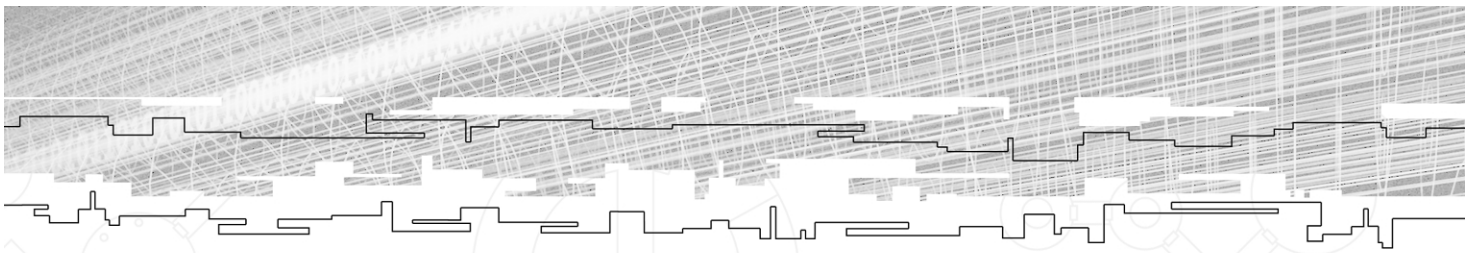
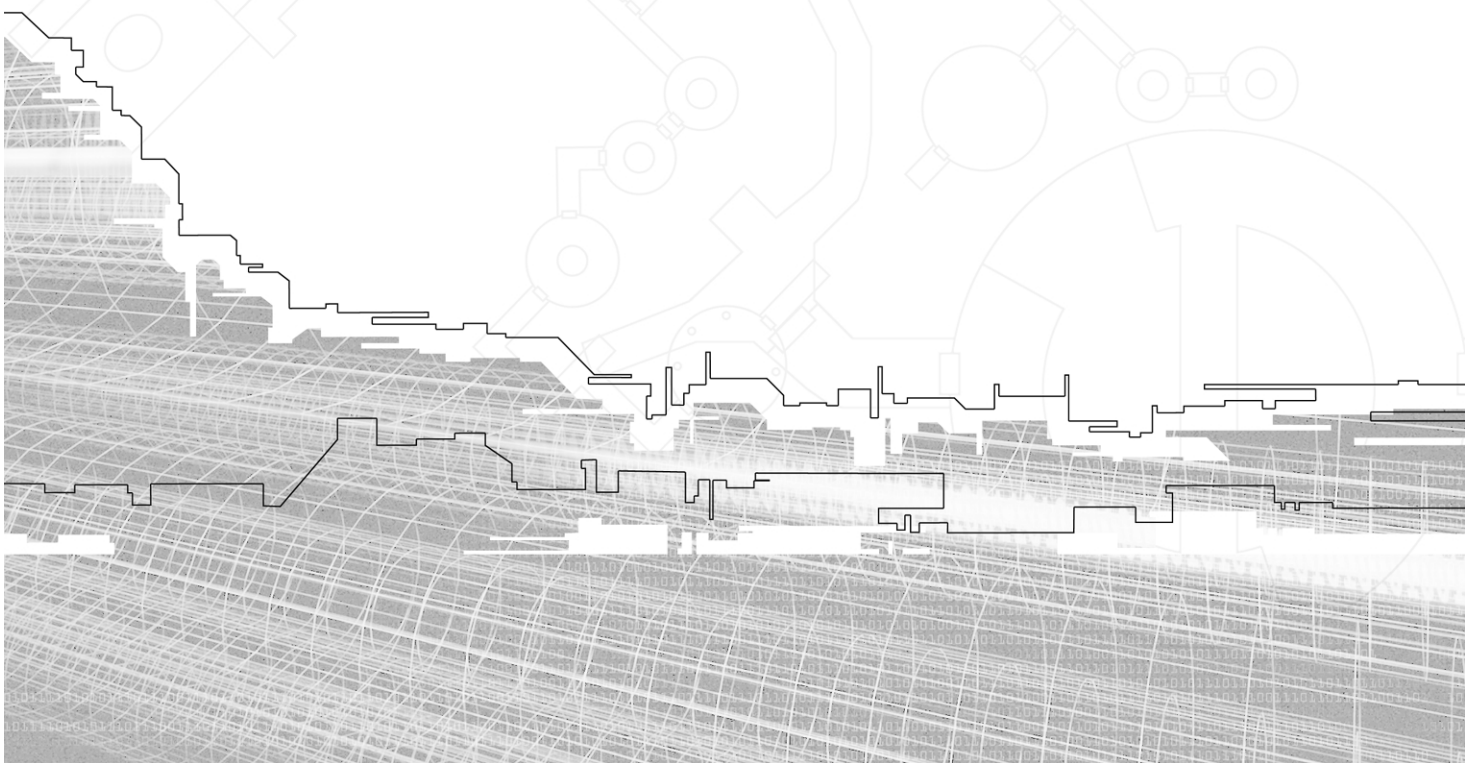


CHAPTER 17

MATRIDIA II: THE PROJECT



You've made it through the work, and now it's time to play—after all, what is all work and no play? In this final chapter, you will experience all that you learned in this book in one package; you will dissect a complete game called *Matridia II*. And during this dissection, you will revisit the following topics and experience how they interact together in a complete game:

- The idea
- The approach
- The setup
- The code
- The future

The Idea

You might be wondering why there is a sequel to a game that you've never heard of before. One of the first games I created for the DOS operating system was called *Matridia*. It was a top-view shooter with a cheesy story, but the real challenge was creating it from scratch. I took from the original idea for this chapter's project, and it led to the sequel.

