Bonus Chapter 24

Game Programming Potpourri

In This Chapter

- Choosing data structures
- Writing a good algorithm
- Understanding optimization theory
- Creating a demo
- Writing a save-game feature
- Implementing multiple players

In this chapter, I cover all those little details that slip through the cracks in any game programming book. I discuss everything from writing games so they can be saved, to making demos, to optimization theory! I think that this chapter will answer any further questions that you may have. If it doesn't, e-mail me at necron@slip.net, and I'll answer it!

Data Structures

Probably one of the most frequent questions I'm asked is this: What kind of data structures should be used in a game? My answer: Use the fastest, most efficient data structure possible for the task at hand. Note, however, that in most cases, the task at hand doesn't require the most advanced, complex data structures that computer science has to offer. Rather, try to keep things simple. When it comes to games in Windows, speed is more important than memory these days. So sacrifice memory before you sacrifice speed!

In the following sections, I cover some of the most common data structures used in games and give you some insight into when to use them.

Static structures and arrays

The most basic of all data structures is, of course, a single occurrence of a data item such as a single structure or class. For example:

```
typedef struct PLAYER_TYP // tag for forward references
        {
        int state; // state of player
        int x,y;   // position of player
        // more fields here...
        } PLAYER, *PLAYER_PTR;
```

In C++, you don't need to use `typedef` on structure definitions to create a type, as in C; a type is automatically created for you when you use the keyword `struct`.

```
PLAYER player_1, player_2; // create a couple of players
```

In this case, a single data structure along with two statically defined records does the job. On the other hand, if the game calls for three or more players, using an array like this is probably a good idea, because you can process all the players with a simple loop:

```
PLAYER players[20]; // the players of the game
```

Okay, great, but what if you don't know the number of players or records until the game runs? When this situation arises, I figure out the maximum number of elements that the array would have to hold in the most demanding case. If the number is less than or equal to 256 and each element is reasonably small (less than 256 bytes) then I usually statically allocate it and use a counter to count how many of the elements are active at any time.

You may think that this process is a waste of memory, and it is; but a preallocated array of a fixed size is easier and faster for the processor to traverse than a linked list or a more dynamic structure. My point: If you know the number of elements ahead of time and that number is small, go ahead and preallocate it or `malloc()` the memory at start up.

Don't get carried away with static arrays. Suppose that you have a 4K structure and you will need from 1 to 256 static records. Allocating 1MB of memory — in case the number *may* increase to 256 at some point — is a poor strategy.

Linked lists

Arrays are fine for simple data structures that can be precounted or estimated at compilation or start up, but data structures that can grow or shrink during run-time should use some form of *linked list*. Figure 24-1 depicts a standard abstract linked list. A linked list consists of a number of nodes, with each node containing information and a link to the next node in the list.

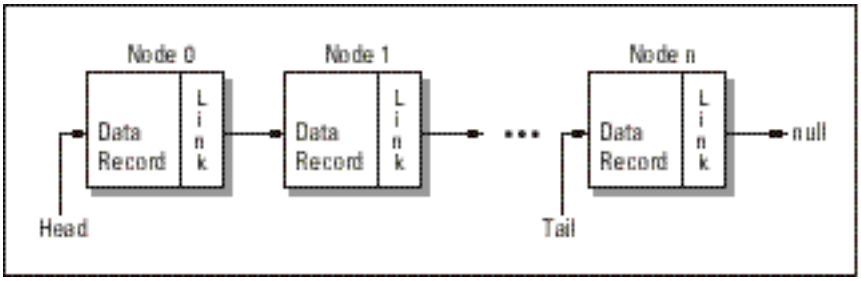

Figure 24-1: A linked list.

Linked lists are cool because you can insert or delete a node anywhere in the list (see Figure 24-2). The capability of a linked list to insert and delete nodes (and, therefore, information) during run-time makes them very attractive as a data structure for games.

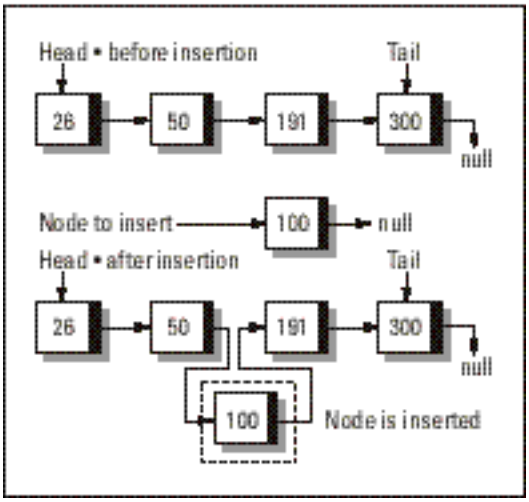

Figure 24-2: Inserting into a linked list.

The only bad thing about linked lists is that you must traverse them node-by-node to find what you are looking for. For example, suppose that you want the 15th element in an array; you can access it like this:

```
players[15]
```

But with linked lists, you need a *traversal algorithm* (which is a method to visit each node in the list) to traverse the list to find the 15th element. In the worst case, the searching of linked lists can take a number of iterations equal to the length of the list, represented mathematically as O(*n*) — read "big O of n." Of course, you can employ optimizations and secondary data structures to maintain a sorted indexed list that allows access almost as fast as the simple array.

