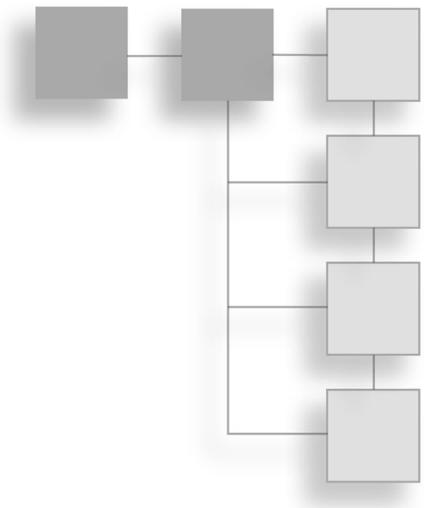
CHAPTER 11



Defining and Using Objects

Big things, little things, round things, and all other sorts of things—the world is full of objects of every size, shape, and description. I'm not just talking about any world here—your gaming world needs to be packed with useful items. Trying to keep track of those useful objects and what they actually do in your role-playing game is a major chore, but with a little knowledge, you can tackle this job blindfolded!

In this chapter, you'll learn how to do the following:

- Define objects in your game
- Create a master list of items
- Use inventory systems to manage items

Defining Objects for Your Game

Frantically I dig through my sack. I know that I put that healing potion in there yesterday; where could it have gone? What a time to lose something—in the midst of a battle, taking hits from all sides, and now that I have a moment, I can't recover my health!

Let's see, there's my dagger, that extra shield, a handful of gold pieces, something I don't recognize, and—oh, there it is—my healing potion! How in the world did I manage to collect all this junk? Oh well, I'll worry about that later; for now, I need to take a gulp of elixir and get back to the job of slaying monsters.

Thankfully my ordeal wasn't really life threatening; I only had to pause the game for a moment, sort my inventory list, and locate the appropriate potion. Fully refreshed and game resumed, I managed to fight on in true warrior spirit!

During the course of a game, you're bound to pick up a few items (also called *objects*), each serving its unique purpose. When creating your game, each object must be accounted for, each designed for a specific use. Weapons, armor, or even healing items, all need definitions. They need form and function.

Form and function—two words to live by when defining an object. *Form* refers to appearance and identity—what an object looks like, what it feels like, how big it is, how much it weighs, and so on. *Function* refers to purpose; every object has a purpose—money buys things, swords aid in an attack, and healing potions heal wounds.

In this section, you learn how to define an object's form and function in a format readily usable in your game project.

Using Form in Objects

Although essential in order for us to visually comprehend an object, form means nothing to a computer, for which an object just needs to be represented by a graphics image or a 3-D model. In Chapter 2, "Drawing with DirectX Graphics," you learned how easy it is to load a bitmap image or an .X file that contains a 3-D mesh, so why not go with using those bitmaps or meshes to define the form of an object?

Assume that you want to create a weapon, or more specifically, a sword. In a 3-D game (for example, the one I describe at the beginning of this chapter), you want players to be able to see the characters holding swords and to examine the swords closely by zooming in. In addition, to show which weapon is equipped (being held), you want to display a bitmap of the sword onscreen. You need only a single mesh and a single bitmap image (see Figure 11.1) to represent the sword.

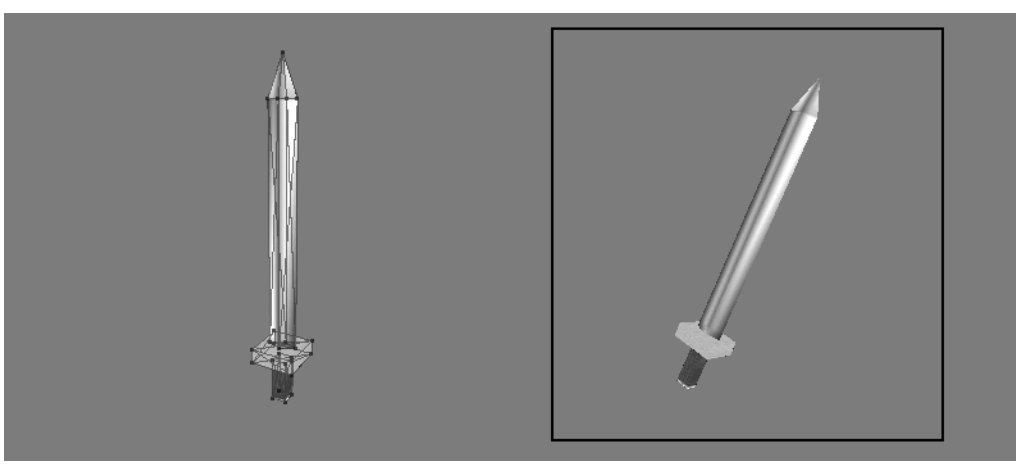


Figure 11.1 You can use the mesh on the left for in-game play (with the player holding it) and in an inventory list (to examine the sword). The bitmap image on the right is displayed during the game to signify that the player is indeed holding a sword.

I know what you're thinking—that's a big, heavy sword in Figure 11.1! In the real world, a sword is "big and heavy." For that matter, you don't want just anyone to be able to wield it. A sword might weigh about five pounds and be about four feet long. If your game is concerned with the physical properties of an object, each item in the game can be assigned dimensions and weight. I opt to measure size in cubic feet.

note

Why would you bother with weight and size in your game? For one thing, do you want a tiny guy or gal wielding a huge sword? What about when the little character tries to put a four-foot sword in a one-cubic-foot backpack? Quite a few games dismiss these problems, but if you want a realistic edge in your game, consider weight and size.

Now you can start entering the item information into your game. Start by creating a structure that will hold the information for the sword (and every other item for that matter):

```
typedef struct sItem
{
  char Name[32];           // A short name for the item
  char Description[128];  // A description of item

  float Weight;  // Weight (in lbs.)
  float Size;    // Size (in cubic feet)

  char MeshFilename[16];   // .X mesh filename
  char ImageFilename[16];  // .BMP filename
} sItem;
```

note

You'll notice the 16-byte size limit of the `MeshFilename` and `ImageFilename` buffers—that's limiting you to use filenames only 16 characters in length (15 plus the `NULL` termination character). If you think you'll need longer filenames, just increase the size of the buffers to something more appropriate.

To quickly define the `Sword` object, declare it as follows:

```
sItem Sword = {
  "Sword", "A big heavy sword.",
  5.0f, 4.0f,
  "Sword.x", "Sword.bmp"
};
```

At this point, the sword is ready for use! Well, not really, because at this point, you've only assigned the physical properties of the sword (along with the filenames to use for the mesh and image). The game engine doesn't have enough information about the item in order to use it. That's where function comes in.

Defining the Functions of Objects

If you were to hand an object to a child (or to anyone for that matter), the child would be able to find some use for it, even if it's not the correct use. Although it might be great to be able to do all kinds of things with objects in a game, that just will not do for your purposes. The items (objects) in your game will be used for specific purposes, which can be in one of the following categories (note that the list is not comprehensive but is a starting place for ideas):

- **Accessory.** Rings, necklaces, belts, and any other form of wearable equipment that can aid an adventurer are classified as accessories. Sometimes an accessory is a magical piece of equipment that helps boost the abilities of the wearer.
- **Armor.** Taking a blow is nothing—if you're wearing the proper protection, that is. Armor can be any form of protective gear, from a full suit of armor to a pair of boots.
- **Collection.** Any item that doesn't serve an actual purpose is considered a collection piece. These can be items that are important to the game story in some way or that really have no definitive use. Items such as a picture, small ornamental figurine, or chair are considered collection items.
- **Edible.** Clam cakes, clam pies, clam à la mode, creamed clams, or BBQ clams—whatever your fancy. Your gaming denizens are bound to get hungry, and whatever they can find and consume is classified as edible, even potions and herbs.
- **Money.** Everybody deals with the big M, no matter what form it takes—coins, bills, clam shells, and so on. Games do not require denominations—one gold coin is as good as the next, and the more coins you have, the better.
- **Transportation.** Whether by bus, boat, plane, or hang glider, people sure know how to get around, and items that can transport belong to characters in the game. A boat can't be lugged around, but your game recognizes a character as the item's owner by noting so in the character's inventory.
- **Weapon.** Whether it's a sword, rock, or piece of lint, some things are just more suitable for dealing out damage.
- **Other.** Anything that doesn't fit into the preceding categories is considered "other."