Lucky for you, you won't have to write any video or sound card code because you already have drivers installed for those purposes. Instead, you will be dissecting the Flash version of *Matridia*, *Matridia II*; doing so will give you that extra push, allowing you to head off on your own into the world of Flash.

As for the game's ideas, I decided I wanted a scrolling star field. I wanted to imitate depth by using different levels of alpha. To improve the effect, I also planned on assigning different sizes and speeds to each star.

I wanted a lot of explosions, missiles and bombs, so I decided to create Movie Clips of all of these. They all have their linkage properties set up too.

In *Matridia II*, you can shoot the bad guys to gain some points when flying through space, but don't let them hit you, because you will lose power. If you make it through all the floating sine control enemies, you will make it to the Boss—this guy loves to spit tons of firepower, so watch out!

If he kills you, you will see a Game Over screen and you will be taken to the splash screen. If you win, you will be congratulated and then taken to the splash screen.

Check out Figure 17.1 for a shot of the splash screen.

Figure 17.1

The Matridia II splash screen

See Figure 17.2 for a look at the game in action.

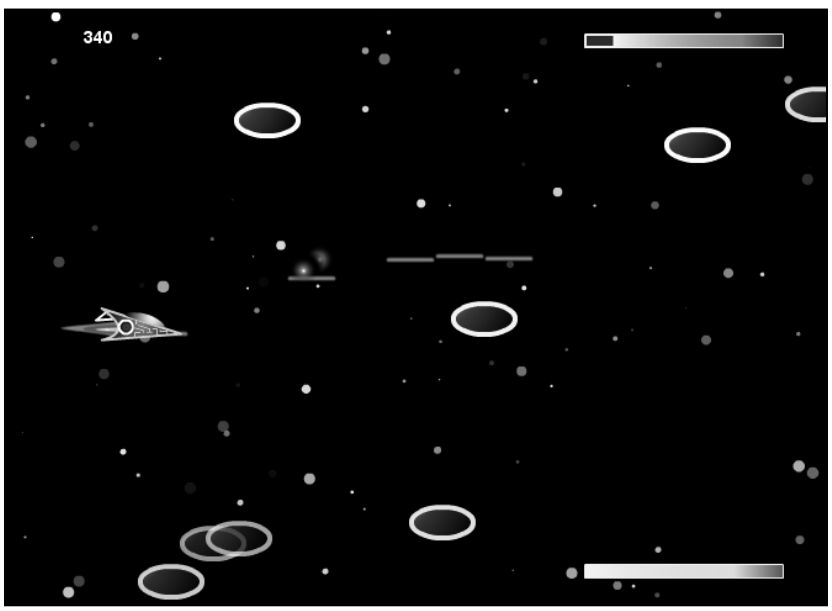


Figure 17.2

Matridia II in action

Figure 17.3 shows the player battling the Boss.

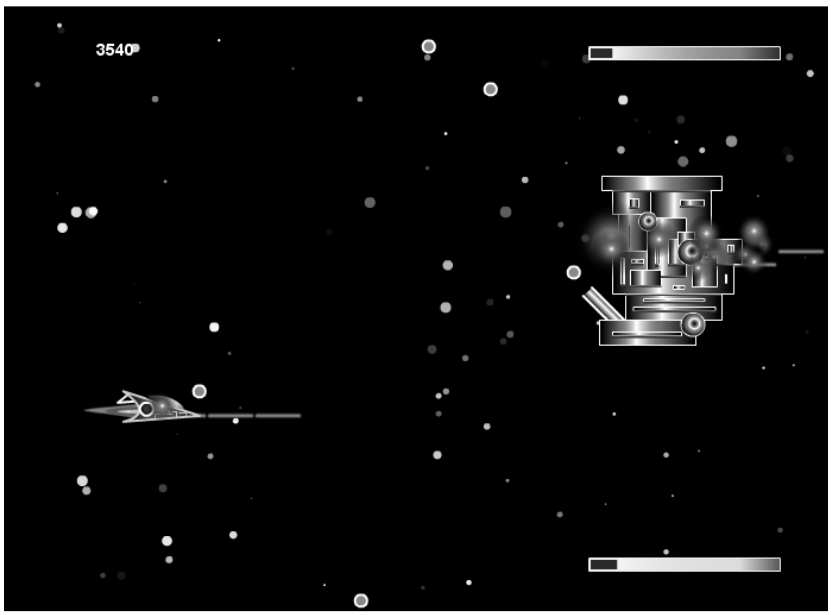


Figure 17.3

Save the universe!

The Approach

I decided to tackle this piece by piece. The first thing I decided to create was the star field. This star field needs to scroll to the left at all times while moving the stars with different speeds, sizes, and opacities. I accomplished this by using, from the library, one linked symbol that was nothing more than a white circle with no stroke. When I created the instances with `attachMovie`, I assigned new `_alpha`, scale and velocity values. To move all of them once they were created, all I did was loop through each instance within each frame while adding the velocities to their x values. This caused the simultaneous motion.

The player's ship was one of the easiest things to create. I first drew the ship with all the built-in animations I wanted (like the engine flame animation) and made it into a symbol that can be linked into the movie.

As far as motion goes, the player is restricted to moving the ship up and down with the respective keyboard arrows.

The player needed some missiles with which to attack, so I also created a symbol for the missiles. This symbol helped the program create a certain number of instances that can be active at one time. In other words, one variable controls how many missiles the player can shoot at once.

The enemies that I created for this game are very dumb. All they do is float to the left while moving in a sine motion. I created the enemies in much the same way I created the star particles—much of the difference is in the motion code.

After a few seconds of battle, I have the big Boss come out to kick some butt. The energy bar at the bottom of the screen is his, the one up top is the player's, and the battle is to the end. The Boss was created in much the same way the main character was created, except that the computer moves him up and down at all times. I wanted the Boss's bombs to launch at random. I also set them up the same way I did the star field and the main character's missiles—in such a way that only a certain number can be launched at once.