Creating a linked list

For an example of a simple linked list, take a look at how to create a linked list, add a node, delete a node, and search for an item with a given key. Here's the basic node:

```
typedef struct NODE_TYP
   {
   int id;          // ID number of this object
   int x,y;         // position of object
   int color;       // color of object
   NODE_TYP *next; // this is the link to the next node
                    // more fields go here
   } NODE, *NODE_PTR;
```

Then to start the list off, you need a *head* pointer and a *tail* pointer that points to the head and tail of the list, respectively. However, because the list is empty, the pointers start off pointing to NULL.

```
NODE_PTR    head = NULL,
            tail = NULL;
```

Traversing a linked list

Ironically, traversing a linked list is the easiest of all operations. To traverse a linked list, follow these steps:

1. **Start at the head pointer.**
2. **Visit the node.**
3. **Link to the next node.**
4. **If the node is not** NULL**, then go to Step 2.**

And here's the code:

```
void Traverse_List(NODE_PTR head)
{
// this function traverses the linked list and prints out
// each node

// first test whether head is null
if (head==NULL)
   {
   printf("\nLinked List is empty!");
   return;
   } // end if
// traverse while nodes
while (head!=NULL)
      {
      // visit the node, print it out, or whatever...
      printf("\nNode Data: id=%d", head->id);
      printf("\nx=%d, y=%d",head->x, head->y);
      printf("\ncolor=%d\n",head->color);
      // advance to next node (simple!)
      head = head->next;
      } // end while
} // end Traverse_List
```

Pretty cool, huh? In the next subsection, I explain how to add a node.

Adding a node (insertion)

The first step in adding a node is to create it. You can use either of two approaches:

- Send the new data elements to the insertion function and let it build up a new node.
- Build up a new node and then pass it to the insertion function.

Both methods achieve the same result. You can choose from a number of ways to insert a node into a linked list. The brute force method is to add it to the front or the end. This approach is fine if you don't care about the order; but if you want the list to remain sorted, use a more intelligent insertion algorithm that maintains order in either ascending or descending order. This process makes searching much faster.

For simplicity's sake, I took the easy way out and inserted at the end of the list, but inserting with sorting is not that much more complex. You first need to scan the list, find the location at which the new element should be inserted, and then insert the new element. Your only problem will be keeping track of the pointers and not losing any nodes or links.

Here's the code to insert a new node at the end of the list (a bit more difficult than the front of the list). Notice the special cases for empty lists and lists with a single element.

```
// access the global head and tail to make code easier;
// in real life, you may choose to use ** pointers and
// modify head and tail in the function
NODE_PTR Insert_Node(int id, int x, int y, int color)
{
// this function inserts a node at the end of the list
NODE_PTR new_node = NULL;

// Step 1: create the new node
new_node = malloc(sizeof(NODE)); // in C++ use new operator
// fill in fields
new_node->id = id;
new_node->x  = x;
new_node->y  = y;
new_node->color = color;
new_node->next = NULL; // good practice

// Step 2: find the current state of the linked list
if (head==NULL) // case 1
   {
   // finding an empty list means using the simplest case
   head = tail = new_node;
   // return new node
   return(new_node);
   } // end if
else
if ((head != NULL) && (head==tail)) // case 2
   {
   // you have exactly one element; this code is really
   // just a little finesse...
   head->next = new_node;
   tail = new_node;
   // return new node
   return(new_node);
   } // end if
else // case 3
   {
   // in case 2 or more elements are in list,
   // simply move to end of the list and add
   // the new node
```

```
   tail->next = new_node;
   tail = new_node;
   // return the new node
   return(new_node);
   } // end else
} // end Insert_Node
```

As you can see, the code is rather simple, but it is easy to mess up because you are dealing with pointers, so be careful! Also, the astute programmer very quickly realizes that, with a little thought, cases 2 and 3 can be combined; however, the preceding code is easier to follow than the code which combines cases 2 and 3.

Deleting a node

Deleting a node is the most complex of all linked-list operations, or at least up there in the record books.

The problem with deletion is that in most cases you want to delete a specific node. The node may be at the head, tail, or in the middle; therefore, you must write a very general algorithm that takes all these cases into consideration. If you're careful, deletion isn't a problem; but if you don't take all the cases into consideration and test them, you'll be sorry!

Now that you're scared of the linked-list police, here's the code to delete a node from a fictitious linked list using the `id` as the key:

```
// this function will modify the globals
// head and tail (possibly)
int Delete_Node(int id) // node to delete
{
// this function deletes a node from
// the linked list given its ID
NODE_PTR curr_ptr = head, // used to search the list
         prev_ptr = head; // previous record
// test whether a linked list to delete from is present
if (!head)
    return(-1);
// traverse the list and find node to delete
while(curr_ptr->id != id)
     {
     // save this position
     prev_ptr = curr_ptr;
     curr_ptr = curr_ptr->next;
     } // end while
```

(continued)

(continued)

```
// at this point we have found either the node
// or the end of the list
if (curr_ptr == NULL)
    return(-1); // couldn't find record

// the record was found, so delete it, but be careful;
// there are a number of cases to test for

// need to test cases
// case 1: one element
if (head==tail)
   {
   // delete node
   free(head);
   // fix up pointers
   head=tail=NULL;
   // return id of deleted node
   return(id);
   } // end if
else // case 2: front of list
if (curr_ptr == head)
   {
   // move head to next node
   head=head->link;
   // delete the node
   free(curr_node);
   // return id of deleted node
   return(id);
   } // end if
else // case 3: end of list
if (curr_ptr == tail)
   {
   // fix previous pointer to point to null
   prev_ptr = NULL;
   // delete the last node
   free(curr_ptr);
   // point tail to previous node
   tail = prev_ptr;
   // return id of deleted node
   return(id);
   } // end if
```

```
else // case 4: node is in middle of list
   {
   // connect the previous node to the next node
   prev_ptr->next = curr_ptr->next;
   // now delete the current node
   free(curr_ptr);
   // return id of deleted node
   return(id);
   } // end else
} // end Delete_Node
```

Note that the code contains a lot of special cases. Each is simple, but you have to think of every possible scenario — which I hope that I did!

