Action Arcade Adventure Set

*Author's note: This chapter from AAAS is being posted with the publisher's permission on the Internet on **ftp.accessnv.com**. This chapter covers sprite animation in an arcade game. Originally, this chapter was written in Word for Windows 6.0. I converted it to Windows Write format because it is a more universal standard. This chapter contains several "side bars" which did not translate well to Windows Write format. They exist as paragraphs embedded in the text. I have highlighted them in italics to show that they are actually side bars, and not part of the continuum of the discussion.*

*I hope you enjoy this chapter from **Action Arcade Adventure Set**. If you have questions, you can reach me at 72000,1642@compuserve.com or FASTGRAPH@AOL.COM.*

Diana Gruber
March 10, 1995.

Chapter 13: Unlocking the Mysteries of Sprite Animation

<deck>Combining action functions with your creativity is a sure-fire way to create a game with smooth animation, interesting characters, and challenging enemies.

[1] The Creativity of Sprite Animation

Now that you know how to store sprites and display them on the screen, let's see if we can make their movements more interesting. In this chapter, we'll see how action functions are executed for each animation frame. The action function looks at the forces applied to the sprite and adjusts the sprite's position accordingly.

The most common forces applied to sprites are background tile attributes, user interaction (keyboard input), interaction with other objects (collision detection), gravity, momentum, friction, and artificial intelligence. To understand how these forces interact, you will first need a thorough understanding of the physics of motion. Get out your 1high-school physics text and look up the formulas for velocity, acceleration, friction, and momentum. (Just kidding--you don't need to do that!) For the purpose of sprite animation, we will act not as scientists, but as artists. We'll subscribe to the philosophy "feels right is good enough." Most of our sprite motion is accomplished through trial and error. The ideas in this chapter will get you off to a good start on sprite motion, but are by no means the final word on the subject. Your own imagination is what will make your sprites come to life.

[1] The Secret of Sprite Motion: The Action Functions

The sprite action functions are the foundation of sprite motion. All the action functions are found in the source code file, ACTION.C. We put them all in one source code file because they are all declared **near**. The **near** declaration allows us to address these functions as integer fields in the sprite structures. (Recall from Chapter 12 that the sprite action function has an integer address that is a member of the 2 object structure.) To support the action functions, we'll also be using a second source file named MOTION.C. This file contains functions that perform collision detection operations and determine how sprites will interact with each other. The action functions frequently call some of the functions in MOTION.C when they need to make decisions about sprite motion.

The functions in ACTION.C are listed in Table 13.1 and the functions in MOTION.C are listed in Table 13.2.

Table 13.1 Functions in ACTION.C

Function	Description
bullet_go()	Bullet action function
enemy_hopper_go()	Grasshopper action function
enemy_scorpion_go()	Scorpion action function
floating_points_go()	Floating score action function
kill_bullet()	Removes a bullet when it is no longer needed
kill_enemy()	Removes an enemy after it has died
kill_object()	Removes any object from a linked list
launch_bullet()	Spawns a bullet object

launch_enemy()	Spawns an enemy object
launch_floating_points()	Spawns a floating-score object
player_begin_fall()	Begins falling
player_begin_jump()	Begins jumping
player_begin_kick()	Begins kicking
player_begin_shoot()	Begins shooting
player_fall()	Falling action
player_jump()	Jumping action
player_kick()	Kicking action
player_run()	Running action
player_shoot()	Shooting action
player_stand()	Standing action
put_score()	Calculates the position of the scoreboard
update_score()	Changes the score

Table 13.2 Functions in MOTION.C

Function	Description
can_move_down()	Checks if the adjacent tile is solid on top
can_move_left()	Checks if the adjacent tile is solid on the right
can_move_right()	Checks if the adjacent tile is solid on the left
can_move_up()	Checks if the adjacent tile is solid on bottom
collision_detection()	Checks if two objects intersect
how_far_left()	Checks to see how far we can move left
how_far_right()	Checks to see how far we can move right
how_far_up()	Checks to see how far we can move up
how_far_down()	Checks to see how far we can move down
test_bit()	Tests a tile attribute

Adding Your Own Action Funtions

To extend the Tommy's Adventures game, the first thing you'll probably want to do is add your own sprite animation or change the animation of one of the exising sprites. This is easy to do by adding new action functions or modifying existing ones. Be sure to add the action function to the ACTION.C source code file so it can be addressed as a ***near*** *function.*

[2] A Closer Look at Action Functions

Although an object, such as a bullet, may have several action functions, only one action function will be called for each frame of animation. In general, the action function performs the following tasks:

> *Calculates the object's position.* The position will change according to the forces applied to the object, such as user interaction, artificial intelligence, interaction with the background, and interaction with other sprites.

Determines the object's sprite. If a character has a six-stage walk, the action function decides which stage we are on currently on, and points the object structure to the right sprite structure.

Collision detection. This function determines if the object runs into anything during the frame. If so, the collision detection determines what should be done.

Spawns new objects and kills off old ones. If a sprite performs an action, such as firing a bullet, the action function must spawn a new object during the frame--in this case a bullet. This is accomplished by calling a function to create the new object. If an object is killed, a function must also be called to remove the object from the linked list and free the allocated memory.

Chooses the action function for the next frame. If an object gets to a transition stage, such as reaching the end of a ledge, a new action function will be assigned to the object, which will be executed during the next animation frame.