Once I had made decisions on all these factors and elements, I began setting up the source file.

The Setup

One of the first things I did to my FLA project was create scenes. This was my first step towards the organization of the game. See Figure 17.4 and check out all the scenes I created in this project.

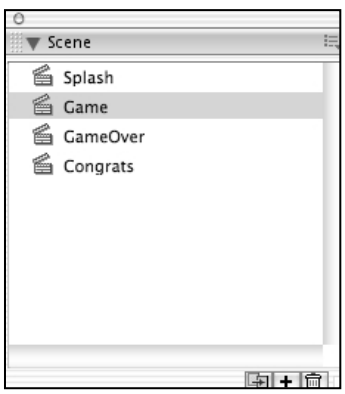


Figure 17.4

Scenes set up for Matridia II

The Splash scene is simply the screen that introduces the game and leads the player to either modify options or to play the game. The Game scene is where all the code is stored. There are three layers here that simply carry the code, a dynamic textbox, and an empty Movie Clip with more code—I'll go over this later.

The GameOver and Congrats scenes are very simple scenes that alert the player to some sort of resolution within the game; if he wins or loses, the program will play the respective scene.

If you take a look into the Library window, you will notice that there are many linked symbols. These symbols are used to create the instances you see within the game. All these symbols were drawn as Movie Clips and they therefore contain properties that can be managed from within ActionScript.

The Code

The code was written in a day's worth of work. It may look really complicated because there's lot of it, but focus on the smaller parts and how they unite as a whole. I approached the code writing just as I approached each idea in the game.

Once of the things you will notice when you look into the Game scene is that there is an empty Movie Clip in the middle of the screen. The purpose of the Movie Clip is to manage all the functions stored in the first frame with the `onClipEvent` handlers.

As you have already guessed, the game is run by two main `onClipEvent` functions. One of them initializes the game and the other helps animate all the frames. Go into the Game scene and click F9 after selecting the blank Movie Clip in the middle of the stage. See the following listing for the code within this clip. Without this code, nothing will happen and all the other functions will be ignored.

NOTE

You might be wondering why I didn't use the `onEnterFrame` and `onLoad` callback functions on the main timeline instead of the `onClipEvent` functions from inside another Movie Clip. The reason is because when the `onEnterFrame` and `onLoad` functions are defined anywhere except the first frame of the first scene, they won't work. This is why I decided to use the similar functions built inside a regular Movie Clip.

```
// Game Development with ActionScript
// By Lewis Moronta (c) 2003

// Initialize the game
onClipEvent(load) {

  // Change this number to change
  // the amount of stars in the field
  _root.MAXSTARS = 100;

  // Change this value to adjust
  // the number of missles and bombs
  // that can be fired at once.
  _root.MAXMISSILES = 6;
  _root.MAXBOMBS = 10;

  // This variable will decide
  // if enemies or boss should attack
  _root.BOSSMODE = false;

  // This decides how many enemies
  // can exist at one time...
  _root.MAXENEMIES = 20;
```

```
  // Setup how many explosions
  // can explode at one time...
  _root.MAXEXPLOSIONS = 13;

  // Hide the cursor
  // Mouse.hide();

  // This plots stars all over the place
  _root.initializeStarField();

  // Create the ship
  _root.initializeShip();

  // Set up the enemies
  _root.initializeEnemies();

  // Set up the missiles
  _root.initializeMissiles();

  // Set up the explosions
  _root.initializeExplosions();

  // Setup the Boss
  _root.initializeBoss();

  // Now that everything is set up
  // let's keep track of the time...
  _root.startTime = getTimer();
}

// Let there be movement!
onClipEvent(enterFrame) {

  // Let's make sure the stars
  // move along and wrap around.
  _root.moveStarField();

  // Allow the player to move
  // the space ship
  _root.moveShip();

  // Attack...
  _root.moveEnemies();

  // Fire!!!
  _root.moveMissiles();
```

```
  // Move the big daddy
  _root.moveBoss();

  if ((getTimer()-_root.startTime) > 30000) {
    _root.BOSSMODE = true;
  }

  _root.scoreDisplay = _root.Player.score;
}
```

The `onClipEvent(load)` function performs a few interesting tasks. First, it initializes a few important variables called *constants* because they dictate a few important attributes to the game, and never change throughout the game.

```
  // Change this number to change
  // the amount of stars in the field
  _root.MAXSTARS = 100;

  // Change this value to adjust
  // the number of missiles and bombs
  // that can be fired at once.
  _root.MAXMISSILES = 6;
  _root.MAXBOMBS = 10;

  // This variable will decide
  // if enemies or BOSS should attack
  _root.BOSSMODE = false;

  // This decides how many enemies
  // can exist at one time...
  _root.MAXENEMIES = 20;

  // Setup how many explosions
  // can explode at one time...
  _root.MAXEXPLOSIONS = 13;
```

You can change the value assigned to `MAXSTARS` to modify the number of stars in the universe, and you can also change the value assigned to `MAXENEMIES` to state how many enemies can be on the stage at one time. By changing the values assigned to `MAXMISSILES` and `MAXBOMBS`, you change the number of missiles that the player can fire at one time and how many bombs the Boss can spit out at once. `MAXEXPLOSIONS` is another variable that you can change to allow any number of explosions to explode on the stage at one time. `BOSSMODE` is a special game state variable that tells the game whether to produce more enemies or to allow the Boss to attack.

