This chapter is the last major discussion of working with *non-player characters* (NPCs) in the book, so my goal here is to develop a working combat system within the game. This requires the use of a complete set of combat animation sequences for the player and NPCs. Fortunately, the sprites available from Reiner’s Tile-sets (http://www.reinerstileset.de) also include the attack animations that you use in this chapter. Combat requires another layer of logic added to the state engine that controls the NPCs. Although higher, more advanced interaction with the player should make the game more realistic, at this point a simple state-based reactionary system has to be used, where the NPCs fight back when attacked by the player.

Here is a breakdown of the major topics in this chapter:

- State-based combat
- Dealing with the player’s death
- Combat animations
- Introducing the Skeleton Knight
- Engaging in combat
- Managing the player’s state
- Managing the NPC states

**Contemplating the Combat System**

The Celtic Crusader project, as of the last chapter, is basically a template game that has most of the functionality you need to actually create a *role-playing game* (RPG), but is lacking most of the finer details. One thing you can do with a basic dialog system (which
was started in the last chapter) is move toward incorporating the game's story into the gameplay. You can add subtle clues in the dialog with NPCs (most notably, noncombatants) that lead the player toward a goal. For instance, suppose you follow the classic “save the girl” plot. You might use this as one of several subquests to lead the player from one corner of the world to the other, seeking out maidens to rescue. These subquests not only progress the storyline of the game, but also provide the player with much-needed experience points.

**note**

This chapter does not provide the complete code changes needed to add the combat system. The key source code routines are explained, but the changes to the project are too numerous to list, so I encourage you to just load up the project from the CD-ROM in \sources\chapter18 and follow along while looking at the completed project.

There is just an enormous amount of detail that must be put into even the simplest of RPGs, so what I’m getting at is that I don’t want you to expect this game to be a finished product by the time you are done with this book. As you might imagine, the size and scope of this book is nowhere near enough to completely build the game (something that I deeply regret, but cannot help), and I do not want to add functionality that I have not explained in each chapter. What I’m providing you here is a working RPG engine with all the tools you need to complete it. By describing it so, I want you to realize that the creative work, the use of your imagination, and the finishing touches should come from you, because this is your game. I believe that with the addition of the combat system in this chapter, you have what you need to turn out a complete game.

**State-Based Combat**

The previous chapter developed the ability for the player to have encounters with NPCs, which is an important first step in the game’s NPC interaction. From this point, you can engage the NPCs in dialog or combat, and the game responds appropriately. Every NPC should behave, in some manner, to attack by the player. A higher level of behavior over the NPCs is also needed to turn this skeleton game into a polished game, a system of behavior that causes NPCs to seek out and engage the player, rather than always responding to the player. At the very least, you can add the ability for NPCs to fight back.

**Fighting Back**

The goal of this chapter is to add combat animations in such a way that it is easy for you to add new NPCs to the game without requiring much extra work. The hostile NPCs need attack animations, while the peasantry do not, so if the player attacks a peasant or any other nonfighting NPC, then you have to add behavior that causes the character to run
away or die, depending on your style. (I recommend adding a state that causes nonfighting NPCs to flee.)

**Respawning NPCs**

When you are fighting with an NPC and kill that character, there should be a death animation. These are not always possible in every case, due to a limited number of sprites. You are limited overall by the availability of artwork, without which you have to get creative with your sprites. Rather than dealing with a whole slew of death animations for each NPC, I have seen some games use the fade effect, where a character blinks out of existence or fades away. You might use the alpha color parameter in `DrawSprite` to cause a character to fade out of existence after dying, which requires some sort of death state.

The important thing is that you recycle your sprites in the game, which means recycling the NPCs. You don’t want the NPCs to just respawn at the same place every time, because then the player can see the spawning taking place (which seriously ruins the realism of the game). In addition, if a player learns where some of the NPCs are respawning on the map, he or she will be able to spawn camp (which refers to hiding out near a spawn point and killing new players that appear) and rack up a ridiculous amount of experience, which also ruins the game.