[2] Examining ACTION.C

Let's look at the complete listing for ACTION.C.

```
/*****************************************************************\
*  action.c -- Tommy game action functions source code            *
*  copyright 1994 Diana Gruber                                    *
*  compile using large model, link with Fastgraph (tm)            *
\*****************************************************************/

void near bullet_go(OBJp objp)
{
 int min_x,max_x;
 register int i;

 /* increment the bullet's horizontal position */
 objp->x += objp->xspeed;

 /* collision detection */
 for (i = 0; i < nenemies; i++)
 {
   if (enemy[i]->frame < 6 && objp->x>enemy[i]->x
       && objp->x<enemy[i]->x+enemy[i]->sprite->width
       && objp->y<enemy[i]->y
       && objp->y>enemy[i]->y-enemy[i]->sprite->height)
   {
     launch_floating_points(enemy[i]);
     enemy[i]->frame = 6;
     objp->action = kill_bullet;
   }
```

```
  }

  /* check if the bullet has moved off the screen */
  max_x = (tile_orgx + objp->tile_xmax) * 16;
  min_x = (tile_orgx + objp->tile_xmin) * 16;

  /* if it has moved off the screen, kill it by setting the action
     function to kill for the next frame */

  if (objp->x > max_x || objp->x < min_x)
     objp->action = kill_bullet;

  if (objp->direction == RIGHT && !can_move_right(objp))
     objp->action = kill_bullet;
  else if (objp->direction == LEFT && !can_move_left(objp))
     objp->action = kill_bullet;
}
/*********************************************************************/
void near enemy_hopper_go(OBJp objp)
{
  if (objp->frame > 4)   /* is this enemy dying? */
  {
     /* after 100 frames, kill this enemy off */
     objp->frame++;
     if (objp->frame > 100)
        objp->action = kill_enemy;
     objp->sprite = enemy_sprite[5];

     /* enemy can fall while dying */
     objp->yspeed = how_far_down(objp,12);
     if (objp->yspeed > 0)
        objp->y += objp->yspeed;

     /* no point in doing anything else while dying */
     return;
  }

  /* this enemy moves every 12 clock ticks */
  objp->time += delta_time;
  if (objp->time < 12)
     return;
  else
     objp->time = 0;

  objp->yspeed = how_far_down(objp,12);    /* falling? */
  if (objp->yspeed > 0)
     objp->y += objp->yspeed;
  else
  {
     /* increment the object's horizontal position */
     if (objp->direction == LEFT)
```

```
    {
      if (!can_move_left(objp))
      {
        objp->direction = RIGHT;
        objp->xspeed = 12;
      }
    }
    else if (objp->direction == RIGHT)
    {
      if (!can_move_right(objp))
      {
        objp->direction = LEFT;
        objp->xspeed = -12;
      }
    }
    objp->x += objp->xspeed;
  }
  objp->frame = 7-objp->frame;          /* increment the frame */
  objp->sprite = enemy_sprite[objp->frame];

  /* if the player hasn't been hit recently, can we hit him now? */
  if (!player_blink)
  {
    if (collision_detection(objp, player) && !kicking)
    {
      player_blink = TRUE;             /* make the player blink */

      nhits++;                        /* seven hits per life */
      if (nhits > 7)
      {
        nlives--;
        if (nlives == 0)              /* three lives per game */
          nlives = 3;
        nhits = 0;
      }

      /* update the action function for the score */
      score->action = update_score;
    }
  }
}
/*****************************************************************/
void near enemy_scorpion_go(OBJp objp)
{
  if (objp->frame > 1)                /* is this enemy is dying? */
  {
    objp->frame++;

    /* after 100 frames, kill this enemy off */
    if (objp->frame > 100)
      objp->action = kill_enemy;
```

```
   objp->sprite = enemy_sprite[2];

   /* enemy can fall while dying */
   objp->yspeed = how_far_down(objp,12);
   if (objp->yspeed > 0)
      objp->y += objp->yspeed;

   /* no point in doing anything else while dying */
   return;
}

/* this enemy moves every 16 clock ticks */
objp->time += delta_time;
if (objp->time < 16)
   return;
else
   objp->time = 0;

objp->yspeed = how_far_down(objp,12);  /* falling? */
if (objp->yspeed > 0)
   objp->y += objp->yspeed;
else
{
   /* increment the object's horizontal position */
   if (objp->direction == LEFT)
   {
      if (!can_move_left(objp))
      {
         objp->direction = RIGHT;
         objp->xspeed = 12;
      }
   }
   else if (objp->direction == RIGHT)
   {
      if (!can_move_right(objp))
      {
         objp->direction = LEFT;
         objp->xspeed = -12;
      }
   }
   objp->x += objp->xspeed;
}
objp->frame = 1-objp->frame;          /* increment frame */
objp->sprite = enemy_sprite[objp->frame];

/* if the player hasn't been hit recently, can we hit him now? */
if (!player_blink)
{
   if (collision_detection(objp, player) && !kicking)
   {
```

```
      player_blink = TRUE;            /* make the player blink */
      /* seven hits per life */
      nhits++;                        /* seven hits per life */
      {
        nlives--;
        if (nlives == 0)              /* three lives per game */
          nlives = 3;
        nhits = 0;
      }

      /* update the score box */
      score->action = update_score;
    }
  }
}
/*********************************************************************/
void near floating_points_go(OBJp objp)
{
  /* update the vertical position */
  objp->y += objp->yspeed;

  /* score goes up 75 frames, then disappears */
  objp->frame++;
  if (objp->frame > 75)
    objp->action = kill_object;
}
/*********************************************************************/
void near kill_bullet(OBJp objp)
{
  /* decrement the bullet count and kill the bullet */
  nbullets--;
  kill_object(objp);
}
/*********************************************************************/
void near kill_enemy(OBJp objp)
{
  register int i;
  int enemy_no;

  for (i = 0; i < nenemies; i++)       /* which enemy is it? */
  {
    if (enemy[i] == objp)
    {
      enemy_no = i;
      break;
    }
  }
  nenemies--;                          /* decrement the enemy count */
  for (i = enemy_no; i < nenemies; i++) /* update the array */
    enemy[i] = enemy[i+1];
  enemy[nenemies] = (OBJp)NULL;        /* last enemy points to NULL */
```

```
  kill_object(objp);                  /* remove node from list */
  player_score += 100;                /* increment the score */
  score->action = update_score;
}
/*****************************************************************/
void near kill_object(OBJp objp)        /* remove node from list */
{
  OBJp node;

  node = objp;
  if (node == bottom_node)            /* remove bottom node */
  {
    bottom_node = node->next;
    if (bottom_node != (OBJp) NULL)
      bottom_node->prev = (OBJp)NULL;
  }
  else if (node == top_node)          /* remove top node */
  {
    top_node = node->prev;
    top_node->next = (OBJp)NULL;
  }
  else                                /* remove middle node */
  {
    node->prev->next = node->next;
    node->next->prev = node->prev;
  }
  free(node);
}
/*****************************************************************/
void near launch_bullet()             /* start a new bullet */
{
  OBJp node;

  if (nbullets > 9) return;           /* max 9 bullets */

  node = (OBJp)malloc(sizeof(OBJ)+3);   /* allocate space */
  if (node == (OBJp)NULL) return;

  if (player->direction == RIGHT)       /* assign values */
  {
    node->direction = RIGHT;
    node->xspeed = 13;
    if (player->sprite == tom_jump[2])  /* jumping */
    {
      node->x = player->x+player->sprite->xoffset+46-node->xspeed;
      node->y = player->y-25;
    }
    else if (player->sprite == tom_jump[3]) /* falling */
    {
      node->x = player->x+player->sprite->xoffset+46-node->xspeed;
      node->y = player->y-25;
```

```
    }
    else if (fg_kbtest(KB_RIGHT))      /* running */
    {
      node->x = player->x+player->sprite->xoffset+40-node->xspeed;
      node->y = player->y-26;
    }
    else                               /* standing */
    {
      node->x = player->x+player->sprite->xoffset+40-node->xspeed;
      node->y = player->y-28;
    }
  }
  else
  {
    node->direction = LEFT;
    node->xspeed = -13;
    node->x = player->x+player->sprite->xoffset-node->xspeed-5;
    if (player->sprite == tom_jump[2])      /* jumping */
      node->y = player->y-25;
    else if (player->sprite == tom_jump[3]) /* falling */
      node->y = player->y-25;
    else if (fg_kbtest(KB_LEFT))           /* running */
      node->y = player->y-26;
    else                                   /* standing */
      node->y = player->y-28;
  }
  node->yspeed    = 0;
  node->tile_xmin = 1;
  node->tile_xmax = 21;
  node->tile_ymin = 0;
  node->tile_ymax = 14;
  node->sprite = tom_shoot[6];             /* assign the sprite */

  node->action = bullet_go;                /* assign action function */

  /* insert the new object at the top of the linked list */
  if (bottom_node == (OBJp)NULL )
  {
    bottom_node = node;
    node->prev = (OBJp)NULL;
  }
  else
  {
    node->prev = top_node;
    node->prev->next = node;
  }
  top_node = node;
  node->next = (OBJp)NULL;

  nbullets++;                              /* increment bullet count */
}
```

```
/*********************************************************************/
void near launch_enemy(int x, int y, int type)  /* start a new enemy */
{
  OBJp node;

  node = (OBJp)malloc(sizeof(OBJ));         /* allocate space */
  if (node == (OBJp)NULL) return;

  node->direction = RIGHT;                  /* assign values */
  node->x = x;
  node->y = y;
  node->xspeed = 8;
  node->yspeed = 0;
  node->tile_xmin = 1;
  node->tile_xmax = 21;
  node->tile_ymin = 0;
  node->tile_ymax = 14;
  node->time = 0;

  /* assign the sprite and action function */
  if (type == 0)
  {
    node->frame = 0;
    node->action = enemy_scorpion_go;
  }
  else
  {
    node->frame = 3;
    node->action = enemy_hopper_go;
  }
  node->sprite = enemy_sprite[node->frame];

  /* insert the new object at the top of the linked list */
  if (bottom_node == (OBJp)NULL )
  {
    bottom_node = node;
    node->prev = (OBJp)NULL;
  }
  else
  {
    node->prev = top_node;
    node->prev->next = node;
  }
  top_node = node;
  node->next = (OBJp)NULL;

  enemy[nenemies] = node;                   /* update enemy array */
  nenemies++;                           /* increment enemy counter */
}
/*********************************************************************/
void near launch_floating_points(OBJp objp)
```

```
{
  OBJp node;

  node = (OBJp)malloc(sizeof(OBJ)+3);       /* allocate space */
  if (node == (OBJp)NULL) return;

  node->direction = RIGHT;                  /* assign values */
  node->xspeed = 0;
  node->yspeed = -1;
  node->x = objp->x+16;
  node->y = objp->y-8;
  node->frame = 0;
  node->tile_xmin = 1;
  node->tile_xmax = 21;
  node->tile_ymin = 0;
  node->tile_ymax = 14;
  node->sprite = tom_score[2];              /* assign the sprite */
  node->action = floating_points_go;        /* assign action function */

  /* insert the new object at the top of the linked list */
  if (bottom_node == (OBJp)NULL )
  {
    bottom_node = node;
    node->prev = (OBJp)NULL;
  }
  else
  {
    node->prev = top_node;
    node->prev->next = node;
  }
  top_node = node;
  node->next = (OBJp)NULL;
}
/*****************************************************************/
void near player_begin_fall(OBJp objp)
{
  /* called once at the start of a fall */

  objp->yspeed = 1;                         /* initialize variables */
  vertical_thrust = 0;
  shoot_time = 0;

  /* any thrust from the arrow keys? */
  if (fg_kbtest(KB_LEFT) || fg_kbtest(KB_RIGHT))
    forward_thrust = 100;
  else
    forward_thrust = 0;

  if (objp->direction == LEFT)
    tom_jump[3]->xoffset = -10;
  else
```

```
    tom_jump[3]->xoffset = -0;

  if (fg_kbtest(KB_ALT))                /* shooting while falling */
    objp->frame = 3;
  else
    objp->frame = 1;

  objp->sprite = tom_jump[objp->frame];    /* assign the sprite */
  objp->action = player_fall;              /* assign action function */
}
/*********************************************************************/
void near player_begin_jump(OBJp objp)
{
  /* called once at the start of a jump */

  objp->yspeed = -15;                   /* initialize variables */
  objp->frame = 0;
  shoot_time = 0;

  if (fg_kbtest(KB_LEFT) || fg_kbtest(KB_RIGHT)) /* walking? */
    forward_thrust = 50;
  else
    forward_thrust = 0;

  if (objp->direction == LEFT)
    tom_jump[3]->xoffset = 25;
  else
    tom_jump[3]->xoffset = 0;

  objp->sprite = tom_jump[objp->frame];    /* assign sprite */
  objp->action = player_jump;              /* assign action function */
}
/*********************************************************************/
void near player_begin_kick(OBJp objp)
{
  /*  called once at the start of a kick */

  int i;

  kicking = TRUE;                       /* initialize variables */
  objp->time = 0;
  nkicks = 0;

  /* is this a left (backward) or a right (forward) kick? */
  if (objp->direction == LEFT)
  {
    objp->frame = 0;
    kick_frame = 3;
    kick_basey = objp->y;
    objp->sprite = tom_kick[objp->frame]; /* assign sprite */
```

```
    /* back him up a little if needed */
    player->x += 36;
    for (i = 0; i < 36; i++)
    if (can_move_left(player))
      player->x--;
  }
  else
  {
    objp->frame = 6;
    kick_frame = 7;
    kick_basey = objp->y;
    objp->sprite = tom_kick[objp->frame]; /* assign sprite */

    /* back him up a little if needed */
    player->x -= 24;
    for (i = 0; i < 24; i++)
    if (can_move_right(player))
      player->x++;
  }
  objp->action = player_kick;             /* assign action function */
}
/*********************************************************************/
void near player_begin_shoot(OBJp objp)
{
  /* called once at the start of shooting */

  register int i;

  objp->frame = 0;                      /* initialize variables */
  objp->time = 0;
  nshots = 0;

  if (objp->direction == RIGHT)
  {
    tom_shoot[0]->xoffset = 2;
    tom_shoot[1]->xoffset = 2;
    tom_shoot[2]->xoffset = 2;

    /* back him up a little if needed */
    if (fg_kbtest(KB_RIGHT))             /* running while shooting? */
    {
      objp->sprite = tom_shoot[3];
      player->x -= 24;
      for (i = 0; i < 24; i++)
      if (can_move_right(player))
        player->x++;
    }
    else
      objp->sprite = tom_shoot[0];       /* assign sprite */
  }
  else
```

```
  {
    tom_shoot[0]->xoffset = -1;
    tom_shoot[1]->xoffset = -20;
    tom_shoot[2]->xoffset = -15;
    if (fg_kbtest(KB_LEFT))             /* running while shooting? */
      objp->sprite = tom_shoot[3];      /* assign sprite */
    else
      objp->sprite = tom_shoot[0];
  }
  objp->action = player_shoot;          /* assign action function */
}
/*********************************************************************/
void near player_fall(OBJp objp)
{
  int tile_x,tile_y;

  /* less than 5 clock ticks? Then skip this function */
  objp->time += delta_time;
  shoot_time += delta_time;
  if (objp->time > 5)
    objp->time = 0;
  else
    return;

  if (fg_kbtest(KB_ALT))                /* shooting while falling? */
  {
    objp->frame = 3;

    /* start a new bullet every 15 clock ticks */
    if (shoot_time > 15)
    {
      launch_bullet();
      shoot_time = 0;
    }
  }
  else
    objp->frame = 1;

  objp->sprite = tom_jump[objp->frame];    /* assign sprite */

  /* increase the rate of speed of the fall */
  if (objp->yspeed < 15)
    objp->yspeed += (vertical_thrust++);

  /* vertical position is based on yspeed */
  objp->y += objp->yspeed;

  /* check the arrow keys, starting with left arrow */
  if (objp->direction == LEFT)
  {
    /* horizontal speed */
```

```
  if (fg_kbtest(KB_LEFT))
  {
    objp->xspeed = -1;
    if (forward_thrust > 50)
      objp->xspeed *= 3;
    else if (forward_thrust > 0)
      objp->xspeed *= 2;
  }
  else
    objp->xspeed = 0;

  /* check for walls, etc. */
  objp->xspeed = -how_far_left(objp,-objp->xspeed);

  /* increment the x position according to the speed */
  objp->x += objp->xspeed;

  /* are we still on visible screen? If not, scroll */
  tile_x = objp->x/16 - tile_orgx;
  if (tile_x < objp->tile_xmin)
    scroll_left(-objp->xspeed);
}

/* same thing for right arrow key */
else
{
  if (fg_kbtest(KB_RIGHT))
  {
    objp->xspeed = 1;
    if (forward_thrust > 50)
      objp->xspeed *= 3;
    else if (forward_thrust > 0)
      objp->xspeed *= 2;
  }
  else
    objp->xspeed = 0;

  tom_jump[3]->xoffset = 0;
  objp->direction = RIGHT;
  objp->xspeed = how_far_right(objp,objp->xspeed);
  objp->x += objp->xspeed;

  /* are we still on visible screen? If not, scroll */