Your game engine determines what each item does, based on each item's category. For example, weapons can be equipped (worn), whereas edible items can be consumed. You

can add subcategories to each as needed to make dealing with items even easier. For example, a healing potion, although edible, can be categorized as a healing item. When your game engine sees a healing item, it knows immediately to restore some level of health to the character.

Each item has additional information for usage, however, so you can't stop here. Weapons have a strength attribute, which increases a character's ability to deal out damage, a healing potion restores heath, and a piece of armor reduces damage to a character during attacks. Now, take a look at what each category of items accomplishes.

Weapons

Characters have the potential to deal out damage with their bare hands. The more powerful the character, the more damage he can administer. Put a weapon in that character's hands and the amount of damage meted out increases. Some weapons do more damage than others, so finding the perfect weapon is more than enough reason to keep adventuring.

Weapons can also have more than one use. In the mythical sense, an enchanted sword can slash through demon hides and, at the same time, cast a potent fireball spell. I refer to such extra functions of an item as *specials*.

Not just everybody can wield every weapon, however; there are some restrictions. Size, weight, power, and value are some restrictions you'll have to consider. How much fun would it be to hand a beginner the most powerful weapon in the game? For that reason, you introduce usage restrictions (you find out more on usage restrictions in the section "Usage Restrictions," later in this chapter).

Getting back on track, some weapons can do more damage than others. The amount of damage a weapon can cause is measured as a number, called an *attack modifier*. The higher the number, the more damage the weapon does. Also, some weapons are easier to wield, so you can hit your targets more often. To measure just how much easier a weapon can hit its target, you use a *to-hit modifier*. The higher a weapon's to-hit modifier, the better chance a character has to hit a target using that weapon. You'll read more about using modifiers and how those modifiers relate to characters in Chapter 12, "Controlling Players and Characters."

note

A *modifier* is something that changes a character's attributes in some way. A *damage modifier*, for example, increases or decreases the amount of damage they do.

Some weapons can also be classified in special groups, called *weapons groups*. Certain weapons tend to do more damage to certain creatures than other weapons—for example, using a fire-enchanted sword against an ice-based monster.

Finally, weapons can be categorized into subgroups such as *hand-to-hand* or *ranged.* Regardless of the type of weapon, each one has a *range of use.* Swords can hit the targets in front of them, while a crossbow may hit a target 40 feet away. In addition, weapons may be able to hit more than one target at once. These are all things to take into consideration when designing weapons.

Armor

The more protection, the better, and in your game every little piece of armor helps. Armor helps add resistance to damage. This resistance amount is called the *defense modifier.* Just like weapons, armor has special uses and usage restrictions and belongs to *armor groups.*

Armor can be split into multiple subcategories, such as helmets, chest and abdomen protection, leggings, boots, gloves, and so on.

Accessories

As mentioned earlier, accessories usually have a specific use. A magic ring can be worn to gain the ability to become invisible. This ability might always be in use once the ring is donned, or it may have to be activated. Accessories can also act like armor; they can increase the resistance to some aspect of the game—for example, making it harder for the wearer to be poisoned.

Edibles

Edible items usually come in a few flavors (pun intended). Food items sustain life, healing items increase health, and poisonous items decrease health. Again, special uses are in effect, but because there are few uses for edible items, you can hardcode those into your game engine.

Collections

A collectable object is usually docile; it's needed only for some small aspect of the game. For example, if a character gives you a picture of himself to deliver to a girl in a neighboring town (and for no other purpose), the picture is considered a collection item. Collection items simply move some part of the game story forward. Perhaps the character delivering the picture will receive, in turn, a special item from the girl in the next town.

Transportation

Getting around on foot is slow and tiring, so other forms of transportation might be needed. The purpose of transportation items is to change the way the characters move around (typically around the map). Transportation can also open up new areas in the game that were previously not accessible. For instance, your character's newly acquired boat can now be used to sail across the lake to an isolated island, or maybe that horse you saw in a nearby town will help you cross a barren desert safely.

Others

"Other" items are pretty much useless because they don't have a defined use. However, don't throw them out. At the least, they can perform some type of action as defined by the engine. For instance, depending on how well you do in combat, your character could be awarded medals. Although these medals are cool to look at, they serve no purpose in the game.

Adding Function to Objects

In the earlier section "Defining the Functions of Objects," you saw how much is needed to define the function of an object. Luckily, because each object is categorized as one thing or another, not all the information is needed—swords do damage, whereas armor prevents it—so there's no need to mix damage and protection data.

Item Categories and Values

In reality, you'll want to categorize each item specifically to fit your game engine, just as I did in the section "Defining the Functions of Objects." Each category of item is numbered for reference (1 is a weapon, 2 is armor, and so on). Each category has a value associated with it, one that determines the modifier (attack or defend), the special use, the healing or damaging value, and an attached script. That's right. Items can use scripting to increase their capabilities.

Except for an attached script, you can represent all values with a single variable, one that represents the modifier amount, healing value, and so on. At this point, the following two variables can be added to the `sItem` structure previously created:

```
// ... previous sItem info

long Category;  // 1-5 representing item category shown above
long Value;     // Modifier, health increase, etc.

// .. More sItem info
```

tip

You can use an enumerator value to represent the categories in the `sItem` structure:

```
enum ItemCategories {
  WEAPON=0,
  ARMOR,
  SHIELD,
  HEALING,
  OTHER
};
```

Assigning Value to Items

Everything in the game is of value. Assigning each item a monetary value helps to determine what a player can buy or sell and at what price. You don't want to clutter each item with multiple amounts; just pick a single amount that the character should pay to buy the item. When being sold, the same item will have an amount lower than what your character would pay to buy it. For instance, an item can be sold for half the price your character paid for it.

An item's value can be inserted in the `sItem` structure as follows:

```
// ... previous sItem info

long Price; // Buying price of item
```

Item Flags

Sometimes, you will not want the character to be able to sell an item—important magic items, for example. A bit flag will take care of this, and while you're at it, add a few more bit flags. Table 11.1 contains a list of possible flags you can use.

Each bit flag is contained within a variable in the `sItem` structure:

```
long Flags; // Item bit flags
// ... more sItem data
```

Table 11.1 Item Bit Flags

Flag	Description
SELLABLE	These items can be sold in shops.
CANDROP	These items can be dropped. Don't use this flag with important items if you don't want characters to drop those items.
USEONCE	These items can be used only once. Once used, they disappear.
UNKNOWN	These items are unknown. You must identify them in order to use them correctly.

You can find examples of these in Sega's online masterpiece game, *Phantasy Star Online*. You can then represent each flag as an `enum` value (with a maximum of 32 flags). To set, clear, or check a flag, use the following macros (using the macros, `v` represents the item flag variable, and `f` is the flag):

```
enum {
  SELLABLE = 0, // Bit 0
  CANDROP,      // Bit 1
  USEONCE,      // Bit 2
```

```
  UNKNOWN       // Bit 3
};

#define SetItemFlag(v,f)   (v |= (1 << f))
#define ClearItemFlag(v,f) (v &= ~(1 << f))
#define CheckItemFlag(v,f) (v & (1 << f))

// Example using macros and flags
long ItemFlags = 0;

// Set item flags to sellable and item can be dropped
SetItemFlag(ItemFlags, SELLABLE);
SetItemFlag(ItemFlags, CANDROP);

// Check if the item is droppable and display a message
if(CheckItemFlag(ItemFlags, CANDROP))
  MessageBox(NULL, "Can Drop Item", "Item", MB_OK);

ClearItemFlag(ItemFlags, SELLABLE); // Clear sellable flag
```

Usage Restrictions

Certain characters in your game might not be able to use a specific item. A magic user, for example, can't wield a huge two-handed battle-axe, and a barbarian can't wield a wizard's staff. In such cases, where only certain characters are allowed to use certain items, you need to restrict usage to specific character classes.

note

A *character class* is a classification or grouping of characters based on their race or profession. For example, all humans belong to the same class, but to be more specific, human fighters are considered a separate class from human wizards (or just fighters and wizards—who says they all have to be human?).

To represent the usage restrictions of an item, another variable is introduced to the `sItem` structure, one that tracks 32 bits of information. Each bit represents a single class, which means that you can track up to 32 classes. If an item is usable by a certain class, that respective bit is set; if an item is restricted in use by the character's class, the appropriate bit is cleared.

