Part II

Isometric Fundamentals

Chapter 10

Tile-Based Fundamentals

- What Does "Tile-Based" mean?
- An Introduction to Rectangular tiles
- Managing Tile Sets
- Tile Map Basics

With this chapter, we break away from the introductory matter that filled Part 1 and start to move into the really cool stuff. Naturally, you aren't going to fly when you haven't walked yet. We will explore tile-based fundamentals, from both the management and user interface sides of the coin.

What Does "Tile-Based" Mean?

You've seen floor tiles, right? Sometimes, especially in older buildings or in malls, different tiles are combined into a pattern (sometimes very elaborate patterns). That's exactly what we'll be doing, but instead of using linoleum or porcelain, we'll be using graphic images.

This is where the comparison ends between tile-based games and floor tiles. When you make a tile-based game, each graphic tile has a different meaning. One might be the floor, and the other solid rock (representing a wall). In a tile-based game, some sort of "characters" or "units" usually occupy tiles, and are moved around by either the player or the computer's AI. These are called *agents* in AI terminology.

The rules of the game determine what happens to the agents as they occupy the various tiles of the game, and they also govern how the agents may move from one tile to another. In a strategy game, an agent may be able to move three tiles per turn. However, different tiles (such as grassland and hills) may have different "movement costs" associated with them. Grassland might only cost 1 to move, but a hill could easily cost 2 or 3. Further addition of things like roads or rivers may reduce these costs. It can all get very complicated very quickly.

Of course, the player isn't really thinking about all this. He just presses the up arrow to move to whatever square is there, and the computer takes care of movement cost. (Even though the player isn't consciously thinking about movement costs, he does know that it takes longer to traverse hills than it does to traverse plains.)

Myths about Tile-Based Games

The first myth about tile-based games is that they are dead. This is wholly untrue. Yes, the days of pure 2D tile-based games are over. These days you have to have 3D rendering to make a really hot game. However, even 3D games can be tile-based, and many are. This is where isometric games come in. I won't go into how isometric tiles work until chapter 11, but suffice it to say that isometric *is* 3D, even if it's done with just 2D rendering. Most real-time strategy games and turn-based strategy games are made using isometric tiles (although the days of pure 2D isometric tile-based games are drawing to a close as well).

The second myth is that no one will buy a tile-based game. Also untrue. In your local computer game store, see for yourself. The strategy genre is filled with tile-based games.

Tile-Based Terminology

While reading this section, keep in mind that these are mainly the terms I use. You might have different names for them. For the most part, these terms are standard.

- **Tile**. A graphic used to render a portion of the background. When using rectangular tiles, this usually means that the entire rectangular area is taken up by the tile. This is not always so, however. You might have "fringe" tiles to take care of coastlines or a transition from one type of terrain to another, in which case the tile may cover only a portion of the rectangular area.
- **Sprite**. An arbitrarily-sized graphic that is usually used for either agents or foreground objects. Really, everything that isn't a tile is a sprite. *Sprite* is just a generic term, like *tile*. You may decide to subclass them into units, buildings, and markers, depending on the type of game.
- **Tileset**. A set of tiles. It is inefficient to store each tile in a separate graphics file. It's easier to just take tiles and group them into logical sets and then place them all into a single graphics file that gets parsed later. A tileset might include sprites or might consist entirely of sprites.
- **Space**. Any arbitrarily-sized and shaped two-dimensional space. Usually a space is rectangular, but not in all cases. You tend to work with nonrectangular spaces using a bounding rectangle as well as whatever other structure describes them.
- **Screen space**. The space on the screen used for rendering the play area, not including any borders, status panels, menu bars, message bars, or any other nonplay area structures. In some cases, the entire display is the screen space. Often, it is not.
- **View space**. The same size as screen space, but the upper-left corner is always at (0,0) for view space. Many times, view space and screen space are the same. View space, in most cases, is purely abstract and plays no part in the rendering process.
- **Tile space**. The smallest space (usually rectangular) that is taken up by an individual tile. In rectangular tiles, this is often the entire rectangle. Tile space can also refer to the space taken up by a sprite.
- **World space**. The space that allows the display of an entire map of tiles and their associated object/agent sprites. In board games and puzzle games, world space may be equal to or smaller than screen space/view space. In larger games, world space might be hundreds of times larger. Figure 10.1 shows the relationship between the screen, view, and world spaces.

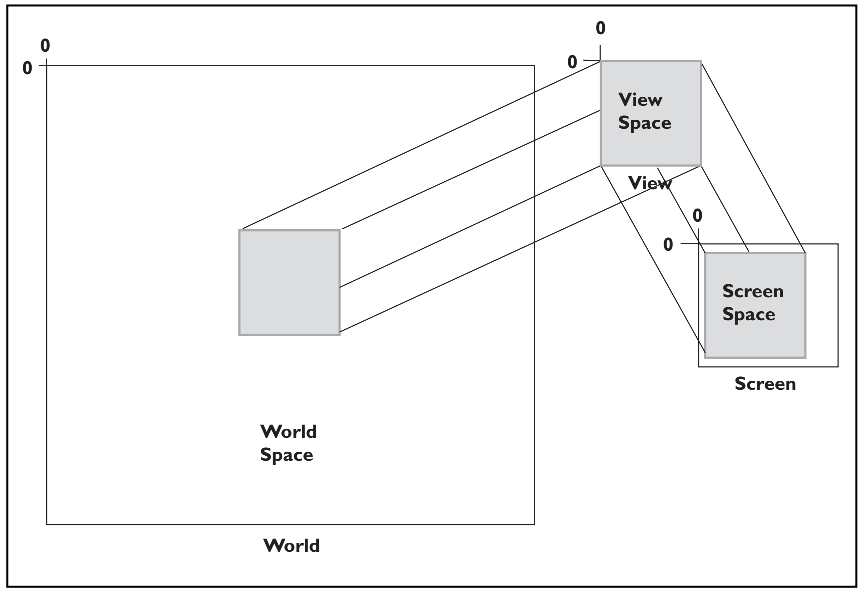

Figure 10.1

Screen, view, and world spaces and their relationship

- **Anchor**. A correlation of one point of a space (usually (0,0)) to another space. An example of this is a correlation of view space to screen space. If you had an 8-pixel border around the main viewing area, you would keep a point that kept track of the relationship from view space to screen space—namely, (8,8). This lets you know how to convert between screen space and view space. Another example would be an anchor that converts from view space to world space. In a scrolling tile engine (with a world space larger than the view space), this anchor helps determine which tiles have to be rendered by translating the tile's world space coordinates into view space coordinates. From there, you can translate them into screen space coordinates.
- ***Anchor space***. A space that defines legal values for the view-to-world anchor. Clipping your anchor point with anchor space lets you easily manage the view-to-world anchor and lets you keep the player from having an illegal view.
- ***Extent***. A rectangle relative to a point (usually an anchor), often with negative left and top values. We will get into this more when I talk about using templates to manage files.
- **Tilemap**. An array containing information about how the world looks—that is, which tiles are in what location. Tilemaps also contain information about objects and agents in the world, even though the structure that contains agents or objects may be a different array, or not even an array at all.
- **Agent**. Any sprite (or sequence of sprites used for animation) that moves, either by AI or by player action.
- **Object**. An unmoving graphic, representing such things as trees, rocks, or other items.

Hope I didn't lose you. If you're fuzzy on the real application of these terms, don't worry. I'll explore the meaning and uses of each as I go, and you will gain full understanding.

An Introduction to Rectangular Tiles

Rectangular tiles are the easiest of all to work with, because of their rectangular-ness. Most of the time, when working in rectangle land, you use square tiles. You can use other sizes, of course, but square seems to be a favorite. An example of a tile is shown in Figure 10.2.

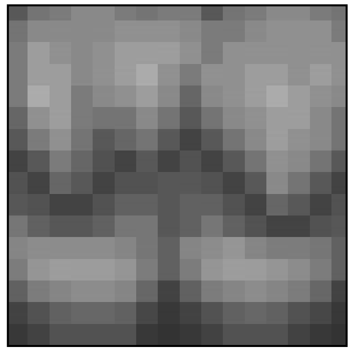

Figure 10.2

A square tile

The point of view for games with square tiles is usually *top down* or *overhead*. This just means that all your graphics must be drawn as though you are looking down on the object. Sometimes you can give your game a slightly angled view so that you are looking mostly down, but you can see some of the front or back, depending on how the agent is facing.

Another point of view for square tiles is the *side scroller* view, where you are looking at the world from its side. This was very popular among older action games like *Super Mario Bros* (Nintendo) and the original *Duke Nukem*.

With the advance of 3D display technology, both the top-down and side-scroller views have become nearly obsolete.

NOTE

Later, when we get to isometric tiles, you will use a view called "3/4," in which you render your agents and object as though you are looking straight at them. It gives the illusion of 3D without perspective correction. Luckily, the human eye is easy to fool, because it automatically corrects for the errors in the projection.

Normally, you will want to group your tiles and sprites into graphical files where more than one tile or sprite is in the file. Normally, you'll want one or two files with the graphics for the background, a file for objects, and then a number of files for the agents (one file to an agent, unless you don't have too many animation sequences).

The examples shown in Figures 10.3, 10.4, and 10.5 are from Ari Feldman's SpriteLib, which is a free graphics package that has been around for a few years. If you aren't graphically inclined (don't be ashamed…you're not alone), you may want to download it from http://www.arifeldman.com.