  tile_x = objp->x/16 - tile_orgx;
  if (tile_x > objp->tile_xmax)
    scroll_right(objp->xspeed);
}

/* decrement the forward thrust */
forward_thrust--;
```

```
  /* are we close to the bottom of the screen? If so, scroll */
  tile_y = objp->y/16 - tile_orgy;
  if (tile_y > objp->tile_ymax)
     scroll_down(objp->yspeed);

  /* have we hit a solid tile yet? If so, stop falling */
  if (!can_move_down(objp))
  {
     objp->y = ((objp->y+1)/16) * 16; /* land on top of tile */
     objp->yspeed = 0;
     objp->action = player_stand;
  }
}
/*****************************************************************/
void near player_jump(OBJp objp)
{
  int tile_x,tile_y;
  register int i;

  /* increment the timer, if it is less than 5, skip it */
  objp->time += delta_time;
  shoot_time += delta_time;
  if (objp->time > 5L)
     objp->time = 0;
  else
     return;

  /* check for arrow keys, left arrow first */
  if (fg_kbtest(KB_LEFT))
  {
     objp->direction = LEFT;
     objp->xspeed = -3;

     /* forward thrust gives a little boost at start of jump */
     if (forward_thrust > 30)
        objp->xspeed *= 4;
     else if (forward_thrust > 0)
        objp->xspeed *= 2;

     /* move left, checking for walls, etc. */
     objp->xspeed = -how_far_left(objp,-objp->xspeed);
     objp->x += objp->xspeed;

     /* need to scroll the screen left? */
     tile_x = objp->x/16 - tile_orgx;
     if (tile_x < objp->tile_xmin)
        scroll_left(-objp->xspeed);
  }

  /* same for right arrow key */
  else if (fg_kbtest(KB_RIGHT))
```

```
{
  objp->xspeed = 3;
  if (forward_thrust > 50)
    objp->xspeed *= 4;
  else if (forward_thrust > 0)
    objp->xspeed *= 2;
  objp->direction = RIGHT;
  objp->xspeed = how_far_right(objp,objp->xspeed);

  tile_x = objp->x/16 - tile_orgx;
  if (tile_x > objp->tile_xmax)
    scroll_right(objp->xspeed);
  objp->x += objp->xspeed;
}

/* decrement forward thrust */
forward_thrust--;

/* additional upward thrust if you hold down the Ctrl key */
if (fg_kbtest(KB_CTRL))
  objp->yspeed++;
else
  objp->yspeed/=4;

/* check bumping head on ceiling */
objp->yspeed = how_far_up(objp,objp->yspeed);
objp->y += objp->yspeed;

/* check if we are shooting */
if (fg_kbtest(KB_ALT))
{
  /* Tommy's jumping and shooting frame */
  objp->frame = 2;

  /* space the bullets 15 clock ticks apart */
  if (shoot_time > 15)
  {
    launch_bullet();
    shoot_time = 0;
  }
}

/* not shooting, just use tommy jumping frame */
else
  objp->frame = 0;

/* set sprite to the correct frame */
objp->sprite = tom_jump[objp->frame];

/* too close to top of screen? scroll the screen up */
tile_y = objp->y/16 - tile_orgy;
```

```
  if (tile_y < objp->tile_ymin)
    scroll_up(-objp->yspeed);

  /* reached top of arc? Tommy start descent */
  if (objp->yspeed >= 0)
    objp->action = player_begin_fall;
}
/*********************************************************************/
void near player_kick(OBJp objp)
{
  register int i;
  int tile_x,tile_y;

  /* collision detection -- did we kick an enemy? */
  for (i = 0; i < nenemies; i++)
  {
    /* frame 6 is the enemy hit frame. enemies are only hit once */
    if (enemy[i]->frame < 6 && collision_detection (objp,enemy[i]))
    {
      /* if you are kicking left, you can only hit enemy
         left of you */

      if (objp->direction == LEFT && enemy[i]->x < objp->x)
      {
        launch_floating_points(enemy[i]);
        enemy[i]->frame = 6;
      }

      /* likewise, right kicks kill enemies on the right */
      else if (objp->direction == RIGHT && enemy[i]->x > objp->x)
      {
        launch_floating_points(enemy[i]);
        enemy[i]->frame = 6;
      }
    }
  }

  /* increment the frame every 10 clock ticks */
  objp->time += delta_time;
  if (objp->time > 10)
  {
    /* case of the left (backwards) kick */
    if (objp->direction == LEFT)
    {
      /* where are we in this kick? */
      if (objp->frame == kick_frame && nkicks < 4
          && fg_kbtest(KB_SPACE))
      {
         /* keep kicking */
      }
```

```
  else
  {
    /* increment the frame */
    objp->frame++;

    /* end of kick */
    if (objp->frame > 5)
    {
      objp->y = kick_basey; /* end kick where you started */
      objp->sprite = tom_stand[0];
      kicking = FALSE;

      /* new action function */
      objp->action = player_stand;
    }

    /* still kicking, set the sprite */
    else
    {
      objp->sprite = tom_kick[objp->frame];

      /* horizontal motion */
      if (can_move_left(objp))
      {
        if (fg_kbtest(KB_LEFT))
          objp->xspeed = -3;
        else
          objp->xspeed = -1;

        objp->x += objp->xspeed;

        /* moved past edge of screen? scroll left */
        tile_x = objp->x/16 - tile_orgx;
        if (tile_x < objp->tile_xmin)
          scroll_left(-objp->xspeed);
      }
    }
  }
}

/* case of the right (forward) kick */
else
{
  /* choose frame */
  if (objp->frame == kick_frame && nkicks < 4
     && fg_kbtest(KB_SPACE))
  {
     /* keep kicking */
  }
  else
  {
```

```
      objp->frame++;
      if (objp->frame > 9)
      {
        objp->y = kick_basey; /* end kick where you started */
        objp->sprite = tom_stand[0];
        kicking = FALSE;
        objp->action = player_stand;
      }
      else
      {
        if (objp->frame > 8)
        {
          objp->sprite = tom_stand[0];
        }
        else if (objp->frame > 7)
        {
          objp->sprite = tom_kick[6];
        }
        else
          objp->sprite = tom_kick[objp->frame];

        /* horizontal motion */
        if (can_move_right(objp))
        {
          if (fg_kbtest(KB_RIGHT))
            objp->xspeed = 3;
          else
            objp->xspeed = 1;

          objp->xspeed = how_far_right(objp,objp->xspeed);
          objp->x += objp->xspeed;

          tile_x = objp->x/16 - tile_orgx;
          if (tile_x > objp->tile_xmax)
            scroll_right(objp->xspeed);

        }
      }
    }
  }

  /* vertical motion */
  if (objp->frame == kick_frame)
  {
    /* put a little vertical bounce in the kick */
    if (objp->y == kick_basey)
    {
      objp->yspeed = -3;

      /* barrier above? */
      objp->yspeed = how_far_up(objp,objp->yspeed);
```

```
        objp->y += objp->yspeed;

        /* need to scroll up? */
        tile_y = objp->y/16 - tile_orgy;
        if (tile_y < objp->tile_ymin)
          scroll_up(-objp->yspeed);
      }
      else
      {
        objp->y = kick_basey;
        nkicks++;
      }
    }

    /* falling? */
    if (objp->y == kick_basey && can_move_down(objp))
    {
      kicking = FALSE;
      objp->action = player_begin_fall;
    }

    /* set the timer back to 0 */
    objp->time = 0;
  }
}
/*********************************************************************/
void near player_run(OBJp objp)
{
  int tile_x;

  /* case where the player is facing left */
  if (objp->direction == LEFT)
  {
    /* gradually increase the speed */
    if (objp->xspeed > -8)
      objp->xspeed--;

    /* change the horizontal position according to the speed */
    if (can_move_left(objp))
    {
      objp->x += objp->xspeed;
      tile_x = objp->x/16 - tile_orgx;

      /* if you have moved out of the visible area, scroll left */
      if (tile_x < objp->tile_xmin)
        scroll_left(-objp->xspeed);
    }
  }

  /* case where the player is facing right */
  else
```

```
{
  if (objp->xspeed < 8)
    objp->xspeed++;

  if (can_move_right(objp))
  {
    objp->x += objp->xspeed;
    tile_x = objp->x/16 - tile_orgx;
    if (tile_x > objp->tile_xmax)
      scroll_right(objp->xspeed);
  }
}

/* is it time to increment the the walking stage yet? */
objp->time += delta_time;
if (objp->time > 3)
{
  objp->time = 0;
  objp->frame++;

  /* it's a six-stage walk */
  if (objp->frame > 5) objp->frame = 0;
  objp->sprite = tom_run[objp->frame];
}

/* are we pressing any arrow keys? */
if (fg_kbtest(KB_LEFT))
{
  /* change the direction if necessary */
  if (objp->direction == RIGHT)
  {
    /* slow down speed in the middle of a direction change */
    objp->xspeed = 0;
    objp->direction = LEFT;
  }
}
else if (fg_kbtest(KB_RIGHT))
{
  if (objp->direction == LEFT)
  {
    objp->xspeed = 0;
    objp->direction = RIGHT;
  }
}

/* if we aren't pressing any keys, then we aren't walking. Change
  the action function to standing. */

else
  objp->action = player_stand;
```

```
  /* are we falling? */
  if (can_move_down(objp))
    objp->action = player_begin_fall;

  /* or kicking or jumping or shooting? */
  else if (fg_kbtest(KB_SPACE))
    objp->action = player_begin_kick;
  else if (fg_kbtest(KB_CTRL))
    objp->action = player_begin_jump;
  else if (fg_kbtest(KB_ALT))
    objp->action = player_begin_shoot;
}
/*********************************************************************/
void near player_shoot(OBJp objp)
{
  register int i;
  unsigned long max_shoottime;
  int tile_x,tile_y;

  objp->time += delta_time;

  /* check for horizontal motion -- arrow keys pressed? */
  if (fg_kbtest(KB_RIGHT))
  {
    /* changing direction? start shooting all over */
    if (objp->direction == LEFT)
    {
      objp->direction = RIGHT;
      objp->action = player_begin_shoot;
    }
    else
    {
      /* spawn bullets more often when walking */
      max_shoottime = 3;
      if (objp->time > max_shoottime)
      {
        if (objp->sprite == tom_shoot[3])
        {
          objp->sprite = tom_shoot[4];
          launch_bullet();
        }
        else if (objp->sprite == tom_shoot[4])
          objp->sprite = tom_shoot[5];
        else
          objp->sprite = tom_shoot[3];

        /* move forward during walking frames */
        if (can_move_right(objp))
        {
          /* move right, checking for barriers */
          objp->xspeed = how_far_right(objp,8);
```

```
        objp->x += objp->xspeed;

        /* need to scroll the screen right? */
        tile_x = objp->x/16 - tile_orgx;
        if (tile_x > objp->tile_xmax)
          scroll_right(objp->xspeed);
      }
      objp->time = 0;
    }
  }

  if (!fg_kbtest(KB_ALT))              /* done shooting? */
    objp->action = player_run;

  else if (can_move_down(objp))        /* falling? */
  {
    objp->yspeed = how_far_down(objp,5);
    objp->y += objp->yspeed;

    /* are we close to the bottom of the screen? If so, scroll */
    tile_y = objp->y/16 - tile_orgy;
    if (tile_y > objp->tile_ymax)
      scroll_down(objp->yspeed);
  }
  else if (fg_kbtest(KB_CTRL))         /* jumping? */
    objp->action = player_begin_jump;
}

/* same thing for left arrow key */
else if (fg_kbtest(KB_LEFT))
{
  if (objp->direction == RIGHT)
  {
    objp->direction = LEFT;
    objp->action = player_begin_shoot;
  }
  else
  {
    max_shoottime = 3;
    if (objp->time > max_shoottime)
    {
      if (objp->sprite == tom_shoot[3])
      {
        objp->sprite = tom_shoot[4];
        launch_bullet();
      }
      else if (objp->sprite == tom_shoot[4])
        objp->sprite = tom_shoot[5];
      else
        objp->sprite = tom_shoot[3];
```

```
        if (can_move_left(objp))
        {
          objp->xspeed = -8;
          objp->x += objp->xspeed;
          tile_x = objp->x/16 - tile_orgx;
          if (tile_x < objp->tile_xmin)
          scroll_left(-objp->xspeed);
        }
        objp->time = 0;
      }
    }

    if (!fg_kbtest(KB_ALT))              /* done shooting? */
      objp->action = player_run;

    else if (can_move_down(objp))        /* falling? */
    {
      objp->yspeed = how_far_down(objp,5);
      objp->y += objp->yspeed;

      /* are we close to the bottom of the screen? If so, scroll */
      tile_y = objp->y/16 - tile_orgy;
      if (tile_y > objp->tile_ymax)
        scroll_down(objp->yspeed);
    }
    else if (fg_kbtest(KB_CTRL))         /* jumping? */
      objp->action = player_begin_jump;

}

/* no arrow keys pressed, standing still */
else
{
    max_shoottime = 16;
    if (objp->time > max_shoottime)
    {
      /* pull out gun */
      if (objp->frame == 0)
      {
        objp->frame++;
          objp->sprite = tom_shoot[objp->frame];
      }
      else if (objp->frame == 1)        /* shooting */
      {
        /* done shooting */
        if (!fg_kbtest(KB_ALT))
          objp->frame++;
        nshots++;

        objp->sprite = tom_shoot[objp->frame];
        launch_bullet();
```

```
      }
      else if (objp->frame == 2)        /* recoil */
      {
        if (fg_kbtest(KB_ALT))
          objp->frame = 1; /* shoot again */
        else
        {
          objp->frame++;
          objp->sprite = tom_shoot[2];
        }
      }
      else if (objp->frame == 3)        /* done shooting */
      {
        objp->frame = 0;
        objp->sprite = tom_stand[0];
        objp->action = player_stand;
      }
      objp->time = 0;
    }
    if (!fg_kbtest(KB_ALT))             /* done shooting? */
      objp->action = player_stand;
    else if (can_move_down(objp))       /* falling? */
      objp->action = player_begin_fall;
    else if (fg_kbtest(KB_CTRL))        /* jumping? */
      objp->action = player_begin_jump;
  }

}
/*****************************************************************/
void near player_stand(OBJp objp)
{
  /* standing still. Start walking? */
  if (fg_kbtest(KB_RIGHT))
  {
    objp->frame = 0;
    objp->xspeed = 1;
    objp->direction = RIGHT;
    objp->action = player_run;
  }
  else if (fg_kbtest(KB_LEFT))
  {
    objp->frame = 0;
    objp->xspeed = -1;
    objp->direction = LEFT;
    objp->action = player_run;
  }

  /* start kicking, jumping or shooting? */
  else if (fg_kbtest(KB_SPACE))
  {
    objp->action = player_begin_kick;
```

```
}
else if (fg_kbtest(KB_CTRL))
{
  objp->action = player_begin_jump;
}
else if (fg_kbtest(KB_ALT))
{
  objp->action = player_begin_shoot;
}

/* look down, look up */
else if (fg_kbtest(KB_DOWN))
{
  if (objp->y - tile_orgy*16 > 48)
    scroll_down(1);
}
else if (fg_kbtest(KB_UP))
{
  if (objp->y - tile_orgy*16 < 200)
    scroll_up(1);
}

/* just standing there */
else if (objp->sprite != tom_stand[0] && objp->frame < 7)
{
  objp->frame = 0;
  objp->sprite = tom_stand[objp->frame];
}
else
{
  /* change Tommy's facial expression */
  objp->time += delta_time;
  if (objp->time > max_time)
  {
    if (objp->frame == 0)
    {
      objp->frame = irandom(7,8);
      objp->time = 0;
      objp->sprite = tom_stand[objp->frame-6];

      /* how long we smile or frown is random */
      max_time = (long)irandom(200,400);
    }
    else
    {
      objp->frame = 0;
      objp->time = 0;
      objp->sprite = tom_stand[0];

      /* how long we stand straight without smiling */
      max_time = (long)irandom(500,1000);
```

```
      }
    }
  }
}
/*********************************************************************/
void near put_score(OBJp objp)
{
  /* determine x and y coords based on the screen origin */
  objp->x = tile_orgx*16 + screen_orgx + 2;
  objp->y = tile_orgy*16 + screen_orgy + 43;
}
/*********************************************************************/
void near update_score(OBJp objp)  /* called when score has changed */
{
  char string[128];
  SPRITE *scoremap;
  int y;
  register int i;

  /* Convert the (long) score to a character string.  Assume 10 digits
     is enough */

  ltoa(player_score,string,10);

  /* clear an area in video memory below the tile space where nothing
     else is going on */

  fg_setcolor(0);
  fg_rect(0,319,680,724);

  /* draw the score box in offscreen memory */
  scoremap = tom_score[0];
  fg_move(0,724);
  fg_drwimage(scoremap->bitmap,scoremap->width,scoremap->height);

  /* set the color to black and display the score */
  fg_setcolor(1);
  center_string(string,5,56,720);

  /* the status bar indicates how many times you have been hit */
  y = nhits*3;

  fg_setcolor(14);
  if (nhits == 0)           /* all blue */
  {
    fg_setcolor(18);
    fg_rect(62,67,701,723);
  }
  else if (nhits >= 8)      /* all white */
  {
```

```
    fg_setcolor(14);
    fg_rect(62,67,701,723);
  }
  else
  {                     /* white and blue */
    fg_setcolor(14);
    fg_rect(62,67,701,700+y);
    fg_setcolor(18);
    fg_rect(62,67,700+y,723);
  }

  scoremap = tom_score[1];    /* tommy one-ups */
  for (i = 0; i < nlives; i++)
  {
    fg_move(80+i*10,716);
    fg_drwimage(scoremap->bitmap,scoremap->width,scoremap->height);
  }

  /* do a getimage to put the score in a bitmap in RAM */
  objp->sprite->width = 80+10*nlives;
  fg_move(0,724);
  fg_getimage(objp->sprite->bitmap,
            objp->sprite->width,objp->sprite->height);

  /* update the x and y coords */
  objp->x = tile_orgx*16 + screen_orgx + 2;
  objp->y = tile_orgy*16 + screen_orgy + 43;

  /* assign action function */
  objp->action = put_score;
}
```