**Simulating Damage**

One aspect of combat you need is some sort of status display showing the hero’s health and other attributes. I think it is a good idea to use the main game window for chatting and combat, which means that most of the bottom toolbar is unused. I recommend using it to display the hero’s attributes, including health (which is calculated using strength and stamina).

The best way to show damage is to cause a character to flicker on the screen—and keep in mind, this is only my opinion based on my experience with RPGs. Some of the most successful RPGs used the flicker/fade method of showing hits. Since that can also be used for a death animation, it makes sense that enough hits cause a character to flicker out completely (which implies death). It also keeps you, the programmer, from having to keep track of dead bodies in the game world. Although a combat-focused game might benefit from showing the carnage of bodies on the ground, it requires a lot of extra work on your part. You basically have to keep all of those sprites in memory just to draw their dead bodies and then create new sprites to respawn the NPCs. This is all just a lot of unnecessary work; the player is plowing through a lot of enemies in the game, anyway. The flicker/fade technique works well overall.
Attack Rolled Against Defense

What really happens when you attack another character in the game? That is the basis of the game's combat system and it has to do with each player's attributes, including weapon and armor class. Usually, the defender's defensive value is compared to the attacker's attack value, and a simulated “roll” of dice is made to determine if the attack even succeeded (before calculating damage).

If the attack value is less than the defense value, then basically you can do no damage to your opponent! So, say you are a new Warrior with an axe that does +10 damage, and you attack a level-10 Viking Berserker with 93 defense points. What happens in this situation? You can stand there and bang against this fellow’s armor all day long with your pathetic little axe and do no damage to him whatsoever! If you don’t like this aspect of game play, maybe you should go back to playing Zelda. (Sorry, I actually love Zelda, especially Link To The Past, but it has a primitive combat system.) In a situation like this, you are helplessly outclassed by this character, who swiftly and easily kills you with a single blow.

This is called the to-hit roll and it adds a nice layer of realism to the game (as opposed to some games where just swinging your sword kills enemies nearby). Knowing that not every swing does damage requires you to use some tactics in your fighting method, and this gives players the ability to be somewhat creative in how they fight enemies. You can swing and run or swing several times in a row, hoping to get a hit. But in general, it’s a hit-or-miss situation (sorry, bad pun).

Many RPGs allow the player to equip modifiers such as rings and special weapons with bonuses for the to-hit value. These modifiers increase your chances of scoring a hit when you attack. Since this game is still a work in progress, I have not had a chance to talk with you about inventory items and equipping your character. This very challenging aspect of the game to program requires you to create an item editor program and use an array of items in the game to display items on the ground that the player can pick up. Items in the player’s inventory (which probably means the player’s backpack) also have modifiers that the player can use in the game, so your forethought on item management is important. Not only is it essential for a good RPG, but working with miscellaneous items as well as different types of swords, shields, armor, helmets, and so on, is an extremely fun part of the game!

Factoring Weapon Values

After the to-hit roll determines that the player did hit his target, determine how much damage was done to the target. This is where the weapon attributes come into play. But wait—I haven’t even talked about weapons yet! Well, now is as good a time as any. Since the inventory system is not possible in this prototype game, I propose basing all attack values directly on the player’s attributes and using a fixed damage value for each character class. The Warrior should do more damage with his axe than a Mage does with his staff.
Default weapon damage values make the combat system functional until a proper inventory system is added to the game.

If the game features real items that you can give your character to use in combat, then it makes a big difference in the game play. For one thing, you can scatter treasure chests around the game world that contain unique quest items (like magical swords, shields, and armor), as well as valuable jewels and gold. These types of items are all modeled and available in the sprites provided by Reiner’s Tilesets. The artwork department is finished and it’s just a matter of adding this feature to the game.