Finally, you may have noticed the drama in the code when deleting nodes from the interior of the list. The problem occurs because, once a node is traversed, you can't get back to it. Therefore, I had to keep track of a previous NODE_PTR to keep track of the last node.

This problem can be solved along with others by using what is called a *double linked list* (as shown in Figure 24-3). The cool thing about a double linked list is that you can traverse in both directions from any point, and insertions and deletions are much easier. And the only change to the data structure is another link field, as shown (in bold) in the following code:

```
typedef struct NODE_TYP
   {
   int id;    // ID number of this object
   int x,y;   // position of object
   int color; // color of object
   NODE_TYP *next; // link to the next node
   NODE_TYP *prev; // link to previous node
   // more fields go here
   } NODE, *NODE_PTR;
```

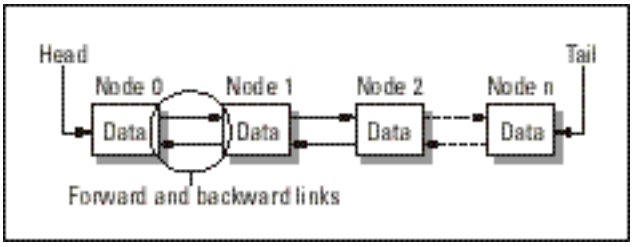

Figure 24-3: A double linked list.

Trees

The next class of advanced data structures are *trees*. Take a look at Figure 24-4 to see a number of different treelike data structures.

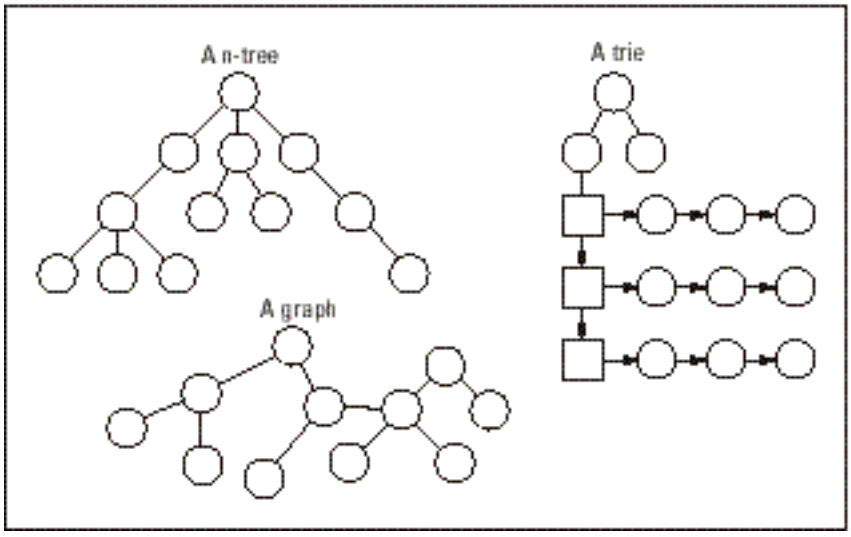

Figure 24-4: Some tree topologies.

Trees were invented to help with searching and storing large amounts of data. The most popular kind of tree is the *binary tree* or *B-tree*. The binary tree is a tree data structure emanating from a single root that is composed of a collection of nodes. Each node has one or two child nodes descending from it — hence, the term *binary*. Moreover, we talk of the *order* or number of levels of a tree, meaning how many layers (or levels) of nodes. For example, the tree in Figure 24-5 is a three-level tree.

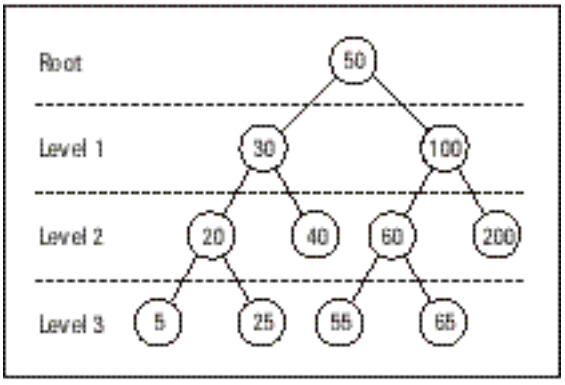

Figure 24-5: A three-level tree.

The interesting thing about trees is how fast the information can be searched. Most B-trees use a single search key to order the data in the tree. Then a searching algorithm searches the tree for the data.

For example, suppose that you want to create a B-tree that contains records of game objects, each with a number of properties. You can use the time of creation as the key. Here's the data structure that you would use to hold a single node:

```
typedef struct TNODE_TYP
   {
   int time;  // time of creation
   int x,y;   // position of object
   int color; // color of object
   NODE_TYP *right; // link to right node
   NODE_TYP *left;  // link to left node
   } TNODE, *TNODE_PTR;
```

Notice the similarity between the tree node and the linked-list node (covered in the earlier subsection "Linked lists"). The only difference is really the way you use the data structure and build up the tree.

