Data Structure Series

This series is actually something I started back when I was part of the Sweet.Oblivion staff, but then some things happened and I was no longer able to complete it. So now, after finally retrieving copies of the original articles, I'm going to make a second attempt at a series of articles that cover step-by-step the design, construction, and implementation of the most common data structures used in software development today.

I will start off the series by briefly introducing basic pointer concepts as well as giving an introduction to our first data structure, linked lists. I will then begin to explore more advanced data structures including, but not limited to stacks, queues, binary trees, AVL trees, splay trees, B-trees, hashes, various types of priority queues (aka heaps), graphs, and quite possibly some topics regarding algorithm analysis and the design of several algorithms that utilize these data structures. I will also make an effort to explain how each structure might be used in your game development.

The compiler of choice for this series is Microsoft Visual C++ 6, and both C and C++ syntax will be used throughout. Also, since my example programs are to be very simple and to the point, all of the programs will be using the console application setup – basically an MS-DOS program.

Now that you have the general idea of this series, let me warn you that this series does have a chance of not being completed, but considering the general lack of *good* tutorials and documents covering this material, I'll do my best to follow through with each part. I just cannot guarantee that between my personal life, work on GameDev.net, and other issues that may arise in the future, that I will be able to complete it. So with this in mind, let's get on with the show.

Introduction to Linked Lists
Part 1

Before we can begin discussing any dynamic data structure, we need to verify that you have a solid background on basic pointer operations.

A Quick Explanation

Dynamic memory is allocated from an area of memory known as the *heap* - a finite supply of memory that can be accessed by the programmer. It is possible to deplete all available memory from the stack or heap, since both are of some definite size that is machine-dependent. A program uses stack memory when a function is called, and this memory is released when the function exits. Allocation and deallocation of stack memory is therefore automatic. This is, however, not the case for heap memory, and the programmer must carefully manage the allocation and deallocation of heap memory.

You could write a set of functions that can help you keep track of how much heap memory you have allocated and how much is being used in the program, but that is beyond the scope of this article and would be a good topic for another article. Such a system could be used in resource management.

Allocating Memory

To use the memory allocation functions, **<STDIO.H>** must be included. Requesting memory from the heap is accomplished through the **malloc()** function, which is defined by this prototype:

void *malloc(size_t size);

The **malloc()** function returns a generic pointer to a chunk of memory containing **size** bytes. Remember that you must typecast the generic pointer returned by **malloc()** to the type of the pointer variable that is being allocated. If the heap is full, and no memory is available for allocation, **malloc()** returns NULL.

Other functions include **calloc()** (also used for memory allocation of an array) and **realloc()** (used to increase or decrease the size of a chunk of memory previously allocated). For our purposes, we will use **malloc()** and **free()**, which frees the memory that has been allocated for the pointer that is passed as **free()**'s parameter.

Using Pointers

Now let's get into some real implementation. Pointers are declared and dereferenced using an asterisk; the variable itself, without the asterisk, refers to the actual address of the first byte of the allocated memory. Remember, pointers contain as their value an address to another variable. The dereferenced pointer actually points to the value of a variable held in memory at the address indicated by the pointer. The & character (the address of operator) is used to determine the address of a variable.

Consider this fragment of code:

```
int i, *ptr1, *ptr2;                // integer and 2 pointers to integers
i = 11;                             // get space for one integer and assign a value
ptr1 = (int *)malloc(sizeof(int));      // note the typecast to integer
*ptr1 = 43;                         // Use already allocated space
ptr2 = &i;
printf("i, *ptr1, and *ptr2 = %d, %d, %d\n", i, *ptr1, *ptr2);
```

This code outputs:

```
i, *ptr1, and *ptr2 = 11, 43, 11
```

Let's run through the code very quickly. The first line declares one integer variable, and two pointers to integers. The second line assigns the value of **11** to integer variable **i**. Line three allocates memory for integer pointer **ptr1**. The fourth line assigns the value of **43** to the memory location that has been allocated by **ptr1**. Line five then sets the integer pointer **ptr2**'s **address** to the memory location occupied by integer variable **i**. The last line then prints out the values of integer variable **i**, and then the values being held in the memory locations pointed to by **ptr1** and **ptr2**.

If you have trouble understanding that, be sure to read over it several times, or look through a C/C++ book or tutorial on pointers. You need to have a solid understanding of pointers before moving on from this point in the series.