Here's the addition to the `sItem` structure, which handles usage restrictions:

```
long Usage; // Usage restrictions

// ... other sItem data
```

To make setting, clearing, and retrieving a usage restriction class bit easier, you can use the following macros (`v` represents the flag variable, and `c` is the class number ranging from 0 to 31):

```
#define SetUsageBit(v,c)   (v |= (1 << c))
#define ClearUsageBit(v,c) (v &= ((~(1 << c))
#define CheckUsageBit(v,c) (v & (1 << c))

// Examples using macros
long Flags = 0;
SetUsageBit(Flags, 5); // Set class 5 bit

if(CheckUsageBit(Flags, 5)) // Check class 5 bit
  MessageBox(NULL, "Usage Set", "Bit", MB_OK);

ClearUsageBit(Flags, 5); // Clear class 5 bit
```

Using the preceding macros (`SetUsageBit`, `ClearUsageBit`, and `CheckUsageBit`), you can quickly check whether a character is allowed to use or equip the item based on his character class. For example, this game places wizards in `class 1` and fighters in `class 2`. When the wizard tries to equip a broadsword (one that has the `class 1` bit clear), the game engine informs the player that the wizard cannot use the item.

Attaching Scripts to Items

To make items more versatile, you can attach scripts to items. A script is triggered any time an item is used, whether it is a potion being consumed, a sword being used in a round of combat, or a user activating the special usage of an item (by using a magic wand, for example).

tip

When using scripts, it's good form to use a specialized action template better suited for items. Refer to Chapter 10, "Implementing Scripts," for more information on creating action templates and using the script editor.

At this point, you need to store only the script's filename in the `sItem` structure:

```
// .. previous sItem data

char ScriptFilename[16];  // .mls script filename
```

Meshes and Images

You want your game's players to see what an item looks like, which means that you need to load a 2-D image or a 3-D mesh to represent the object. You achieve this using the following additions to the `sItem` structure:

```
  // .. previous sItem data
  char MeshFilename[16];    // .X mesh filename
  char ImageFilename[16];   // .bmp image filename
} sItem; // Close structure
```

The Final Item Structure

At this point, the `sItem` structure is ready for use! Here it is again in its entirety (including supporting macros):

```
enum ItemCategories { WEAPON=0,ARMOR,SHIELD,HEALING,OTHER };

#define SetUsageBit(v,c)   (v |= (1 << c))
#define ClearUsageBit(v,c) (v &= ((~(1 << c))
#define CheckUsageBit(v,c) (v & (1 << c))

enum {
  SELLABLE = 0, // Bit 0
  CANDROP,      // Bit 1
  USEONCE,      // Bit 2
  UNKNOWN       // Bit 3
};

#define SetItemFlag(v,f)   (v |= (1 << f))
#define ClearItemFlag(v,f) (v &= ~(1 << f))
#define CheckItemFlag(v,f) (v & (1 << f))

typedef struct sItem
{
  char  Name[32];  // A short name for the item
  char  Description[128]; // A description of item
  float Weight;    // Weight (in lbs.)
  float Size;      // Size (in cubic feet)
  long  Category;  // Category of item
  long  Value;     // Modifier, health increase, etc.
  long  Price;     // Buying price of item
  long  Flags;     // Item bit flags
  long  Usage;     // Usage restrictions

  char  ScriptFilename[16];  // .mls script filename
  char  MeshFilename[16];    // .X mesh filename
  char  ImageFilename[16];   // .bmp image filename
} sItem;
```

note

Recall from earlier that the `ScriptFilename`, `MeshFilename`, and `ImageFilename` buffers are each limited to 16 bytes, which means that the `sItem` structure will store only the actual filename and not the path to the appropriate file. Also, although the `sItem` structure stores the filenames, you are responsible for loading the files to use in the game.

If you're going to store actual paths or use longer filenames for your scripts, meshes, or images, go ahead and change the buffer sizes appropriately.

With the complete `sItem` structure in place, it's time to get back to building the sword item. Say that the sword item uses a +10 modifier on damage (which means that you add 10 to the damage factor in combat). The sword normally sells for 200 monetary units in the game, and only fighter classes (class two) can use it:

```
// Character class definitions
#define WIZARD 1
#define WARRIOR 2

sItem Sword = {
  "Sword", "A big heavy sword",    // name and description
  5.0f, 4.0f, // weight and size
  WEAPON, 200, SELLABLE | CANDROP, // category, price, and flags
  (1 << WARRIOR), // usage class 2 (warrior)
  "", "Sword.x", "Sword.bmp"       // Script, mesh, image files
};
```

Now that the sword item is defined, you can use it in the game. But what good is a single item? Your game world is going to be packed with items! How can you possibly deal with all those objects?

The Master Item List

Every item in your game needs to be defined, and to keep things tidy, you need to keep all item descriptions in a *master item list* (*MIL*). Think of the MIL as a catalog of objects, much like the one shown in Figure 11.2. Each object is numbered for reference, and only one of each item is shown.

Any time you need a new object or need to retrieve the attributes of a specific object, you consult the MIL. At a basic level, your game's MIL can be stored as an array of `sItem` structures or a single sequential file composed from a list of item structures (similar to the one shown in Figure 11.3). How can you go about creating your own MIL? Well, let's take a closer look.

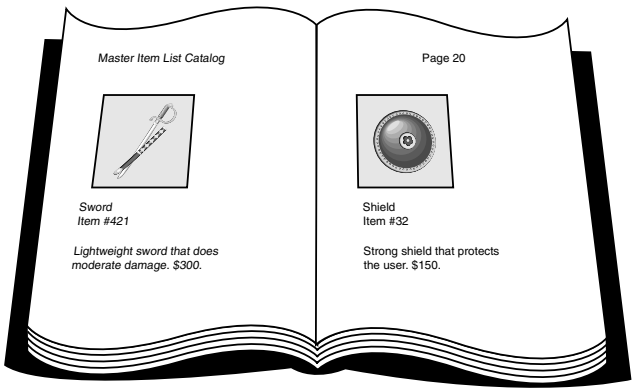


Figure 11.2 Much like a department store catalog, a master item list helps you keep every object in the world in order.

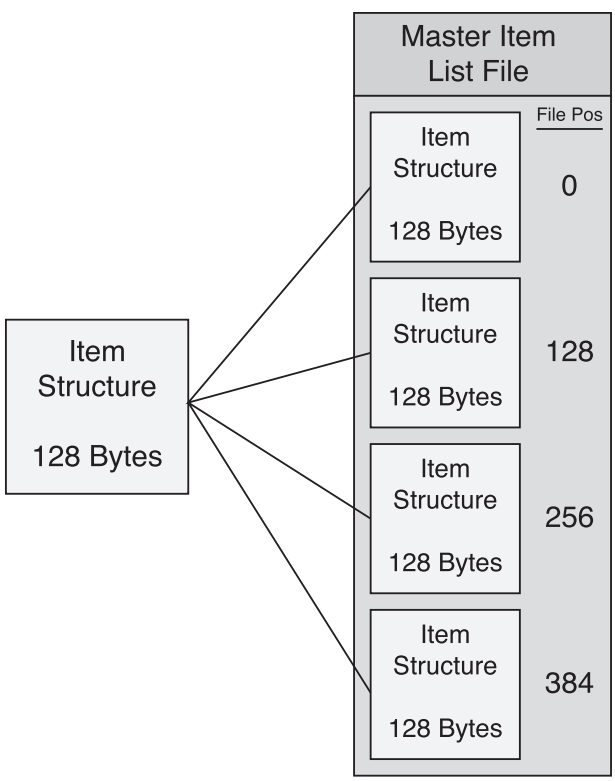


Figure 11.3 A sequential MIL file keeps track of every item in your game. Each item's data is fixed in size, ensuring that you can quickly access the file when you need to retrieve data.