**Dealing with the Player’s Death**

One drawback to combat is that you can die. It’s a cold, hard, truth, I realize, but it can happen. What should you do, as the game’s designer and programmer, when the player’s character (PC) dies? That is a tough decision that requires some thought and should be based on the overall design of your game. You might let the player save and load games, but that takes away from the suspension of disbelief. Remember that concept that I introduced to you back in Chapter 3, “Designing the Game”? You want the player to be completely immersed in the game and unaware of a file system, an operating system, or even of the computer. You want your players to be mesmerized by the content on the screen, and something as cheesy as a load/save feature takes away from that. I’ll admit, though, most players abuse the save/load game feature and complain if you don’t have one. After all, you want the player to be able to quit at a moment’s notice without going through any hassle. Let’s face it: Sometimes the real world asserts itself into the reverie you are experiencing in the game, and you have to quit playing.

But just for the sake of game play, what is the best way to deal with the PC’s death, aside from having a save/load feature? I recommend just respawning the PC at a nearby town at this point. You don’t want the player to get too frustrated with having to walk clear across the world again after dying, so respawning at the starting point is a bad idea. (Remember how big this world is!)

**Implementing the Combat System**

I have made some changes to the Celtic Crusader project that you find on the CD-ROM in \sources\chapter18. The player/hero code has been moved to a separate module called Hero.BAS and the main game loop has been cleaned up as a result. I have also added some new sprites to the game for this chapter.

**Combat Animations**

Before you can engage in combat, one might argue that you need a weapon first. Granted, the hero in this game has been carrying a sword around for quite a while. The problem is
that he doesn’t know how to use it, so it’s what you might call a decorative sword at present. What this hero needs is the ability to swing away at the bad guys, and that calls for some new animations!

Wait a second. I want to do something a little different this time. Take a look at one of the other character classes for this chapter. What do you say? I really like the Mage character’s artwork, so look into using the Mage in this chapter. He won’t have any magic spells and just swings his staff like a blunt weapon, but it is cool to see a different character this time.

**tip**

The fully prepared sprites for several character classes of the player are available on the CD-ROM in the \bitmaps folder, available in both walking and fighting animations.

First, like usual, I downloaded from Reiner’s Tilesets the character animations of the Mage character (which Reiner calls “staffstan”). Take a look at one of the character frames in Pro Motion shown in Figure 18.1.

After the eight animation strips have been saved (from the original bitmap images) by Pro Motion, I then combine the strips and set the background to pink in Paint Shop Pro, which you can see in Figure 18.2.

![Figure 18.1 The animated Mage character is being converted to an animation strip in Pro Motion.](image-url)
In addition to the walking animation, I need a combat animation of the Mage. I have also exported the animation strips for the Knight Hero character, but want to use the Mage Hero in the updated version of Celtic Crusader for this chapter. The combat animations are really good for the Mage sprite, with 13 frames for each direction for a total of 104 animation frames—just to swing the staff! See for yourself in Figure 18.3.

While I’m on the subject of combat animations, I’ve got the attack frames for the Viking ready to go in this chapter as well! Figure 18.4 shows this really cool character that I totally love; check out his huge battle axe! While you can’t tell from the figure here, this character has red hair and is a very imposing-looking figure (which is perfect for a Viking).

**Introducing the Skeleton Knight**

In addition to the attack animations for the Hero and Viking, I have added a skeleton to the mix to demonstrate how the game looks with different NPCs present. Figure 18.5 shows the Skeleton Knight walking animation in Pro Motion.

*Figure 18.2* The Mage animation strips are combined into a single bitmap file.
Chapter 18  ■  Engaging in Combat with NPCs

Figure 18.3 The combat animations for the Mage character.

Figure 18.4 A combat animation for the Viking Warrior character.
Since this chapter now includes the ability to engage in combat, the Skeleton Knight needs some attack animations. Figure 18.6 shows one of the attack frames for this character.

I had to combine the animation strips for the two versions of the Skeleton Knight into individual bitmap files using Paint Shop Pro. Again, this is technically something you can do in Pro Motion, but I find it more time consuming to insert frames into Pro Motion rather than just combining the images in Paint Shop Pro (shown in Figure 18.7).

**Engaging in Combat**

There are two basic things you need to do to allow the player to fight with NPCs:

- Make sure the player is close enough to an enemy to hit him.
- Make sure the player is facing an enemy while attacking.