The Linked List

Now that you understand basic pointer concepts, you may be asking yourself, "What is this linked list thing?" Well, a *linked list* is a dynamic data structure consisting of records (called *nodes*) that hold data and are "'linked" to each other by a method determined by the programmer. The most common linking method is through pointers (addresses), such that each record contains the address of the next *node* in the list in addition to the record's regular data. The first *node* in the list, called the *head*, is used as the list's key identifier. *You must always keep track of the* ***head*** *of the list*. Why? Because it is the starting point that will be used later to retrieve the information that you have stored in the list. Lose the head, and you have lost the entry point to the entire list, particularly in our implementation here of the singly linked list. A list expands and shrinks (hence *dynamic data structure*) as data is added and deleted, allowing the list to accommodate an arbitrary number of elements. Compare this concept to the allocation of an array, which remains the same size during its lifetime. Figure 1.1 shows a visual example of a singly linked list, which we will be discussing throughout this article.

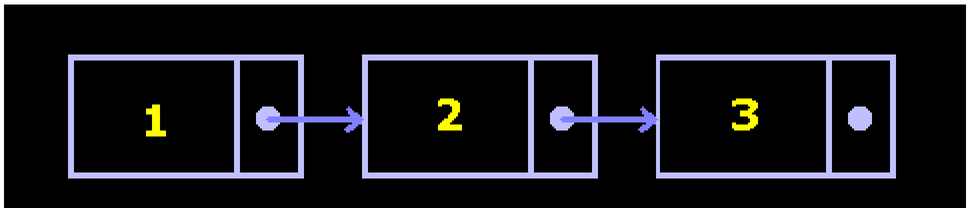

Fig 1.1 - Three Node Linked List

A linked list is a linear data structure. All operations on the list must begin by accessing the first node on the list, then the second node, then the third node, etc. Compared to arrays, this sequential access can be a significant performance drawback. Arrays can be accessed randomly – ever hear of the binary search algorithm? A linked list node can only be accessed after all preceding nodes have been accessed; this results in slower searching algorithms. There are, however, ways around this by manipulating and modifying the structure of the basic linked list that has been described here. We will explore these in a later part of the series.

The main advantage of linked lists is their dynamic nature. A list can grow to be quite big with dynamic allocation. New nodes can be added in between existing nodes with a few simple pointer manipulations, and deletions may be performed with a call to **free()** and some pointer redirection.

Some of the operations that we will be covering on the basic linked list include:

- List initialization
- Search
- Create a new node
- Node insertion
- Node deletion
- List traversal

Linked List Node Design

Now we will discuss the design of the *node* that can be used in your linked list implementations. The data placed inside a *node* is dependent upon the application of the list. For instance, let's say that you would like to use a linked list to keep track of the enemy ships that are currently alive and flying around in your space shooter. There are several ways to represent enemy ships with your nodes, but for the sake of simplicity, we will let each node hold the x and y coordinate of the ship on the screen.

In this example, we will create a **struct** to package the data.

```
typedef struct node
{
        int x, y;               // x and y coordinate
        struct node *next;      // pointer ("link") to the next node
} node_t;
```

There we have our basic node structure with the type defined as **node_t**. Note that you specify **struct node *next;** because it is a pointer to an incomplete type. The following code demonstrates how we can use this type to declare our variables.

```
node_t  nodeRec;                // a single node

node_t  *head;                  // head node of a linked list

typedef node_t  *node_ptr;      // create a node pointer type

node_ptr  head;                 // head node of a linked list
```

The first declaration, **node_t nodeRec**, declares a variable of the node structure. You can't really use this for your linked lists.

The second declaration, **node_t *head**, is what we're looking for. This declares a node pointer that we will use for the head of the linked list.

The third and fourth declarations are fairly self-explanatory. They create a new pointer type that allows for better code readability, and then declare the head of the linked list using this new pointer type.

Also keep in mind that the second and fourth lines are equivalent.

In Figure 1.2, you can see the visual representation of a single node. We will use this representation throughout the series for all the nodes in all the data structures we explore. The node has a data area, where all the node's information is stored, as well as a link area, where the links to other nodes are defined.