Continuing with the example, suppose that I have five objects with the following creation times: t={0,25,3,12,10}. Figure 24-6 depicts two different B-trees that contain this data. However, a number of topologies exist that would maintain the properties of a B-tree.

In Figure 24-6, I use the convention that any right child is greater than or equal to its parent and any left child is less than its parent. You can use a different convention as long as you stick to it.

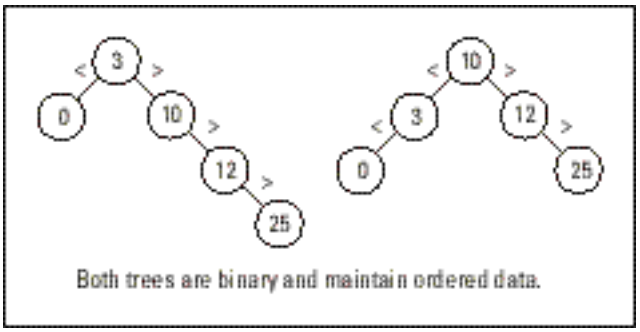

Figure 24-6: B-tree encoding of data set (0,25,3,12,10).

Unfortunately, I don't have time to cover the code for creating, searching, and working with B-trees, so you'll have to get a book or do some more research if you're interested (try *Programs and Data Structures In C*, by Leendert Ammeraal, Wiley Press). But I can tell you what B-trees bring to game programming.

Binary trees can hold enormous amounts of data, and that data can be quickly searched by using a binary search. This property is a manifestation of the binary structure of the tree. For example, if you have a tree with a million nodes, then at most it will take you 20 comparisons to find any desired record! Is that crazy or what? The reason for such a small number of comparisons is that at each iteration of your search (as you compare the key you are looking for against the current node you are visiting), you cut half the nodes out of the search space.

The above statement about search time is only true for balanced trees (trees that have an equal number of right and left children per level). If a tree is totally unbalanced, it degrades into a linked list and search time degrades into a linear function.

The next cool thing about B-trees is that if you take a branch (a subtree) and process it separately, the branch maintains the properties of a B-tree. Therefore, if you know where to look, you can search only the branch for whatever it is you're looking for.

When do you use B-trees? I suggest that you use treelike structures when the problem or data is treelike to begin with. If you find yourself drawing out the problem and you see branches to the left and right, then a tree is definitely for you. For example, in Bonus Chapter 23 on artificial intelligence, I speak of creating memories for the game characters. A tree structure would be perfect for memory. Each node could represent a room, and the children off of each node could represent the various objects that exist in each room.

Algorithmic Xtasy

Algorithm design and algorithmic analysis are complex subjects and usually are senior-level computer science material, but I can at least touch upon some common-sense techniques and ideas to help you out when you start writing more complex algorithms — because brute-force, sloppy programming just isn't good enough in many cases.

A good algorithm is better than all the assembly language or optimization in the world. For example, just by re-ordering your data, you can reduce the amount of time necessary to search for a data element by orders of

magnitude. So the moral of the story is to select a good solid algorithm that fits the problem and the data, but at the same time to pick a data structure that can be accessed and manipulated with a good algorithm. I mean, if you always use linear arrays, you're never going to get better than linear search time (unless you use secondary data structures); but if you use sorted arrays, you can get logarithmic search time.

The first step to *writing* good algorithms is having some clue about how to *analyze* them. The art of analyzing algorithms is called *asymptotic analysis* and is usually calculus-based, so I'm just going to skim some of the concepts.

The basic idea of analyzing an algorithm is to compute how many times the main loop is executed for *n* elements, whatever *n* means. Of course, how long each execution takes plus the overhead of setup can also be important after you have a good algorithm, but the first place to start is the general counting of how many times. Take a look at two examples:

```
for (int index=0; index<n; index++)
    {
    // do work, 50 cycles
    } // end for index
```

In this case, the loop is going to execute for n iterations, thus the execution time is of the order *n*, or O(*n*). As explained in the earlier section called "Linked lists," Big O is a very rough upper estimate of execution time. You can be more precise in this case because you know that the inner computation takes 50 cycles; so the total execution time is:

```
n*50 cycles
```

Right? *Wrong!* If you are going to count cycles, then you had better count the cycles that it takes for the loop itself. This calculation consists of an initialization of a variable, a comparison, an increment, and a jump for each iteration. Adding in these factors, you end up with something like this:

$$Cycles_{initialization}+(50+Cycles_{inc}+Cycles_{comp}+Cycles_{jump})*n$$

This estimate is much more accurate. Of course, $Cycles_{inc}$, $Cycles_{comp}$, and $Cycles_{jump}$, are the number of cycles for the increment, comparison, and jump, respectively, and are each around 1 to 2 cycles on a Pentium-class processor. Therefore, in this case, the loop itself contributes just as much to the overall time of the inner loop as does the work performed by the loop!

Loop overhead is a key point. For example, many game programmers write a pixel-plotting function as a function instead of a macro or inline code. Because a pixel-plotting function is so simple, the call to the function takes

more time than the pixel plotting! So make sure that you do enough work within your loop to warrant the usage of a loop in the first place. If the work within the loop "drowns" out the loop mechanics, then you should be okay.