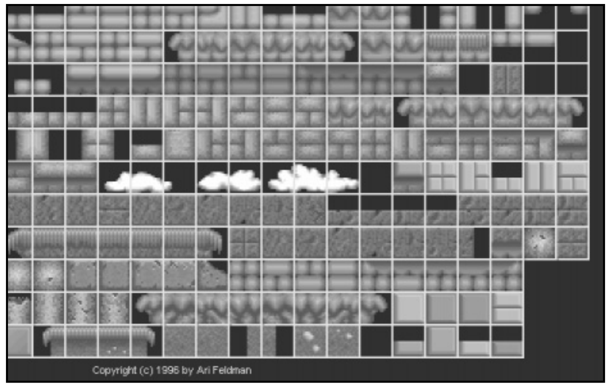

Figure 10.3

Background tiles

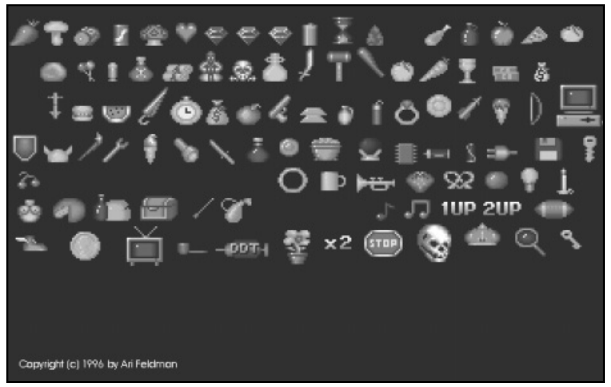

Figure 10.4

Object tiles

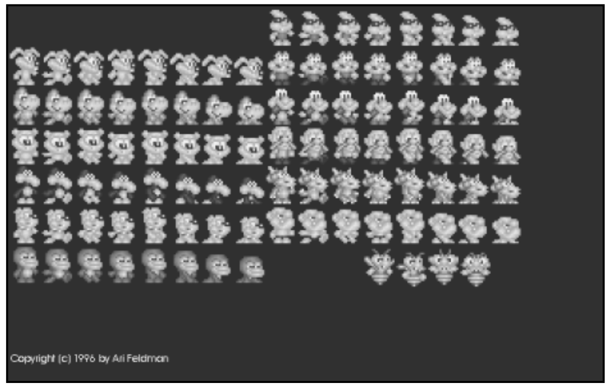

Figure 10.5

Character tiles

Managing Tilesets

The tilesets and sprite sets you just saw are great, but they aren't exactly in a form that is easy to work with for programmers like you and me. If you wanted to work with them, you'd have to store a bunch of rectangles in a text file or other configuration file, or (*gasp!*) you'd have to hardcode image rectangles. Later, if you decided to change the art, this would be a maintenance horror show. You'd have to go back and change around the rectangle lists. Of course you'd forget one, and naturally you wouldn't find out until one of your beta testers got really far into the game. . . well, I think you get the idea.

So, what's the solution to this dilemma? Templates. A template is used in the first example of a tileset—the one with the white boxes around it (Figure 10.3). That's one way to do a template. However, it's not the best way, because you still have to either hardcode or put into a configuration file the width and height of the template.

Load IsoHex10_1.bmp into your favorite graphics editing program. Figure 10.6 shows what it looks like.

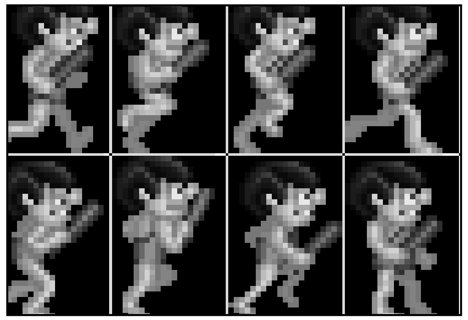

Figure 10.6

A sample tileset

You can see the border around each of the images. Unlike the tileset shown in Figure 10.3, the border is green instead of white. Or is it?

Take a really close look at the top cell (zoom in as far as the program will let you), shown in Figure 10.7.

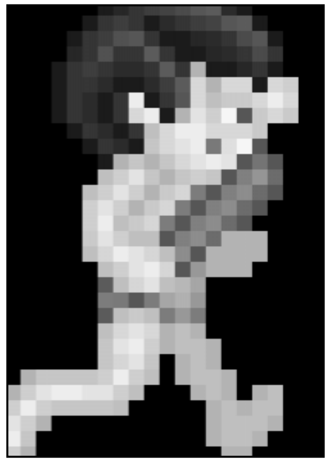

Figure 10.7

Zooming in on the caveman

There is more than just green. . . there is also white and cyan (you can't see it too well in the book, but you can see it just fine in a graphics program). As you might have guessed, each of the different colors means something. The green dots span the width and height of the image. White dots are part of the frame but outside of the image. The black dot in the corner is what designates the corner of a tile cell, and also the transparent color of the tileset. The cyan dot (or blue dot) designates a coordinate for the tile's anchor. (I use cyan when the anchor is within the bounds of the tile image itself. In other words, it would otherwise be a green dot, and blue when the anchor point would otherwise be white.)

Why these colors? Why not a completely different set of colors? Quite frankly, you *could* use a different set of colors, and this type of template supports just that. Take a look and zoom in on the upper-right corner of the tileset (shown in Figure 10.8).

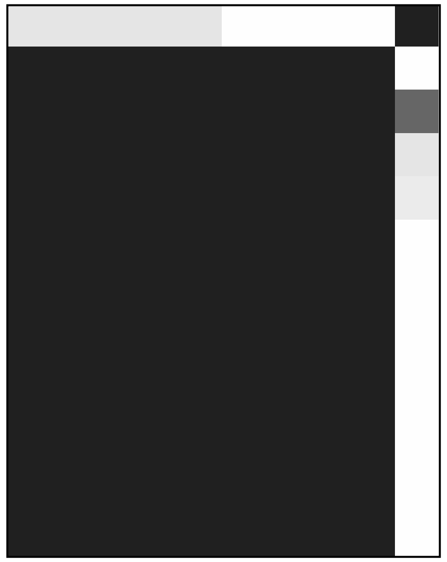

Figure 10.8

The upper-right corner of the tileset, demonstrating control colors

There are five pixels in the rightmost column, in the following order: black, white, blue, green, and cyan. These specify corner, frame, anchor, inside, and inside anchor, respectively. If you wanted to, you could change one of these colors to red (for example), and put it in the proper control color position, and use it instead of the color used here.

Using an extended template like this gives you a great deal of freedom. You can make a template and later change the width or height of the cells, and it will still load the same way. The green and blue and cyan pixels let you calculate tile spaces, anchor points, and tile extents, which you can parse into arrays of rectangles and points. You can move an image's anchor point and have it show up in a different location. An extended template takes pressure off programmers and removes stress from artists, who, when using it, are less constrained by the normally tight restrictions for tile-based graphics.

Before you finish building your utopian society, though, you have to write code that will parse a graphics file into arrays of rectangles and points. Let's start by figuring out what information you need about each tile. Presumably, these templated graphics will be on an `IDirectDrawSurface7` somewhere, and you want to optimize your data structures for using `Blt` and `BltFast`. Since both of these use source rectangles, you'll definitely want to keep an array of `RECT`s for that. The coordinates held in these `RECT`s will be pixel coordinates measured from (0,0) in the tileset's image. Figure 10.9 shows what one of these `RECT`s might look like.

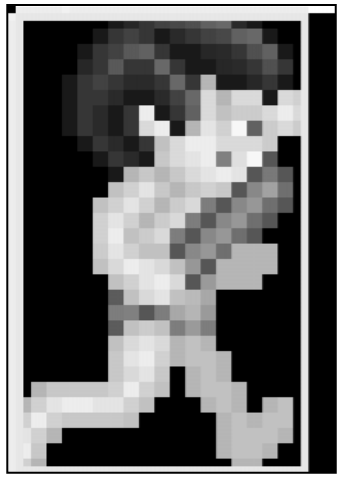

Figure 10.9

Source `RECT`

In Figure 10.9, the RECT has the coordinates (1,1)–(39,61). Remember that you have to add one to the bottom and right because of how RECTs work. Doing so gives you a resulting RECT of (1,1)–(40,62).

Because you might or might not be referencing off the upper-left of these source RECTs (the blue and cyan points might lie elsewhere), you need an array of POINTs to keep track of the tile anchors. Like the source rectangles, these POINTs contain coordinates into the tileset image, meaning that a tile with a rectangle of (100,100)–(200,200) has its anchor point within the x and y range of that RECT (rather than having the anchor point in reference to the tile cell's upper-left corner, which is another way you could have done this that would have added unnecessary computations). Figure 10.10 shows the anchor point.

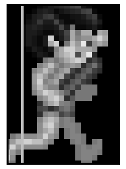

Figure 10.10

Anchor point

In Figure 10.10, the anchor point is (7,1), which is within the range of your source RECT, as I said it would be.