[2] Examining MOTION.C

The functions in MOTION.C are often called by the action functions to modify the position of an object, or make decisions about an object's actions. Here is the complete code:

```
/*********************************************************************\
*  motion.c -- Tommy game source code file                            *
*  copyright 1994 Diana Gruber                                        *
*  compile using large model, link with Fastgraph (tm)                *
\*********************************************************************/

#include "gamedefs.h"
/*********************************************************************/
int can_move_down(OBJp objp)
{
  /* can the object fall? */

  int tile_x,tile_y,tile_num;
```

```
  /* test left side */
  tile_x = (objp->x)/16;
  if (tile_x < 0)
     return(FALSE);

  tile_y = (objp->y+1)/16;
  tile_num = (int)background_tile[tile_x+1][tile_y];

  /* are we at the bottom of the map? */
  if (tile_y >= nrows)
     return(FALSE);

  /* is the tile solid on the top? */
  if (test_bit(background_attributes[tile_num],0))
     return(FALSE);

  /* test the right side too */
  tile_x = (objp->x + objp->sprite->bound_width)/16;
  tile_num = (int)background_tile[tile_x-1][tile_y];
  return (!test_bit(background_attributes[tile_num],0));
}
/*****************************************************************/
int can_move_left(OBJp objp)
{
  int tile_x,tile_y,tile_num;

  /* test the bottom of the sprite */
  tile_x = (objp->x-1)/16;
  if (tile_x <= 0)
     return(FALSE);
  tile_y = objp->y/16;
  tile_num = (int)background_tile[tile_x][tile_y];

  /* is the tile solid on the right? */
  if (test_bit(background_attributes[tile_num],3))
     return(FALSE);

  /* check the top of the sprite too */
  tile_y = (objp->y - objp->sprite->height)/16;
  tile_num = (int)background_tile[tile_x][tile_y];
  return (!test_bit(background_attributes[tile_num],2));
}
/*****************************************************************/
int can_move_right(OBJp objp)
{
  int tile_x,tile_y,tile_num;
  int width;

  tile_x = (objp->x + objp->sprite->bound_width)/16;
  if (tile_x >= ncols-1)
```

```
    return(FALSE);

  /* test the bottom of the sprite */
  tile_y = objp->y/16;
  tile_num = (int)background_tile[tile_x][tile_y];

  /* is the tile solid on the left? */
  if (test_bit(background_attributes[tile_num],2))
    return(FALSE);

  /* check top of sprite too */
  tile_y = (objp->y - objp->sprite->height)/16;
  tile_num = (int)background_tile[tile_x][tile_y];
  return (!test_bit(background_attributes[tile_num],3));
}
/*****************************************************************/
int can_move_up(OBJp objp)
{
  int tile_x,tile_y,tile_num;

  tile_x = objp->x/16;      /* test left side */
  tile_y = (objp->y-objp->sprite->height-1)/16;
  if (tile_y < 0)           /* test top of map */
    return(FALSE);

  tile_num = (int)background_tile[tile_x][tile_y];

  /* is the tile solid on the bottom? */
  if (test_bit(background_attributes[tile_num],1))
    return(FALSE);

  /* test the right side too */
  tile_x = (objp->x + objp->sprite->width)/16;
  tile_num = (int)background_tile[tile_x][tile_y];

  return (!test_bit(background_attributes[tile_num],1));
}
/*****************************************************************/
int collision_detection (OBJp objp1,OBJp objp2)
{
  int xmin1,xmax1,xmin2,xmax2;
  int ymin1,ymax1,ymin2,ymax2;

  /* x coordinates of object 1 */
  xmin1 = objp1->x+objp1->sprite->xoffset;
  xmax1 = xmin1+objp1->sprite->width;

  /* x coordinates of object 2 */
  xmin2 = objp2->x+objp2->sprite->xoffset;
  xmax2 = xmin2+objp2->sprite->width;
```

```
  /* y coordinates of object 1 */
  ymax1 = objp1->y+objp1->sprite->yoffset;
  ymin1 = ymax1-objp1->sprite->height;

  /* y coordinates of object 2 */
  ymax2 = objp2->y+objp2->sprite->yoffset;
  ymin2 = ymax2-objp2->sprite->height;

  /* object 2 entirely to the left of object 1 */
  if (xmax2 < xmin1) return(FALSE);

  /* object 2 entirely to the right of object 1 */
  if (xmin2 > xmax1) return(FALSE);

  /* object 2 entirely to the below object 1 */
  if (ymax2 < ymin1) return(FALSE);

  /* object 2 entirely to the above object 1 */
  if (ymin2 > ymax1) return(FALSE);

  /* the objects overlap */
  return(TRUE);
}
/*********************************************************************/
int how_far_down(OBJp objp,int n)
{
  register int i;
  register int temp;

  temp = objp->y;        /* save the current position */

  /* increment the position until you can't move right any further */
  for (i = 0; i < n; i++)
  {
     objp->y++;
     if (!can_move_down(objp))
     {
        objp->y = temp;
        return(i);
     }
  }
  objp->y = temp;        /* restore the current position */
  return(n);             /* return how far right */
}
/*********************************************************************/
int how_far_left(OBJp objp,int n)
{
  register int i;
  register int temp;

  temp = objp->x;       /* save the current position */
```

```
  /* increment the position until you can't move left any further */
  for (i = 0; i < n; i++)
  {
    objp->x--;
    if (!can_move_left(objp))
    {
      objp->x = temp;
      return(i);
    }
  }
  objp->x = temp;      /* restore the current position */
  return(n);           /* return how far left */
}
/*********************************************************************/
int how_far_right(OBJp objp,int n)
{
  register int i;
  register int temp;

  temp = objp->x;     /* save the current position */

  /* increment the position until you can't move right any further */
  for (i = 0; i < n; i++)
  {
    objp->x++;
    if (!can_move_right(objp))
    {
      objp->x = temp;
      return(i);
    }
  }

  objp->x = temp;     /* restore the current position */
  return(n);          /* return how far right */
}
/*********************************************************************/
int how_far_up(OBJp objp,int n)
{
  register int i;
  register int temp;

  temp = objp->y;     /* save the current position */

  /* increment the position until you can't move right any further */
  for (i = 0; i > n; i--)
  {
    objp->y--;
    if (!can_move_up(objp))
    {
       objp->y = temp;
```

```
        return(i);
      }
   }
   objp->y = temp;    /* restore the current position */
   return(n);         /* return how far right */
}

/*********************************************************************/

test_bit(char num,int bit)
{
   /* test bit flags, used for tile attributes */
   return((num >> bit) & 1);
}
```

[1] A Simple Action Function
Let's start by looking at a simplified action function to see how it is constructed:

```
void near player_stand(OBJp objp)
{
   if (fg_kbtest(KB_CTRL))
      objp->action = player_start_jump;
}
```

This isn't the actual **player_stand()** function used in ACTION.C. Here we've reduced this function down to its basic componets so that you can easily follow along.