Constructing the MIL

The following code bit creates a small item structure that contains the item's name, weight, and size. This is the same `sItem` structure you've come to love up to this point in the chapter—you're just going to redefine it to store information in the master list of items. You will use this structure to construct a simple MIL:

```
typedef struct sItem
{
  char Name[32];  // Name of item
  float Weight;   // Weight (in lbs.)
  float Size;     // Size (in cubic ft.)
};
```

From here, say that you want to store five items in the MIL, all represented in an array of `sItem` structures:

```
sItem Items[5] = {
  { "Big Sword",   5.0f, 4.0f },
  { "Small Sword", 2.0f, 2.0f },
  { "Magic Wand",  0.5f, 1.0f },
  { "Rock",         1.0f, 0.5f },
  { "Potion",      0.5f, 0.5f }
};
```

Now that you have defined your MIL (using an array of `sItem` structures), you may want to save the list out to a file for later retrieval. Such is the case if you are using a separate program that creates the MIL file for you, much like the program you'll see in the upcoming section "Using the MIL Editor." As for here, take a look at the following bit of code that will create a file (called `items.mil`) and save the `Items` array to the file:

```
FILE *fp=fopen("items.mil", "wb");

for(short i=0;i<5;i++)
  fwrite(&Items[i], 1, sizeof(sItem), fp);
fclose(fp);
```

Although short and to the point, the preceding example for creating a MIL file is wholly unusable in a real-world application such as a role-playing game. Item descriptions need to contain much more information, and you could theoretically work with thousands of items. Doing all that by hand is a waste of time. What you need is an item editor to help you create and maintain the MIL—and, so, behold the *MIL Editor*.

Using the MIL Editor

The need for quick and easy item creation gave birth to the Master Item List Editor (MIL Editor). Much like the MLS Editor discussed in Chapter 10, "Implementing Scripts," the MIL Editor consists of a single application window that enables you to navigate through a list of items, editing the attributes of each item as you go. You can save and load MILs, but the list of item attributes remains fixed (until you reprogram them for your own purposes).

The complete source code to the MIL Editor is included on the CD-ROM that comes with this book (look for \BookCode\Chap11\MILEdit). When you start the MIL Editor found on the CD, the Master Item List Editor dialog box appears as shown in Figure 11.4. The Master Item List Editor dialog box consists of a list box that contains each item in the list, plus buttons to edit each item's information and to save and load an item list. The list provides room for 1,024 items, which means that you can store an item number within a 16-bit variable (ranging from 0 to 1,023).

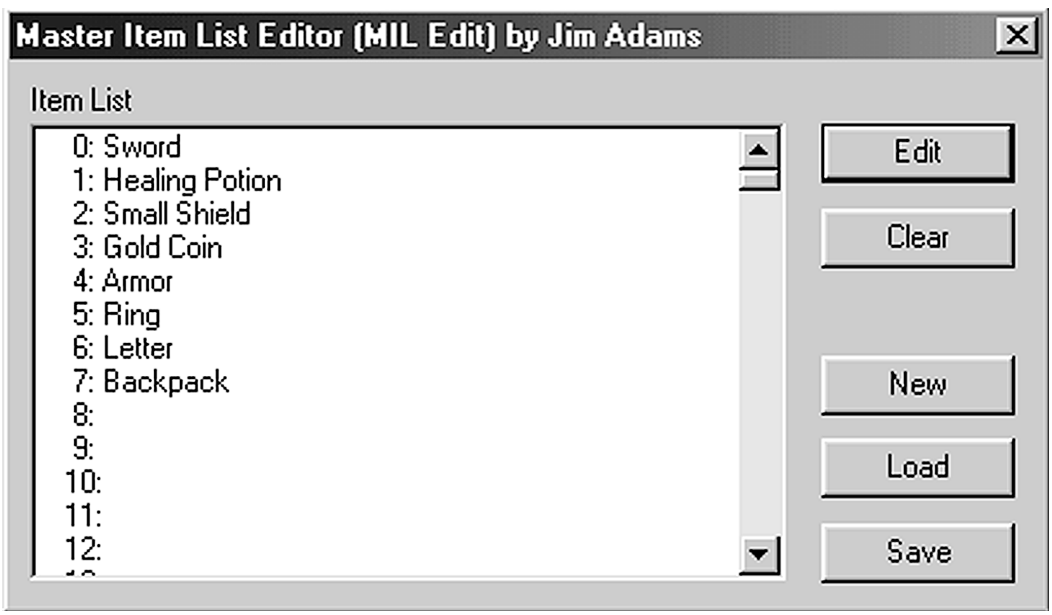


Figure 11.4 The Master Item List Editor is used to create and edit your game's objects. The list of items shown here has eight items defined.

To begin using the Master Item List Editor, locate and execute the MILEdit.exe file (look in \BookCode\Chap11\MILEdit). At the Master Item List Editor dialog box, you can perform the following steps to add or edit items and then save them to disk:

1. Select an item from the Item List by double-clicking the item (or selecting the item and clicking the Edit button), or add an item by clicking the Add button. The Modify Item dialog box appears (shown in Figure 11.5).

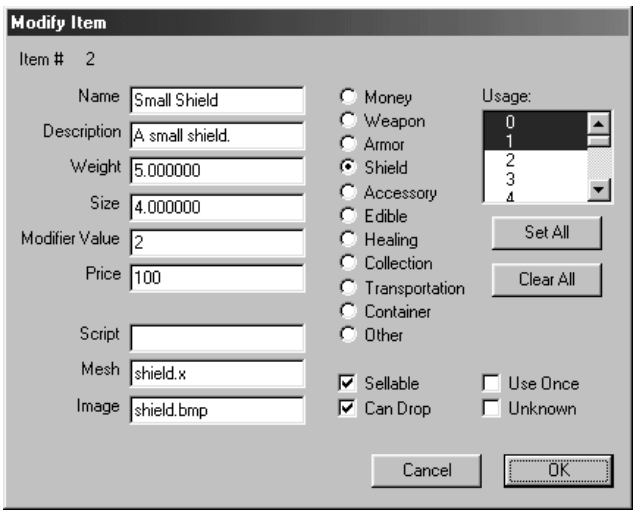


Figure 11.5 The Modify Item dialog box enables you to modify each item's vital information.

2. Edit the appropriate fields in the Modify Item dialog box.
3. When you finish editing, click OK to apply the changes and return to the Master Item List Editor dialog.
4. To make the changes permanent, click Save on the Master Item List Editor dialog box, and in the Save MIL File dialog box, enter a filename and save the MIL to disk.

note

Loading a MIL is just as easy as saving one; click the Load button on the Master Item List Editor dialog box. When the Load MIL File dialog box appears, select the file you want to load.

MIL files typically use a .MIL file extension. You can use a different extension for your lists, but the sample programs for this book use .MIL.

The MIL Editor uses the same version of the `sItem` structure shown earlier in this chapter, but I added some extra item categories. Those extra categories are *Shield*, *Healing*, and *Container* (a container object such as a backpack). Here these extra categories are added to the `ItemCategories enum` list, shown previously in the section "Item Categories and Values":

```
enum ItemCategories
{
```

```
  MONEY = 0,  WEAPON,       ARMOR,
  SHIELD,     ACCESSORY,    EDIBLE,
  HEALING,    COLLECTION,   TRANSPORTATION,
  CONTAINER,  OTHER
};
```

If you decide to modify the MIL Editor to use different item attributes or categories, modify the `sItem` structure as well. When you're ready, you can start using the item data you created in your game project.

Accessing Items from the MIL

Once you have a MIL, you can load the entire list into an array of `sItem` structures using the following code:

```
sItem Items[1024]; // Array of sItem structures

// Open the Default.mil file
FILE *fp = fopen("Default.mil", "rb");

// Read in all items
for(short i=0;i<1024;i++)
  fread(&Items[i], 1, sizeof(sItem), fp);

// Close file
fclose(fp);
```

At this point, I'm assuming that your item structure is relatively small and that you are using no more than 1,024 items in your MIL. What happens if you extend each item's `sItem` structure or you begin storing more items in the MIL? We're talking about some serious memory usage.

To avoid loading each and every single item in memory from the MIL at once, you can access individual items directly from the MIL. Because the size of each item structure is fixed, you can access each item's data by moving the file pointer to the appropriate position and reading in the structure, as in the following code bit:

```
// ItemNum = reference # of item to load
sItem Item;

// Open the MIL file titled items.mil
FILE *fp=fopen("items.mil", "rb");

// Seek to the appropriate position in file
// based on the size of the sItem structure and
```

```
// the number of the item to load.
fseek(fp, sizeof(sItem) * ItemNum, SEEK_SET);

// Read in the item structure
fread(&Item, 1, sizeof(sItem), fp);

// Close the file
fclose(fp);
```

And there you go—quick and easy access to every item in the MIL! Now, it's time to do something with those items.

Managing Items with Inventory Control Systems

With your items ready to be scattered around the world, it's only a matter of time before players start trying to pick up everything that isn't nailed down. When that happens, the players will need a way of managing the items, which includes using an *Inventory Control System* (*ICS*) to sort things out.