Finally, you add another array of RECTs to hold the tile extents. Extents can be calculated after the source RECT and anchor POINT have been determined, like so:

```
//copy source rect
CopyRect(&rcExtent,&rcSrc);
//offset by anchor point
OffsetRect(&rcExtent,-ptAnchor.x,-ptAnchor.y);
```

In this case, the extent is (–6,0)–(32,60). The derivation of these values is as follows:

Upper Left:

From source RECT	(1,1)
Minus anchor point	(7,1)
Combine coordinates	(1–7,1–1)
Solve	(–6,0)

Bottom Right:

From source RECT	(40,62)
Minus anchor point	(7,1)
Combine	(40–7,62–1)
Solve	(33,61)

Yes, there's a negative left coordinate; this is quite common for this type of tileset, where you might want to reference a tile from a point other than the upper left. It would not be a stretch to use the character's feet or center. Just use whatever works to give an animation continuity and smoothness. For this tileset, the horizontal aspect of the anchor lines up with the back of the caveman's hair.

The idea here is that you want to be able to simply specify a single (x,y) screen coordinate and tell it which tile to blit, and have it come out right. When blitting, the (x,y) point corresponds to the tile's anchor point. This means that if you tell this tile to blit to screen coordinate (100,100), you want screen coordinate (100,100) to correspond to the tileset image's coordinate (7,1).

You want the extent so that you can simplify the process of determining the coordinates of the destination RECT (or the destination (x,y) for BltFast). Taking the source RECT and subtracting the point gives you the extent. This way, based on a single set of coordinates (dstX and dstY), you can determine the proper destination rectangle. For example:

```
//for Blt
CopyRect(&rcDst,&rcExtent);
OffsetRect(&rcDst,dstX,dstY);
//perform the Blt

//for BltFast
dstX+=rcDst.left;
dstY+=rcDst.top;
//perform bltfast
```

If you didn't precalculate the extents, you would have to calculate them from the source rectangle, anchor point, and destination point each time you wanted to render the tile. Although doing so isn't too much more work (about a dozen add or subtract operations), it *is* work, and in game programming, you want to avoid any work that you can. Precalculating tile extents might give you just one extra frame per second or even only half a frame per second, which doesnt' sound like much, but if you have two optimizations that each give you an extra half-frame per second, you've just earned yourself another frame per second. Game programming is a game of inches.

A TileSet Class

So, now that you've decided what information you want, you just have to go in and get it. I made a class to work with these sorts of templates. It's called `CTileSet`, and you can find the code for it in TileSet.h and TileSet.cpp.

The Class Declaration

First, I designed a struct to contain important information about tiles, including source rectangle, anchor point, and destination extent. I put this information into `TILEINFO`.

```
//tileset information structure
struct TILEINFO
{
       RECT rcSrc;//source rectangle
       POINT ptAnchor;//anchoring point
       RECT rcDstExt;//destination extent
};
```

The members of `TILEINFO` are explained in Table 10.1.

Table 10.1 TILEINFO Members

TILEINFO Member	Meaning
`rcSrc`	Source `RECT` for the tile
`ptAnchor`	Anchor `POINT` for the tile
`rcDstExt`	Destination extent `RECT` for the tile

Next, here's the class itself:

```
class CTileSet
{
private:
       //number of tiles in tileset
       DWORD dwTileCount;
       //tile array
       TILEINFO*  ptiTileList;
       //filename from which to reload
       LPSTR lpszReload;
       //offscreen plain directdrawsurface7
       LPDIRECTDRAWSURFACE7 lpddsTileSet;
public:
       //constructor
       CTileSet();
       //destructor
       ~CTileSet();
       //load (initializer)
       void Load(LPDIRECTDRAW7 lpdd,LPSTR lpszLoad);
       //reload (restore)
       void Reload();
       //unload (uninitializer)
       void Unload();
       //get number of tiles
       DWORD GetTileCount();
       //get tile list
```

```
        TILEINFO* GetTileList();
        //get surface
        LPDIRECTDRAWSURFACE7 GetDDS();
        //retrieve filename
        LPSTR GetFileName();
        //blit a tile
        void PutTile(LPDIRECTDRAWSURFACE7 lpddsDst,int xDst,int yDst,int
iTileNum);
};
```

The private members contain all of the information needed to process the tileset. These are listed in Table 10.2.

Table 10.2 CTileSet Private Members

CTileSet Private Member	Meaning
`dwTileCount`	The number of tiles contained in the tileset
`ptiTileList`	A pointer to an array of `TILEINFO` that describes each tile
`lpszReload`	The file name from which this tileset was loaded
`lpddsTileSet`	The `IDirectDrawSurface7` pointer that is the off-screen surface containing the tileset

The member functions in the public section perform all necessary operations on the tileset. Table 10.3 explains these member functions.

Table 10.3 CTileSet Public Member Functions

CTileSet Public Member Function	Purpose
`CTileSet`	Constructor that initializes all variables to 0 or `NULL`
`~CTileSet`	Destructor that calls `Unload`
`Load`	Loads and parses an image
`Reload`	Reloads the image (if for some reason the surface has been freed, such as resulting from an Alt+Tab)
`Unload`	Frees the resources associated with the tileset
`GetTileCount`	Returns the number of tiles
`GetTileList`	Returns the tile info pointer
`GetDDS`	Returns a pointer to the `IDirectDrawSurface7` containing the tileset
`GetFileName`	Returns the name of the file from which the tileset was loaded
`PutTile`	Puts a tile on a surface, given a coordinate and a tile number

The constructor and destructor don't do much and aren't very interesting, but the other functions are more important, so I'll explain them in more detail.

CTileSet::Load

This function loads a bitmap and places it onto a DirectDraw surface and also parses the image into its component tiles.

```
void CTileSet::Load(LPDIRECTDRAW7 lpdd,LPSTR lpszLoad);
```

The `lpdd` parameter is a pointer to an `IDirectDraw` object, which is used to initially create the tileset surface. The `lpszLoad` parameter is the name of the file to load that contains the image you want for this tileset.

This function is quite long, because of the image parsing. It performs the following tasks:

1. Loads the image.
2. Grabs the control colors from the upper-right corner.
3. Counts and measures the horizontal and vertical cells.
4. Allocates the tile list.
5. Scans each tile's left and top for anchor points and inside points (using default values if these control points are not specified).
6. Calculates destination tile extents.

All of the main work is done here, at load time, so that after a call to `Load`, you can immediately start using `PutTile`, and you never really have to worry about it ever again.

CTileSet::Reload

`CtileSet::Reload` reloads an image if and when it is lost due to a display mode change or Alt+Tab incident.

```
void CTileSet::Reload();
```

If, as a result of an Alt+Tab or other such misfortune, your tileset's surface is lost, a call to `IDirectDraw7::RestoreAllSurfaces` may be required. After that, you can call `CTileSet::Reload`, and the image will be reloaded (but not reparsed).

CTileSet::Unload

This frees all the resources used by the tileset. It is called during the destructor and whenever `Load` is called.

```
void CTileSet::Unload();
```

Very likely, you will never call this function directly, since it is taken care of in the destructor. Even if you wanted to load a different image into a tileset, you could just call `Load`. The only time you would ever want to call `Unload` is if you were trying to conserve video memory for other images. It is here mainly for completeness.

CTileSet::GetTileCount

This one's a no-brainer.

```
DWORD CTileSet::GetTileCount();
```

This function returns the number of tiles in the set.

CTileSet::GetTileList

This function gives you access to the tile information, which is very important if you want to implement clipping yourself rather than relying on a DirectDraw clipper and `CTileSet::PutTile`.

```
TILEINFO* CTileSet::GetTileList();
```

This function returns the pointer to the tile array. You can use the result of this function just as you would an array.

```
//make tileset
CTileSet tsExample;
tsExample.Load(lpdd,"Sample.bmp");
//retrieve the info about tile zero
TILEINFO ti=txExample.GetTileList()[0];
```

CTileSet::GetDDS

This function allows access to the DirectDraw surface on which dwell the tiles.

```
LPDIRECTDRAWSURFACE7 CTileSet::GetDDS();
```

This function returns the `IDirectDrawSurface7` pointer that contains the image of the tileset. If for some reason you wanted to modify or read from the surface, this would be the function you'd start with. Keep in mind that any changes you make to the surface will not survive a call to `CTileSet::Reload`. Also, if you want to keep a copy of the surface pointer for a long time, it might be best to use `AddRef` so that the surface isn't inadvertently deleted in the interim.

CTileSet::GetFileName

This function is pretty self-explanatory.

```
LPSTR CTileSet::GetFileName();
```

This returns a pointer to the file name that is used to reload the tileset.

CTileSet::PutTile

This function is the reason for the whole show. It's the workhorse of the `CTileSet` class.

```
void CTileSet::PutTile(LPDIRECTDRAWSURFACE7 lpddsDst,int xDst,int yDst,int
iTileNum);
```

This takes care of putting a tile onto a destination surface (`lpddsDst`), with `xDst,yDst` corresponding to the anchor point of the specified tile (`iTileNum`). Tiles are numbered starting with 0 and are ordered left to right, top to bottom.

An Animated Sprite Example

One of the many uses for a tileset is for an animated sprite sequence. Earlier in this chapter, I showed you a tileset consisting of some caveman images from spritelib, which is a good example of one such animation sequence. Load up IsoHex10_1.cpp, which makes use of `CTileset` (among other things; see the top of IsoHex10_1.cpp). If you load and run it, you will see the caveman running in place, as shown in Figure 10.11.

Figure 10.11

Animation demo

This example is based on IsoHex1_1.cpp, just like the rest of the examples. The main differences exist in `Prog_Init`, `Prog_Loop`, and `Prog_Done`.

