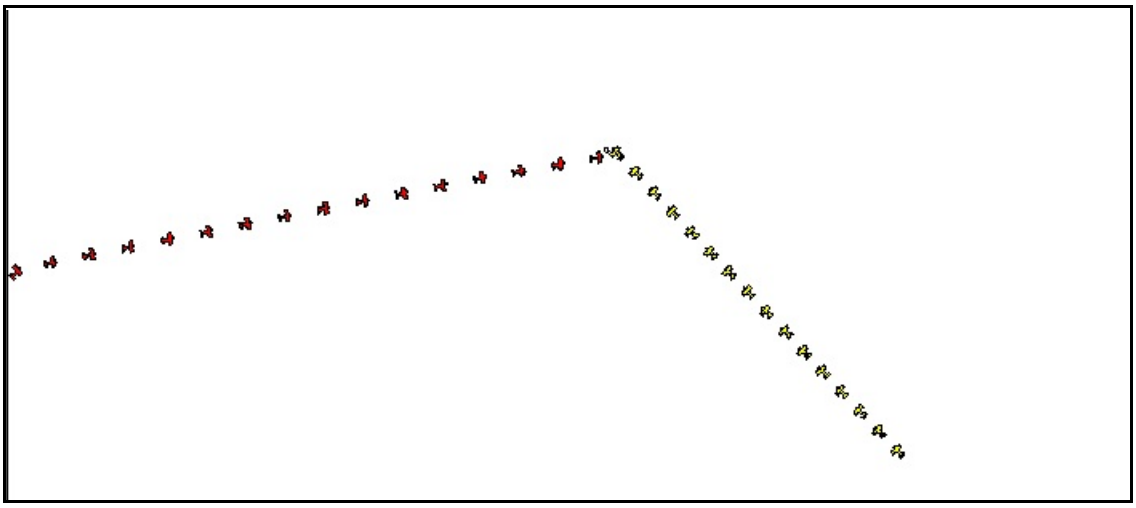

Figure 7 Interception

What are the drawbacks to this algorithm? Sometimes interceptions cannot be made. For example a slow attacker which ends up behind the defender might not be able to catch up. Another way it can fail is that the attacker gets way ahead of the defender, predicts a closing point that is so far ahead of both of them that neither reach it. You can determine whether the attacker is ahead of or behind the defender by looking at the normalized y resultant local variable. If y is negative, the defender is behind the attacker and a good move might be to circle around to get a better shot; to do this, use the pure LineOfSightChase and then revert back to the intercept version. If y is positive, the defender is ahead of the attacker.

Patterned Movement

Closely related is the patterned movement situation. The idea is to add realism by giving the computer controlled objects the illusion that they are doing something, such as patrolling the perimeter of their camp, performing barrel rolls while attacking, washing clothes, working in fields, and so on. The patterned movement does not have to be a rectangular path, but can be any path allowed in the game world, as complex as needed.

Patterned movement is implemented by constructing an array of control data that define the pattern of movement to be taken by the computer controlled object. For example, an NPC on guard duty might march in a rectangular pattern around the encampment. Workers in a field might move up and down long rows of crops. Enemy biplanes might perform a signature barrel roll as they sweep down to attack you.

Obviously, the precise form that the control data takes varies between a continuous environment and a tile-based one. In the continuous environment, the structure could contain members that specified an amount to turn left, turn right, speed up or slow down, a move to this

location, fire weapons, drop depth charge here, cast a certain spell, and so on. In a tile-based game, locations are specified by providing row and column locations of the tiles to move to instead of angles of direction and positions in two or three dimensional space, x, y, z coordinates.

Suppose that the orc camp is located within the rectangle defined by (10, 10) and (15, 15) (x, y) values. An orc guard paces the perimeter of the camp. Thus, it begins at say (10, 10) and moves from there to (10, 15). From there, it moves to (15, 15); next, it goes to (15, 10); and finally it moves back to the starting point, (10, 10). This is the first step in creating a patterned movement, defining the locations of each segment of movement, forming the pattern. To actually compute the individual tiles to which to move, we ought to use Bresenham's algorithm to yield the best looking path between each of the points.

Thus, at first glance, it would seem that we only need to enter the four sets of data in the form of x, y from and x, y to: (10,10, 10, 15), (10, 15, 15, 15), (15, 15, 15, 10), (15, 10, 10, 10). However, and this is a big however, if we kept our data this specific, then if the orc camp moved to another location with the same size, our patterned movement would be invalid! Assuming that the camp size and shape remained the same no matter where it relocated, then we desire to enter the path in a normalized format such that it could be executed from any orc camp location!