The following code example has a much worse running time than *n*:

```
// outer loop
for (i=0; i<n; i++)
    {
    // inner loop
    for (j=1; j<2*n; j++)
        {
        // do work
        } // end for j
    } // end for i
```

In this code block, I'm assuming that the "work" part takes much more time than the actual code that supports the loop mechanics, so I'm not interested in the loop mechanics. What I am interested in is how many times this loop executes. The outer loop executes *n* times and the inner loop 2×*n*–1 times; thus the total amount of time the inner code will be executed is:

$$n\times(2\times n-1) = 2\times n^2-n$$

Look at these two terms for a moment. The $2\times n^2$ term is the dominant term and will drown out the *n* term as *n* gets larger (see Figure 24-7).

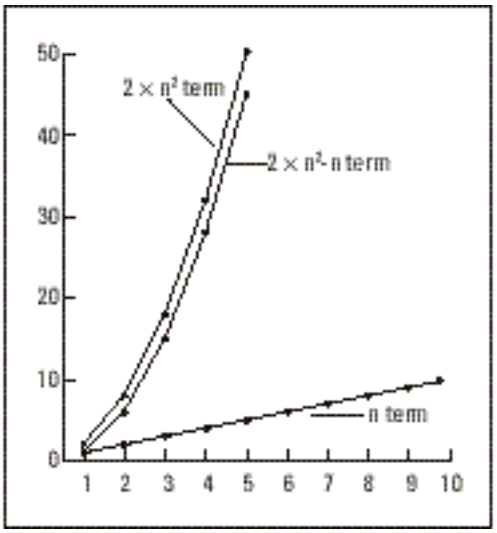

Figure 24-7: Rates of growth for the term of $2\times n^2 n$.

For a small *n* — for example, when *n* equals 2 — the *n* term *is* relevant:

$$2\times(2)^2-2 = 6$$

In this case, the *n* term contributed to subtracting 25 percent of the total time away. But take a look at what happens when *n* gets larger; for example, when *n* equals 1,000.

$$2\times(1{,}000)^2-1{,}000 = 1{,}999{,}000$$

In this case, the n term contributes a decrease of only .05 percent; hardly important. Thus, you can see that the dominant term is indeed the $2\times n^2$ term, or more simply the n^2 itself. Therefore, this algorithm is $O(n^2)$. This result is very bad. Algorithms that run in n^2 time will just kill you — well, at least will kill the performance of your code — so if you come up with an algorithm like this, then try, try again!

That's it for asymptotic analysis; the bottom line is that you must be able to roughly estimate the run-time of your loops. This estimation will help you pick out the best algorithms and recode areas that need work.

Optimization Theory

No other programming has the kind of performance requirements that games do. Video games have always pushed the limits of hardware and software and will continue to do so. The reason for this: Enough is never enough. Game programmers always want to add one more creature, effect, or sound, as well as increase or improve the AI. Therefore, optimization is of the utmost importance. In this section, I cover some optimization techniques to get you started. If you are interested in reading more about this subject, a number of good books on the subject are available (try *Black Art of 3D Game Programming,* by André LaMothe; Waite Group Press).

Using your head

The first key to writing optimized code is understanding the compiler, data types, and the way your C/C++ is finally transformed into executable machine language. The best idea is to use simple programming and simple data structures. The more complex and contrived your code is, the more difficult time the compiler is going to have converting to machine code and, thus, the slower your code is going to execute (in most cases). Here are some basic rules to keep in mind:

- Use 32-bit wide data as much as possible; 8-bit data may take up less space, but Intel processors like 32-bit data are optimized to access it.
- Use inline functions for small functions that you call a lot.
- Use globals as much as possible without making ugly code.
- Avoid floating-point numbers for addition and subtraction.
- Use integers whenever possible, even though the floating point processor is almost as fast as the integer processor. Integers are exact, so if you don't need decimal accuracy, use integers.
- Align all data structures to 32-byte boundaries. You can do this manually or with compiler directives on most compilers.
- Never pass data to functions as *value* if the data is anything other than a simple type; always use a pointer.
- Don't use the `register` keyword in your code. Although Microsoft says that this keyword makes faster loops, it starves the compiler of registers and ends up making horrible code.
- If you're a C++ programmer, then it's okay for you to use classes and virtual functions; just don't go crazy with inheritance and layers of software.
- The Pentium-class processors use an internal data and code cache. Be aware of this arrangement and try to keep the size of your functions relatively small so they can fit into the cache (16K to 32K). In addition, when you store data, store it in the way it will be accessed. This method minimizes cache thrashing and main memory or secondary cache access, which is ten times slower than the internal cache.
- Be aware that Pentium-class processors have RISC-like cores, and they like simple instructions, allowing two or more instructions to execute in more than one execution unit. Don't write contrived code on a single line. Writing simpler code lines is better, even though you can mash the same functionality on the same line.

Working mathematical sorcery

Because a great deal of game programming is mathematical in nature, it pays to know advanced ways to perform math functions. You can use a number of general tricks and methods to enhance math performance and speed up operations.