Setting up

The `Prog_Init` does the required DirectDraw setup (creating the DirectDraw interface, creating the primary surface and the back buffer). It also loads the tileset into `tsCaveMan` (a global `CTileSet` variable).

```
bool Prog_Init()
{
        //create IDirectDraw7
        lpdd=LPDD_Create(hWndMain,DDSCL_FULLSCREEN | DDSCL_EXCLUSIVE |
DDSCL_ALLOWREBOOT);
        //set display mode
        lpdd->SetDisplayMode(800,600,16,0,0);
        //create primary surface
        lpddsMain=LPDDS_CreatePrimary(lpdd,1);
        //get back buffer
        lpddsBack=LPDDS_GetSecondary(lpddsMain);
        //clear out back buffer
        DDBLTFX ddbltfx;
        DDBLTFX_ColorFill(&ddbltfx,0);
        lpddsBack->Blt(NULL,NULL,NULL,DDBLT_WAIT | DDBLT_COLORFILL,&ddbltfx);
        //load in tileset
        tsCaveMan.Load(lpdd,"IsoHex10_1.bmp");
        return(true);//return success
}
```

The Main Loop

In `Prog_Loop`, three things happen. First, the back buffer is cleared out. Second, one of the cells of the tileset is written to the approximate middle of the screen. Third, the application is locked to 15 frames per second.

```
void Prog_Loop()
{
        //start timer
        DWORD dwTimeStart=GetTickCount();
        //clear out back buffer
        DDBLTFX ddbltfx;
        DDBLTFX_ColorFill(&ddbltfx,0);
        lpddsBack->Blt(NULL,NULL,NULL,DDBLT_WAIT | DDBLT_COLORFILL,&ddbltfx);
        //put the caveman
        tsCaveMan.PutTile(lpddsBack,400,300,dwCaveManFrame);
        //change the frame number
        dwCaveManFrame++;
        dwCaveManFrame%=8;
        //flip
        lpddsMain->Flip(NULL,DDFLIP_WAIT);
        //lock to 15 FPS
        while(GetTickCount()-dwTimeStart<66);
}
```

You can see that using CTileSet is a great deal easier than setting up RECTs and going down that path. The tileset makes sprite and tile management easy and doesn't add that much overhead.

Cleaning up

You don't have to call the Unload function, because CTileSet's destructor automatically does so, and you can essentially ignore your tileset in Prog_Done. You can just destroy the primary surface and the IDirectDraw and be done with it.

```
void Prog_Done()
{
        //destroy primary surface
        LPDDS_Release(&lpddsMain);
        //destroy IDirectDraw7
        LPDD_Release(&lpdd);
}
```

Taking Control

Although just watching a caveman run in place is fun, you'd probably rather control him. For this, I wrote IsoHex10_2.cpp. This example is mostly the same as IsoHex10_1, except that now you respond to the arrow keys and use that information to move the caveman back and forth across the screen. The major change happens in Prog_Loop.

```
void Prog_Loop()
{
        //start timer
        DWORD dwTimeStart=GetTickCount();
        //clear out back buffer
        DDBLTFX ddbltfx;
        DDBLTFX_ColorFill(&ddbltfx,0);
        lpddsBack->Blt(NULL,NULL,NULL,DDBLT_WAIT | DDBLT_COLORFILL,&ddbltfx);
        //put tile

tsCaveMan[dwCaveManFace].PutTile(lpddsBack,dwCaveManPosition,300,dwCaveManFrame);
        //move
        if(MoveLeft^MoveRight)
        {
                if(MoveLeft)
                {
                        //moving left
                        dwCaveManFace=1;
                        //update position
                        dwCaveManPosition+=796;
                        dwCaveManPosition%=800;
                        //update animation frame
                        dwCaveManFrame+=1;
                        dwCaveManFrame%=7;
                }
                else
                {
                        //moving right
                        dwCaveManFace=0;
                        //update position
                        dwCaveManPosition+=4;
                        dwCaveManPosition%=800;
                        //update animation frame
                        dwCaveManFrame+=1;
                        dwCaveManFrame%=7;
                }
        }
        else
        {
                //standing
                dwCaveManFrame=7;
        }
        //flip
        lpddsMain->Flip(NULL,DDFLIP_WAIT);
        //lock to 15 FPS
        while(GetTickCount()-dwTimeStart<66);
}
```

The global variables named MoveLeft and MoveRight are bools, and you change their status in response to WM_KEYUP and WM_KEYDOWN.

```
        case WM_KEYDOWN:
                {
                        //on escape, destroy main window
                        if(wParam==VK_ESCAPE)
                        {
                                DestroyWindow(hWndMain);
                        }
                        //movement keys
```

```
                    if(wParam==VK_LEFT)
                    {
                        MoveLeft=true;
                    }
                    if(wParam==VK_RIGHT)
                    {
                        MoveRight=true;
                    }
                    return(0);//handled
                }break;
        case WM_KEYUP:
                {
                    //movement keys
                    if(wParam==VK_LEFT)
                    {
                        MoveLeft=false;
                    }
                    if(wParam==VK_RIGHT)
                    {
                        MoveRight=false;
                    }
                    return(0);//handled
                }break;
```

In `Prog_Loop`, you can see that, depending on which key is being pressed, the facing (contained in `dwCaveManFace`), the position (`dwCaveManPosition`), and the animation frame (`dwCaveManFrame`) are updated. Nothing happens if both keys are pressed at the same time.

This is about it for your crash course in tile and sprite management. Throughout the rest of the book, you will make heavy use of `CTileSet`. I hope I've shown you that this stuff isn't so hard after all, as long as you have the proper tools and classes to help you.

NOTE

You may have noticed that no subtraction is done—only addition. This is because all the variables are `DWORD`s, or unsigned longs, which have no negative values. You can see that all the additions are shortly followed by a modulus (%) operation. Combining addition and modulus, you get a net subtraction.

Tilemap Basics

A single tile, or even a sequence of tiles depicting an animated character, isn't in itself very useful. In order to be useful, a variety of sprites and tiles must be used together. Now that you've seen how easy it is to manipulate tilesets, the time has come to get into tilemaps. When creating a tile-based world, you must have a way to represent it in your computer's memory. Usually, you do so with some sort of array, although there are other more complicated but more flexible solutions.

Since we are still in rectangle land, our tilemaps are more intuitive than they will be once we get into isometric and hexagonal tilemaps. They are simply two-dimensional arrays, like so:

```
int iTileMap[WIDTH][HEIGHT];
```

`WIDTH` and `HEIGHT` can be any old value—whatever you need to make your tilemap the proper size. In a chess or checkers game, `WIDTH` and `HEIGHT` would both have a value of 8. A side-scroller might have a `HEIGHT` equal to the screen height divided by the tile height, but the width of the map times the width of the tiles might be several times the width of the screen. The `WIDTH` and `HEIGHT` values depend entirely on what kind of game you are making.

The meaning of the numbers in this array remains in question. Intrinsically, they have none; the meaning of the numbers is entirely up to you. You may not even have ints in the array, but instead a completely different custom structure. Again, this is entirely game-dependent.

For example, in a checkers game, the board squares are alternately black and red, as shown in Figure 10.12. You might put this in the number as a bit flag (if bit 0 is set or not set, for example). On the other hand, you might decide that a board whose x and y add up to be an even number is black, and an odd number is red, like so:

```
if((tilex+tiley)&1)
{
    //red square
}
else
{
    //black square
}
```

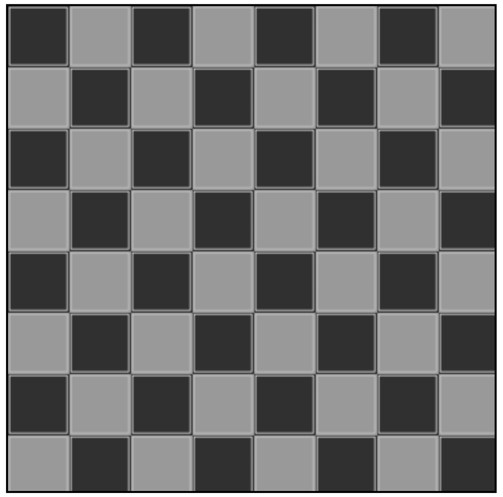

Figure 10.12

A checkerboard

Figure 10.13 shows the calculations for (x+y) & 1 for the sample checkerboard.

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

Figure 10.13

Alternating odd/even checkerboard

You may want to only contain in your checkerboard's tile array which piece is or is not there. There are a total of five options: black piece, red piece, black king, red king, and empty; you might create an `enum` to keep track of them.

```
enum{EMPTY=0,BLACKPIECE=1,REDPIECE=-1,BLACKKING=2,REDKING=-2};
```

In this scheme, all black pieces are positive numbers, and all red pieces are negative. This provides an easy way to differentiate them and conveniently leaves 0 for representing empty. The starting board configuration tilemap values are shown in Figure 10.14.

Y↓ X→	0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0	1
2	1	0	1	0	1	0	1	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	-1	0	-1	0	-1	0	-1
6	-1	0	-1	0	-1	0	-1	0
7	0	-1	0	-1	0	-1	0	-1

Figure 10.14

Starting board configuration

More Complicated Tilemaps

Checkers is a good example of a game for which to use a very simple map structure. There isn't much variety in the tiles this map can hold. This is true of most board and puzzle games, like chess, Reversi, and so on. However, more complicated games like turn-based or real-time strategy games are more visually rich and thus have a more complicated map structure. Also, these types of maps tend to be layered.