As instances had to be created based on these constants, I created initialization functions that do just that:

```
// This plots stars all over the place
_root.initializeStarField();

// Create the ship
_root.initializeShip();

// Set up the enemies
_root.initializeEnemies();

// Set up the missiles
_root.initializeMissiles();

// Set up the explosions
_root.initializeExplosions();

// Set up the Boss
_root.initializeBoss();

// Now that everything is setup
// let's keep track of the time...
_root.startTime = getTimer();
```

The `initializeStarField` function creates `MAXSTAR` number of stars. It assigns additional properties to each star, as you will soon see. The `initializeShip` creates an instance of the player's ship and restricts it to an axis. The `initializeEnemies`, `initializeExplosions` and `initializeMissiles` are very similar to one another—they all create a certain number of enemies, explosions, and missiles, and place them in their proper positions.

The `initializeBoss` function creates the Boss's instance hiding off the side of the screen until the game enters `BOSSMODE`.

The variable `startTime` records the time at which the game started. `getTimer` is used in this case, because it returns the number of milliseconds from the beginning of the movie. `startTime` will be used later to decide when `BOSSMODE` should be entered.

Now that I've finished initializing the game, I now enter the `onClipEvent(enterFrame)` function. This function starts off by moving the star field.

```
// Let's make sure the stars
// move along and wrap around.
_root.moveStarField();
```

After moving the star field, the player is moved by receiving input that the following function interprets:

```
// Allow the player to move
// the space ship
_root.moveShip();
```

I needed a function that could generate enemies at random, so I created the `moveEnemies` function which waits until an enemy dies to create a new one. It can also consider an enemy dead if it flies off the left side of the stage.

```
// Attack...
_root.moveEnemies();
```

In order to manage the firing missiles that are launched from the player's ship, I had to create a `moveMissiles` function.

```
// Fire!!!
_root.moveMissiles();
```

The last thing I wanted to move (and only if the game is in `BOSSMODE`) is the Boss. I wrote the following lines to do so:

```
// Move the big daddy
_root.moveBoss();
```

One of the things I needed was to shift the game into `BOSSMODE` after a certain period of time. I did this by checking to see whether 30 seconds had passed—if so, I then switched to `BOSSMODE` by setting the `BOSSMODE` variable to true.

All of the core functions that we've just gone over were written within the first frame of the Actions layer of the Game scene on the main timeline. I will discuss these functions in the following sections.

If you took a peek at how long the code on the main Timeline is, there probably is a defeated look on your face, but think of it this way: The code could have been much more complicated. I kept it simple because after all, this is a beginner book. You should be able to understand all of this code without a problem. Let's jump into the code.

The Star Field Functions

There are only two star field functions that create the effect of flying through the universe. The first function to explore is the `initializeStarField` function. Take a look at Figure 17.5 for a screen shot of some stars.

Figure 17.5

Take a look at the stars! All of them are instances of the same symbol.

```
function initializeStarField() {

  // Let's create MAXSTARS Movie Clips
  for (var i = 0; i < MAXSTARS; i++) {

        // Attach the movie from the linked symbol
        attachMovie("StarLayer", "star"+i, i);

        // Store the clip for easy reference
        var star = _root["star"+i];

        // Plot it at a random place
        star._x = Math.random()*640;
        star._y = Math.random()*480;

        // Adjust the alpha channel for
        // a depth effect...
        star._alpha = Math.random()*100;

        // Adjust the scale for a greater
        // depth effect...
        star._xscale = Math.random()*80;
        star._yscale = _root["star"+i]._xscale;

        // Make sure stars move from 2 to 22 pixels per frame
        star.velocity = -(Math.round(Math.random()*20)+2);
  }
}
```

The first thing you bump into within this code is a loop that is looping for `MAXSTARS` times. Every time the loop iterates, it attaches a new instance and stores the instance name in a simple variable for easy access.

```
        // Attach the movie from the linked symbol
        attachMovie("StarLayer", "star"+i, i);

        // Store the clip for easy reference
        var star = _root["star"+i];
```

After the instance has been created, the star instance is placed in a random position.

```
        // Plot it at a random place
        star._x = Math.random()*640;
        star._y = Math.random()*480;
```

As an added effect, I decided to modify the opacity of the white instance—this helps enhance the depth effect.

```
        star._alpha = Math.random()*100;
```

Just to vary things, I scaled each star at a random interval.

```
        // Adjust the scale for a greater
        // depth effect...
        star._xscale = Math.random()*80;
        star._yscale = _root["star"+i]._xscale;
```

And finally, to make the particles an even more interesting piece of the game, I decided to make them move at different speeds.

```
        // Make sure stars move from 2 to 22 pixels per frame
        star.velocity = -(Math.round(Math.random()*20)+2);
```

As each star is initialized with random values, this causes an interesting star field that becomes a big part of this game.

The next function we will check out will be `moveStarField`. This function starts off by looping through all of the stars available.

```
function moveStarField() {

  // Let's move ALL the star clips
  for (var i =  0; i < MAXSTARS; i++) {

        // Store for easy reference
        var star = _root["star"+i];

        // Move it along with its
        // own velocity property.
        star._x += star.velocity;

        // Wrap it around if
        // it's off the screen.
        if (star._x < -star._width)
          star._x = star._width+640;
  }
}
```

Within each loop, the symbol's velocity moves each star by adding its own velocity property that was initialized in the previous function.

The last thing the program does is check to see whether the star is off the left side of the screen. If it is, the star is forced to wrap around to the right side of the screen.

And this concludes the star field code. Remember that the stars are loaded and moved from within the empty Movie Clip I discussed in previous sections.

The Ship Functions

Now that you know exactly how the star field was created, you should have very little trouble figuring out how the player's ship was created and moved. Take a look at Figure 17.6 so you can see what the ship looks like.