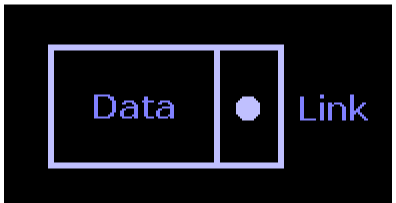

Fig 1.2 - The Node

When designing nodes, keep in mind that nodes have a **data** portion and a **link** portion. You can put any type of data that you want in the **data** portion, including pointers, structs, classes, or the more common ordinal types. However, for the **link** portion of the node, you must only create variables that will be used as links to other nodes. Here in the basic design of the linked list node, we created a single link that links to the next node in the list. In future articles, we will expand on this idea and add more links for more complex data structures.

In the meantime however, we need to discuss some of the basic operations that you can perform on linked lists.

Node Allocation

List nodes are created on demand. When data needs to be inserted, a new list node must be allocated to hold it. The **malloc()** function presented earlier is the basis for this allocation. The statement

```
newPtr = (node_ptr)malloc(sizeof(node_t));

or

newPtr = (node_t*)malloc(sizeof(node_t));
```

allocates our new node. Assuming that **malloc()** does not return NULL, we may now begin to assign the data portion of our node with values. Here's a quick example, using the coordinate node defined earlier:

```
newPtr->x = 10;
```

Now that we have the basis for allocating nodes, we create a function that will encapsulate all of the node allocation functionality into one block of code.

```
node_t* Allocate()
{
        node_t *newNode;                // our new node

        // request memory for our node
        newNode = (node_t*)malloc(sizeof(node_t));

        // error checking
        if (newNode == NULL)
                printf("Error: Unable to allocate new node.\n");
        else
                newNode->next = NULL;        // allocation succeeded, set the next
link to NULL

        return newNode;
}
```

There we have our basic node allocation function. We use this function like so:

```
node_t *myNode;
myNode = Allocate();
```

Despite the fact that the function **Allocate()** prints out an error message if the pointer is NULL, we should still verify that myNode is not NULL before trying to use it. Should this be the case, you may now use the data portion of the node and fill it with values.

Initialization

Initialization of a linked list is very quick and easy. We simply assign the value of NULL to the head of the list. This function only needs to be called once, and some of you may choose to not even bother. However, for the sake of this article, we will use it.

```
void Initialize(node_t **ptr)
{
        *ptr = NULL;
}
```

Using the definition of **head** given earlier, you would call this function as

```
Initialize(&head);
```

Node Insertion – Unsorted

We will now cover how to insert freshly allocated nodes into a linked list. Let's take a step back for a moment and consider insertion into an array. When using an array, you would typically add data to the end of the array, unless the data was sorted, in which case you would insert the data using some other method. We could do the same for an unsorted linked list, but this would require finding the last node because only the next to last node "knows" where the last node is located in memory. However, the **head** of the linked list points to the first node in the list, so a new node can be inserted at the *front* of the list without any extra work. You can see the steps of inserting three nodes into a linked list in Figure 1.3.

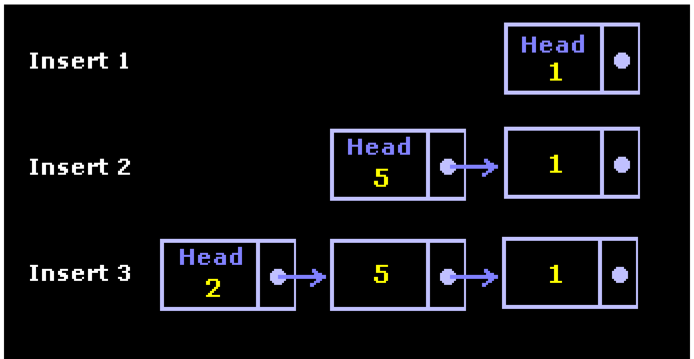

Fig 1.3 - Unsorted Node Insertion

And now the code:

```
void InsertFront(node_t **head, node_t newRecord)
{
        node_t*  newNode;               // pointer to a new node
        newNode = Allocate();

        if (newNode == NULL)
        {
                printf("Error: Not enough memory to allocate node.\n");
                return;
        }

        // fill the node with data
        newNode->id = newRecord.id;

        // insert at the front of the list
        newNode->next = *head;

        // move the head of the list to the new node
        *head = newNode;
}
```

To try to sum up this function in English: a new node is allocated and its data is filled; the new node's **next** field is set to point to the head node; the head is then reset to point to the new node. This function also works if the list is empty (*head = NULL) since we want the list to be NULL-terminated.