For example, you might decide that your turn-based strategy game will have several different types of terrain: ocean, plains, forest, hills, and mountains. These would become your basic terrain types. In addition, you might want to have rivers and roads connecting various map squares. Roads and rivers would be contained in different layers. Also, you'll undoubtedly want to have cities and units on the map, and this can add even more layers. To accomplish all this layering, you might have a struct like the following to describe your tilemap areas:

```
struct TILEMAPSQUARE
{
        char BasicTerrain;//0=ocean;1=plains;2=forest;3=hills;4=mountains;
        unsigned char RoadFlags;//bit 0=north; bit 1=northeast; bit 2=east; etc.
        unsigned char RiverFlags;//bit 0=north;bit 1=east;bit2=south;bit3=west
        UNIT* Unit;//pointer to a unit
};
```

I think you get the idea. The more rich the world, the more complicated becomes the map structure. For now, we will stick with simplistic tilemaps. We'll get into more complicated structures in later chapters.

Rendering a Tilemap

Storing a tilemap somewhere in an array is important. Doing so allows you to persist a world without having to hardcode it. With such a tilemap, you can save to and load from disk, and create an editor that allows you to modify the map. Creating an editor is a great idea; you can distribute it with your game so that your players can create their own levels if they wish, thus enhancing the replay value of your game. Take, for example, the popularity of *Civilization II*, which was written several years ago but is still played heavily. Entire Web sites are dedicated to modified tilesets and scenarios that can be played within the game.

Having said that, let's talk about how to render a tilemap, and then we'll make a simple map editor that uses a tileset from spritelib. I'll bring up and talk about some of the terms I mentioned earlier in this chapter.

Screen Space

First, we'll talk about screen space in more depth. Screen space is nothing more than a rectangle describing the play area shown on-screen. This could be the entire screen, or it could be a smaller portion. Most modern games have some sort of status bar on the side or bottom of the screen, so quite often screen space is smaller than the entire screen.

For the editor that we will be making, let's use 800×600×16 mode. We will use a 600×600 area for editing the map on the left side of the screen, leaving 200×600 on the right for tile selection. The tiles we will be using are 32×32. The tileset is shown in Figure 10.15.

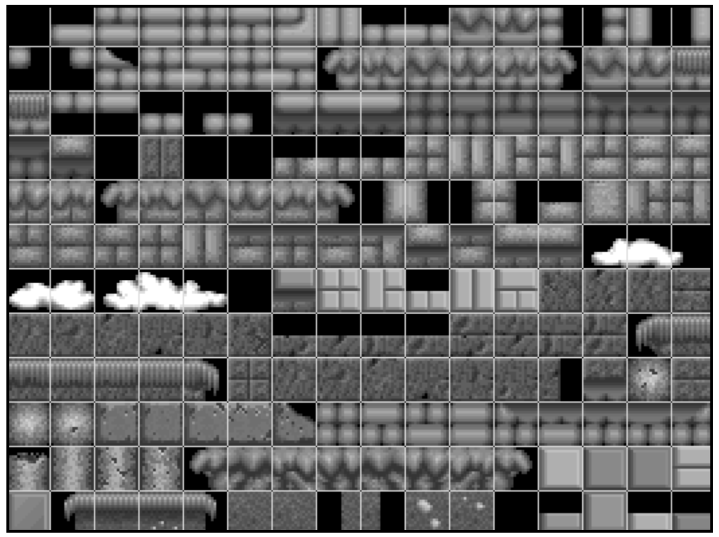
Figure 10.15

Tileset for the editor

Of course, neither 200 nor 600 is evenly divisible by 32. 200/32=6.25, and 600/32=18.75, leaving extra pixels. For this reason, there will be borders around both the editing panel and the tile selection panel, as shown in Figure 10.16. This makes the map panel 576×576 (or 18 tiles by 18 tiles), and the tile selection panel 192×576 (or 6 tiles by 18 tiles).

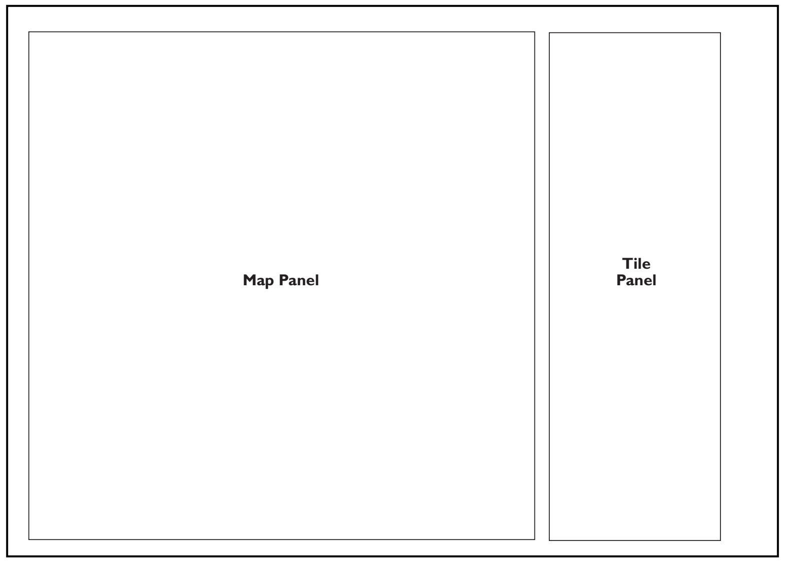

Figure 10.16

Layout of the map editor

You want your map panel centered within the 600×600 rectangle, and you want the tile selection panel centered within the 200×600 rectangle on the right. This will give your map panel `RECT` the value of (12,12)–(588,588) and your tile selection `RECT` the value of (12,604)–(796,588). This gives you not one, but two screen spaces. In the map panel, draw the current representation of the map based on your map array, which contains indices into the tileset (make the tilemap 18×18 so that it conveniently fits). In the tile selection panel, draw all the tiles, in order, and outline the one that is currently selected.

But now you have more tiles in the set than will fit in the tile selection box. You can fit 6×18 tiles (108 tiles), but you have 192. In order for the editor to be any good, you must either reduce the number of tiles in the set—something you don't want to do—or make it so that all the tiles can be selected by allowing some sort of scrolling mechanism. This is a better solution. You may, at some point, want to handle a variably-sized tileset, so not locking yourself into a fixed-size tileset is wise.

World Space and View Space

You have already decided to have an 18×18 tile grid, and this will be the total of your world space. Since each tile is 32*32, this makes the pixel measurement of world space 576×576. Since you are making world space 0-based, the world space `RECT` is (0,0)–(576,576).

Your view space is based on your screen space. Since screen space spans from (12,12)–(588,588), you simply must subtract (12,12) from each coordinate pair to determine your view space. This makes view space 0-based, which makes conversion from one space to another much easier. The point (12,12) is called the screen-to-view anchor.

Upper Left:
Screen coordinate (12,12)
Minus anchor (12,12)
Combine (12–12,12–12)
Solve (0,0)

Lower Right:
Screen coordinate (588,588)
Minus anchor (12,12)
Combine (588–12,588–12)
Solve (576,576)

Conveniently, your view space `RECT` works out to be (0,0)–(576,576), which is exactly the same as your world space `RECT`, meaning that no conversion is necessary to go from world to view space. So, to convert from world to screen space, simply add the coordinate (12,12). To do the reverse, subtract (12,12).

A Simple TileMap Editor

Load up IsoHex10_3.cpp. This example demonstrates what we've been talking about for the last several pages. It sets up a map panel and a tile panel. The map panel is your screen space for the tilemap. The tile panel shows the variety of tiles that you can place in the tilemap. Figure 10.17 shows sample output for this example.

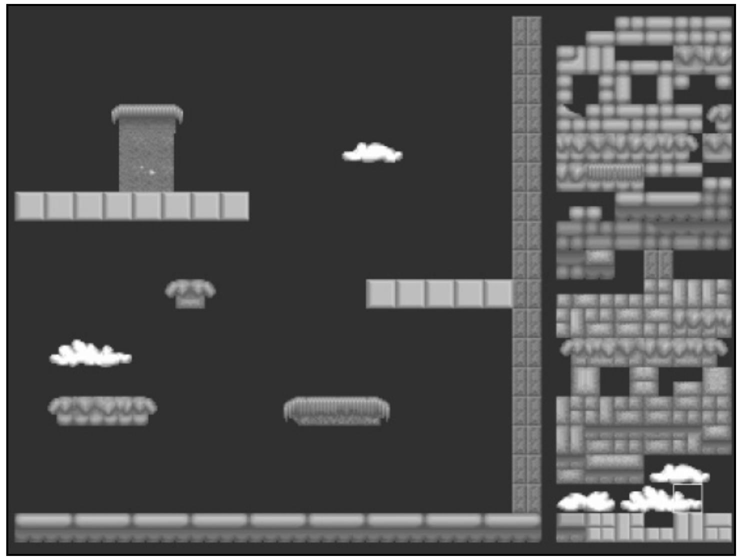

Figure 10.17

A simple `TileMap` editor

The controls for this example are rather simple, and the features rather slim. Clicking anywhere in the map panel puts the selected tile there. Clicking in the tile panel selects a new tile. Clicking above or below the tile panel scrolls the tile panel up or down. All in all, this example is pretty spartan. It doesn't save, it doesn't load, it doesn't really do much except let you play with the tileset. Still, I think it's a pretty good example of what a tilemap editor looks like at its very core. Let's take a look at how it works.