Don't be fooled; an ICS doesn't just apply to player characters. It applies to the entire world. Items can belong to a map, a character, or even to a different item (a backpack with other items inside it, for example). That means items need to be assigned *ownership*. In addition, an owner can have *multiple instances* of an object—coins, for example.

An owner's collection of items is called an *inventory list*, and any object can belong within this list (and many instances of the object as well). The relationships between owners, inventory lists, and quantities are illustrated in Figure 11.6. (*Note:* Don't think of items as belonging to an owner; rather, think of owners as having a collection of items.)

The Inventory Control System works hand in hand with the MIL. Whereas the MIL contains only a single, unique instance of each object in the world, the ICS works with many instances of any object. Any time the Inventory Control System needs information about an item, it can refer to the Master Item List for the specifics. In that way, you can conserve memory by using only the ICS to store reference numbers to objects in the MIL (as illustrated in Figure 11.7).

For your game's maps and levels, a simple Inventory Control System (called a *map ICS*) consists of only a list of items and their locations within the map, which is just fine because you can place objects throughout—ready for characters to pick them up. The real problem comes when those characters pick them up and add them to their inventory. Multiple instances pile up, new items are added, and other items are used or dropped. Things quickly become a real jumble. Handling a collection of character's objects is the job of a character Inventory Control System, which is a little more complicated than its map counterpart.

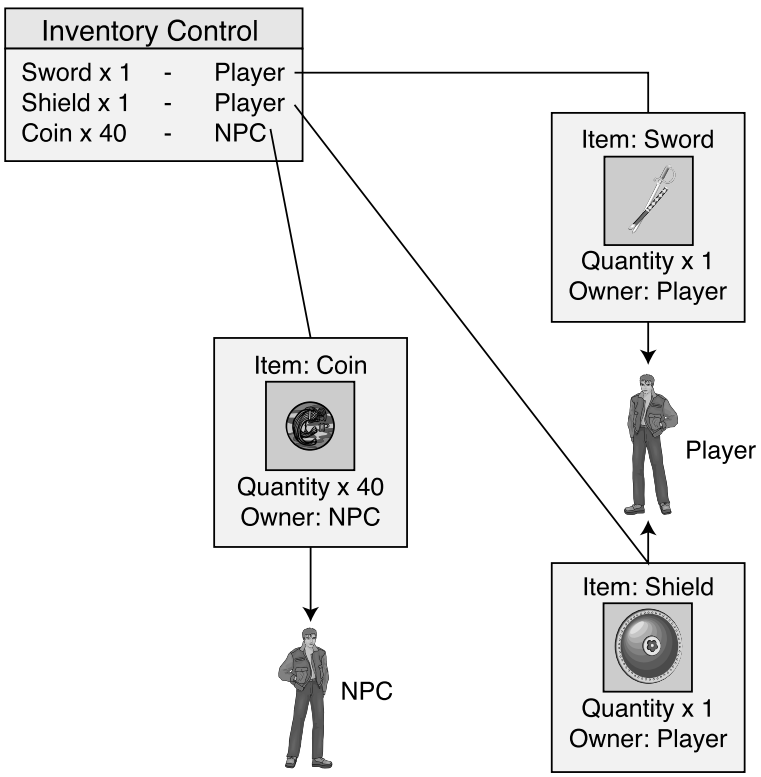


Figure 11.6 An inventory control system keeps track of each owner's items, including the number of items.

Developing a Map ICS

The map ICS tracks items that are placed within levels, including items that are contained within other items—a sword contained within a treasure chest, for example. The type of map you use determines how you position items within the map. In 3-D maps, you use three coordinates for positioning an item—the x-, y-, and z-coordinates. Also, if you divide your world up into multiple maps, you can track each of those maps' items by using an item file for each map.

You can represent the map ICS with a structure and a class:

```
typedef struct sMapItem
{
  long  ItemNum;           // MIL item number
  long  Quantity;          // Quantity of item (i.e. coins)
  float XPos, YPos, ZPos; // Map coordinates

  sMapItem *Prev, *Next;  // linked list pointers
```

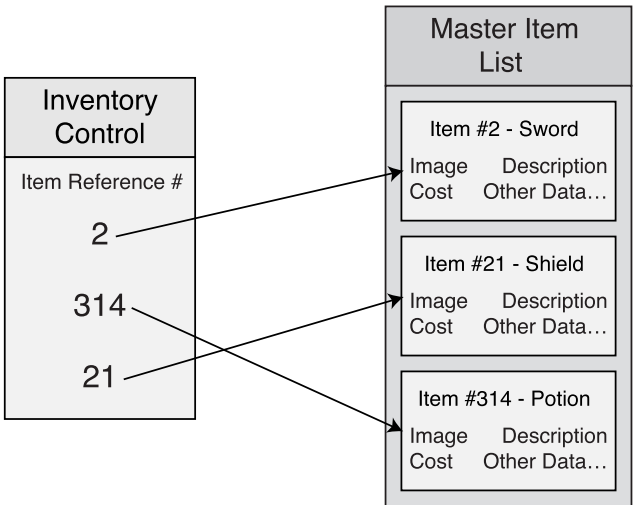


Figure 11.7 Assuming that the Inventory Control System contains only item references, you can save massive amounts of memory by letting the Master Item List store the rest of the item's information.

```
  long Index;              // This item's index #
  long Owner;              // Owner index #
  sMapItem *Parent;        // Parent of a contained item

  sMapItem()
  {
    Prev = Next = Parent = NULL;
    Index = 0; Owner = -1;
  }

  ~sMapItem() { delete Next; Next = NULL; }
} sMapItem;

class cMapICS
{
  private:
    long       m_NumItems;     // # items in map
    sMapItem *m_ItemParent;  // Linked list parent map item

    // Functions to read in next long or float # in file
    long GetNextLong(FILE *fp);
    float GetNextFloat(FILE *fp);
```

```
  public:
    cMapICS();   // Constructor
    ~cMapICS();  // Destructor

    // Load, save, and free a list of map items
    BOOL Load(char *Filename);
    BOOL Save(char *Filename);
    BOOL Free();

    // Add and remove an item on map
    BOOL Add(long ItemNum, long Quantity,                    \
             float XPos, float YPos, float ZPos,             \
             sMapItem *OwnerItem = NULL);
    BOOL Remove(sMapItem *Item);

    // Retrieve # items or parent linked list object
    long GetNumItems();
    sMapItem *GetParentItem();
    sMapItem *GetItem(long Num);
};
```

First, you see the `sMapItem` structure, which holds the information for every item in the map. `ItemNum` is the MIL item reference number (which ranges from 0 to 1,023 if you used the MIL Editor program), and `Quantity` is the number of `ItemNums` (for example, to allow a horde of coins to be represented as a single object). Then you see the item's map coordinates `XPos`, `YPos`, and `ZPos`.

Next comes the `Prev` and `Next` pointers. You insert them to track a linked list of `sMapItem` structures. The next couple of variables, `Index` and `Owner`, are used when loading and saving the items in a map. `Index` stores the current index number of an item in the linked list. If an item is owned by another item, the `Owner` variable holds the index number of the parent object (otherwise, `Owner` is set to -1).

When loading (or adding) an object, you set the final variable in `sMapItem (Parent)` to point to the actual owner item's structure. You can see the `sMapItem` structure link list concept illustrated in Figure 11.8.

The `sMapItem` uses both a constructor and destructor function called whenever a structure instance is allocated or re-allocated.

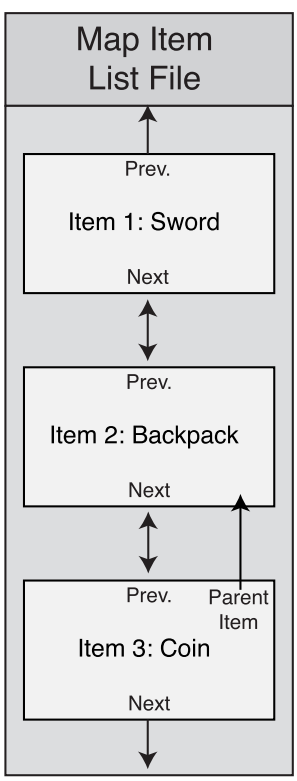


Figure 11.8 You store the map's list of items as a linked list. Items are numbered as they are read in from disk, which helps to match child-to-parent objects.