Figure 17.6

Matridia II's main character

The following function is the `initializeShip` function that creates the player's instance:

```
function initializeShip() {

  // Let there be Player!
  attachMovie("Ship", "Player",8000);

  // Position him...
  Player._x = 100;
  Player._y = 240;

  // Y-Axis Velocity
  Player.yv = 0;

  // Y-Axis Acceleration
  Player.ya = 4;

  // Setup Score
  Player.score = 0;

  // Setup the Power Bar
  initPowerBar();
}
```

The ship was given a depth of 8000—it started out as an arbitrary number but as I wrote the game, the depth became a relative thing. What I mean by this is that I wanted the ship above its missiles, and explosives over the ship and missiles, and so on.

The ship was then given a position and velocity and acceleration properties. The score property was also assigned; this property will keep track of all the points the player earns.

You will also notice an `initPowerBar` function. This function creates and places an energy bar on the upper right of the screen. You'll see how it is adjusted later on.

The `moveShip` function is a bit long but simple. Let's jump right into it.

```
function moveShip() {
```

```
// Adjust the velocities
if (Key.isDown(Key.UP)) {
   Player.yv += -Player.ya;
 }

 if (Key.isDown(Key.DOWN)) {
   Player.yv += Player.ya;
 }

 var MAXVEL = 20;

 // Cap Velocity
 if (Player.yv > MAXVEL)
   Player.yv = MAXVEL;

 if (Player.yv < -MAXVEL)
   Player.yv = -MAXVEL;

 // Move the player
 Player._y += Player.yv;

 // Add Friction
 if (Player.yv > 0)
   Player.yv--;
 if (Player.yv < 0)
   Player.yv++;

 // Set boundaries
 if (Player._y < 0)
   Player._y = 0;

 if (Player._y > (480-Player._height))
   Player._y = (480-Player._height);


////////////////////////////////////////////////
//// Check If Player Fired ////////////////////
////////////////////////////////////////////////
  if (Key.isDown(Key.SPACE)) {

    for (var i = 0; i < MAXMISSILES; i++) {
          // Store for easy reference
          var missile = _root["missile"+i];

          // Find an inactive one and activate it.
          if (!missile.fired) {
```

```
                // Set flags
                missile.fired = true;
                missile._visible = true;

                // Make it follow ship with
                // some compensation values
                missile._x = Player._x-42;
                missile._y = Player._y+18;
                break;
            }
          }
    }
}
```

One of the first things the function is doing is detecting when the user presses either the Up or Down keys. These keys cause the acceleration to be added to its velocity value, which in turn causes the ship to move vertically either up or down.

I needed to cap this value so that the ship doesn't fly off the screen as a result of exaggerated values.

```
var MAXVEL = 20;

  // Cap Velocity
  if (Player.yv > MAXVEL)
    Player.yv = MAXVEL;

  if (Player.yv < -MAXVEL)
    Player.yv = -MAXVEL;
```

And finally, to make the ship move no matter what the velocity is, I made sure the velocity variable is added to its current y position.

```
  // Move the player
  Player._y += Player.yv;
```

There is very little friction in space, but I decided to break some rules and add a lot of it. All I did was subtract 1 from the velocity until the velocity is 0. It worked out perfectly.

```
// Add Friction
  if (Player.yv > 0)
    Player.yv--;
  if (Player.yv < 0)
    Player.yv++;
```

And of course, I had to add constraints to the ship. I didn't want the player to fly off the screen.

```
// Set boundaries
  if (Player._y < 0)
    Player._y = 0;
```

```
  if (Player._y > (480-Player._height))
    Player._y = (480-Player._height);
```

The next section is quite a big section within this function. It's a segment of code that fires a missile if there is one available to fire. This is only done if the user presses or holds the spacebar.

```
if (Key.isDown(Key.SPACE)) {

    for (var i = 0; i < MAXMISSILES; i++) {
          // Store for easy reference
          var missile = _root["missile"+i];

          // Find an inactive one and activate it.
          if (!missile.fired) {
               // Set flags
               missile.fired = true;
               missile._visible = true;

               // Make it follow ship with
               // some compensation values
               missile._x = Player._x-42;
               missile._y = Player._y+18;
               break;
          }
       }
  }
```

What I did here was loop through all the missiles looking for one that was not fired. If an unfired missile is found, I then set its status to "fired" and "visible." To finish the initialization, I set up a position that looks as if it is coming out of the ship's guns.

The `moveMissiles` function will detect the missiles whose status was switched to "fired" and move them.

The Missiles Functions

Let's jump right into the `initializeMissiles` function. Many of the familiar constructs are being used here.

```
function initializeMissiles() {
  for (var i = 0; i < MAXMISSILES; i++) {

        // Attach the movie from the linked symbol
        attachMovie("Ammo", "missile"+i, 7000+i);

        // Store the clip for easy reference
        var missile = _root["missile"+i];
```

```
        // Make sure they are inactive
        missile._visible = false;

        // Boolean value that
        // decides to launch ammo
        missile.fired = false;
  }
}
```

The initialization function goes through the usual loop—it iterates for `MAXMISSILES`. It then attaches an instance and then stores this Movie Clip within a variable for easy access.

All of the missiles are made invisible until the player triggers them. They are also all set to "not fired" status. The "not fired" status will give them the passive state that we want.