If you can take care of these two problems, then you can create a combat system for the game. Tackle these two key problems in order. Figure 18.8 shows the general goal. You want to be able to acknowledge that a hit has occurred when the player is in attack mode and also facing the enemy.
Chapter 18 ■ Engaging in Combat with NPCs

Figure 18.6 The attack animation for the Skeleton Knight character.

Figure 18.7 The Skeleton Knight character animation strips are being combined in Paint Shop Pro.
The first thing you need to check on before you can handle a combat strike is whether the player is close enough to an enemy character to actually hit him. That is accomplished with a simple call to the Collision function (a technique that you learned back in Chapter 14, “Core Technique: Collision Detection”). If there is no collision between the two sprites, then there definitely can’t be an attack, and the swing misses! After determining if the two sprites are close enough for an attack, see if the attacker is at least facing the enemy. It is important that the player actually faces an enemy before you start to tally the attacks. Otherwise it’s possible to hit the enemy by just swinging a weapon anywhere in close proximity to him (using the earlier collision routine).

I wrote a function called IsFacing that returns True or False depending on whether one sprite is facing another sprite. This is also useful for pitting NPCs against each other, not just for the player.

Public Function IsFacing( _
    ByVal spr1 As TSPRITE, _
    ByVal spr2 As TSPRITE) As Boolean

Figure 18.8 Dealing damage to the enemy occurs only if you are facing the enemy character.
```vba
Dim n As Long
Dim a As point
Dim b As point

' are both sprites in range of each other?
If Not Collision(spr1, spr2) Then
    IsFacing = False
    Exit Function
End If

a.x = spr1.x + spr1.width / 2
a.y = spr1.y + spr1.height / 2
b.x = spr2.x + spr2.width / 2
b.y = spr2.y + spr2.height / 2

Select Case spr1.AnimSeq
    ' looking up
    Case 7, 0, 1
        If b.y < a.y Then IsFacing = True

    ' looking down
    Case 5, 4, 3
        If b.y > a.y Then IsFacing = True

    ' looking left
    Case 6
        If b.x < a.x Then IsFacing = True

    ' looking right
    Case 2
        If b.x > a.x Then IsFacing = True
End Select

End Function
```

**Managing the Player’s State**

After you have the combat code ready to go, there’s just one little problem: The source code written in the game so far just draws the walking version of the player. With combat, the player has to swing his weapon too. The main game loop has to be modified so that you can check the player’s state and then draw either the walking or the attacking animations based on what the player is doing.
I modified the game loop so that all the player update code is replaced with a single call to UpdateHero, which is listed here:

Public Sub UpdateHero()
    Dim state As String

    Select Case PlayerData.state

        Case HERO_STOPPED
            state = "STOPPED"
            DrawSprite heroImgWalk, heroSprWalk, C_WHITE

        Case HERO_WALKING
            state = "WALKING"

            'animate the walking hero
            If heroSprWalk.Animating Then
                AnimateSprite heroSprWalk
            End If

            'draw the walking hero
            DrawSprite heroImgWalk, heroSprWalk, C_WHITE

        Case HERO_ATTACKING
            state = "ATTACKING"

            'animate the attacking hero
            If heroSprAttack.Animating Then
                AnimateSprite heroSprAttack

            'done attacking? go back to walking
            If heroSprAttack.Animating = False Then
                PlayerData.state = HERO_STOPPED
            End If
            End If

            'draw the walking hero
            DrawSprite heroImgAttack, heroSprAttack, C_WHITE

            'check for a hit
            CheckForHits

    End Select

End Sub
Managing the NPC States

As you can see for yourself, there is a lot more to drawing the Hero’s sprite now that combat is involved. Before, it was easy because just one type of animation was being used—walking. Now, though, two different states have to be monitored and the correct sprite has to be animated. State engines are very helpful when you need to keep track of a complicated series of conditions in a game, because each condition is isolated from the rest, allowing you to write code for each condition separately.