Both functions ensure that the linked list pointers are in check, and whenever a structure is deleted, all subsequent `sMapItem` structures in the linked list are deleted as well.

caution

If you're removing only a single instance of `sMapItem` from the linked list, you first have to set the instance's `Next` variable to `NULL`. Doing so ensures that all subsequent instances in the linked list are not deleted as well.

The `cMapICS` class has two private functions (`GetNextLong` and `GetNextFloat`) used to read in text and convert it into a `long` or `float` value. The `cMapICS` class also has eight usable public functions. Take a closer look at those public functions.

cMapICS::Load, cMapICS::Save, and cMapICS::Free

As their names suggest, this trio of functions loads, saves, and frees a list of items that belong to a map. The first of the three, `Load`, loads and creates a list of items. For simplicity, store all items in a text file, using the following format:

```
MIL_ItemNum
Quantity
XPos
YPos
ZPos
ParentID
```

Each item uses six lines of text, and each entry (group of six lines) is numbered sequentially (the first item in the file is item #0, the second item is #1, and so on). Here's a sample file that contains two items:

```
// Item #0 as follows:
10                // MIL Item # (long value)
1                 // Quantity (long value)
10.0              // XPos (float value)
0.0               // YPos (float value)
600.0             // ZPos (float value)
-1                // Owner (-1 = none, index # otherwise)
// Item #1 as follows:
1                 // MIL Item #
1                 // ...
10.0
0.0
600.0
0                 // belongs to item #0 (first item in file)
```

The preceding comments are for clarification; the actual storage file does not use them. When reading in a list of items such as the preceding ones, the `Load` function converts the text into usable numbers. Using those numbers, it creates a `sMapItem` structure for each item in the map to be loaded, constructing a linked list as the items are loaded. After every item is read in, all objects that belong to another are matched up (using the `Parent` pointer in the `sMapItem` structure).

There's really nothing too difficult here, so jump right into the `cMapICS::Load` code:

```
BOOL cMapICS::Load(char *Filename)
{
  FILE *fp;
  long LongNum;
  sMapItem *Item, *ItemPtr = NULL;

  Free(); // Free a prior set

  // Open the file
  if((fp=fopen(Filename, "rb"))==NULL)
    return FALSE;
```

Once the file has been opened (as shown in that last bit of code), you can then begin loading items. The first bit of item data you need to load is the item number. Once the item number is loaded, you then create a new map item structure and link it into the list of map items. Also, all items that belong to other items (such as a potion belonging to a backpack, since the potion is in the backpack, are linked). This process continues until all items are loaded.

```
  // Loop forever reading in items
  while(1) {
    // Get next item number (break if no more items,
    // which is represented by a return value of -1).
    if((LongNum = GetNextLong(fp)) == -1)
      break;

    // Create a new map pointer and link it in
    Item = new sMapItem();
    if(ItemPtr == NULL)
      m_ItemParent = Item;
    else {
      Item->Prev = ItemPtr;
```

```
      ItemPtr->Next = Item;
    }
    ItemPtr = Item;

    // Store MIL item number
    Item->ItemNum = LongNum;

    // Get quantity
    Item->Quantity = GetNextLong(fp);

    // Get coordinates
    Item->XPos = GetNextFloat(fp);
    Item->YPos = GetNextFloat(fp);
    Item->ZPos = GetNextFloat(fp);

    // Get owner #
    Item->Owner = GetNextLong(fp);

    // Save index # and increase count
    Item->Index = m_NumItems++;
  }

  // Close the file
  fclose(fp);

  // Match objects that belong to others
  ItemPtr = m_ItemParent;
  while(ItemPtr != NULL) {
    // Check if this item belongs to another
    if(ItemPtr->Owner != -1) {
      // Find matching parent item
      Item = m_ItemParent;
      while(Item != NULL) {
        if(ItemPtr->Owner == Item->Index) {
          // A match, point to parent
          ItemPtr->Parent = Item;
          break; // Stop scanning for parents
        }
        Item = Item->Next;
      }
    }
```

```
    // Go to next item
    ItemPtr = ItemPtr->Next;
  }

  return TRUE;
}
```

note

Much like all the code in this book, the `cMapICS` class functions return a value of `TRUE` if the function call succeeded or a value of `FALSE` if the call failed.

`Save` takes an internal list of items and, using the filename you specify, saves that list to a file on disk. The `Save` function is typically used to update the game data, because players might consistently pick up and drop items.

The `Save` function first assigns an index value to each `sMapItem` structure in the linked list (based on their order). The first item in the linked list is `0` (zero), the second item is `1`, and so on. Each child item's `Owner` variable is updated as well at this point, and finally all data is written to a file:

```
BOOL cMapICS::Save(char *Filename)
{
  FILE *fp;
  sMapItem *Item;
  long Index = 0;

  // Open the file
  if((fp=fopen(Filename, "wb"))==NULL)
    return FALSE;

  // Assign index numbers to items
  if((Item = m_ItemParent) == NULL) {
    fclose(fp);
    return TRUE; // no items to save
  }

  while(Item != NULL) {
    Item->Index = Index++;
    Item = Item->Next;
  }

  // Match child items to parents
```

```
  Item = m_ItemParent;
  while(Item != NULL) {
    if(Item->Parent != NULL)
      Item->Owner = Item->Parent->Index;
    else
      Item->Owner = -1;
    Item = Item->Next;
  }

  // Save 'em out
  Item = m_ItemParent;
  while(Item != NULL) {
    // Item number
    fprintf(fp, "%lu\r\n", Item->ItemNum);

    // Quantity
    fprintf(fp, "%lu\r\n", Item->Quantity);

    // Coordinates
    fprintf(fp, "%lf\r\n%lf\r\n%lf\r\n", Item->XPos,           \
                                         Item->YPos,           \
                                         Item->ZPos);

    // Owner #
    fprintf(fp, "%ld\r\n", Item->Owner);

    // Next item
    Item = Item->Next;
  }

  fclose(fp); // Close the file
  return TRUE; // Return success!
}
```

Finally, you use the `Free` function when destroying the class (thus, deleting the linked list of items). Here's the code for `Free`:

```
BOOL cMapICS::Free()
{
  m_NumItems = 0;
  delete m_ParentItem;
  m_ParentItem = NULL;
```

```
  return TRUE;
}
```

You're just deleting the item linked list and getting the class ready for further use.

cMapICS::Add and cMapICS::Remove

As items are added to the map (as the result of a player dropping them, for example), you need to call `Add` to make sure those dropped items make it into the list of map objects. The `Add` function does this by first allocating a `sMapItem` structure, filling it with the appropriate item information that you give it and then linking it into the map's list of items:

```
BOOL cMapICS::Add(long ItemNum, long Quantity,                  \
                  float XPos, float YPos, float ZPos,           \
                  sMapItem *OwnerItem)
{
  sMapItem *Item;

  // Create a new item structure
  Item = new sMapItem();

  // Insert into top of list
  Item->Next = m_ItemParent;
  if(m_ItemParent != NULL)
    m_ItemParent->Prev = Item;
  m_ItemParent = Item;

  // Fill the item structure
  Item->ItemNum  = ItemNum;
  Item->Quantity = Quantity;
  Item->XPos     = XPos;
  Item->YPos     = YPos;
  Item->ZPos     = ZPos;
  Item->Parent   = OwnerItem;

  return TRUE;
}
```

Just as you use `Add` function to add objects to the map's list of items, you'll need to use `Remove` to remove items from a map. You call `Remove` using the item's identifier that you wish to remove from the map's list. `Remove` also deletes the allocated item structure and takes care of items that belong to the removed item:

```
BOOL cMapICS::Remove(sMapItem *Item)
{
```

```
  sMapItem *ItemPtr, *NextItem;

  // Remove child objects first
  if((ItemPtr = m_ItemParent) != NULL) {
    while(ItemPtr != NULL) {
      NextItem = ItemPtr->Next;
      if(ItemPtr->Parent == Item)
        Remove(ItemPtr);
      ItemPtr = NextItem;
    }
  }

  // Remove from linked list and reset root
  // if it's the current head of list.
  if(Item->Prev != NULL)
    Item->Prev->Next = Item->Next;
  else
    m_ItemParent = Item->Next;
  if(Item->Next != NULL)
    Item->Next->Prev = Item->Prev;

  // Clear link list
  Item->Prev = Item->Next = NULL;

  // Free memory
  delete Item;

  return TRUE;
}
```

cMapICS::GetNumItems, cMapICS::GetParentItem, and cMapICS::GetItem

You use these three functions to retrieve the number of items that belong to the map and to retrieve the parent `sMapItem` or specified item structure in the linked list. The first two of the following three functions return a single variable while the third function does the hard work by scanning through the linked list of objects, returning the specified item in the list:

```
long cMapICS::GetNumItems()       { return m_NumItems;   }
sMapItem cMapICS::GetParentItem() { return m_ParentItem; }

sMapItem *cMapICS::GetItem(long Num)
{
```

```
  sMapItem *Item;

  Item = m_ItemParent; // Start at parent item
  while(Num--) { // Loop until reached item num
    if(Item == NULL)
      return NULL; // no more items to scan, return error
    Item = Item->Next; // go to next item in linked list
  }

  return Item; // return resulting item
}
```

note

With the parent item structure pointer returned from `GetParentItem`, you can scan the entire linked list of items by utilizing each item structure's `Next` pointer. If you want a specific item structure based on its index in the list, use the `GetItem` function.