The `moveMissiles` function needs to be dissected carefully for you to understand all the parts. Take a look at the function:

```
function moveMissiles() {
  for (var i = 0; i < MAXMISSILES; i++) {

    // Store for easy reference
    var missile = _root["missile"+i];

    if (missile.fired) {

      // Move it if it was fired
      missile._x += missile._width;

      // Kill it if it went off the screen
      if (missile._x > 640) {
        missile.fired = false;
        missile._visible = false;
      }

      for (var j = 0; j < MAXENEMIES; j++) {

        var badGuy = _root["Enemy"+j];

        if (missile.hitTest(badGuy)) {

          Player.score += 20;

          badGuy.alive = false;
          badGuy._visible = false;

                if (!BOSSMODE) {
```

```
            badGuy._x = badGuy._width + 640;
            badGuy._y = Math.round(Math.random()*480);

          } else {

                // Draw it outside of the screen...
                // this avoids any random explosions
                // caused by the hitTest function
                // testing to true...
                badGuy._x = -badGuy._width;
               badGuy._y = 0;
          }

          missile.fired = false;
          missile._visible = false;

         // Start and explosion
          for (var k = 0; k < MAXEXPLOSIONS; k++) {

           // Store the clip for easy reference
           var Explosion = _root["Explosion"+k];

           if (!Explosion.active) {
             Explosion._x = missile._x+(missile._width/2);
             Explosion._y = missile._y;
             Explosion.active = true;
             Explosion.gotoAndPlay(2);
             break;
           }
          } // End for k

      } // End if missile.hitTest
} // End for j

if (missile.hitTest(bigBoss)) {

    Player.score += 40;
   bossPower.Bar._xscale -= 0.5;

  // Game Over!
  if (bossPower.Bar._xscale < 0) {
    bossPower.Bar._xscale = 0;

    unLoadGame();
    gotoAndPlay("Congrats", 1);
  }
```

```
        // Start and explosion
        for (var j = 0; j < MAXEXPLOSIONS; j++) {

          // Store the clip for easy reference
          var Explosion = _root["Explosion"+j];

          if (!Explosion.active) {

            Explosion._x = missile._x+(missile._width/2);
            Explosion._y = missile._y;
            Explosion.active = true;
            Explosion.gotoAndPlay(2);
            break;
          }
        } // End for j
      }

    } // End if missile.fired
  } // End for i
} // End moveMissiles
```

Beyond the usual, the program checks to see whether the missile is fired before acting upon it. If it has been fired, the program then moves the missile along. If it has gone off the screen, the missile will be reset.

```
if (missile.fired) {

      // Move it if it was fired
      missile._x += missile._width;

      // Kill it if it went off the screen
      if (missile._x > 640) {
        missile.fired = false;
        missile._visible = false;
      }
```

The program then loops through all the enemies to see which ones have been hit. It then adds to the player score if there was a collision and kills the enemies that have been hit.

```
for (var j = 0; j < MAXENEMIES; j++) {

        var badGuy = _root["Enemy"+j];

        if (missile.hitTest(badGuy)) {

          Player.score += 20;
```

```
        badGuy.alive = false;
        badGuy._visible = false;
```

To clean up a little, the missile is also removed temporarily:

```
            missile.fired = false;
            missile._visible = false;
```

To add to the effect, I then looked for an inactive explosion to use, then set it off where the point of contact occurred.

```
      for (var j = 0; j < MAXEXPLOSIONS; j++) {

        // Store the clip for easy reference
        var Explosion = _root["Explosion"+j];

        if (!Explosion.active) {

          Explosion._x = missile._x+(missile._width/2);
          Explosion._y = missile._y;
          Explosion.active = true;
          Explosion.gotoAndPlay(2);
          break;
        }
      } // End for j
```

This code can be easily implemented in any game you create, so make sure you understand it.

The Enemy Functions

As I said before, the enemies are not programmed much differently from the rest of the game. They are initialized then moved. See Figure 17.7 for a screen shot of a bunch of the enemies in action.

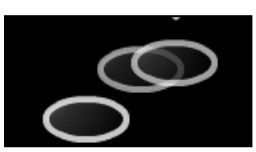


Figure 17.7

Enemies in action

Examine the following initialization function:

```
function initializeEnemies() {
  for (var i = 0; i < MAXENEMIES; i++) {

        attachMovie("sineEnemy", "Enemy"+i, 9000+i);

        var badGuy = _root["Enemy"+i];
```

```
        badGuy._x = badGuy._width + 640;
        badGuy._y = Math.round(Math.random()*440+40);

        badGuy.xv = -Math.round(Math.random()*10+10);
        badGuy.yv = 0;

        badGuy.alive = false;
        badGuy._visible = false;

        badGuy.degrees = Math.round(Math.random()*360);
        badGuy.radians = Math.PI/180*badGuy.degrees;

        badGuy._alpha = Math.round(Math.random()*60+40);
  }
}
```

Each enemy was placed randomly on the stage. They were also given random velocities. To start off, they were made invisible and their `alive` status property was set to false.

In order to have them follow their own sine path on the y axis, I added two new properties, `degrees` and `radians`.

To make things more interesting, I also played with the enemies' alpha channels. They will be transparent from a level of 40 to 100 percent.

```
function moveEnemies() {

  // If the Boss steps on stage,
  // make sure no new enemies are spawned.
  if (!BOSSMODE) {
    for (var i = 0; i < MAXENEMIES; i++) {

          var badGuy = _root["Enemy"+i];

          // Spit out another enemy only if random()
          // returns 1 (or true) and the badGuy is not alive...
          if (!badGuy.alive && (Math.round(Math.random()*30) == 3)) {
            badGuy.alive = true;
            badGuy._visible = true;
            break;
          }

    }
  }

  for (var i = 0; i < MAXENEMIES; i++) {

        var badGuy = _root["Enemy"+i];
```

```
// If alive, move the enemy along...
if (badGuy.alive) {

  badGuy._x += badGuy.xv;

  if (badGuy.degrees++ > 360)
    badGuy.degrees = 0;

 badGuy.radians = Math.PI/180 * badGuy.degrees;
 badGuy.yv = Math.round(Math.sin(badGuy.radians)*2);
 badGuy._y += badGuy.yv;


 // If an enemy hits the Player,
 // dock the Player 2 percent...
 if (badGuy.hitTest(Player)) {
      Power.Bar._xscale -= 2;

       // Game Over!
       if (Power.Bar._xscale < 0) {
         Power.Bar._xscale = 0;

         unLoadGame();
         gotoAndPlay("GameOver", 1);
      }

    // Start and explosion
    for (var j = 0; j < MAXEXPLOSIONS; j++) {

     // Store the clip for easy reference
     var Explosion = _root["Explosion"+j];

     if (!Explosion.active) {
           Explosion._x = Player._x;
          Explosion._y = Player._y+12;
          Explosion.active = true;
           Explosion.gotoAndPlay(2);
           break;
      }
    }
 }

 // If off the left side, reset.
 if (badGuy._x < -badGuy._width) {
   badGuy.alive = false;
   badGuy._visible = false;
```

```
            badGuy._x = badGuy._width + 640;
            badGuy._y = Math.round(Math.random()*440+40);
          }
        }
  }
}
```