The first I cover briefly is fixed-point math, which is an advanced subject, and I refer you to my other book, *The Black Art of 3D Game Programming* (published by Waite Group Press) for a more complete treatise on this topic. However, here is a list of math tricks you can use to speed up operations:

- With regard to data types, always use integers with integers and floats with floats. Conversion from one to another kills performance. Hence, hold off on the conversion of data types to the very last minute.
- Integers can be multiplied by any power of 2 by shifting to the left. And likewise, they can be divided by any power of 2 by shifting to the right. Multiplication and division other than by power of 2 is accomplished by using sums or subtractions of shifts. For example, 640 is not a power of two, but 512 and 128 are, so here's the best way in C code to multiply a number by 640 using shifts:

```
product=(n<<7) + (n<<9); // n*128 + n*512 = n*640
```

- If you use matrix operations in your algorithms, then make sure that you take advantage of the sparseness of those operations.
- When you create constants, make sure that they have the proper casts, so that the compiler doesn't reduce them to integers or interpret them incorrectly. The best idea is to use the C++ `const` directive; for example:

```
const float f=12.45;
```

- Avoid square roots, trigonometric functions, or any complex mathematical functions. In general, find a simpler way to accomplish the operation by taking advantage of certain assumptions or making approximations. However, you can always make a lookup table as shown in the section "Appreciating lookup tables."
- If you have to zero out a large array of floats, use a `memset()` like this:

```
memset((void*)float_array,0,sizeof(float)*num_floats);
```

  However, you can only use `memset()` in this situation, because floats are encoded in IEEE format and the only value that is the same in both integer and float values is 0.

- When you perform mathematical calculations, see if you can reduce the expressions manually before coding them. For example, $n\times(f+1)\div$ is equivalent to $(f+1)$ because the multiplication and division of n cancel out.
- If you perform a complex mathematical operation and you need it again a few lines down in the code, then cache it; for example:

```
// compute term that is used in more
// than one expression
float n_squared = n*n;

// use term in two different expressions
pitch = 34.5*n_squared+100*rate;
magnitude = n*squared / length;
```

- And last, but not least, make sure that you set the compiler options to use the floating point processor and create code that is *fast* (runs the quickest) rather than *small* (takes up the least amount of RAM).

Unrolling the loop

The next optimization trick is *loop unrolling*, which was one of the best optimizations possible back in the 8- and 16-bit days, but today it can backfire on you.

Unrolling the loop means to take apart a loop which iterates some number of times and to manually code each line as the loop would have mechanically. Here's an example:

```
// loop before unrolling
for (int index=0; index<8; index++)
    {
    // do work
    sum+=data[index];
    } // end for index
```

The problem with this loop is that the "work" section takes less time than the loop does for the increment, comparison, and jump. Hence, the loop code itself doubles or triples the amount of time the code requires!

To fix this problem with the code, unroll the loop like this:

```
// the unrolled version
sum+=data[0];
sum+=data[1];
sum+=data[2];
sum+=data[3];
sum+=data[4];
sum+=data[5];
sum+=data[6];
sum+=data[7];
```

This approach is much better.

However, consider these two caveats to the code listed above:

- If the loop body is much more complex than the loop mechanics itself, then you really don't need to unroll it. For example, if you are computing square roots in the "work" section of the loop, then a few more cycles in each iteration isn't going to help you.

Pentium processors have internal caches, and unrolling a loop too much may cause it to be unable to fit in the internal cache. This situation is disastrous and will bring your code to a halt. I suggest unrolling (if appropriate) 8 to 32 times, depending on the situation.

Appreciating lookup tables

This is my personal favorite optimization. *Lookup tables* are precomputed values of some computation that you know you will perform during runtime. You simply compute all possible values at startup and then run the code.

For example, suppose that you need the sine and cosine of the angles from 0 to 359 degrees. Computing them by using `sin()` and `cos()` would kill your math performance if you use the floating point processor; but by utilizing a lookup table, your code can compute `sin()` or `cos()` in a few cycles because the process involves just grabbing the number from a lookup table. Here's an example:

```
// storage for look up tables
float SIN_LOOK[360];
float COS_LOOK[360];
// create lookup table
for (int angle=0; angle < 360; angle++)
    {
    // convert angle to radians because the math library
    // uses rads instead of degrees
    // remember that 2*pi rads are in 360 degrees
    float rad_angle = angle * (3.14159/180);

    // fill in the remaining entries in lookup tables
    SIN_LOOK[angle] = sin(rad_angle);
    COS_LOOK[angle] = cos(rad_angle);
    } // end for angle
```

As an example of using the lookup table, here's the code to draw a circle of radius 10:

```
for (int ang = 0; ang<360; ang++)
    {
    // compute the next point on circle
    x_pos = 10*COS_LOOK[angle];
    y_pos = 10*SIN_LOOK[angle];
    // plot the pixel
    Plot_Pixel((int)x_pos+x0, (int)y_pos+y0, color);
    } // end for ang
```

Of course, lookup tables take up memory, but they are well worth it. If you can precompute a set of values that you'll need in your code, then put the set in a lookup table. That's my motto. (And if you have a hard time believing that the really cool and complex games today don't use lookup tables, think again; how do you think that *Doom* and *Quake* work?)

Using assembly language

The final optimization I want to talk about is using *assembly language*.

So you have the killer algorithm and all your data structures are good, but you just want a little bit more *oomph* to your code's speed. Hand-crafted code written in assembly language doesn't make code go 1,000 times faster with 32-bit processors like it did with 8- and 16-bit processors, but it can get you 2 to 10 times more speed, and that result is definitely worth it.

However, make sure that you only try to convert sections of your game that need converting. Don't mess with converting the menu program to assembly, because that's a waste of time. Use a profiler or similar analysis program to see where all your game's CPU cycles are being eaten up (probably in the graphics sections) and then target those for conversion to assembly language.

In the old days (a few years ago), most compilers didn't have inline assemblers, and if they did, the inline assemblers were awful and supported very few features of an external assembler. Today, the inline assemblers that come with Microsoft, Borland, or Watcom compilers are really good and just about as full featured as a standalone assembler for small jobs that range from a few dozen lines to a couple hundred. Therefore, I suggest using the inline assembler in your compiler if you want to do any assembly language.