Using the cMapICS Class

Every map in your game will have an associated list of items that belong to it. The map ICS will load those items and provide them to your engine whenever it needs to render the map or add a specific item to a player's inventory (when an item contained in the map is picked up).

Take a look at the following code bit, which loads a sample list of items, adds a single item, removes another item, and saves the list of items:

```
cMapICS MapICS;

MapICS.Load("sample.mi"); // Load the file

// Add 1 of item # 10
MapICS.Add(10, 1, 0.0f, 0.0f, 100.0f, NULL);

// Remove 2nd item from list
MapICS.Remove(MapICS.GetItem(1));

// Save list back out
MapICS.Save("sample.mi");
```

Although this is a simple example of modifying a map's item list, why not go ahead and see just how complicated it can become?

The MapICS demo (see Figure 11.9) contains the full `cMapICS` class shown in this chapter and the `sItem` structure from the MIL Editor program. You can load the MapICS demo from the CD-ROM at the back of this book (look for \BookCode\Chap11\MapICS). You use the map ICS and MIL to render a list of objects spread around a simple level.

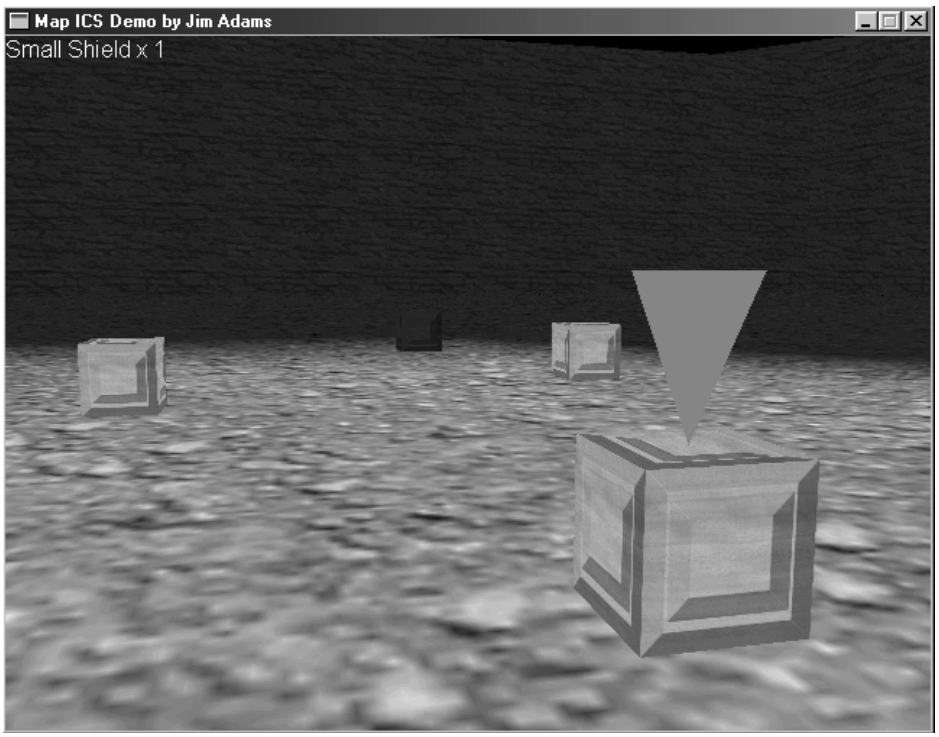


Figure 11.9 The MapICS demo allows you to walk around a simple level, picking up and dropping items.

The MapICS loads the map items and uses their coordinate data to draw a generic item mesh in the scene. Whenever you approach an item, a targeting icon appears and displays the name of the item.

Nothing in the MapICS should be new to you, so I'll skip the code at this point. Basically, the MapICS demo is like the NodeTree demo shown in Chapter 8, "Creating 3-D Graphics Engines."

note

The MapICS demo allows you to walk around a simple level by using the arrow keys and mouse. You can pick up items by standing in front of one and pressing the left mouse button. Pressing the right mouse button causes you to drop a random item.

Once the level is rendered out, each item in the map is scanned and drawn if in view. The closest item is detected, and its name is printed to the screen. The code is well commented, so you should have no problem breezing through it.

Developing a Character ICS

Although developing a character's inventory system might make you cringe at first, let me reassure you that it's not much different from developing a map inventory control system. You have the capability to add and remove items, but you don't have the problem of dealing with the item coordinates on the map. Instead, a player's ICS keeps track of order, which means that players can rearrange the items as they see fit.

Of course, this ordering is just a matter of arranging the linked list, but what about the items that can hold other items, such as backpacks? As long as you properly categorize the items as containers in the MIL Editor, you don't need to worry.

Speaking of categorizing, the real magic happens when you want to equip or use an item. Because each item is categorized, the character ICS can quickly determine what to do with the item in question. If the item is a weapon, the character ICS can ask to equip the item. If it's a healing item, the player can consume it. Beginning to get the idea?

Finally, a character ICS should allow the player to examine objects, which is the reason for the mesh and image parameters in the MIL Editor. Whenever the game's player examines the object, the specific mesh or image is loaded and displayed.

Now, turn your attention to putting together a character ICS and example.

Defining the `cCharICS` Class

The character ICS class and supporting structure developed for this book are defined as follows:

```
typedef struct sCharItem
{
  long ItemNum;              // MIL item number
  long Quantity;             // Quantity of item (i.e. coins)

  sCharItem *Prev, *Next;   // linked list pointers

  long Index;                // This items index #
  long Owner;                // Owner index #
  sCharItem *Parent;        // Parent of a contained item

  sCharItem()
  {
```

```
    Prev = Next = Parent = NULL;
    Index = 0; Owner = -1;
  }

  ~sCharItem() { delete Next; Next = NULL; }
} sCharItem;

class cCharICS
{
  private:
    long        m_NumItems;     // # items in inventory
    sCharItem *m_ItemParent;   // Linked list parent item

    // Functions to read in next long or float # in file
    long GetNextLong(FILE *fp);
    float GetNextFloat(FILE *fp);

  public:
    cCharICS();   // Constructor
    ~cCharICS();  // Destructor

    // Load, save, and free a list of items
    BOOL Load(char *Filename);
    BOOL Save(char *Filename);
    BOOL Free();

    // Add and remove an item
  BOOL Add(long ItemNum, long Quantity,                          \
           sCharItem *OwnerItem = NULL);
  BOOL Remove(sCharItem *Item);

  // Retrieve # items or parent linked list object
  long GetNumItems();
  sCharItem *GetParentItem();
  sCharItem *GetItem(long Num);

  // Re-ordering functions
  BOOL Arrange();
  BOOL MoveUp(sCharItem *Item);
  BOOL MoveDown(sCharItem *Item);
};
```

Much like the `cMapICS` class, the `cCharICS` class uses a special structure (`sCharItem`) that tracks the MIL item numbers and quantity and maintains a linked list. Unlike the `sMapItem` structure, however, `sCharItem` doesn't care about the item's coordinates.