Notice that the action function is declared as **void near**. None of the action functions return values and they are all declared **near** to force all of the code to be stored in the same code segment. We do this for two reasons. First, **near** functions will execute a bit faster--in this kind of game every millisecond counts. Since the action functions are executed many times per second, keeping them in near memory makes sense. Second, the near address allows us to store the pointer to this function as an integer field in the object structure. This is important because a sprite's action function will change constantly as it confronts different obstacles. For instance, to execute the proper action function each frame, we let Tommy's object point to it. If the action changes, the object points to a new function.

We don't have a lot of room in a code segment, only 64K, so we must allocate that space carefully. By default, functions may reside in separate code segments and are activated through far calls in the medium- and large-memory models. We are using the large-memory model, so we expect all functions to reside in other code segments unless we explicitly declare them **near**. We will only declare the action functions to be near, so we will have room for as many of them as possible.

Returning to **player_stand()**, notice that only one argument is passed--a pointer to a structure of type **OBJp**. In this case, the object will always be the player. We won't pass a

grasshopper or a scorpion to this action function, or any of the other player action functions. That makes things a little easier. Our main player, Tommy, has his own set of action functions which he doesn't share with any of the other objects. Only one of Tommy's action functions will be executed in a given frame.

[1] A Chain of Action Functions

An object's action functions can be thought of in terms of a chain of events. Each time an action function is executed, it is a link in a chain. The next link may be the same action function, or it may be a different one. The current action function determines the next link in the chain.

For example, the **player_stand()** action function will continue to be excuted once every frame until a Ctrl keypress is detected. At that point, the **player_stand()** function determines it is time for a change. It decides it is time for Tommy to stop standing and start jumping. Rather than handle the jumping action itself, the **player_stand()** function calls another function. It does this by assigning the object's action function pointer to another function. In this case, it tells the object to point to **player_start_jump()**. Subsequently, in the next frame, the **player_stand()** function will not be executed, but the **player_start_jump()** function will. The **player_start_jump()** function will execute for one frame, and then will pass control on to another function, the **player_jump()** function. That function, in turn, will execute for a while and then pass control on to something else, most likely the **player_start_fall()** function. Eventually, Tommy will be finished with his jumping and falling, and control will return to the **player_stand()** function. This sequence of events is repeated many times during the game. The various action functions will pass control to each other depending on the variables and forces they are aware of. Nearly all the animation in the game is controlled by this chain of action functions.

[1] The Low-Level Keyboard Handler

One of the most common forces acting on the player object is the keyboard. As you press keys, you expect Tommy to run, jump, kick, and shoot in a responsive manner. The player action functions intercept these keystrokes and perform actions accordingly. In the the **player_stand()** function we discussed earlier, the Ctrl key is detected and causes Tommy to begin jumping. We detect this key by using Fastgraph's low-level keyboard handler. The low-level keyboard handler replaces the BIOS keyboard handler for interrupt 09 hex. Keystrokes are intercepted before they get to the BIOS keyboard handler. The result is fast, continuous detection of keypresses without the problem of filling up the BIOS keyboard buffer. It also has the advantage of being able to detect two keypresses at the same time, for example the Ctrl and left arrow key detected simultaneously will cause Tommy to jump to the left. A low-level keyboard handler is an essential element of responsive action arcade games.

Fastgraph's **fg_kbtest()** function is used to detect keypresses in the player action functions.

fg_kbtest()

The **fg_kbtest()** function determines if the key having the specified scan code is now pressed or released.

int fg_kbtest (int *scan_code*);

scan_code is the scan code of the key to check.

[1] The Player Action Functions

When we discussed the **player_stand()** function we didn't look at everything this function does. Let's take a closer look at the entire function now:

```
void near player_stand(OBJp objp)
{
 /* standing still. Start walking? */
 if (fg_kbtest(KB_RIGHT))
 {
   objp->frame = 0;
   objp->xspeed = 1;
   objp->direction = RIGHT;
   objp->action = player_run;
 }
 else if (fg_kbtest(KB_LEFT))
 {
   objp->frame = 0;
   objp->xspeed = -1;
   objp->direction = LEFT;
   objp->action = player_run;
 }

 /* start kicking, jumping or shooting? */
 else if (fg_kbtest(KB_SPACE))
 {
   objp->action = player_begin_kick;
 }
 else if (fg_kbtest(KB_CTRL))
 {
   objp->action = player_begin_jump;
 }
 else if (fg_kbtest(KB_ALT))
 {
   objp->action = player_begin_shoot;
 }

 /* look down, look up */
 else if (fg_kbtest(KB_DOWN))
 {
   if (objp->y - tile_orgy*16 > 48)
     scroll_down(1);
 }
 else if (fg_kbtest(KB_UP))
```

```
  {
    if (objp->y - tile_orgy*16 < 200)
      scroll_up(1);
  }

  /* just standing there */
  else if (objp->sprite != tom_stand[0] && objp->frame < 7)
  {
    objp->frame = 0;
    objp->sprite = tom_stand[objp->frame];
  }
  else
  {
    /* change Tommy's facial expression */
    objp->time += delta_time;
    if (objp->time > max_time)
    {
      if (objp->frame == 0)
      {
        objp->frame = irandom(7,8);
        objp->time = 0;
        objp->sprite = tom_stand[objp->frame-6];

        /* how long we smile or frown is random */
        max_time = (long)irandom(200,400);
      }
      else
      {
        objp->frame = 0;
        objp->time = 0;
        objp->sprite = tom_stand[0];

        /* how long we stand straight without smiling */
        max_time = (long)irandom(500,1000);

      }
    }
  }
}
```

Here, keystrokes are processed as before, but we look at more cases. If a right or left arrow key is pressed, Tommy begins to run to the right or left. The appropriate structure fields are modified: the direction is set to **RIGHT** or **LEFT**, the speed is set to 1 or -1, the frame is set to 0, and the action function is set to **player_run()**. Similarly, if keys are intercepted for jumping, kicking, or shooting, the appropriate action function pointer is assigned to the **objp->action** field. If no keystroke is intercepted, the frame is set to 0 and the sprite image is set to **tom_stand[0]**, which is Tommy's standing still frame. As long as Tommy is not moving, the **tom_stand()** action function will continue to execute once each frame.

[2] Making Tommy Fidget
Characters are most endearing when they seem to have a mind of their own. Tommy is no exception. When he is supposed to be standing still, his personality shows through. He fidgets. Sometimes he grins and shrugs his shoulders. Other times he frowns. This is done at random intervals in the **player_stand()** function.

[3] Looking at the Time
To keep track of the random time intervals, we need to look at two variables. One of them is the **time** member of Tommy's object structure. The other is a global variable called **delta_time**. Whenever Tommy is doing nothing, the time interval is added to Tommy's time field, as follows:

```
objp->time += delta_time;
```

The **delta_time** variable is the amount of time elapsed since the last frame. As you recall from Chapter 12, the system clock has been accelerated to eight times the normal speed. That means the clock interrupt is called 145 times per second. The **delta_time** variable represents the number of clock ticks between the beginning of the last frame and the beginning of the current frame.Tommy's time field is increased by **delta_time** and then it is compared to a target value called **max_time**:

```
if (objp->time > max_time)
```

When the target is reached, it is time to change Tommy's expression.

Changing Tommy's expression is as easy as reassigning Tommy's sprite image. The structure field we are interested in is **objp->image**. This field points to the sprite representing the current incarnation of Tommy. If he is standing still and neither smiling nor shrugging, **objp->image** will point to **tom_stand[0]**. When he smiles or shrugs, **objp->image** will point at either **tom_stand[1]** or **tom_stand[2]**.

If Tommy is currently smiling or frowning, we set **max_time** to a random interval between 200 and 400 clock ticks. If Tommy is standing still, **max_time** is a random interval between 500 and 1,000 clock ticks. We give him a longer time interval for standing still than fidgeting. There is no exact formula to determine when Tommy should fidget. The above numbers were derived through trial and error. If you run the game and watch Tommy, you will see him smile and frown at random intervals. If you think Tommy should smile more often, you can decrease the range of values for **max_time**.

Tommy's **frame** member keeps track of his current frame of animation. Since Tommy has a six-stage walk, frames 1 through 6 are are associated with his walking frames. When his frame is 0, Tommy is standing still. We assign frames 7 and 8 to his

fidget frames. When Tommy's frame is 7, his image is **tom_stand[1]**. When his frame is 8, his image is **tom_stand[2]**.

[2] Making Tommy Jump

When the **player_stand()** action function detects a Ctrl keypress, it passes control of the Tommy sprite to the **player_begin_jump()** function.This function retains control for only one frame, as it prepares Tommy for his jump:

```
void near player_begin_jump(OBJp objp)
{
  /* called once at the start of a jump */

  objp->yspeed = -15;                   /* initialize variables */
  objp->frame = 0;
  shoot_time = 0;

  if (fg_kbtest(KB_LEFT) || fg_kbtest(KB_RIGHT)) /* walking? */
    forward_thrust = 50;
  else
    forward_thrust = 0;

  if (objp->direction == LEFT)
    tom_jump[3]->xoffset = 25;
  else
    tom_jump[3]->xoffset = 0;

  objp->sprite = tom_jump[objp->frame];    /* assign sprite */
  objp->action = player_jump;              /* assign action function */
}
/*********************************************************************/
void near player_begin_kick(OBJp objp)
{
  /*  called once at the start of a kick */

  int i;

  kicking = TRUE;                       /* initialize variables */
  objp->time = 0;
  nkicks = 0;

  /* is this a left (backward) or a right (forward) kick? */
  if (objp->direction == LEFT)
  {
    objp->frame = 0;
    kick_frame = 3;
    kick_basey = objp->y;
    objp->sprite = tom_kick[objp->frame]; /* assign sprite */

    /* back him up a little if needed */
```

```
    player->x += 36;
    for (i = 0; i < 36; i++)
    if (can_move_left(player))
      player->x--;
  }
  else
  {
    objp->frame = 6;
    kick_frame = 7;
    kick_basey = objp->y;
    objp->sprite = tom_kick[objp->frame]; /* assign sprite */

    /* back him up a little if needed */
    player->x -= 24;
    for (i = 0; i < 24; i++)
    if (can_move_right(player))
      player->x++;
  }
  objp->action = player_kick;             /* assign action function */
}
/*****************************************************************/
void near player_begin_shoot(OBJp objp)
{
  /* called once at the start of shooting */

  register int i;

  objp->frame = 0;                      /* initialize variables */
  objp->time = 0;
  nshots = 0;

  if (objp->direction == RIGHT)
  {
    tom_shoot[0]->xoffset = 2;
    tom_shoot[1]->xoffset = 2;
    tom_shoot[2]->xoffset = 2;

    /* back him up a little if needed */
    if (fg_kbtest(KB_RIGHT))              /* running while shooting? */
    {
      objp->sprite = tom_shoot[3];
      player->x -= 24;
      for (i = 0; i < 24; i++)
      if (can_move_right(player))
        player->x++;
    }
    else
      objp->sprite = tom_shoot[0];       /* assign sprite */
  }
  else
  {
```

```
    tom_shoot[0]->xoffset = -1;
    tom_shoot[1]->xoffset = -20;
    tom_shoot[2]->xoffset = -15;
    if (fg_kbtest(KB_LEFT))             /* running while shooting? */
      objp->sprite = tom_shoot[3];      /* assign sprite */
    else
      objp->sprite = tom_shoot[0];
  }
  objp->action = player_shoot;          /* assign action function */
}
```

Tommy's vertical speed is initialized to -15 at the start of the jump. As the jump progresses, the vertical speed will be incremented until it reaches 0. When upward speed is 0, Tommy is no longer going up and he will start to fall.