To normalize a specific path, take the first coordinate pair, here 10, 10, and subtract those values from each x, y pair in the entire pattern, yielding a normalized path that looks like this: (0, 0, 0, 5), (0, 5, 5, 5), (5, 5, 5, 0), (5, 0, 0, 0). When the program is about to display the patterned movement for a specific orc camp, specify the upper left corner x, y indices and add them to each pair of values in the normalized pattern. For example, if the orc camp was now located at 6, 5 (the upper left corner of the camp), the display patterned movement of the orc guard would first add those values to the normalized path values, yielding: (6, 5, 6, 10), (6, 10, 11, 10), (11, 10, 11, 5), (11, 5, 6, 5).

One caution on normalization. By subtracting whatever is in the very first (x, y) pair in the patterned movement from all other coordinates, a potential problem arises when a subsequent point is above and left of the original one. Consider this path: (1, 1) to (0, 0). Normalization yields (0, 0, -1, -1)! The pair of -1's indicate the end of the path, not another point in the sequence. Thus, it is wisest to always begin tracing the patterned movement from the upper left corner of the sequence.

Programming-wise, we have several options. The original patterned path could be hard-coded into the program, necessitating a normalization function, and then a specific pattern created for the current camp. A normalized pattern could be built originally and hard-coded into the program, and then the specific pattern for the current camp created. Another approach is to store the patterned movement data in a file and load it into arrays when needed. The benefit of storing the patterned movements in a file is that they can be created independent of the program itself. In fact, one could create a file that stored all of the possible patterned movements for the entire game. One could store the data in binary format for fast input.

Let's use the file method, storing our patterned movement in a simple text file. In reality, one could create a helper program that would allow the designer to enter a pattern and then the program would normalize it before storing it into the file. In this chapter, let's assume that the inputted patterned movement has been normalized. That is, the first x, y pair are both zero. Then, the game engine only needs to input the pattern and create a specific instance of it based upon where the patterned movement should take place.

Here is the oasis.txt file I used for the orc guard patrolling the perimeter of the orc encampment.

```
0 0 6 0
6 0 7 3
7 3 4 6
4 6 0 6
0 6 0 0
```

Pgm01c illustrates how patterned movement is implemented. Figure 8 shows the final screen shot outlining the path taken by the guard. Notice that the path can be anything desired, it does not have to be a boring rectangle or square. The path can even be back and forth along a single line.

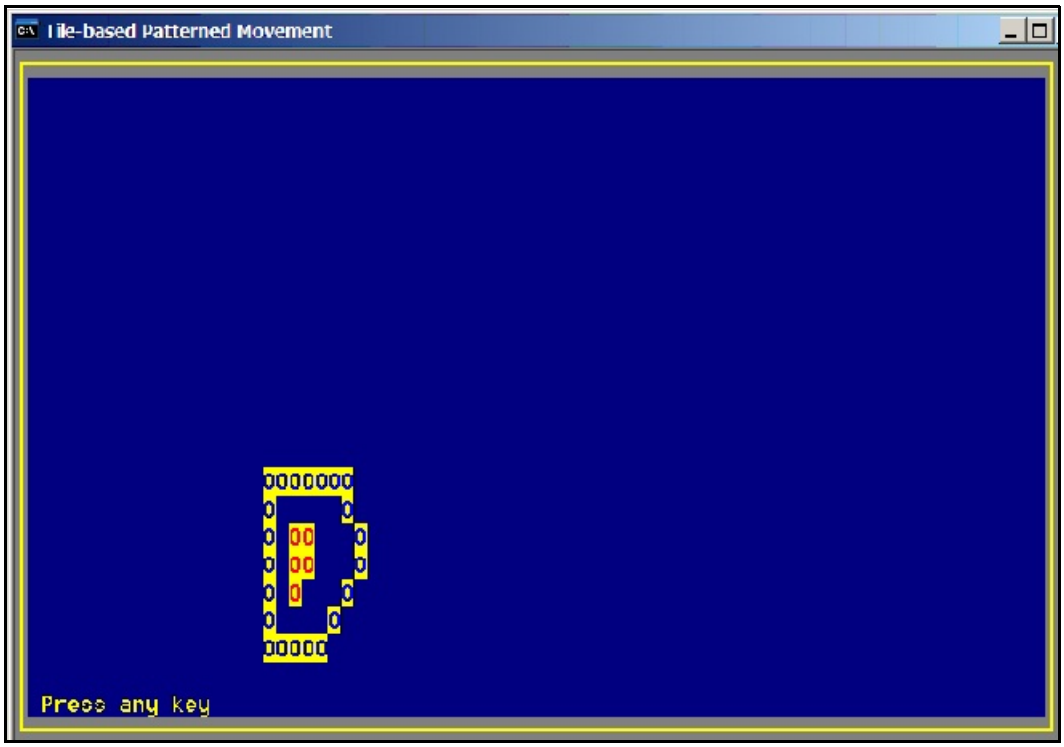

Figure 8 Orc Guard's Patterned Movement (Tile-based)

Implementation consists of three actions. First, the file containing the normalized pattern must be loaded and each line segment must be created using the Bresenham method for creating an optimum path. Once the normalized path has been created, it must be made specific for the current orc guard. Second, a modified Bresenham function is used to make the optimum path for each line segment. Third, the final resultant path must be displayed.