If the game is not in `BOSSMODE`, the game will keep generating enemies until it is.

```
if (!BOSSMODE) {
    for (var i = 0; i < MAXENEMIES; i++) {

          var badGuy = _root["Enemy"+i];

          // Spit out another enemy only if random()
          // returns 1 (or true) and the badGuy is not alive...
          if (!badGuy.alive && (Math.round(Math.random()*30) == 3)) {
            badGuy.alive = true;
            badGuy._visible = true;
            break;
          }

    }
  }
```

Once the function is finished spawning new enemies, it finds the enemies that are alive and moves them.

```
for (var i = 0; i < MAXENEMIES; i++) {

        var badGuy = _root["Enemy"+i];

        // If alive, move the enemy along...
        if (badGuy.alive) {

          badGuy._x += badGuy.xv;
```

Depending on the enemy's `degrees` value, the enemy will move along a sine curve—this makes each enemy look as if it is a lower life form.

If by chance the bad guy hits the player, the code will decrease the scale of the player's power bar. If the scale is below zero, the game will remove all of the Movie Clips and will play the Game Over screen.

```
          if (badGuy.hitTest(Player)) {
               Power.Bar._xscale -= 2;

                // Game Over!
                if (Power.Bar._xscale < 0) {
                  Power.Bar._xscale = 0;
```

```
        unLoadGame();
        gotoAndPlay("GameOver", 1);
    }
```

What the code then does is set off an explosion. This is, of course, is relative to where the impact occurred.

```
for (var j = 0; j < MAXEXPLOSIONS; j++) {

    // Store the clip for easy reference
    var Explosion = _root["Explosion"+j];

    if (!Explosion.active) {
          Explosion._x = Player._x;
          Explosion._y = Player._y+12;
          Explosion.active = true;
          Explosion.gotoAndPlay(2);
          break;
    }
 }
```

The final piece of code in this function tests to see if the enemy is off the left side of the screen—if it is, it is killed.

```
if (badGuy._x < -badGuy._width) {
  badGuy.alive = false;
  badGuy._visible = false;
  badGuy._x = badGuy._width + 640;
  badGuy._y = Math.round(Math.random()*440+40);
}
```

The Explosions Function

The explosions themselves required more setup than the other objects because there is an animation to the explosion Movie Clip. When first triggered, this animation is stopped and blank on the first frame. When played, the animation would have to start playing from Frame 2 and the script on the last frame of the clip would make it jump and pause it again in Frame 1; this makes the explosion symbol reset ready for yet another animation.

As you understand the structure of the Movie Clip, just check out the code that I wrote for it. The function you'll see here is `initializeExplosions`. This is the only function that is needed to make any other part of the program ready to cause some explosions.

```
function initializeExplosions() {

  for (var i = 0; i < MAXEXPLOSIONS; i++) {
```

```
        // Attach the movie from the linked symbol
        attachMovie("Explosion", "Explosion"+i, 11000+i);

        // Store the clip for easy reference
        var Explosion = _root["Explosion"+i];

        Explosion.active = false;
  }
}
```

As you can see, this function is nothing different from what we have been doing before. The only thing that needs to be explained is that I assigned a new property, called `active`, to each new clip. This helps the program track which explosion is dormant so I can wake it up when I need to.

The Boss Functions

The big Boss functions complete the game and can easily be rewritten for a different boss AI. In order to have the Boss load up, I had to write up an initialization function called `initializeBoss`. You probably would have guessed that already.

```
function initializeBoss() {

  attachMovie("Boss", "bigBoss", 10000);

  bigBoss._x = 640+bigBoss._width;
  bigBoss._y = 240-(bigBoss._height/2);

  bigBoss.yv = 10;

  initBossPowerBar();
  initializeBombs();
}
```

As you can see, the Boss function sets up its own bombs and power bar. This leaves no worries for the rest of the game. I gave the Boss a vertical velocity and also placed him off the screen so that he can scroll in with the following code—the `moveBoss` code:

```
function moveBoss() {

  if (BOSSMODE) {
        bigBoss._x += -5;

        if (bigBoss._x < 430)
          bigBoss._x = 430;

        bigBoss._y += bigBoss.yv;
```

```
        if (bigBoss._y < 0)
          bigBoss.yv = -bigBoss.yv;
        if (bigBoss._y > (480-bigBoss._height))
          bigBoss.yv = -bigBoss.yv;

        moveBombs();
  }
}
```

If the game enters `BOSSMODE`, the Boss will start scrolling left until it hits a certain point on the screen. Thereafter, the Boss will start moving up and down according to this code. At the same time, the `moveBombs` code is called. You'll see what that does shortly.