Constants

First, I made a number of constants to keep track of the sizes in the editor. Quite a few of them are dependent on other constants.

```
//map and tile constants
const int TILEWIDTH=32;
const int TILEHEIGHT=32;
const int MAPWIDTH=18;
const int MAPHEIGHT=18;
//panels
const int MAPPANELX=12;
const int MAPPANELY=12;
const int MAPPANELWIDTH=MAPWIDTH*TILEWIDTH;
const int MAPPANELHEIGHT=MAPHEIGHT*TILEHEIGHT;
const int TILEPANELX=604;
const int TILEPANELY=12;
const int TILEPANELCOLUMNS=6;
const int TILEPANELROWS=18;
const int TILEPANELWIDTH=TILEPANELCOLUMNS*TILEWIDTH;
const int TILEPANELHEIGHT=TILEPANELROWS*TILEHEIGHT;
```

Globals

Besides our usual globals (window handle, our DirectDraw pointer, and our primary and back surfaces), there are a few extras with which to keep track of the state of the editor.

```
//tileset
CTileSet tsTileSet;
//tilemap
int iTileMap[MAPWIDTH][MAPHEIGHT];
//tile selection
int iTileTop=0;
int iTileSelected=0;
```

The `tsTileSet` variable contains the tileset you'll be using. `iTileMap` is the array in which you contain your tilemap. The `iTileTop` and `iTileSelected` variables are for managing the tile selection panel. `iTileSelected` keeps track of what tile is currently selected for drawing, and `iTileTop` tracks what tile is shown at the top of the tile selection panel.

Set up and Clean up

The changes to `Prog_Init` are minor. You set up DirectDraw, load your tileset, and clear out your tilemap. I won't list the function's contents here. In `Prog_Done`, there are effectively no changes, since you neither have to deallocate the tilemap nor destroy the tileset.

The Main Loop

The main loop itself (`Prog_Loop`) does virtually nothing. It delegates to `ShowMapPanel` and `ShowTilePanel` and then performs a flip.

ShowMapPanel

This function has no parameters, returns no value, and carries out two tasks. The first task is clearing out the entire map panel with black. The second is looping through all the tiles in the tilemap and putting them onto the map panel.

```
void ShowMapPanel()
{
        //clear out map panel
        //set up fill rect
        RECT rcFill;
        SetRect(&rcFill,MAPPANELX,MAPPANELY,MAPPANELX+MAPPANELWIDTH,MAPPANELY+MAP-
PANELHEIGHT);
        //set up ddbltfx
        DDBLTFX ddbltfx;
        DDBLTFX_ColorFill(&ddbltfx,0);
        lpddsBack->Blt(&rcFill,NULL,NULL,DDBLT_WAIT | DDBLT_COLORFILL,&ddbltfx);
        //loop through map
        for(int mapy=0;mapy<MAPHEIGHT;mapy++)
        {
                for(int mapx=0;mapx<MAPWIDTH;mapx++)
                {
                        //put the tile
                        tsTileSet.PutTile(lpddsBack,MAPPANELX+mapx*TILEWIDTH,
                                MAPPANELY+mapy*TILEHEIGHT,iTileMap[mapx][mapy]);
                }
        }
}
```

ShowTilePanel

ShowTilePanel is responsible for displaying all of the tiles in the tile panel and for placing a white box around the currently selected tile.

```
void ShowTilePanel()
{
        //clear out map panel
        //set up fill rect
        RECT rcFill;
        SetRect(&rcFill,TILEPANELX,
TILEPANELY,TILEPANELX+
TILEPANELWIDTH,TILEPANELY+
TILEPANELHEIGHT);
        //set up ddbltfx
        DDBLTFX ddbltfx;
        DDBLTFX_ColorFill(&ddbltfx,0);
        lpddsBack->Blt(&rcFill,NULL,NULL,
DDBLT_WAIT | DDBLT_COLORFILL,&ddbltfx);
        //set tile counter to first tile
        int tilenum=iTileTop;
        //loop through columns and rows
        for(int tiley=0;tiley<TILEPANELROWS;tiley++)
        {
                for(int tilex=0;tilex<TILEPANELCOLUMNS;tilex++)
                {
                        //check for tilenum's existence in tileset
                        if(tilenum<tsTileSet.GetTileCount())
                        {

tsTileSet.PutTile(lpddsBack,TILEPANELX+tilex*TILEWIDTH,TILEPANELY+tiley*TILE-
HEIGHT,tilenum);
                                //check for selected tile
                                if(tilenum==iTileSelected)
                                {
                                        //grab the dc
                                        HDC hdc;
                                        lpddsBack->GetDC(&hdc);
                                        //calculate outline rect
                                        RECT rcOutline;
SetRect(&rcOutline,TILEPANELX+
tilex*TILEWIDTH,
TILEPANELY+
tiley*TILEHEIGHT,
TILEPANELX+
tilex*TILEWIDTH+
TILEWIDTH,
TILEPANELY+
tiley*TILEHEIGHT+
TILEHEIGHT);
                                        //select a white pen into dc
                                        SelectObject(hdc,
(HPEN)GetStockObject(WHITE_PEN));
                                        //place selection rectangle

MoveToEx(hdc,rcOutline.left,
rcOutline.top,NULL);

                                        LineTo(hdc,rcOutline.right-1,rcOutline.top);
                                        LineTo(hdc,rcOutline.right-1,rcOutline.bottom-
1);

                                        LineTo(hdc,rcOutline.left,rcOutline.bottom-1);
                                        LineTo(hdc,rcOutline.left,rcOutline.top);

                                        //release the dc
                                        lpddsBack->ReleaseDC(hdc);
                                }
                        }
                        //increase tile counter
                        tilenum++;
                }
        }
}
```

Accepting Input

The only topic left to cover is accepting input and making things happen. I'm only going to show the event handler for WM_LBUTTONDOWN, since the handler of WM_MOUSEMOVE is almost identical, and because of the sheer size of the handler.

In essence, the WM_LBUTTONDOWN handler takes the position of the mouse and places it in a POINT variable called ptMouse. Then it sets up a series of RECTs—one for the map panel, one for the tile panel, one for the area above the tile panel, and one for the area below the tile panel. It checks to see if the mouse is within these RECTs, and if it is, it carries out the appropriate action: place a tile if within the map panel, select a tile if within the tile panel, scroll the tile panel up if above or down if below.

WM_MOUSEMOVE does mostly the same thing, except for the scrolling of the tile panel if above or below.

```
case WM_LBUTTONDOWN:
       {
              //point to contain mouse coords
              POINT ptMouse;
              ptMouse.x=LOWORD(lParam);
              ptMouse.y=HIWORD(lParam);
              //RECT used for zone checking
              RECT rcZone;
              //other variables
              int mapx=0;
              int mapy=0;
              int tilex=0;
              int tiley=0;
              int tilenum=0;
              //check the map panel
       SetRect(&rcZone,MAPPANELX,MAPPANELY,
MAPPANELX+MAPPANELWIDTH,
MAPPANELY+MAPPANELHEIGHT);
              if(PtInRect(&rcZone,ptMouse))
              {
                     //in map panel
                     //calculate what tile mouse is on
                     mapx=(ptMouse.x-MAPPANELX)/TILEWIDTH;
                     mapy=(ptMouse.y-MAPPANELY)/TILEHEIGHT;
                     //change map tile to currently selected tile
                     iTileMap[mapx][mapy]=iTileSelected;
                     return(0);//handled
              }
              //check the tile panel
              SetRect(&rcZone,TILEPANELX,TILEPANELY,
TILEPANELX+TILEPANELWIDTH,
TILEPANELY+TILEPANELHEIGHT);
              if(PtInRect(&rcZone,ptMouse))
              {
                     //calculate which tile was selected
                     tilex=(ptMouse.x-TILEPANELX)/TILEWIDTH;
                     tiley=(ptMouse.y-TILEPANELY)/TILEHEIGHT;
```

```
                     tilenum=iTileTop+tilex+tiley*TILEPANELCOLUMNS;
                     //check for valid tile
                     if(tilenum<tsTileSet.GetTileCount())
                     {
                            //assign current tile
                            iTileSelected=tilenum;
                     }
                     return(0);//handled
              }
              //scroll tileset up
       SetRect(&rcZone,TILEPANELX,0,TILEPANELX+
TILEPANELWIDTH,TILEPANELY);
              if(PtInRect(&rcZone,ptMouse))
              {
                     //check if we can scroll up
                     if(iTileTop>0)
                     {
                            //scroll up
                            iTileTop-=TILEPANELCOLUMNS;
                     }
              }
              //scroll tileset down
              SetRect(&rcZone,TILEPANELX,TILEPANELY+
TILEPANELHEIGHT,TILEPANELX+
TILEPANELWIDTH,600);
              if(PtInRect(&rcZone,ptMouse))
              {
                     //check if we can scroll down
if((iTileTop+TILEPANELCOLUMNS)<
              tsTileSet.GetTileCount())
                     {
                            //scroll up
                            iTileTop+=TILEPANELCOLUMNS;
                     }
              }
              return(0);//handled
       }break;
```

A Few Words about the TileMap Editor

Even though the sample map editor doesn't do much, it does illustrate important points about all map editors. Just about every map editor I've made or used includes something similar to the map panel and something similar to the tile panel (although usually with a more obvious way of scrolling through the tileset).