The main function is very straightforward. I define a series of orcs, the first one will be the guard and its starting coordinates will become the initial location for the patterned movement. An array to hold the patterned movement path, pathX and pathY, is defined and passed to the function LoadPatternMovement. Maximum flexibility is retained by passing the load function the name of the file, the answer arrays, and the specific initial location for the start of the patterned movement.

```
// define the orcs and set their initial coordinates
// the orc guard is index 0
const int MAXORCS = 6;
const int OrcsStartX[MAXORCS] = {19, 21, 22, 21, 22, 21};
const int OrcsStartY[MAXORCS] = {15, 17, 17, 18, 18, 19};

const int MAXPATH = 200; // max number of pattern moves
int pathX[MAXPATH];      // pattern movement in X
int pathY[MAXPATH];      // pattern movement in Y

// load the patterned guard's path ready for display
int maxSteps = LoadPatternMovement ("oasis.txt", pathX, pathY,
                          MAXPATH, OrcsStartX[0], OrcsStartY[0]);
int i;
// display all orcs at their starting locations
for (i=0; i<MAXORCS; i++) {
 s.OutputUCharWith ('O', OrcsStartY[i], OrcsStartX[i],
                    Screen::BrightYellow, Screen::BrightRed);
}
```

Showing the patterned movement of the orc guard is merely a matter of displaying the orc at each step in the array.

```
for (i=0; i<maxSteps; i++) {
 s.OutputUCharWith ('O', pathY[i], pathX[i],
                     Screen::BrightYellow, Screen::BrightBlue);
 Sleep (100); // delay 100 milliseconds
}
```

The loading function opens the file. If it is successful, the answer arrays are initialized to a -1, indicating this element is not used, and the number of steps taken in the entire patterned movement, numSteps, is set to 0.

```
int LoadPatternMovement (const char* filename,
                         int pathX[], int pathY[], int maxPath,
                         int startX, int startY) {
```

```
 ifstream infile (filename);
 if (!infile) {
  cerr << "Error: cannot open patterned movement file: "
       << filename << endl;
  exit (1);
 }
 int i;
 // initialize whole path array to unused (-1)
 for (i=0; i<maxPath; i++)
  pathX[i] = pathY[i] = -1;

 int numSteps = 0;
 int xstart, ystart, xend, yend;
```

Next, the main primed loop begins, obtaining the first pair of start and end x y coordinates. If the four values are inputted successfully, the BuildPathSegment() function is called to append this line segment of the path onto the total path, using Bresenham's algorithm to form the line. The process is repeated until the end of file is reached.

```
 infile >> xstart >> ystart >> xend >> yend;
 while (infile && numSteps < maxPath) {
  // add this segment to the total path
  BuildPathSegment (pathX, pathY, maxPath, xstart, ystart, xend,
                    yend, numSteps);
  // get next pair of start and end x y coordinates
  infile >> xstart >> ystart >> xend >> yend;
 }
 if (!infile.eof ()) {
  cerr << "Error: bad data in patterned movement file: "
       << filename << endl;
  infile.close ();
  exit (1);
 }
 infile.close ();
```

Finally, the normalized path must be made specific to a given actual starting location of the orc. The initial location for the orc is added to each x, y pair in the entire path to be followed. The total number of steps in the complete path is returned to the caller.

```
 for (i=0; i<numSteps; i++) {
  pathX[i] += startX;
  pathY[i] += startY;
 }
 return numSteps;
}
```

BuildPathSegment() appends a new straight line path onto the existing path array. While the basic algorithm is not altered, provision must be made for the appending action. This means that the current starting index must be passed from the caller and updated by this function as it

adds on more steps. Hence, the currentStep is now passed by reference into the function.

```
void BuildPathSegment (int pathX[], int pathY[], int max,
                int startAtX, int startAtY,
                int endAtX, int endAtY, int& currentStep) {
 // nextX and nextY hold the orc's next position
 int nextX = startAtX;
 int nextY = startAtY;

 // deltaX and deltaY hold the difference between the locations
 int deltaX = endAtX - startAtX;
 int deltaY = endAtY - startAtY;

 // stepX and stepY hold how much to inc or dec each time
 int stepX = deltaX < 0 ? -1 : 1;
 int stepY = deltaY < 0 ? -1 : 1;
```