The Bomb Functions

The bomb functions are the last functions I needed for this to be a complete game. Just like every other part of this game, I initialized these bombs with the following code:

```
function initializeBombs() {
  for (var i = 0; i < MAXBOMBS; i++) {
        attachMovie("Bomb", "Bomb"+i, 9500+i);

        var bossBomb = _root["Bomb"+i];

        bossBomb._x = bigBoss._x+2;
        bossBomb._y = bigBoss._y+96;
        bossBomb.fired = false;

        bossBomb.force = 2;
  }
}
```

One thing that you noticed here is that the bombs were placed all relative to the `bigBoss`. They start out as all dormant, and I also added a mysterious gravity-like force property to them. I thought the force made their motion interesting.

The `moveBombs` function became a little more exciting with all the action written within it. Check it out:

```
function moveBombs() {
  for (var i = 0; i < MAXBOMBS; i++) {

        var bossBomb = _root["Bomb"+i];

        // Decide when to fire...
        if (Math.round(Math.random()*30) == 3) {
          if (!bossBomb.fired) {
                bossBomb.fired = true;
```

```
            bossBomb._x = bigBoss._x+2;
            bossBomb._y = bigBoss._y+96;

            bossBomb.xv = -Math.round(Math.random()*10+10);
            bossBomb.yv = -20;
        }
      }

      if (bossBomb.fired) {
        bossBomb._x += bossBomb.xv;
        bossBomb._y += bossBomb.yv;

        bossBomb.yv += bossBomb.force;
    }

if (bossBomb._x < -bossBomb._width) {
  bossBomb.fired = false;
}

if (bossBomb._y > 480) {
  bossBomb.fired = false;
}

if (bossBomb.hitTest(Player)) {
  Power.Bar._xscale -= 2;

  // Game Over!
if (Power.Bar._xscale < 0) {
   Power.Bar._xscale = 0;

     unLoadGame();
     gotoAndPlay("GameOver", 1);
}

  // Start and explosion
  for (var j = 0; j < MAXEXPLOSIONS; j++) {

     // Store the clip for easy reference
     var Explosion = _root["Explosion"+j];

     if (!Explosion.active) {
       Explosion._x = Player._x;
       Explosion._y = Player._y+12;
       Explosion.active = true;
       Explosion.gotoAndPlay(2);
       break;
```

```
        }
      }
    }
  }
}
```

After entering the loop and going through all the bombs available in the game, you'll see the code that decides when to fire another bomb.

```
if (Math.round(Math.random()*30) == 3) {
  if (!bossBomb.fired) {
        bossBomb.fired = true;

        bossBomb._x = bigBoss._x+2;
        bossBomb._y = bigBoss._y+96;

        bossBomb.xv = -Math.round(Math.random()*10+10);
        bossBomb.yv = -20;
  }
}
```

The code will only fire a new bomb if a random number from the range of 0 to 30 is 3. The chances of that are 1 out of 30. This causes the bombs to vary their behavior a bit. So if the bomb is not fired, it will position itself by the Boss's cannon and get itself ready to fire.

If the bomb has fired, the following block causes it to move in a projectile way.

```
if (bossBomb.fired) {
  bossBomb._x += bossBomb.xv;
  bossBomb._y += bossBomb.yv;

  bossBomb.yv += bossBomb.force;
}
```

If by chance the bomb doesn't hit the player and it goes off the side or bottom of the screen, the following code was written to disable it and reserve it for the next throw:

```
if (bossBomb._x < -bossBomb._width) {
  bossBomb.fired = false;
}

if (bossBomb._y > 480) {
  bossBomb.fired = false;
}
```

If the bomb hits the player, 2 percent is subtracted from the player's power bar.

```
if (bossBomb.hitTest(Player)) {
   Power.Bar._xscale -= 2;
```

```
    // Game Over!
    if (Power.Bar._xscale < 0) {
      Power.Bar._xscale = 0;

      unLoadGame();
      gotoAndPlay("GameOver", 1);
}
```

And finally, if a hit was detected, the following code starts up some fire:

```
for (var j = 0; j < MAXEXPLOSIONS; j++) {

      // Store the clip for easy reference
      var Explosion = _root["Explosion"+j];

      if (!Explosion.active) {
        Explosion._x = Player._x;
        Explosion._y = Player._y+12;
        Explosion.active = true;
        Explosion.gotoAndPlay(2);
        break;
      }
}
```

The Future

If you dedicate any time to upgrading this game, you will bump into the bugs and frustrations that are part of every programmer's life. What you can do is plan carefully before adding anything to this program. Don't forget to use the Debugger, your friendly trace command, and any other strategic algorithms that can get you out of a programmer's pit.

If you follow the same style *Matridia II* was written in, you should have no problems adding more enemies, power-ups, power-shields, and explosives. Just remember one thing: the limitations of your viewer's computer. Not everybody has a power computer. Flash MX 2004 has been improved and is at least 25% more speed-efficient than Flash MX, but the user's computer will forever affect your code.

Now that you have dissected a complete game, the next step would be to create your own.

Summary

You broke apart a complete game in this chapter. All the principles that you have learned throughout the book were combined here in one chapter. You went through the idea, setup, approach, and code that went into creating *Matridia II*. You learned how to install a nice star field into a working game. You also learned how to create explosives, bombs, enemies, and even bosses that work together to make a complete game. Enjoy!

Questions & Answers

Q. How does the flame stay animated off the ship's back?

A. The animated flame is nothing more than a Movie Clip that is tweened with the same frame in the beginning and end of its Timeline. This causes the smooth transition.

Q. The enemies were coming out at random—is there a way I can write something more stable?

A. Yes. You can store a pattern in an array and have the game check it at a certain interval. This will cause the enemies to come out in a predictable manner.

Exercises

1. Add an option screen to this game that allows the user to modify the constants that were set in the initialization part of the game. HINT: Make them global.
2. Make the game have at least three levels and create three different bosses for each stage.
3. Give the player three lives—in other words, three chances to finish the game. Every time he loses a life, have the bar go back to 100%.
4. Create a nice animated ending that will only play if the player can get through the game without dying.