Here's how you invoke the inline assembler in Microsoft Visual C++ 2.0+:

```
_asm
{
[assembly language code here]
} // end asm
```

The cool thing about the inline assembler is that it enables you to use variable names that have been defined by C/C++. For example, here's how to write a 32-bit memory fill function using inline assembly language:

```
void qmemset(void *memory, int value, int num_quads)
{
// this function uses 32-bit assembly language based
```

```
// on the string instructions to fill a region of memory
_asm
   {
   CLD                // clear the direction flag
   MOV EDI, memory    // move pointer into EDI
   MOV ECX, num_quads // ECX hold loop count
   MOV EAX, value     // EAX hold value
   REP STOSD          // perform fill
   } // end asm
} // end qmemset
```

To use the new function, all you do is this:

```
qmemset(&buffer, 25, 1000);
```

And 1,000 quads would be filled with the value 25 starting at the address of buffer.

If you're not using Microsoft Visual C++, then take a look at your particular compiler's Help file to see the exact syntax needed for inline assembly. In most cases, the changes to the prior code block are an underscore here and there and nothing more.

Making Demos

So you've got this killer game and you need a demo mode. You can use two main methods to implement a demo mode:

- Play the game yourself, record your own moves, and then play the moves back.
- Use an AI player that plays the game unattended.

Recording game play turns out to be the most common choice, because writing an AI player that can play as well as a human is difficult. In addition, it's difficult to let the AI demo player know that it needs to make a good impression on potential buyers by playing the game in a "cool" way. The next sections take a brief look at how each of these methods are implemented.

Prerecorded

To record a demo, follow these steps:

1. **Record the state of all the input devices each cycle as you create the demo.**
2. **Write the data to a file.**
3. **Play back the demo as if it were the input of the game.**

Take a look at Figure 24-8 to see this point graphically. The idea is to create your demo so that the game doesn't know whether the input is from the keyboard (input device) or from a file, so it simply plays the game back.

For this process to work, you need to have a *deterministic* game. This term means that if you play the game again and do the exact same moves, then the game creatures will also respond the same way. As well as recording the input devices, you must record the initial random-number seed as well, so that the starting state of a game is recorded as well as the input. This step ensures that the game will play back in the exact same way as you recorded it.

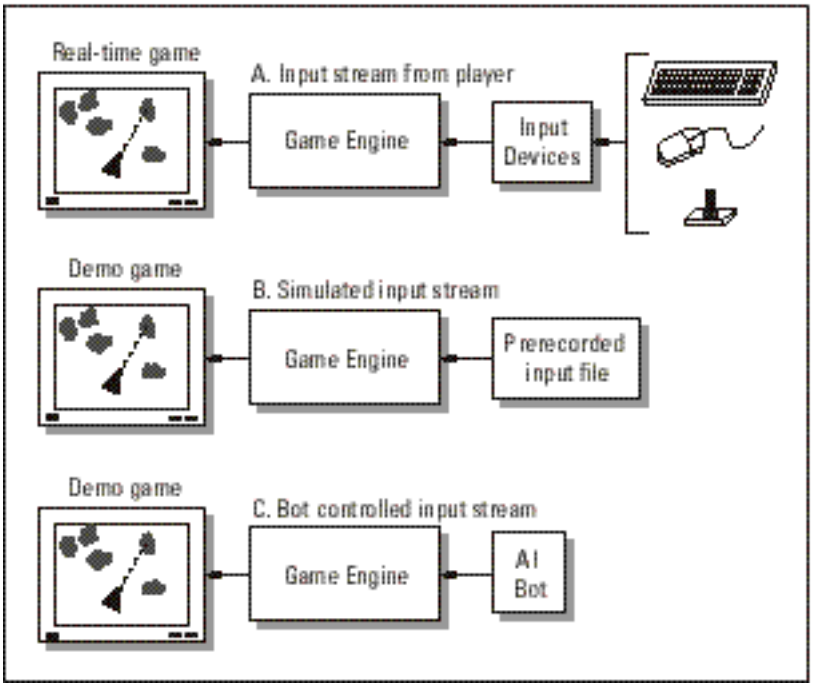

Figure 24-8: Demo playback.

To record a game, the best approach is to not sample the input at time intervals, but to sample the input at each frame. Therefore, if the game is played on a slower or faster computer, the playback data won't get out of synchronization with the game. Here are the steps your code should follow:

1. **Create a general input record.**
2. **Merge all the input devices into the single record each cycle.**
3. **As the game runs, write each input record to a file (one for each frame).**

Also, at the beginning of the file, I place any state information or random numbers that I played the demo with, so that these values can be loaded back in.

For example, the playback file may look something like this:

```
Initial State Information
Frame 1: Input Values
Frame 2: Input Values
Frame 3: Input Values . . .
Frame N: Input Values
```

After you have the file, you reset the game and simply start it up. Then you read the file as if it were the input devices. The game doesn't know the difference and simply plays!

The single mistake that you can make in creating the demo is sampling the input at the wrong time when you write records. Make absolutely certain that the input you sample and record is the actual input that the game uses for that frame. A common mistake newbies make is to sample the input for the demo mode at a point in the event loop before or after the normal input is read. Hence, you are sampling different data! It's possible that the play may have the fire button down in one part of the event loop and not in another; thus you must sample at the same point you normally read the input for the game.