The **player_start_jump()** function introduces two new global variables: **shoot_time** and **forward_thrust**. The **shoot_time** variable determines the amount of time between bullets. When the fire key (Alt) is pressed, bullets will be spawned at the rate of approximately one every 15 clock ticks. The value of **shoot_time** is incremented until it reaches 15, then the bullet is spawned, and **shoot_time** is set back to 0. Again, this value was chosen by trial and error.

The **forward_thrust** variable affects the horizontal motion when an arrow key is pressed. If a left or right arrow is pressed, Tommy will move forward during the jump, but his amount of forward motion decreases, so he will move in a natural-looking arc. When the arrow key is released, forward_thrust is reset to 0.

Next, Tommy's frame is set to 0. This coincides with the frame in the **tom_jump[]** sprite list. The 0 frame is the image of Tommy jumping upward without shooting. This image will continue to be displayed until Tommy begins descending or starts shooting.

Finally, Tommy's action function is set to **player_jump(),** which will be active as long as Tommy is accelerating. Let's take a closer look at **player_jump()**.

```
void near player_jump(OBJp objp)
{
  int tile_x,tile_y;
  register int i;

  /* increment the timer, if it is less than 5, skip it */
  objp->time += delta_time;
  shoot_time += delta_time;
  if (objp->time > 5L)
    objp->time = 0;
  else
    return;
```

```
/* check for arrow keys, left arrow first */
if (fg_kbtest(KB_LEFT))
{
  objp->direction = LEFT;
  objp->xspeed = -3;

  /* forward thrust gives a little boost at start of jump */
  if (forward_thrust > 30)
    objp->xspeed *= 4;
  else if (forward_thrust > 0)
    objp->xspeed *= 2;

  /* move left, checking for walls, etc. */
  objp->xspeed = -how_far_left(objp,-objp->xspeed);
  objp->x += objp->xspeed;

  /* need to scroll the screen left? */
  tile_x = objp->x/16 - tile_orgx;
  if (tile_x < objp->tile_xmin)
    scroll_left(-objp->xspeed);
}

/* same for right arrow key */
else if (fg_kbtest(KB_RIGHT))
{
  objp->xspeed = 3;
  if (forward_thrust > 50)
    objp->xspeed *= 4;
  else if (forward_thrust > 0)
    objp->xspeed *= 2;
  objp->direction = RIGHT;
  objp->xspeed = how_far_right(objp,objp->xspeed);

  tile_x = objp->x/16 - tile_orgx;
  if (tile_x > objp->tile_xmax)
    scroll_right(objp->xspeed);
  objp->x += objp->xspeed;
}

/* decrement forward thrust */
forward_thrust--;

/* additional upward thrust if you hold down the Ctrl key */
if (fg_kbtest(KB_CTRL))
  objp->yspeed++;
else
  objp->yspeed/=4;

/* check bumping head on ceiling */
objp->yspeed = how_far_up(objp,objp->yspeed);
```

```
  objp->y += objp->yspeed;

  /* check if we are shooting */
  if (fg_kbtest(KB_ALT))
  {
    /* Tommy's jumping and shooting frame */
    objp->frame = 2;

    /* space the bullets 15 clock ticks apart */
    if (shoot_time > 15)
    {
      launch_bullet();
      shoot_time = 0;
    }
  }

  /* not shooting, just use Tommy jumping frame */
  else
    objp->frame = 0;

  /* set sprite to the correct frame */
  objp->sprite = tom_jump[objp->frame];

  /* too close to top of screen? scroll the screen up */
  tile_y = objp->y/16 - tile_orgy;
  if (tile_y < objp->tile_ymin)
    scroll_up(-objp->yspeed);

  /* reached top of arc? Tommy start descent */
  if (objp->yspeed >= 0)
    objp->action = player_begin_fall;
}
```

Th **player_jump()** function begins by regulating sprite motion according to the real-time clock. If fewer than five clock ticks have passed since the last frame, we skip the rest of the action function this frame. That means Tommy will be displayed at the same position this frame as he was in the last frame. So while the frame rate will vary on different computers, Tommy will move at approximately the same speed. Tommy's speed is dependent on the system clock, not the frame rate.

The **shoot_time** variable is also incremented every frame. Bullets are released at the rate of approximately one every 15 clock ticks, regardless of whether Tommy has moved or not. We have to increment the **shoot_time** variable every frame in order for the bullets to be evenly spaced, whether or not Tommy is moving or shooting during this particular frame.

If sufficient time has passed, the action function goes to work. The first thing it does is check for specific key presses, starting with the left arrow key. If the left arrow key is

pressed, Tommy's horizontal speed is set appropriately. The speed is modified by the amount of horizontal thrust, which was set in the **player_start_jump()** function, and is decremented later in this function. This is how Tommy jumps to the left.

[3] How Far Can He Go?
The horizontal speed is modified by the function **how_far_left()**, which determines how far away Tommy is from a wall or other barrier. So while Tommy's speed determines how far left he will move, this value can be cut short by **how_far_left()**. The **how_far_left()** function is in the file MOTION.C and looks like this:

```
int how_far_left(OBJp objp,int n)
{
  register int i;
  register int temp;

  temp = objp->x;       /* save the current position */

  /* increment the position until you can't move left any further */
  for (i = 0; i < n; i++)
  {
    objp->x--;
    if (!can_move_left(objp))
    {
      objp->x = temp;
      return(i);
    }
  }
  objp->x = temp;      /* restore the current position */
  return(n);           /* return how far left */
}
```

The **how_far_left()** function saves Tommy's x position in a temporary variable called **temp**. It then decrement's Tommy's x coordinate and calls **can_move_left()** sequentially, until either **can_move_left()** fails, or we have gone as far as we wanted to go in the first place. Then the temporary variable is copied back into Tommy's x coordinate. The purpose of this function is not to actually modify the x coordinate, just to report how much it can be modified in the left direction.

The **can_move_left()** function calls the **test_bit()** function to check the tile attributes of the adjacent tile. If the tile is solid on the right, the object can not move left.

```
int can_move_left(OBJp objp)
{
  int tile_x,tile_y,tile_num;

  /* test the bottom of the sprite */
  tile_x = (objp->x-1)/16;
  if (tile_x <= 0)
```

```
    return(FALSE);
  tile_y = objp->y/16;
  tile_num = (int)background_tile[tile_x][tile_y];

  /* is the tile solid on the right? */
  if (test_bit(background_attributes[tile_num],3))
    return(FALSE);

  /* check the top of the sprite too */
  tile_y = (objp->y - objp->sprite->height)/16;
  tile_num = (int)background_tile[tile_x][tile_y];
  return (!test_bit(background_attributes[tile_num],2));
}
```

For good measure, tiles at the top and the bottom of the sprite are checked. Tommy cannot move forward if either his head or his feet will bump into a wall.

Both **how_far_left()** and **can_move_left()** work on other objects besides Tommy. Bullets and enemies are also restricted in their motion. Bullets should not shoot through walls, for example. The other motion functions work in a similar manner: **how_far_right()** calls **can_move_right()**, **how_far_up()** calls **can_move_up()**, and **how_far_down()** calls **can_move_down()**.

The **test_bit()** function is quite simple. It just returns the value of a bit in a byte.

```
test_bit(char num,int bit)
{
  /* test bit flags, used for tile attributes */
  return((num >> bit) & 1);
}
```

[3] Time to Scroll

We can't let Tommy move too far left or he will move off the edge of the screen. We need to calculate Tommy's position with respect to the tile origin. If he has moved beyond the minimum tile tolerance, the screen needs to scroll. We choose to scroll the screen left the same number of pixels as Tommy moved left. The scroll will look smoother if it scrolls at the same speed as Tommy's horizontal movement. Here is the code to check Tommy's position relative to the tile origin and then scroll left:

```
    tile_x = objp->x/16 - tile_orgx;
    if (tile_x < objp->tile_xmin)
      scroll_left(-objp->xspeed);
```

The **scroll_left()** function expects a positive number as the number of pixels to scroll. Since Tommy's **xspeed** field is negative when he walks left, we pass the negative **xspeed** to **scroll_left()** and it works out to a positive number.

The same code is executed for the right arrow key. Obviously, Tommy can't move left and right at the same time, so if both the left and right arrow keys are pressed, the left motion will take precedence over the right motion.

[3] Shooting while Jumping

Tommy can also shoot while he is jumping. The **player_stand()** function next checks for the Alt key, which signals that Tommy is shooting at something while he jumps. The shooting while jumping image is frame 3 in the **tom_jump[]** sprite list. The frame is set to 3 whether or not a bullet is going to be released this frame. As we said earlier, bullets are only started when **shoot_time** exceeds 15, which means at least 15 clock ticks have elapsed since the last time a bullet was spawned. The function **start_bullet()** spawns a bullet and adds it to the linked list. More about **start_bullet()** in a minute.

If Tommy is not shooting, his image is set to frame 0 of the **tom_jump[]** sprite list. Once Tommy's image is set to the proper sprite, it's time to consider vertical motion.

As with horizontal motion, the amount Tommy moves depends on whether a key is being pressed. If the Ctrl key is pressed, Tommy will move up faster than when the Ctrl key is released. A slight tap on the Ctrl key makes Tommy jump a tiny amount, and a prolonged press of the Ctrl key causes Tommy to reach his maximum height. As with horizontal movement, Tommy can only move vertically until he reaches a barrier. The **how_far_up()** function modifies the vertical speed so that he will not continue going up if he bumps his head on the ceiling.

Again, Tommy's position is compared to the edge of the screen. If he has moved beyond the minimum vertical tile tolerance, the screen will scroll vertically to accommodate him.

[3] What's Next?