A Tile-Based Example: Reversi

Now that we've delved a bit into the tile-based world, let's put this knowledge into practice. The first example I'd like to show you is a game called Reversi. (It's also called Othello, but Othello is trademarked by Milton Bradley, so we'll call ours Reversi.)

The basic idea of Reversi is pretty simple. In case you aren't familiar with the game or the rules, here's a brief breakdown: the game pieces are a board, divided into an 8×8 grid of 64 squares, and at least 64 two-sided pieces of contrasting color (usually black and white). At the beginning of the game, the center four squares are filled with pieces, two black and two white, as shown in Figure 10.18.

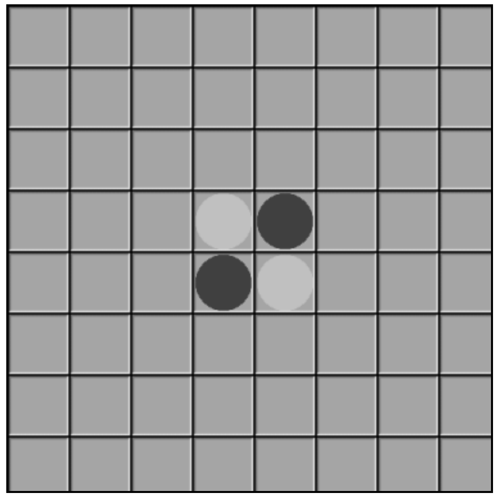

Figure 10.18

The Reversi board at the beginning of play

Two players alternate taking turns placing a single piece on the board and capturing any opposing pieces that they outflank. To outflank means to have one piece of your color on each end of a horizontal, vertical, or diagonal row of your opponent's pieces. You cannot outflank across your own pieces or across open squares. If on a player's turn there is no valid square on which he can play a piece and outflank his opponent, he forfeits that turn. Play progresses until no valid moves for either player are left (usually this happens when the board is full, although it can happen earlier).

Having said that, let's make the game.

Designing Reversi

I have Milton Bradley's Othello sitting on my game shelf, so I looked to that to model this game. The board is green with a black border separating the squares. Two cells in from the corners, there is a small square on the junction of the black lines, apparently to separate the sides and corners from the middle of the board. The pieces are double-sided and two-colored, with white on one side and black on the other.

I wanted to have some sort of method with which to highlight the possible squares to which the player can move on his turn, so I also made yellow versions. I wanted an animated "flipping over" of the pieces, so I made a sequence of ellipses to show that. Figure 10.19 shows the tileset I came up with for this game. You can also find it in the source code for this chapter, under the name IsoHex10_4.bmp. I used magenta instead of black as the transparent color. (Originally, I considered having a black piece instead of the dark gray that I later settled on.)

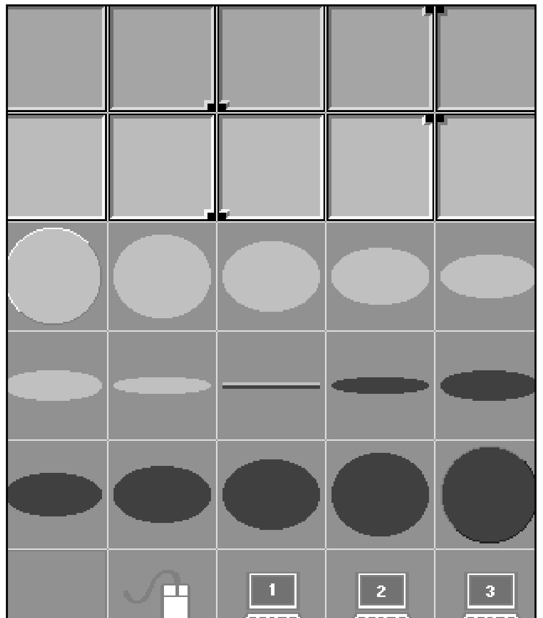

Figure 10.19

Tileset for Reversi

The first row of tiles is the nonhighlighted version of a board background tile. The second row is the highlighted version. Rows three through five are the animation sequence for the piece flip, with the actual pieces for both sides on opposite ends of the sequence. The last row consists of extra graphics I needed to finish up the UI. There is a red square to represent the last move made, and four icons to show the AI level chosen for the players.

AI Levels

I decided on four levels of AI for this example (none of them are very difficult to beat). These levels are represented by constants defined in the source.

```
//ai levels
const int AI_HUMAN=0;
const int AI_RANDOM=1;
const int AI_GREEDY=2;
const int AI_MISER=3;
const int AI_COUNT=4;
```

Table 10.4 explains these AI levels.

Table 10.4 AI Levels and Their Tactics

Level	Tactic
AI_HUMAN	None. Waits for input from the mouse.
AI_RANDOM	Picks a random valid move.
AI_GREEDY	Picks the valid move that will give it the greatest score.
AI_MISER	Picks the valid move that best limits the opponent's movement.

NOTE

AI_COUNT is not an AI level, but rather a constant to keep track of the number of levels that exist, in case you later want to add more AI levels.

Game States

As with all games, there are a number of major game states in which Reversi might dwell at any given time. I was able to reduce it to only five states.

```
//game states
const int GS_NONE=-1;
const int GS_WAITFORINPUT=0;
const int GS_NEWGAME=1;
const int GS_NEXTPLAYER=2;
const int GS_FLIP=3;
```

Table 10.5 explains these states.

Table 10.5 Reversi Game States

Game State	Meaning
GS_NONE	A neutral state. The board is drawn, but no other action takes place.
GS_WAITFORINPUT	If the current player is computer-controlled, a move will be made. Otherwise, it waits for mouse input.
GS_NEWGAME	Sets up a new game
GS_NEXTPLAYER	Checks for game over. If the game is not over, it selects the next player.
GS_FLIP	In this state, the pieces captured during this turn are taken through the animation sequence.

Tile Information Structure

Reversi may seem like a simple board game, but the struct that keeps track of the tile information is a little more complicated than just a simple array of integers.

```
//tile information structure
struct REVERSITILE
{
       int iTileNum;//base tile number for square
       bool bHilite;//hilited, or not hilited
       int iPiece;//piece occupying square
       bool bLastMove;//last move made
};
```

iTileNum

This member keeps track of the background and specifies one of the first five tiles of the tileset. Most squares contain tile zero, but a few contain the others. I could have easily just used tile zero for the entire board, but that would have been boring.

bHilite

When the current player can make a valid move on a given square, `bHilite` is true. If the square is not a valid move for the player, `hHilite` is false. `bHilite`, when used in conjunction with `iTileNum`, provides the background tile. When `bHilite` is true, 5 is added to `iTileNum`.

iPiece

This member has four meaningful values: `PIECEEMPTY(-1)`, `PIECEBLACK(0)`, `PIECEWHITE(1)`, and `PIECETRANSIT(2)`. The empty, black, and white are self-explanatory. The transit piece is for use with the `GS_FLIP` state. It specifies which pieces undergo the animation sequence.

bLastMove

Only one square at a time will ever have `bLastMove` set to `TRUE`. `bLastMove` specifies that the red rectangle (tile 25 of the tileset) is to be shown over the background, thus indicating that the square was the most recent move. Keeping track of this is not absolutely necessary, but I find it helpful when playing the game.

Score Indication

I wanted to have a score indication that did not require a font to implement. I could have used some extra tiles for the numerals 0 through 9 in the tileset, but I just didn't like that idea. Instead, I decided to use vertical stacks of the pieces alongside the board. Both stacks are on the left side of the board, so they can easily be compared to see who is winning.

AI Level Control

I didn't want to make a configuration screen, so I had to work in some sort of AI level control right on the screen itself. What I came up with was to put two of the colored pieces in the bottom-left corner (aligned with the score stacks), and I would blit icons representing what AI levels controlled which color. The icons are from the wingdings font, but I colored them in to make them look better.

Implementation of Reversi

With the design in mind, here's some of the implementation detail for Reversi. Because of space concerns, I can't get into every minute detail, but the full source code can be found in IsoHex10_4.cpp. I'm going to concentrate on the main game loop (`Prog_Loop`) and break it down by game state.

Major Global Variables

Reversi uses full-screen DirectDraw, set to an 800×600×16 resolution. The major global variables are shown next.