AI controlled

The second method of recording a game is by writing an AI bot that plays, much like people do for Internet games such as *Quake*. The bot plays the game while in demo mode as if it were one of the AI characters in the game. The only problem (other than the technical complexity) is that the bot may not necessarily show off all the cool rooms, weapons, and so on, because it doesn't know that it's making a demo. On the other hand, the cool thing about having a bot play is that each demo is different and the attract mode of the game will never get boring.

Implementing a bot to play your game is like using any other AI character: You connect it to the input port of your game and override the normal input stream (refer to Figure 24-8). Then you write the AI algorithms for the bot and give it some main goals, such as finding its way out of the maze or killing everything in sight. Finally, you simply let the bot loose to demo until the player wants to play.

Saving the Game

One of the biggest pains in the butt is writing a save-game feature. This task is one that all game programmers do last and do by the seat of their pants, in most cases. The key is to write your game with the idea that you want to give the player a save-game option at some point, so that you don't dig yourself into a corner.

To save a game at any point in the game means to record the state of every single variable in the game and the state of every single object in the game. Therefore, you must record in a file all global variables along with the state of every single object.

The best way to approach this task is by adopting an object-oriented thought process. Instead of writing a function that writes out the state of each object and all the global variables, teach each object how to write and read its own state to a disk file.

Then to save a game, all you need to do is write the globals and create a simple function that requests each game object to write its own state. To load the game back in, all you need to do is read the globals back into the system and load the state of all the objects back into the game.

This way, if you add another object or object type, the loading/saving process is localized in the object itself, rather than strewn about all over the place in your code.

Implementing Multiple Players

The last little tidbit of game programming legerdemain is implementing multiple players. Of course, if you want to implement a networked game, that's a whole other story, but DirectPlay makes the communication part easy at least. However, if all you want to do is let two or more players play your game at the same time or by taking turns, then that flexibility requires nothing more than extra data structures and a bit of housekeeping.

Taking turns

Implementing turn-taking is simple and difficult at the same time. The task is simple because if you can implement one player, then implementing two or more is nothing more than having more than one player record. But the task is difficult because you must save the game for each player when switching players. Hence, you need to implement a save-game option if you want to allow for turn-taking. Obviously, the players shouldn't know that the game is being saved as they take turns, but that's what's really going on.

Here's a list of the steps to allow two players to play, one after the other:

1. **Start game; player 1 begins.**
2. **Player 1 plays until she dies.**
3. **The state of player 1's game is saved, and player 2 begins.**
4. **Player 2 plays until he dies.**
5. **The state of player 2's game is saved.**

   Here comes the transition.

6. **The previously saved game of player 1 is reloaded and player 1 continues.**
7. **Go back to Step 2.**

As you can see, Step 5 is where the action starts happening and the game starts pinging back and forth between players. And if you want more than two players, you simply play them one at a time until you're at the end of the list and then you start over.

Appearing on-screen at the same time

Playing two or more players on the same screen is a little more difficult than swapping, because you have to write the game a little more generally as far as game play, collision, and interaction between the players goes. Moreover, now that two or more players are on the screen at the same time, you must allocate a specific input device for each player. This device is usually a joystick for each player, or maybe one player uses the keyboard and one uses the joystick.

The other problem with putting two or more players on the screen at the same time is that some games just don't work well with two players at the same time. For example, if the game is a scrolling game, one player may want to go one way while the other wants to go another way. This dilemma

can cause a conflict, and you'll have to think about it as you program. Thus, the best games for implementing more than one player are games that are single-screen, such as fighting games or other games in which the players stay relatively near each other.

If you want to allow the players to roam around freely, you can always generate more than one view — create a split-screen display (as shown in Figure 24-9). The only problem with a split-screen display is the split-screen display! You must generate two or more views of the game. This step can be technically challenging, moreover, because the players may not be able to see what's going on if the screen is too small to accommodate two views. The bottom line is this: If you can pull it off, then it's a cool option.

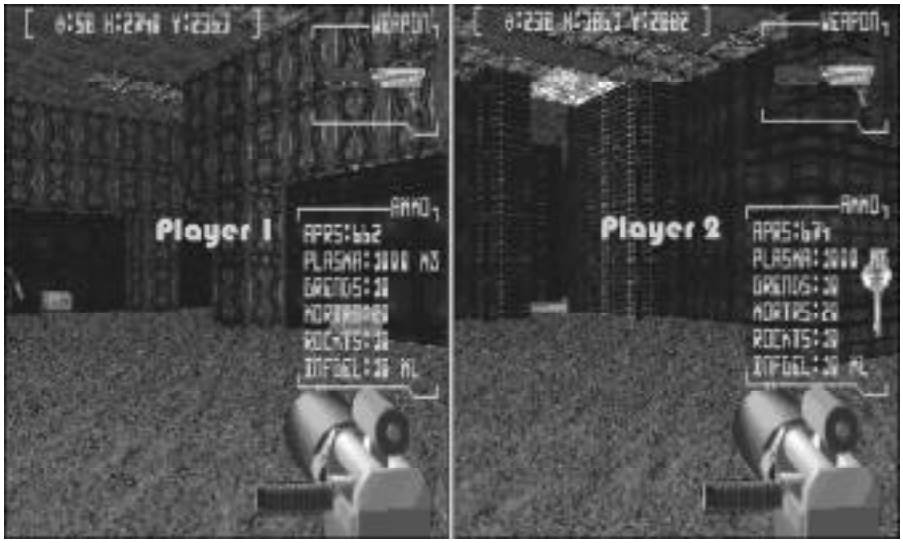

Figure 24-9: Split-screen game display.