When Tommy ascends, his vertical speed is negative. The vertical speed is decreased a little each frame, either by incrementing it when the Ctrl key is pressed, or by dividing it by four when the Ctrl key is not pressed. Either way, eventually Tommy's vertical speed will reach 0. When that happens, Tommy is no longer moving up. He has reached the top of his jump and is ready to start descending. When Tommy's vertical speed reaches 0, the **player_jump** action function is replaced by **player_start_fall()**. The **player_start_fall()** action function initializes the falling variables in a manner similar to **player_start_jump()**. It then sets the action function to **player_fall()**. The **player_fall()** function is very similar to the **player_go_up()** function, so I'm not going to list it here. The biggest difference between the jumping and falling functions is that Tommy's vertical speed increases in the falling function, and he will continue to go down until he hits a solid tile. The **player_fall()** function, along with all of Tommy's action functions, are

included on the companion disk.

[1] Bullet Action Functions
So far, we have only looked at Tommy's action functions, but Tommy is not the only object that has them. All the objects have action functions. In fact, one action function is executed for each object every frame. The non-Tommy action functions are interesting when they modify the linked list. For an example of some action functions that are not Tommy's, let's take a look at the bullets.

[2] Launching Bullets
Bullets are spawned and killed quite often. Every time the Alt key is pressed, Tommy fires a bullet, at the rate of one every 15 clock ticks. Tommy can fire a bullet while standing, running, or jumping. If Tommy's action function determines it is time to fire a bullet, it will call the **launch_bullet()** function. While **launch_bullet()** is not an action function itself (it is not called as a pointer to a function in an object structure) it launches a new object and assigns control of that object to its associated action function. Let's take a look at how this works:

```
void near launch_bullet()               /* start a new bullet */
{
  OBJp node;

  if (nbullets > 9) return;             /* max 9 bullets */

  node = (OBJp)malloc(sizeof(OBJ)+3);   /* allocate space */
  if (node == (OBJp)NULL) return;

  if (player->direction == RIGHT)       /* assign values */
  {
    node->direction = RIGHT;
    node->xspeed = 13;
    if (player->sprite == tom_jump[2])  /* jumping */
    {
      node->x = player->x+player->sprite->xoffset+46-node->xspeed;
      node->y = player->y-25;
    }
    else if (player->sprite == tom_jump[3]) /* falling */
    {
      node->x = player->x+player->sprite->xoffset+46-node->xspeed;
      node->y = player->y-25;
    }
    else if (fg_kbtest(KB_RIGHT))      /* running */
    {
      node->x = player->x+player->sprite->xoffset+40-node->xspeed;
      node->y = player->y-26;
    }
    else                                /* standing */
```

```
    {
      node->x = player->x+player->sprite->xoffset+40-node->xspeed;
      node->y = player->y-28;
    }
  }
  else
  {
    node->direction = LEFT;
    node->xspeed = -13;
    node->x = player->x+player->sprite->xoffset-node->xspeed-5;
    if (player->sprite == tom_jump[2])      /* jumping */
      node->y = player->y-25;
    else if (player->sprite == tom_jump[3]) /* falling */
      node->y = player->y-25;
    else if (fg_kbtest(KB_LEFT))        /* running */
      node->y = player->y-26;
    else                                /* standing */
      node->y = player->y-28;
  }
  node->yspeed    = 0;
  node->tile_xmin = 1;
  node->tile_xmax = 21;
  node->tile_ymin = 0;
  node->tile_ymax = 14;
  node->sprite = tom_shoot[6];          /* assign the sprite */

  node->action = bullet_go;             /* assign action function */

  /* insert the new object at the top of the linked list */
  if (bottom_node == (OBJp)NULL )
  {
    bottom_node = node;
    node->prev = (OBJp)NULL;
  }
  else
  {
    node->prev = top_node;
    node->prev->next = node;
  }
  top_node = node;
  node->next = (OBJp)NULL;

  nbullets++;                           /* increment bullet count */
}
```

The first thing **launch_bullet()** does is see how many bullets are currently active. Through trial and error, I determined nine bullets on the screen at one time are about enough. If Tommy stands in the middle of the screen and fires towards one edge, the first bullet will go off the edge of the screen before the ninth bullet is spawned. Even when Tommy is standing at the edge of the screen, there is no noticeable gap between bullets

when the maximum is set to nine. Feel free to change this number if you want to. As with most of the arbitrary values in this game, the programmer should choose unique values so his game will not look like everybody else's.

We keep track of the number of bullets in the variable **nbullets**. This value is incremented as bullets are added, and decremented when bullets are killed. If there are fewer than nine bullets are currently flying, **launch_bullet()** proceeds to launch a new one. It starts by using the C runtime library function **malloc()** to allocate space for the object. If **malloc()** is unable to allocate space for the bullet (which may happen if there are already many objects on the screen), we return without creating this bullet. In most cases, there will be no problem allocating the room for the bullet[3], so we proceed to initialize the object by plugging values into the members of the object structure.

The direction of the bullet depends on the direction of the player. If the player is facing right, the bullet will move to the right, and if the player is facing left, the bullet will move to the left. The x and y position of the bullet also depends on the position of the player. We want the bullet to come out of the end of Tommy's gun, not out of his knee or his foot. The exact location of the end of Tommy's gun depends on what Tommy is currently doing. We look at Tommy's image and the keyboard to determine if Tommy is currently jumping, falling, running, or standing still, and calculate the x and y position accordingly.

The bullet has no vertical speed; it always moves horizontally at the rate of 13 pixels per frame. The tile extents are set so the bullet will be killed if it goes beyond the edge of the screen in any direction. The image field for the bullet is set to **tom_shoot[6]**. The bullet's action function is set to **bullet_go()**.

[3] Sharing Action Functions
The pointer to the object structure, **objp**, is always passed to the action function. In the case of Tommy, the object will always be the player--we only have one main character. That is not the case for most of our objects, however. Most objects have multiple copies. You may have up to nine bullets on the screen at one time, for example. When that happens, all the bullet objects will point to the same action function, as shown in Figure 13.1.

Figure 13.1 Several objects pointing to the same action function.

[2] A Linked List of Objects

To keep our objects properly organized, we store them in a linked list. The bullet is added to the top of the linked list, and the pointers to the next and previous nodes are properly initialized. The bullet becomes the top node of the linked list. Objects are always added to the top of the list, and the last object spawned is always the top node. It is a safe bet this bullet won't stay on top for long. It is very likely another bullet will be spawned

in about 15 clock ticks, and this bullet will move down the list and the next bullet will become the top node. Bullets come and go frequently.

As we said earlier, **launch_bullet()** is not an action function. It does, however, assign an action function to the bullet, called **bullet_go()**. The **bullet_go()** function is the main action function for the bullet.

```
void near bullet_go(OBJp objp)
{
 int min_x,max_x;
 register int i;

 /* increment the bullet's horizontal position */
 objp->x += objp->xspeed;

 /* collision detection */
 for (i = 0; i < nenemies; i++)
 {
   if (enemy[i]->frame < 6 && objp->x>enemy[i]->x
       && objp->x<enemy[i]->x+enemy[i]->sprite->width
       && objp->y<enemy[i]->y
       && objp->y>enemy[i]->y-enemy[i]->sprite->height)
   {
     launch_floating_points(enemy[i]);
     enemy[i]->frame = 6;
     objp->action = kill_bullet;
   }
 }

 /* check if the bullet has moved off the screen */
 max_x = (tile_orgx + objp->tile_xmax) * 16;
 min_x = (tile_orgx + objp->tile_xmin) * 16;

 /* if it has moved off the screen, kill it by setting the action
    function to kill for the next frame */

 if (objp->x > max_x || objp->x < min_x)
   objp->action = kill_bullet;

 if (objp->direction == RIGHT && !can_move_right(objp))
   objp->action = kill_bullet;
 else if (objp->direction == LEFT && !can_move_left(objp))
   objp->action = kill_bullet;
}
```

The first thing **bullet_go()** does is increment (or decrement) the horizontal position of the bullet as determined by the bullet's speed. In other words, the bullet moves 13 pixels to the left or the right. Then the action function checks for a collision between the

bullet and any of Tommy's enemies. It does this by scanning an array of pointers to enemy objects. If a collision is detected, the enemy object is flagged by setting its frame number to 6. The enemy's action function will handle the death throes of the enemy. The bullet's action function is only concerned with the activity of the bullet.

[2] Collision Detection

We are often quite interested in happens when two objects collide. When a bullet collides with an enemy, we will want to initiate the sequence of events that leads to the death of both the bullet and the enemy, and more points for Tommy. To detect a collision, we call the **collision_detection()** function.

```
int collision_detection (OBJp objp1, OBJp objp2)
{
  int xmin1,xmax1,xmin2,xmax2;
  int ymin1,ymax1,ymin2,ymax2;

  /* x coordinates of object 1 */
  xmin1 = objp1->x+objp1->image->xoffset;
  xmax1 = xmin1+objp1->image->width;

  /* x coordinates of object 2 */
  xmin2 = objp2->x+objp2->image->xoffset;
  xmax2 = xmin2+objp2->image->width;

  /* y coordinates of object 1 */
  ymax1 = objp1->y+objp1->image->yoffset;
  ymin1 = ymax1-objp1->image->height;

  /* y coordinates of object 2 */
  ymax2 = objp2->y+objp2->image->yoffset;
  ymin2 = ymax2-objp2->image->height;

  /* object 2 entirely to the left of object 1 */
  if (xmax2 < xmin1) return(FALSE);

  /* object 2 entirely to the right of object 1 */
  if (xmin2 > xmax1) return(FALSE);

  /* object 2 entirely to the below object 1 */
  if (ymax2 < ymin1) return(FALSE);

  /* object 2 entirely to the above object 1 */
  if (ymin2 > ymax1) return(FALSE);

  /* the objects overlap */
  return(TRUE);
}
```

This function uses a simple rectangular collision detection. It checks for two objects for four cases: If object 2 is entirely to the left or the right of object 1, or if it is entirely above or below object 1, then there is no collision. Otherwise, there is a collision. See figure 13.2 for a diagram of the collision detection scheme.

Figure 13.2 The collision_detection() function checks for four cases

Note that the **collision_detection()** function only detects collisions between two objects. Collisions between 4objects and tiles are detected in the motion functions such as **can_move_left()**.

There are more accurate and complicated collision detection alogrithms. Our function is inexact, but exact collision detection is not required for this kind of game. The action in *Tommy's Adventures* is fast and furious, it is difficult to see whether a bullet has actually hit an enemy, or just brushed past its tail. In an action game like **Street Fighter** or **Mortal Kombat**, accuate collision detection is more critical. You would not want your player to get points for throwing air punches!

The feel of our collision detection scheme was gauged through trial and error. After trying the simple detection, and discovering it felt reasonable, I discarded the more complicated collision detection algorithms. Most side-scroller games use a simple rectangular collision detection.

The accuracy of the collision detection function could be improved somewhat by using the bounding box information as the basis of the collisions, instead of the position and width and height of the sprite. For some odd-shaped sprites, that enhancement could be useful, but implementing bounding box collision detection is left as an excercise for you.

[2] Killing Bullets

Bullets don't do much. They move forward until they die. They die when they collide with an enemy, hit a wall, or if they go off the edge of the screen. If any of these events happens, the bullet is not killed immediately. Instead, its action function is set to **kill_bullet()**, and the object is killed on the next frame. Here is the **kill_bullet()** action function.

```
void near kill_bullet(OBJp objp)
{
 /* decrement the bullet count and kill the bullet */

 nbullets--;
 kill_object(objp);
}
```

This function decrements the number of bullets, and also calls **kill_object()**. The **kill_object()** function removes an object from the linked list. Any killable objects, such as bullets and enemies, are killed using this function. Here is the **kill_object()** function.

```
void near kill_object(OBJp objp)
{
 /* remove the object from the linked list */

 OBJp node;

 node = objp;
 if (node == bottom_node)
 {
   bottom_node = node->next;
   if (bottom_node != (OBJp) NULL)
     bottom_node->prev = (OBJp)NULL;
 }
 else if (node == top_node)
 {
   top_node = node->prev;
   top_node->next = (OBJp)NULL;
 }
 else
 {
   node->prev->next = node->next;
   node->next->prev = node->prev;
 }
 free(node);
}
```

The **kill_object()** function uses a traditional method for removing a node from a linked list: It checks the position of the object within the list and reassigns the pointers accordingly. The object is then freed using C's **free()** function. The memory allocated for this node is now free to be re-allocated for a new object.