Linked List Traversal

In order to access each node in the list for data processing, we need to create a function that will *traverse* the list. There are several different uses for list traversal, including printing, searching, data manipulation, etc. Regardless of the use, you must traverse your list in order to really do anything useful with this data structure.

In this example, we will create a function called **DisplayList** that will display the x and y coordinates of every "enemy" in our list.

```
void DisplayList(node_t *head)
{
        node_t *current;                // our current node (position)

        // begin at the head of the list
        current = head;

        // loop until done
        while (current != NULL)
        {
                printf("ID = %d\n", current->id);
                current = current->next;
        }
}
```

Figure 1.4 shows the visualization for list traversal. As you can see, we move the **current** pointer through each node of the list until it reaches the **NULL** link at the end.

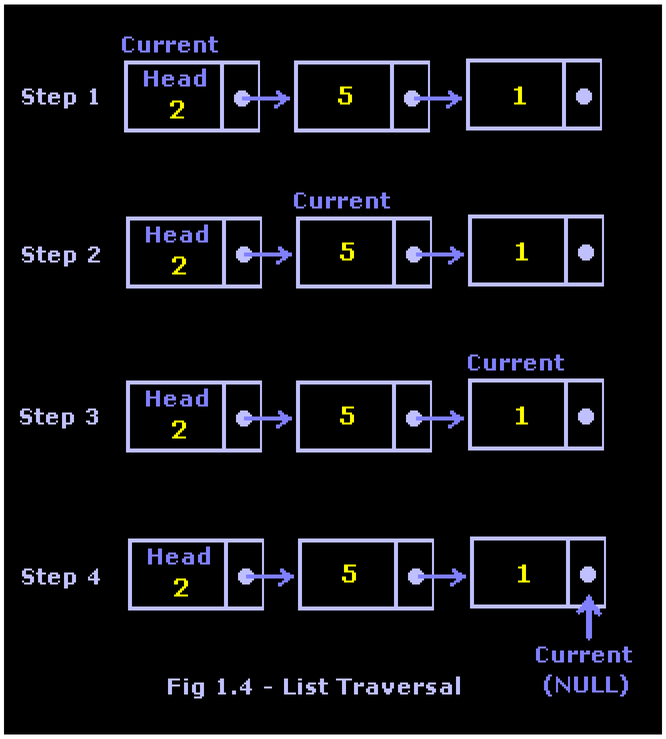

Fig 1.4 - List Traversal

Searching

Searching a linked list is very much like the traversal algorithm just described. The only real difference is that as you go to each node in the list, you compare the data in the node to the "key" data that you are searching for. In this example, I will show you the algorithm using an entire node record as the "key", but keep in mind that this is not the only way to accomplish the search. I leave the other avenues of exploration to you.

First, let's see how to search the linked list visually by looking at Figure 1.5.

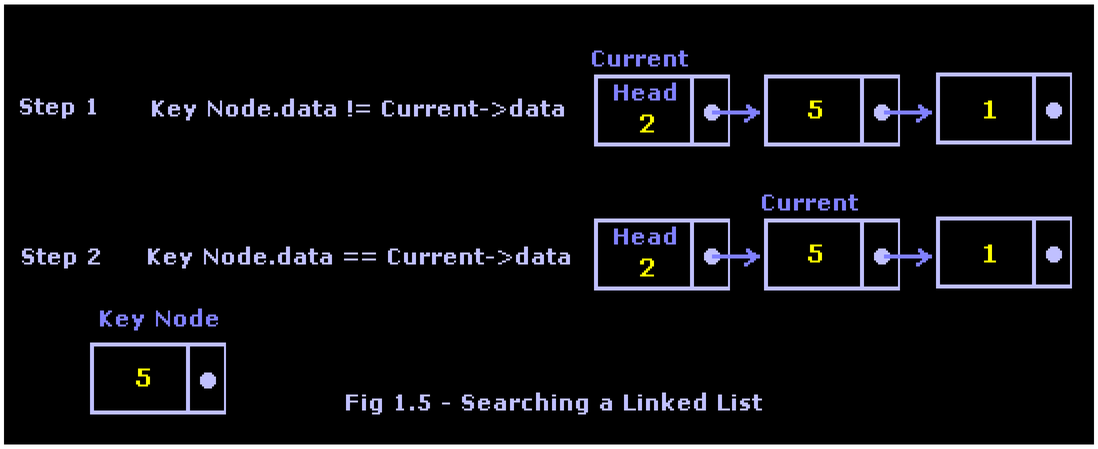
Step 1
Key Node.data != Current->data
Current
Head
2
5
1
Step 2
Key Node.data == Current->data
Current
Head
2
5
1
Key Node
5