Here is the new list of states being used for each NPC:

Public Enum NPCSTATES
    NPC_STOPPED = 0
    NPC_WALKING = 1
    NPC_PAUSED = 2
    NPC_TALKING = 3
    NPC_DYING = 4
    NPC_KILLED = 5
    NPC_ATTACKING = 6
End Enum

The arrays that keep track of character classes, images, and sprites have also been changed to accommodate the new combat system:

Public Const NUMCHARS As Long = 2
Public charWalk(NUMCHARS) As Direct3DTexture8
Public charAttack(NUMCHARS) As Direct3DTexture8
Public charClasses(NUMCHARS) As TCHARACTER

'unique data for each individual NPC
Public Const NUMNPCS As Long = 10
Public charStates(NUMNPCS) As TNPC
Public charWalkSpr(NUMNPCS) As TSPRITE
Public charAttackSpr(NUMNPCS) As TSPRITE
Checking for Attack Hits on NPCs

The section of code under the HERO_ATTACKING state includes a call to a subroutine called CheckForHits:

```vba
Public Sub CheckForHits()
    'this is temporary—replace with weapon attack value
    Const ATTACKVALUE As Long = 1
    Dim n As Long

    For n = 0 To NUMNPCS - 1
        If IsFacing(heroSprAttack, charWalkSpr(n)) Then
            AttackNPC charStates(n), ATTACKVALUE
            Exit For
        End If
    Next n
End Sub
```

CheckForHits looks at all of the NPCs in the game and then calls IsFacing on each one to see if the player is close to and facing the NPC. If these two conditions are met, then the player hits the NPC with the weapon swing. If no NPC is in range (in front of the player) then the swing doesn't hit anything! See how easy it is when a state engine is being used?

Doing Damage to an NPC

Now take a look at the AttackNPC subroutine that is called from the preceding routine you just looked at. This new routine is actually only called when the player has definitely hit an NPC. When this happens, the NPC’s health needs to be cut down by an appropriate amount, and he dies if health is 0! AttackNPC has some test code that prints a message above the player, and a message above the target NPC during an attack, to tell you that the game registered the hit. When the NPC’s health reaches 0, the state of the character is set to NPC_DYING.

```vba
Public Sub AttackNPC(ByRef target As TNPC, ByVal attack As Long)
    'fight back!
    target.state = NPC_ATTACKING

    'decrease health
    target.health = target.health - attack
    If target.health < 1 Then
        target.state = NPC_DYING
    End If
```
display a message to indicate the NPC was hit!
PrintText fontImg, fontSpr, _
    heroSprAttack.x, heroSprAttack.y, C_WHITE, _
    "Take that! (* & attack & " pts)"

make the target respond to the hit
Dim p As point
p.x = target.curpos.x - ScrollX
p.y = target.curpos.y - ScrollY
PrintText fontImg, fontSpr, _
    p.x, p.y, C_WHITE, _
    "Argh, I've been hit! (" & target.health & ")"

End Sub

Death Sequence
There is a death sequence where the NPC is frozen and fades into nothingness (a simple
way to show that the NPC has died). If this happens, then the NPC’s state takes over the
death, allowing your player’s code to continue without worrying about dealing with the
NPC’s resting place. The state engine in Characters.bas manages the state of the NPCs.
When the dying state has played out (using a simple counter to keep the faded body visi-
ble for a short time), then the state of the bad guy is set to NPC_KILLED. This state triggers
the calling of KillNPC, which respawns the character. Figure 18.9 shows an example of the
dying sequence for an NPC.
Public Sub KillNPC(ByRef dude As TNPC)
    Dim p As point

    p.x = PLAYERSTARTX * TILEWIDTH + Random(1000)
p.y = PLAYERSTARTY * TILEHEIGHT + Random(1000)

    With dude
        .startpos = p
        .curpos = p
        .SpeedDelay = 1
        .SpeedCount = 0
        .health = 20    'added in chapter 18
        .state = NPC_WALKING
    End With
    SetRandomDestination dude
End Sub
Moving the State-Based NPC

With all of these different states to handle walking, attacking, and dying, the code that moves and draws the NPCs has to be modified to take them into account. Here is the current MoveNPCs subroutine, which is called by the main game loop:

Public Sub MoveNPCs()
    Dim n As Long

    'loop through all of the NPCs and move them
    For n = 0 To NUMNPCS - 1

        Select Case charStates(n).state

            Case NPC_ATTACKING
                'stop attacking if the player leaves or if I'm dead...
                If charStates(n).health < 0 Then
                    charStates(n).state = NPC_STOPPED
                End If

Figure 18.9 This NPC's health has reached 0, so he is about to die.