[1] Enemy Action Functions

The other major kind of objects in our game are enemies. Enemies are important because they add an element of challenge to the game. Producing unusual and creative enemy sprites is one of the most important elements of a game's success.

Something usually triggers the spawning of an enemy. For example, an enemy may be launched when Tommy moves into a new area in a level. Tommy may step on a tile that triggers the spawning of an enemy. Sometimes enemies are time based, and a new enemy will appear every minute or two. Sometimes enemies appear at random. Most often, all the enemies are created at the beginning of the level in fixed locations, and once they are killed they are gone for good. This last method seems to be the most appealing to players. It's okay to have an occasional random enemy, but if most of your enemies are in

fixed locations, the player will have a chance to learn the level and anticipate the appearance of enemies.

Boss Enemies

It is customary in side-scrolling games to have a boss enemy. This is a large, fierce enemy usually encountered at the end of a level or at the end of an episode. It is usually very difficult to defeat. Unlike the regular enemies, it must be hit many times at just the right angle in order to be killed. Boss enemies serve as a dramatic climax to a game, and prolong the play time of a game at the end because they are so hard to kill.

Different classes of enemies have different action functions. In our game, Tommy faces some huge, scary insects. One is a grasshopper, and the other is a pink scorpion. We use the same code to launch both types of enemies because they are so similar.

```
void near launch_enemy(int x, int y, int type)  /* start a new enemy */
{
 OBJp node;

 node = (OBJp)malloc(sizeof(OBJ));        /* allocate space */
 if (node == (OBJp)NULL) return;

 node->direction = RIGHT;                /* assign values */
 node->x = x;
 node->y = y;
 node->xspeed = 8;
 node->yspeed = 0;
 node->tile_xmin = 1;
 node->tile_xmax = 21;
 node->tile_ymin = 0;
 node->tile_ymax = 14;
 node->time = 0;

 /* assign the sprite and action function */
 if (type == 0)
 {
   node->frame = 0;
   node->action = enemy_scorpion_go;
 }
 else
 {
   node->frame = 3;
   node->action = enemy_hopper_go;
 }
 node->sprite = enemy_sprite[node->frame];

 /* insert the new object at the top of the linked list */
 if (bottom_node == (OBJp)NULL )
 {
```

```
    bottom_node = node;
    node->prev = (OBJp)NULL;
  }
  else
  {
    node->prev = top_node;
    node->prev->next = node;
  }
  top_node = node;
  node->next = (OBJp)NULL;

  enemy[nenemies] = node;                /* update enemy array */
  nenemies++;                          /* increment enemy counter */
}
```

The grasshopper enemy is not terribly complicated. All it does is walk left or right at a fixed speed. If it comes to the end of a platform, it falls off. If it is hit by a bullet or kicked, it dies. Here is the grasshopper's main action function:

```
void near enemy_hopper_go(OBJp objp)
{
  if (objp->frame > 4)   /* is this enemy dying? */
  {
    /* after 100 frames, kill this enemy off */
    objp->frame++;
    if (objp->frame > 100)
      objp->action = kill_enemy;
    objp->sprite = enemy_sprite[5];

    /* enemy can fall while dying */
    objp->yspeed = how_far_down(objp,12);
    if (objp->yspeed > 0)
      objp->y += objp->yspeed;

    /* no point in doing anything else while dying */
    return;
  }

  /* this enemy moves every 12 clock ticks */
  objp->time += delta_time;
  if (objp->time < 12)
    return;
  else
    objp->time = 0;

  objp->yspeed = how_far_down(objp,12);    /* falling? */
  if (objp->yspeed > 0)
    objp->y += objp->yspeed;
  else
  {
```

```
    /* increment the object's horizontal position */
    if (objp->direction == LEFT)
    {
      if (!can_move_left(objp))
      {
        objp->direction = RIGHT;
        objp->xspeed = 12;
      }
    }
    else if (objp->direction == RIGHT)
    {
      if (!can_move_right(objp))
      {
        objp->direction = LEFT;
        objp->xspeed = -12;
      }
    }
    objp->x += objp->xspeed;
  }
  objp->frame = 7-objp->frame;          /* increment the frame */
  objp->sprite = enemy_sprite[objp->frame];

  /* if the player hasn't been hit recently, can we hit him now? */
  if (!player_blink)
  {
    if (collision_detection(objp, player) && !kicking)
    {
      player_blink = TRUE;            /* make the player blink */

      nhits++;                      /* seven hits per life */
      if (nhits > 7)
      {
        nlives--;
        if (nlives == 0)             /* three lives per game */
          nlives = 3;
        nhits = 0;
      }

      /* update the action function for the score */
      score->action = update_score;
    }
  }
}
```

When hit, the grasshopper dies slowly. It appears as a dead grasshopper sprite, which stays on the screen for 75 frames. The object's frame field is used to count the dying frames. Meanwhile, a floating score is launched and moves upward as the grasshopper dies. After the prolonged death throes of the grasshopper, it is killed the same way a bullet is killed: Its action function is set to **kill_enemy()**, which removes it from the

enemy array and the linked list. At this point, you could say the enemy is out of the loop.

Floating Scores
A floating score has become traditional in Apogee-style, side-scrolling games. It is a number, usually a three-digit number, that floats upward over the corpse of an enemy. It indicates how many points you earned for the kill.

It is always good to give the player some satisfaction for defeating enemy sprites. Similar reward devices include making the enemy flicker, change into something else, or explode.

In non-violent games, it is common for enemies to be stunned rather than killed, or to be changed into something friendly. In Goodbye Galaxy, for example, enemies look dazed when shot, as indicated by a halo of stars circling above their heads. In Sonic the Hedgehog, evil robots are turned into happy woodland creatures.

The enemy action function has an interesting anomaly--it modifies the action function of another object. That is, when the grasshopper dies, it sets the action function of the score object to **update_score()**. In general, action functions only change the actions of their own objects but this is a special case. We'll discuss the score object some more in the next section.

The scorpion behaves in a manner very similar to the grasshopper (for the purposes of this book, we've kept our enemies *very* simple-minded). I'm sure you can come up with better enemies than this. Enemies that fly or jump are interesting. Enemies that have enough artificial intelligence to hunt you down and kill you are also fun. This is one of the areas of game design where your creative ideas will separate an excellent game from an ordinary game. I encourage you to spend a lot of time designing interesting enemies.

[1] Score Action Functions
The score object is a special object. It is displayed in the same place in the upper-left corner of the screen every frame unless it is turned off, in which case it is not displayed in any frame. To optimize the score for speed, we don't redraw the numbers every frame. If there are four digits in the score, it would require four bitmaps to be drawn every frame, as well as the outline of the scoreboard. This would cause only a tiny speed degradation, but even tiny ones are significant when added up over hundreds of frames. To avoid drawing unnecessary characters every frame, we'll update the score only when it changes.

In the frames when the score has not changed, the **put_score()** action function looks like this:

```
void near put_score(OBJp objp)
{
```

```
  /* determine x and y coords based on the screen origin */
  objp->x = tile_orgx*16 + screen_orgx + 2;
  objp->y = tile_orgy*16 + screen_orgy + 43;
}
```

This function simply adjusts the x and y coordinates according to the screen origin so the score will appear in the same place every frame, but does nothing to change the sprite bitmap.

If the score has changed, for example, when Tommy shoots a grasshopper, then the sprite must be updated to reflect the new score. This is done in the **update_score()** action function:

```
void near update_score(OBJp objp)  /* called when score has changed */
{
  char string[128];
  SPRITE *scoremap;
  int y;
  register int i;

  /* Convert the (long) score to a character string.  Assume 10 digits
     is enough */

  ltoa(player_score,string,10);

  /* clear an area in video memory below the tile space where nothing
     else is going on */

  fg_setcolor(0);
  fg_rect(0,319,680,724);

  /* draw the score box in offscreen memory */
  scoremap = tom_score[0];
  fg_move(0,724);
  fg_drwimage(scoremap->bitmap,scoremap->width,scoremap->height);

  /* set the color to black and display the score */
  fg_setcolor(1);
  center_string(string,5,56,720);

  /* the status bar indicates how many times you have been hit */
  y = nhits*3;

  fg_setcolor(14);
  if (nhits == 0)           /* all blue */
  {
    fg_setcolor(18);
    fg_rect(62,67,701,723);
  }
```

```
  else if (nhits >= 8)       /* all white */
  {
    fg_setcolor(14);
    fg_rect(62,67,701,723);
  }
  else
  {                          /* white and blue */
    fg_setcolor(14);
    fg_rect(62,67,701,700+y);
    fg_setcolor(18);
    fg_rect(62,67,700+y,723);
  }

  scoremap = tom_score[1];    /* tommy one-ups */
  for (i = 0; i < nlives; i++)
  {
    fg_move(80+i*10,716);
    fg_drwimage(scoremap->bitmap,scoremap->width,scoremap->height);
  }

  /* do a getimage to put the score in a bitmap in RAM */
  objp->sprite->width = 80+10*nlives;
  fg_move(0,724);
  fg_getimage(objp->sprite->bitmap,
            objp->sprite->width,objp->sprite->height);

  /* update the x and y coords */
  objp->x = tile_orgx*16 + screen_orgx + 2;
  objp->y = tile_orgy*16 + screen_orgy + 43;

  /* assign action function */
  objp->action = put_score;
}
```

The **update_score()** action function creates a new bitmap. It does this by drawing the score box in offscreen video memory. First the **fg_rect()** function is used to draw a rectangle. Then **fg_drwimage()** is used to put the scoreboard outline on top of the rectangle. Finally, the score is drawn on top of the score box. Fastgraph's **fg_getimage()** function is used to grab the image and store it in a sprite bitmap in RAM. See Figure 13.4 for a picture of video memory with the score sprite being redrawn below the tile area.

Figure 13.4 Score sprite being redrawn in offscreen video memory.

The shaded area in Figure 13.4 is where the score is redrawn in video memory. Tommy one-ups are also drawn in this area. The one-ups are represented as miniature pictures of Tommy.

One-Ups

A one-up is an extra life for the main player. Traditionally, a character begins a game with three one-ups. Every time a character dies, the number of one-ups decreases. If the character dies three times, it uses the last one-up, and the game is over.

Some games allow the player to find one-ups in the level, or earn additional one-ups by finding other objects. One-ups should not be confused with energy, which is reduced incrementally as the player encounters hazards, or continues, which are chances to restart the game at the current level when all the one-ups are exhausted.

[1] The Creativity of Sprite Animation

This chapter has covered the basic elements of controlling sprite movements through action functions. Once again, I would like to stress that creativity is an absolute necessity in this part of game programming. The action functions we have looked at are the bare-bones minimum amount of code needed to create sprite motion. You can use these functions as templates for your own sprites, but plan on modifying and adding to them. A game needs to have interesting sprites to be a success, and you'll spend a lot of time designing them. Sprite design is an inexact science. The best way to approach it is through trial and error. Implement your ideas, try them out, and see if they feel right. Keep experimenting with the action functions until you are satisfied that your sprites express the personality, emotion, and challenge that will set your game apart from all the others.