Your basic run-of-the-mill `IDirectDraw7` pointer:

```
//IDirectDraw7 Pointer
LPDIRECTDRAW7 lpdd=NULL;
```

A primary surface and the attached back buffer:

```
//surfaces
LPDIRECTDRAWSURFACE7 lpddsMain=NULL;
LPDIRECTDRAWSURFACE7 lpddsBack=NULL;
```

The main tileset to contain all of the graphics used:

```
//tileset
CTileSet tsReversi;
```

The main board and a temporary storage area:

```
//the board
REVERSITILE Board[8][8];
//backup board
REVERSITILE BackUpBoard[8][8];
```

A variable to keep track of the current player:

```
//current player
int iPlayer=0;
```

An animation counter for use during `GS_FLIP`:

```
//counter for animated "flipping" of pieces
int iAnimation=0;
```

An array to keep track of what AI controls each color:

```
//ai level for the players
int iAILevel[2];
```

The main game state:

```
//gamestate
int iGameState=GS_NONE;
```

All Game States

Regardless of game state, a certain amount of code runs each loop. This code prepares a new frame for the game and then displays it.

```
//clear out back buffer
DDBLTFX ddbltfx;
DDBLTFX_ColorFill(&ddbltfx,0);
lpddsBack->Blt(NULL,NULL,NULL,DDBLT_WAIT | DDBLT_COLORFILL,&ddbltfx);
//***OMITTED CODE***
//show the board
ShowBoard();
//show the scores
ShowScores();
//show players
ShowPlayers();
//flip
lpddsMain->Flip(NULL,DDFLIP_WAIT);
```

This bit is pretty simple. First, you clear out the back buffer, and then you draw the board, draw the scores, draw the AI levels, and finally flip the page. It's a pretty to-the-point snippet. You can take a look at the constituent functions in the source code if you're interested. The following sections offer a brief rundown of the major function calls.

ShowBoard

This function loops through all of the board squares and follows approximately these steps:

1. Based on `iTileNum` and `bHilite` for this board square, determine which tile to use as the background tile.
2. Determine what piece, if any, is resting on this square. If it is `PIECEBLACK` or `PIECEWHITE`, show the appropriate tiles. If it is `PIECETRANSIT`, determine what tile to show based on the global variable `iAnimation`.
3. If this square has `bLastMove` set, put the red square on top.

ShowScores

This function shows the scores for each color, representing the score with a vertical stack of pieces. For each piece on the board, `ShowScores` renders one piece. The first piece is rendered with the top of the piece at y=0, and y increases by 4 for each additional piece on the board. This allows a nice, easy way to tell who is winning while avoiding numerals.

ShowPlayers

This function shows the AI levels of both colors in the bottom-left corner of the screen. A black piece sits next to a white piece. On top of these pieces the function renders an icon that represents the AI level for that color. A mouse represents a human player, and computers with the numerals 1, 2, and 3 represent the three levels of computer AI.

GS_NONE

This game state does almost nothing. In fact, there is no case for it in the `iGameState` switch in `Prog_Loop`. Only in the `WM_LBUTTONUP` event handler does `GS_NONE` get a mention. If the board is clicked on while in `GS_NONE`, the game moves to `GS_NEWGAME`.

```
case WM_LBUTTONUP:
        {
                //grab mouse position
                POINT ptMouse;
                ptMouse.x=LOWORD(lParam);
                ptMouse.y=HIWORD(lParam);
                //test rectangle
                RECT rcTest;
                //get tile width and height
                int iTileWidth=tsReversi.GetTileList()[0].rcSrc.right-
                        tsReversi.GetTileList()[0].rcSrc.left;
```

```
          int iTileHeight=tsReversi.GetTileList()[0].rcSrc.bottom-
                tsReversi.GetTileList()[0].rcSrc.top;
          //calc board rect
          SetRect(&rcTest,(400-iTileWidth*4),
                (300-iTileHeight*4),(400+iTileWidth*4),
                (300+iTileHeight*4));
                //point on board?
          if(PtInRect(&rcTest,ptMouse))
          {
          //***CODE OMITTED
                //if a game is over, start a new game by clicking on the
                board
                if(iGameState==GS_NONE)
                {
                      iGameState=GS_NEWGAME;
                }
          }
          //***CODE OMITTED***
    }break;
```

GS_NEWGAME

GS_NEWGAME starts a new game, and is actually one of the simpler game states. First, it makes a call to SetUpBoard, which does all of the reinitialization necessary to start out with a clean board. Then it sets the player to PLAYERTWO and sends the game into GS_NEXTPLAYER. I could have done this another way, by setting iPlayer to PLAYERONE and sending it into GS_WAITFORINPUT.

```
case GS_NEWGAME:
    {
          //clear the board
          SetUpBoard();
          //set player
          iPlayer=PLAYERTWO;
          //change game state
          iGameState=GS_NEXTPLAYER;
    }break;
```

GS_WAITFORINPUT

This game state is the central game state. All AI moves are done here. When the game first enters GS_WAITFORINPUT, it checks the current player's AI level. If AI_HUMAN is indicated, the game does nothing. If it is a computer AI (AI_RANDOM, AI_GREEDY, or AI_MISER), it calls the appropriate AI function.

```
case GS_WAITFORINPUT:
    {
          //make move appropriate to the AI
          switch(iAILevel[iPlayer])
          {
          case AI_RANDOM:
                {
                      MakeRandomMove(iPlayer);
                }break;
          case AI_GREEDY:
                {
                      MakeGreedyMove(iPlayer);
                }break;
          case AI_MISER:
                {
                      MakeMiserMove(iPlayer);
                }break;
          }
    }break;
```

Note that the AI level of AI_HUMAN isn't even represented in this snippet. That is because all of the AI_HUMAN stuff for GS_WAITFORINPUT is handled in the WM_LBUTTONUP event handler. (AI and GS and WM. . . oh my!)

```
//***CODE OMITTED***
//point on board?
if(PtInRect(&rcTest,ptMouse))
{
    //if we are waiting for input and the ai is "human," check for inside the
board
    if((iGameState==GS_WAITFORINPUT) &&
          (iAILevel[iPlayer]==AI_HUMAN))
    {
          //find board position
          int BoardX=(ptMouse.x-rcTest.left)/iTileWidth;
          int BoardY=(ptMouse.y-rcTest.top)/iTileHeight;
```

```
        //check for a valid square
        if(ValidMove(iPlayer,BoardX,BoardY))
        {
        //make the move
            MakeMove(iPlayer,BoardX,BoardY);
            SetLastMove(BoardX,BoardY);
            iGameState=GS_FLIP;
        }
    }
    //***CODE OMITTED*** (the GS_NONE check)
}
//***CODE OMITTED***
```

GS_FLIP

After a move has been made, the newly captured pieces are not set to the color of the player who captured them. Instead, they are changed to `PIECETRANSIT`, and `GS_FLIP` is the game state responsible for making sure that the animation sequence for capturing these pieces is shown.

```
case GS_FLIP:
    {
        switch(iPlayer)
        {
        case PLAYERTWO:
            {
                if(iAnimation==0)
                {
                    FinishMove(iPlayer);
                    iGameState=GS_NEXTPLAYER;
                }
                else
                {
                    iAnimation—;
                }
            }break;
        case PLAYERONE:
            {
                if(iAnimation==14)
                {
                    FinishMove(iPlayer);
                    iGameState=GS_NEXTPLAYER;
                }
                else
                {
                    iAnimation++;
                }
            }break;
        }
    }break;
```

The main purpose of `GS_FLIP` is to modify `iAnimation`, which controls what part of the animation sequence you are on. When it is `PLAYERONE`'s turn, `iAnimation` starts at 0 and is incremented until it hits 14, at which point the move finishes (by a call to `FinishMove`, which changes all `PIECETRANSIT`s to a color's piece). Similarly, on `PLAYERTWO`'s turn, `iAnimation` starts at 14 and moves backwards until it hits 0. In either case, after `GS_FLIP` is finished, the game moves into `GS_NEXTPLAYER`.

GS_NEXTPLAYER

After a move has been completed, this game state checks to see if the game is over or sets the next active player. If the game is over (there are no valid moves for either player), it sends the game into `GS_NONE`. If the game is not over, it checks to see if the opposing player has a valid move. If the opposing player does not have a valid move, it goes to `GS_WAITFORINPUT` without changing the player. If the opposing player does have a valid move, it sets the current player to the opposing player and moves into `GS_WAITFORINPUT`.

```
case GS_NEXTPLAYER:
    {
        //scan for moves
        ScanForMoves(iPlayer);
        //if no more valid moves, game over
        if((!AnyValidMoves(PLAYERTWO)) && (!AnyValidMoves(PLAYERONE)))
        {
            iGameState=GS_NONE;
        }
        else
        {
            //find if opponent has any moves
            if(AnyValidMoves(1-iPlayer))
            {
                iPlayer=1-iPlayer;
            }
```

```
                    //scan for moves by current player
                    ScanForMoves(iPlayer);

                    //get next move
                    iGameState=GS_WAITFORINPUT;
                }
            }break;
```

Miscellaneous Actions

Before we complete our treatment of Reversi, I have a few last things left that I want to point out.

- **Changing AI Levels.** During every loop, the current AI levels are shown at the bottom left of the screen. You can change the level by clicking on the indicators. Each time you click, you increase the AI level by 1. Clicking on the highest level brings you back to the lowest level (`AI_HUMAN`).
- **Keyboard Controls.** Esc exits the program, no matter what game state you are in. F2 starts a new game, no matter what game state you are in.

Final Words on Reversi

This simple little game of Reversi is far from complete. Yes, it is fully functional and playable, but it lacks any extras. Just like the Breakout game in the preceding chapter, I'm leaving it for you to finish. Here's a brief list of features I think it needs:

- A title screen
- Some sort of "bells and whistles" when you win
- Sound/music

And I'm sure you'll come up with 50 ways to improve the program. Have fun with it.

Summary

In this chapter, you took a step into a larger world. You explored the power that graphical tiles can give you. I went into great detail on the topic of tileset management, and for good reason. From here on out, just about everything you do will be done using the `CTileSet` class, in some fashion or another.