The `cCharICS` class, again, is much like its `cMapICS` counterpart, except for the addition of three more public functions—`Arrange`, `MoveUp`, and `MoveDown`. You use these functions to sort the character's list of items. Their code is as follows:

```
BOOL cCharICS::Arrange()
{
  sCharItem *Item, *PrevItem;

  // Start at top of linked list and float
  // each item up that has a lesser ItemNum.
  // Break if past bottom of list
  Item = m_ItemParent;
  while(Item != NULL) {

    // Check previous item to float up
    if(Item->Prev != NULL) {

      // Keep floating up while prev item has
      // a lesser ItemNum value or until top
      // of list has been reached.
      while(Item->Prev != NULL) {
        PrevItem = Item->Prev; // Get prev item pointer

        // Break if no more to float up
        if(Item->ItemNum >= PrevItem->ItemNum)
          break;

        // Swap 'em
        if((PrevItem = Item->Prev) != NULL) {
          if(PrevItem->Prev != NULL)
            PrevItem->Prev->Next = Item;
          if((PrevItem->Next = Item->Next) != NULL)
            Item->Next->Prev = PrevItem;
          if((Item->Prev = PrevItem->Prev) == NULL)
            m_ItemParent = Item;
          PrevItem->Prev = Item;
          Item->Next = PrevItem;
        }
      }
```

```
    }

    // Go to next object
    Item = Item->Next;
  }

  return TRUE;
}

BOOL cCharICS::MoveUp(sCharItem *Item)
{
  sCharItem *PrevItem;

  // Swap item and item before it
  if((PrevItem = Item->Prev) != NULL) {
    if(PrevItem->Prev != NULL)
      PrevItem->Prev->Next = Item;
    if((PrevItem->Next = Item->Next) != NULL)
      Item->Next->Prev = PrevItem;
    if((Item->Prev = PrevItem->Prev) == NULL)
      m_ItemParent = Item;

    PrevItem->Prev = Item;
    Item->Next = PrevItem;
  }

  return TRUE; // Return success
}

BOOL cCharICS::MoveDown(sCharItem *Item)
{
  sCharItem *NextItem;

  // Swap item and item after it
  if((NextItem = Item->Next) != NULL) {
    if((Item->Next = NextItem->Next) != NULL)
      NextItem->Next->Prev = Item;
    if((NextItem->Prev = Item->Prev) != NULL)
      Item->Prev->Next = NextItem;
    else
      m_ItemParent = NextItem;
```

```
    NextItem->Next = Item;
    Item->Prev = NextItem;
  }

  return TRUE; // Return success
}
```

`Arrange` sorts the linked list of items based on each item's MIL item number, from lowest to highest. If, on the other hand, you want to specifically order the list yourself, you can utilize the `MoveUp` and `MoveDown` functions, which take a pointer to a `sCharItem` structure that is already contained in the list.

The `MoveUp` function moves the specified `sItem` structure up in the linked list, and `MoveDown` moves the specified structure down in the linked list. Figure 11.10 illustrates the concept of using the `Arrange`, `MoveUp`, and `MoveDown` functions on a sample linked list of items.

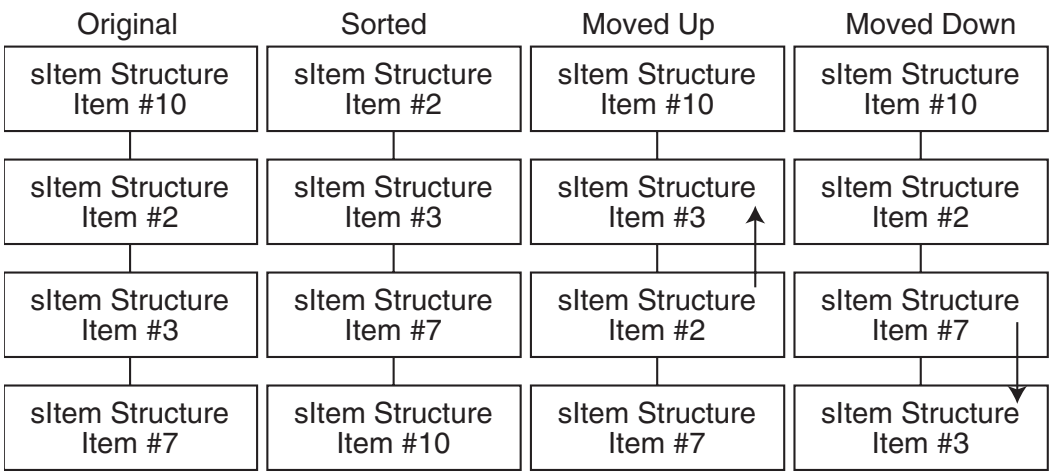


Figure 11.10 From left to right, you see the original unsorted list, the arranged (sorted) list, the list after moving an item up, and the list after moving an item down.

The rest of the functions in `cCharICS` are identical in their functionality to the `cMapICS` class, with the obvious exclusion of the item coordinates used when adding an item to the list. Even the storage format for character items is identical to the map item format, except for the coordinates.

Using the `cCharICS` Class

To demonstrate the use of the character ICS system, I created a demo application named CharICS that maintains a list of items contained in the `default.mil` master item list file. You can find the demo on the CD-ROM at the back of this book (look for

\BookCode\Chap11\CharICS). When you start the demo, you'll see the Inventory dialog box (shown in Figure 11.11). In the Inventory dialog box, the list box contains an imaginary character's inventory, which you can manipulate by using the buttons on the dialog box.

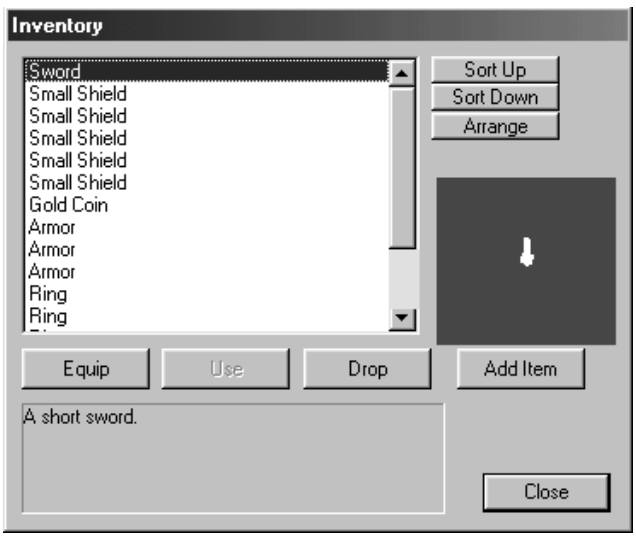


Figure 11.11 The CharICS demo allows you to peruse an imaginary character's inventory, adding, dropping, or sorting items as you see fit.

To use the CharICS demo, follow these steps:

1. Click the Add Item button. The CharICS demo will add a random item from the MIL to the inventory list.
2. Select an item from the list. Items are classified, so the Use and Equip buttons are enabled or disabled depending on which item you select from the list. Clicking Equip has no effect; it comes into play later when you deal with characters. Clicking Use, however, removes an item if it is flagged as `USEONCE`.
3. Click the Drop button to drop an item (remove the item from the inventory list) if it is flagged as `CANDROP`.
4. To arrange the items in the inventory, you can click an item and then click either Sort Up or Sort Down. The selected item then moves up or down in the list. To arrange the items by their item number in the list, click Arrange.

As a last, special treat, items that have matching meshes appear in the box on the right in the demo. The 3-D object spins around slowly so that you have a full view of all sides. Now, that's cool!

Wrapping Up Objects and Inventory

Don't let the complexity of working with objects in your game fool you. The simple fact is that items in your game are actually small, easily handled bits of data. It's when characters start using those items that things become a bit more complicated. In this chapter, you saw just how you use inventory control systems to handle items belonging to a map and a character.

To expand the usefulness of the MIL Editor (and for your own game's items), I recommend adding your own item attributes. This involves a little Windows programming knowledge, because you must modify the dialog box that handles the modifications. In addition, you have to rewrite the `sItem` structure to hold the new attributes that you're adding. Of course, you don't have to worry about that challenge; you're becoming a role-playing game programming wizard.

Programs on the CD-ROM

The \BookCode\Chap11\ directory contains these three programs, which demonstrate how to use a master item list and how to use the two inventory control systems developed in this chapter:

- **MILEdit.** The Master Item List Editor program discussed in this chapter. The editor aids in the creation of items you can use in your own game project. Location: \BookCode\Chap11\MILEdit\.
- **MapICS.** An example of a map inventory control system. Items loaded from disk are scattered about a map that the user can scroll around using the mouse and arrow keys. Location: \BookCode\Chap11\MapICS\.
- **CharICS.** An example character inventory control system that enables you to view a list of items. Items can be examined, used, or equipped. Location: \BookCode\Chap11\CharICS\.