Fig 1.5 - Searching a Linked List

As you can see, searching is in fact almost the same as traversing the list. In the code, the only real difference is an extra conditional statement for comparing the key node data with the current node's data that will force the traversal loop to exit should the statement be true. Take a look at the function **Find**, which returns a pointer to a node. If **Find** returns **NULL**, then no matches were found.

Keep in mind that this is a linear search, and that the linear search tends to be slow. This is the only way we can search this particular data structure because of the nature of the links. You can do more advanced searches on linked lists by adding more links and changing the nature of the linked list overall. We will get into that in a later article. In the meantime, here's the **Find** code:

```
node_t *Find(node_t *head, node_t keyRecord)
{
        node_t *current;
        bool found;

        current = head;
        found = false;

        while ((current != NULL) && (!found))
        {
                if (current->id == keyRecord.id)
                        found = true;
                else
                        current = current->next;
        }

        return current;
}
```

Node Deletion

We’re getting close to the end now, as it’s time to talk about node deletion. Node deletion is a little bit different compared to the other linked list functions in that you must think of all the special cases that may come up. If you don’t think of every special case, you might end up with severe memory leaks, crashes, or in the not-so-bad case, just a program that refuses to delete the desired node.

For all of the dynamic data structures we discuss in this series, we will need to go through and determine all the cases that will come up when deleting nodes for the data structure we discuss. So let’s begin by determining the cases involved in deleting nodes from a linked list.

To start off, let’s discuss the simplest case: deletion of a node in the middle of the list. Let’s say we already know what data we want to delete, and we put it in a key node. The first thing we need to do is find the pointer to that node. To accomplish that, we use the **Find** function that we just created and call the found node **current**. The next thing we need to do is find the node that links to the node that was just found, which we will call **previous**. From here, we can now bypass the link to the **current** node by changing the **previous** node’s link to point to the **current** node’s link. This will “skip” the **current** node in the linked list chain, but we have not lost this memory since we still have the **current** pointer. Now that the linked list is intact, we can free the memory used by the **current** pointer. Take a look at Figure 1.6 to see how this is done.