The original coding to initialize the resultant path array to -1's has been removed and replaced by installing the initial location, only if there is no points yet in the final path, currentStep being 0.

```
 if (currentStep == 0) {
  pathX[currentStep] = startAtX;
  pathY[currentStep++] = startAtY;
 }
```

The remainder of the Bresenham solution is exactly the same as before.

```
 // double the total difference between locations
 // and take the absolute value to remove negative signs
 deltaX = abs (deltaX * 2);
 deltaY = abs (deltaY * 2);

 int fraction;

 // two cases: is the x length greater than the y length?
 if (deltaX > deltaY) {
  // x > y, so constantly move x but control when to move y
  // fraction controls times to move in y
  fraction = deltaY * 2 - deltaX;

  // now fill in all steps the orc must take to reach player
  while (nextX != endAtX && currentStep < max) {
   if (fraction >= 0) { // fraction is still > 0, must move in y
    nextY += stepY;     // add another y step to total y move
    fraction -= deltaX; // remove one column amount
   }
   // here, we have moved enough in y to justify moving in x
   nextX += stepX;
   // now add in another y move for next iteration
```

```
   fraction += deltaY;
   // store this move in the answer array
   pathX[currentStep] = nextX;
   pathY[currentStep++] = nextY;
  }
 }

 // here, y is greater than the x length
 else {
  // fraction controls times to move in x for each y move
  fraction = deltaX * 2 - deltaY;
  // fill in all steps the orc takes to get to the player
  while (nextY != endAtY && currentStep < max) {
   if (fraction >= 0) { // fraction > 0, so must move in x
    nextX += stepX;     // add another x to total move
    fraction -= deltaY; // remove one row amount
   }
   // here we have moved enough in x to justify moving in y
   nextY += stepY;
   // add in the x move to fraction for next iteration
   fraction += deltaX;
   // store this move in the answer array
   pathX[currentStep] = nextX;
   pathY[currentStep++] = nextY;
  }
 }
}
```

Patterned Movement in a Continuous Environment

When in a continuous environment, totally new problems face game designers. They all stem from the fact that the simulation is based upon the physics of motion of the objects. Suppose that we are implementing an air combat game. Each plane has certain physical properties which impact the way the plane operates. Additionally, we are simulating the flying of airplanes. Thus, one cannot utilize the patterned movements of the tile based games as is. For example, you cannot have an airplane making instantaneous right turns. Doing so violates the physics of the flying plane, ruining the simulation for everyone. The computer cannot be allowed to break all the operational rules of physics while the player must follow them.

Nevertheless patterned movements can be implemented in such a way that they work with the physics of the simulation. The control structures now contain relative requests for an action, which the physics engine then carries out over as many turns as necessary to accomplish the requested action. With an airplane, the following might represent a control structure.

```
struct PATTERNMOVE {
     double altitudeChange;
     double turnToHeading; // in degrees + => to the right
     double alterSpeedTo;
     double travelThisDistance;
};
```

In other words, the control structure requests an altitude change, leaving the physics engine to accomplish that task in as many turns as is required to achieve the result. The main thing to keep in mind is not to use absolute values, but relative ones. Climb another hundred feet, not move to this x, y position. That way, the pattern can be utilized whenever it is needed. Patterns might describe a barrel roll, for example. If you keep the movements relative, then the barrel roll can be executed by any plane anywhere it is physically possible to do so.

In the implementation phase, since any pattern request may take several turns to accomplish, another control structure must be added to aid in carrying out the request. This structure would contain the initial settings and the accumulated changes thus far. Each turn, the implementation loop of the pattern can check to see if it has finally accomplished the request. If not, continue applying the changes. If so, end that pattern and do what is next in line.

Problems

Problem 1-1 Interception in a Tile-based Game

The text discussed implementation of interceptions in a continuous environment. They can also be done in a tile-based game. Using the sample programs provided, allow Frodo to move in some direction at some rate of speed. Have one orc at some other position move to intercept him.

When the program begins and before you create your Screen instance, prompt the user for the initial setup values. Allow the user to specify the initial position of Frodo and his direction of travel and likewise for the orc. Run the simulation until one of the two runs into the edge of the screen or they meet. The edge of the screen is row becoming 0 or at the bottom or column becomes 0 or 79. They meet when they are in the same location, row and column indexes are identical.

Use the Vector class as desired.

Problem 1-2 Guards on Patrol

Draw the outline of a castle's outer wall which occupies a rectangular area approximately half the height and width of the screen, that is 40x13, for example. Use Grey on Black for the color scheme to display the wall.

Next, create four guards who are to patrol the walls by marching designated paths along each side of the walls on the outside of the walls. Create a text file with the four patterned movements of the guards. Assume that the guards move one tile per turn. Show the guards position, sleep for 100 milliseconds and then move the guards to their next location, repeating the sequence 100 times before stopping. Invent a good symbol and color scheme for the guards.