Moving the State-Based NPC

With all of these different states to handle walking, attacking, and dying, the code that moves and draws the NPCs has to be modified to take them into account. Here is the current MoveNPCs subroutine, which is called by the main game loop:

Public Sub MoveNPCs()
    Dim n As Long

    'loop through all of the NPCs and move them
    For n = 0 To NUMNPCS - 1

        Select Case charStates(n).state

            Case NPC_ATTACKING
                'stop attacking if the player leaves or if I'm dead...
                If charStates(n).health < 0 Then
                    charStates(n).state = NPC_STOPPED
                End If

End If
If Not Collision(charWalkSpr(n), heroSprWalk) Then
    charStates(n).state = NPC_STOPPED
End If

Case NPC_TALKING
    FacePlayer n

Case NPC_PAUSED
    SetRandomDestination charStates(n)

Case NPC_WALKING
    MoveNPC n

Case NPC_STOPPED
    SetRandomDestination charStates(n)

Case NPC_DYING
    charStates(n).destpos = charStates(n).curpos
    charStates(n).health = charStates(n).health - 1
    If charStates(n).health < -100 Then
        charStates(n).state = NPC_KILLED
    End If

Case NPC_KILLED
    KillNPC charStates(n)

End Select
Next n
End Sub

**Drawing the State-Based NPC**

In addition to moving the NPCs differently based on state, the drawing code also has to take into account the character's state. Different sequences for the walking and attacking animations have to be accounted for in the draw routine. This is where the dying sequence takes place as well. Figure 18.10 shows a skeleton that has been dealt the fatal blow. When the state is NPC_DYING, the sprite is drawn using a gray color that renders the sprite with about 50-percent translucency. (The color is &H99FFFFFF, which has an RGB for white, but a 50-percent alpha or thereabouts.)
Public Sub DrawNPCs()
    Dim n As Long

    'loop through all of the NPCs and draw them
    For n = 0 To NUMNPCS - 1
        Select Case charStates(n).state
        Case NPC_ATTACKING
            DrawNPC n, C_RED
            charStates(n).state = NPC_WALKING
        Case NPC_TALKING
            DrawNPC n, C_WHITE
            charStates(n).state = NPC_WALKING
            If diState.key(KEY_SPACE) > 0 Then
                TalkToPlayer n
            End If
        End Select
    Next n
End Sub

Figure 18.10 Alpha blending is used to draw a sprite with partial translucency.
Chapter 18  ■  Engaging in Combat with NPCs

Case NPC_PAUSED
    DrawNPC n, C_WHITE
    charStates(n).state = NPC_WALKING

Case NPC_WALKING
    DrawNPC n, C_WHITE

Case NPC_STOPPED
    DrawNPC n, C_WHITE
    charStates(n).state = NPC_WALKING

Case NPC_DYING
    DrawNPC n, &H99FFFFFF
End Select

Next n
End Sub

The DrawNPCs routine calls on the more specific DrawNPC subroutine to do the actual work. This routine also checks the state to draw the attack animation. When you attack an NPC, that character goes into the NPC_ATTACKING state to fight back. The NPCs are still pretty dumb, because they go about their business as if nothing happened if you stop fighting with them. As long as you’re attacking them, though, the NPCs fight back. Figure 18.11 shows a Viking taking swings at the player’s sprite.

The alpha channel support is utilized by drawing the NPC in red when the NPCs are engaging the player in combat (as shown in Figure 18.12). I wanted to clearly show when an NPC is attacking your player because the attack animations are so similar to the walking animations; it’s hard to tell exactly which NPC is fighting back. The red coloration of the sprite is a fantastic effect! In fact, I like it so much that I think it should be a part of the game and left in place! It would be cool to use this coloring effect for other states, and you can use it with some great results for things like spells and so on.