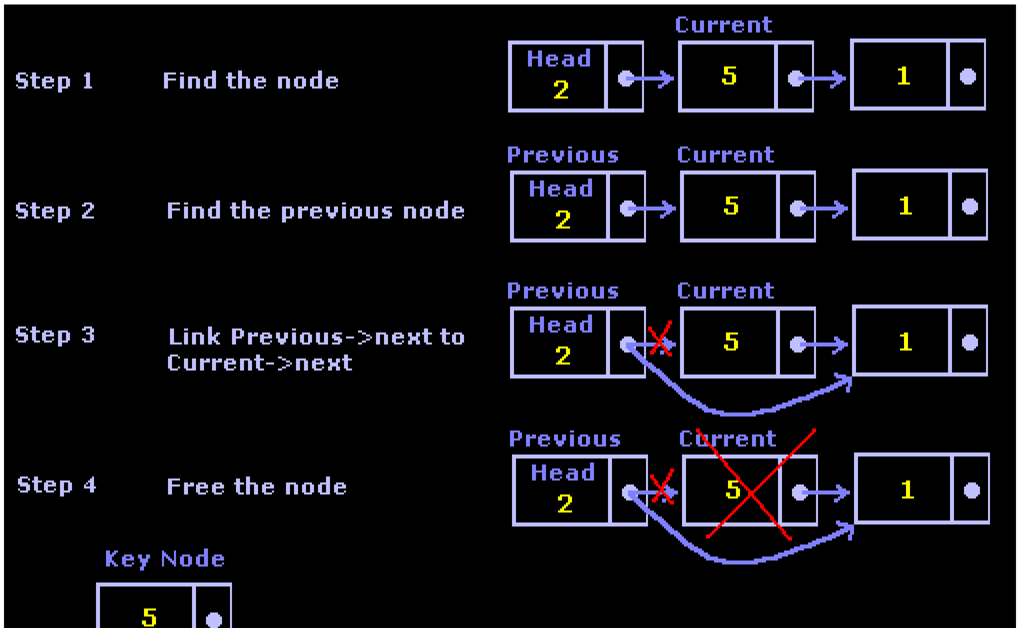

Fig 1.6 - Linked List Deletion

Now we need to think about another potentially hazardous case of node deletion: the desired node is the first node in the list, but not the only node. In a case like this, all we need to do is set **current** to the **head** node, and then move the **head** pointer to the next node in the list. From there, we just free the **current** node pointer. Take a look at Figure 1.7 for a better idea.

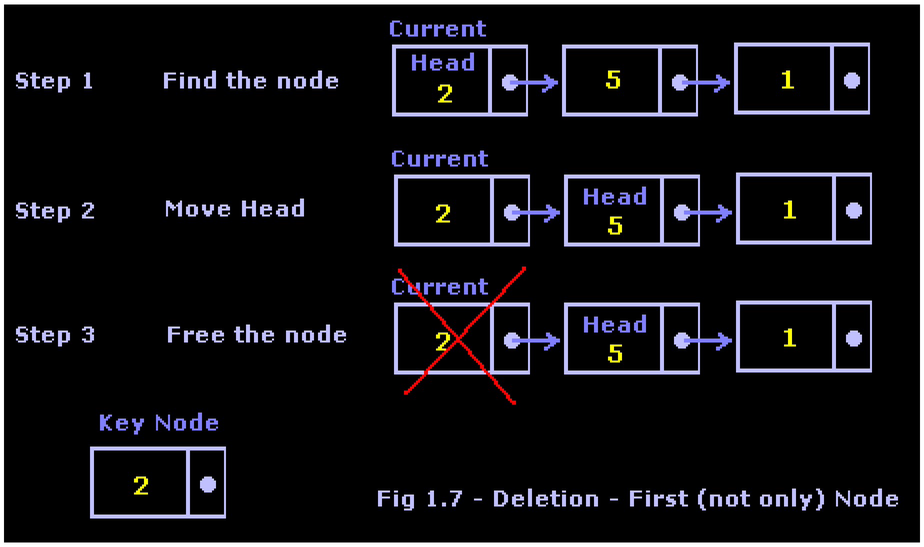

Fig 1.7 - Deletion - First (not only) Node

That's it. There are no more cases for deletion in a singly linked list. You do, however, need to put a verification that the desired node was in fact found, or you could run into some access violation problems.

Here is our delete function:

```
void DeleteNode(node_t **head, node_t keyRecord)
{
        node_t *delNode;                // node to delete
        node_t *previous;               // node before the deleted node

        // find our node to delete
        delNode = Find(*head, keyRecord);

        // if desired record is not in the list, exit the function
        if (delNode == NULL)
        {
                printf("Record not found.\n");
                return;
        }

        if (delNode == *head)
        {
                // first node in the list, but not the only node
                // move the head to the second node in the list
                *head = delNode->next;
                free((void*)delNode);
        }
        else
        {
                // any other case
                previous = *head;

                // search through the list for the node before our deleted node
                while (previous->next != delNode)
                {
                        previous = previous->next;
                }

                // link the previous node to the node after our deleted node
                previous->next = delNode->next;

                if (delNode != NULL)
```

```
                        {
                                free((void*)delNode);// free the memory
                                delNode = NULL;
                        }
                }
        }
```

End Of File

That concludes our “brief” introduction to the singly linked list. This particular data structure forms the basis for all of the data structures that we are going to discuss throughout the rest of the series.

Now that we’ve gone through the important functions involving linked lists, next time we will discuss some of the abstract data types that can be derived and implemented using the base singly linked list. I will also briefly cover some derivations of singly linked lists such as circular linked lists, doubly linked lists, and a few other methods that you might want to experiment with on your own.

Be sure to download the example code to get a good idea of how the linked lists are used. The program is very simple and just performs all of the functions on a list of integers.

If you have any questions, suggestions, or comments, please email me at kevin@gamedev.net.

Special thanks to Anne.