As you know, the NPCs need to be drawn even when they aren’t just walking or attacking, because the other states (such as NPC_TALKING) must have the sprite being updated on the screen. DrawNPC checks for new states and then assumes NPC_WALKING for any state that is not explicitly programmed to handle everything else that the NPC might be doing. If you add animations to the NPCs, you need to add the state condition here to account for it.

Public Sub DrawNPC(ByVal num As Long, ByVal color As Long)
    Dim r As RECT
    Dim classindex As Long
Implementing the Combat System

Figure 18.11  This Viking Warrior is attacking the player!

Figure 18.12  Another combat in progress, this time with a Skeleton Knight.
324  Chapter 18  ■  Engaging in Combat with NPCs

' grab a shortcut to these long variable names
r.Left = charStates(num).curpos.x
r.Top = charStates(num).curpos.y
r.Right = r.Left + charWalkSpr(num).width
r.Bottom = r.Top + charWalkSpr(num).height

' remember, images are referred to using the NPC’s classindex!
' the sprite and state arrays are for every single unique NPC,
' but the bitmap image and class data are shared by all NPCs
classindex = charStates(num).classindex

' now check to see if the sprite is within the scrolling viewport
' sprite’s position is actually global, so determine if it’s visible
If r.Left > ScrollX - 1 And r.Right < ScrollX + SCREENWIDTH + 1 And _
    r.Top > ScrollY - 1 And r.Bottom < ScrollY + SCREENHEIGHT + 1 Then

    Select Case charStates(num).state
    Case NPC_ATTACKING
        AnimateSprite charAttackSpr(num)
        charAttackSpr(num).x = charStates(num).curpos.x - ScrollX
        charAttackSpr(num).y = charStates(num).curpos.y - ScrollY
        charAttackSpr(num).AnimSeq = charStates(num).facing
        DrawSprite charAttack(classindex), charAttackSpr(num), color
    Case Else
        ' update animation frame if walking
        AnimateSprite charWalkSpr(num)

        ' draw the sprite—remember, it’s using the shared image
        charWalkSpr(num).x = charStates(num).curpos.x - ScrollX
        charWalkSpr(num).y = charStates(num).curpos.y - ScrollY
        charWalkSpr(num).AnimSeq = charStates(num).facing
        DrawSprite charWalk(classindex), charWalkSpr(num), color
    End Select

End If

End Sub
A Note about Making the Game Modular

The Celtic Crusader game is now becoming rather large and complex. When you think about how far we’ve come and how simply the game started out, it’s pretty amazing how big it is now. At this point, it is important to make the game more modular and expandable. Switching the NPCs and PC to a state-driven model rather than procedural was a big help. Otherwise it can be daunting to maintain the complexity of many animation sequences simultaneously in memory along with tracking what the player is doing and so on.

But the state engines are not good enough to make this game a success. Since the code has now grown a little difficult to wade through, consider how to proceed if you continue completing the game (with your own vision and imagination, of course). I have set up the game so that NPCs have two arrays of animations available—one for walking, another for attacking. If you plan to add more types of animations, insert additional state conditions and new sprite arrays for each type of animation. Giving each character a standard, default number of animations helps you easily insert new characters without having to modify any code for bitmap files with a different number of sprite columns, animation frames, and so forth.

Level Up

This chapter filled in the most important aspect of the game thus far. A combat system is essential to a good RPG, but your list of core techniques were not up to the challenge until this point in the book. Now that you have a rudimentary combat system available, there is no limit to what you can do with this game engine. You have come so far since the first few chapters on building the tile-based scroller! It’s incredible that things have progressed so much; it feels like ancient history. You are now working with high-level concepts that brought the game to life! I’ve been running the game in full-screen mode and it looks fantastic. This is entirely due to the high-quality artwork made available by Reiner’s Tilessets, and I take no credit for the artwork (although properly using it is no simple feat, as you have seen so far). What’s next? This concludes Part IV. Part V begins in the next chapter, where you learn how to liven up the game world with inanimate objects such as trees, buildings, and so on.