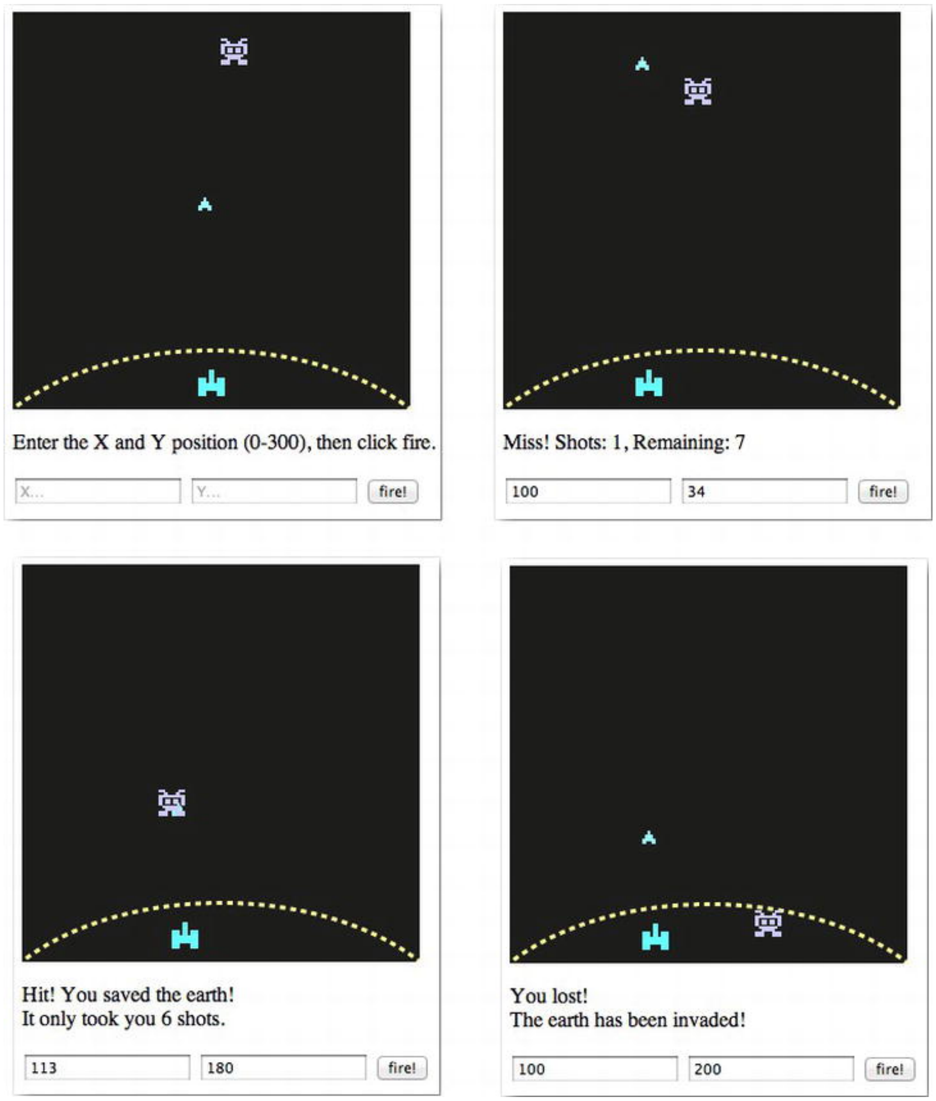
Imagine that instead of guessing a mystery number, you're trying to guess the position of an invading space alien so that you can shoot it down with nuclear missiles. That sounds like a lot of fun to me! You'll see a working example of this in the **`alienAttack.html`** file. Figure 3-21 shows what you'll see.


Enter the X and Y position (0-300), then click fire.

X... Y...

Miss! Shots: 1, Remaining: 7

100 34

Hit! You saved the earth!
It only took you 6 shots.

113 180

You lost!
The earth has been invaded!

100 200

Figure 3-21. Try and guess the X and Y positions of the invading alien

Try and guess the alien's position to shoot it down. Enter X and Y positions between 0 and 300 and click the Fire! button. The X position is the horizontal distance in pixels, from the left side. The Y position is the vertical distance in pixels, from the top of the screen. If you miss, the alien moves closer to Earth and chooses a new random X position. If it reaches Earth within eight turns, you lose. But if you hit it, you save Earth! Each time you make a guess, your cannon moves to the X position you entered, and your missile moves to the X and Y positions that you guessed.

Does it seem like this would be complicated to make? Don't be fooled, it's nothing more than our plain-old number-guessing game, all dolled up for a night out on the town. But instead of guessing just one number, you're now guessing two: the alien's X and Y positions. And the technique used to display the game graphics is exactly the same as those we used to display and change the arrow in the number-guessing game. But this time around you're repositioning three images, not just one.

Here's the entire code listing, and I'll take you through a step-by-step tour of how it works. You'll soon see that it's much simpler than you might think at first glance.

```
<!doctype html>
<meta charset="utf-8">
<title>Alien attack</title>

<style>

#stage
{
  width: 300px;
  height: 300px;
  position: relative;
}

#background
{
  width: 300px;
  height: 300px;
  position: absolute;
  top: 0px;
  left: 0px;
  background-image: url(../images/background.png);
}

#cannon
{
  width: 20px;
  height: 20px;
  position: absolute;
  top: 270px;
  left: 140px;
  background-image: url(../images/cannon.png);
}
```

```
#alien
{
  width: 20px;
  height: 20px;
  position: absolute;
  top: 20px;
  left: 80px;
  background-image: url(../images/alien.png);
}

#missile
{
  width: 10px;
  height: 10px;
  position: absolute;
  top: 240px;
  left: 145px;
  background-image: url(../images/missile.png);
}

</style>

<div id="stage">
  <div id="background"></div>
  <div id="cannon"></div>
  <div id="missile"></div>
  <div id="alien"></div>
</div>

<p id="output">Enter the X and Y position (0-300), then click fire.</p>
<input id="inputX" type="text" placeholder="X...">
<input id="inputY" type="text" placeholder="Y...">
<button>fire!</button>

<script>

//Game variables
var alienX = 80;
var alienY = 20;
var guessX = 0;
var guessY = 0;
var shotsRemaining = 8;
var shotsMade = 0;
var gameState = "";
var gameWon = false;

//The game objects
var cannon = document.querySelector("#cannon");
var alien = document.querySelector("#alien");
var missile = document.querySelector("#missile");
```

```
//The input and output fields
var inputX = document.querySelector("#inputX");
var inputY = document.querySelector("#inputY");
var output = document.querySelector("#output");

//The button
var button = document.querySelector("button");
button.style.cursor = "pointer";
button.addEventListener("click", clickHandler, false);

function render()
{
  //Position the alien
  alien.style.left = alienX + "px";
  alien.style.top = alienY + "px";

  //Position the cannon
  cannon.style.left = guessX + "px";

  //Position the missile
  missile.style.left = guessX + "px";
  missile.style.top = guessY + "px";
}

function clickHandler()
{
  playGame();
}

function playGame()
{
  shotsRemaining = shotsRemaining - 1;
  shotsMade = shotsMade + 1;
  gameState = " Shots: " + shotsMade + ", Remaining: " + shotsRemaining;

  guessX = parseInt(inputX.value);
  guessY = parseInt(inputY.value);

//Find out whether the player's x and y guesses are inside
//The alien's area

  if(guessX >= alienX && guessX <= alienX + 20)
  {
  //Yes, it's within the X range, so now let's
  //check the Y range

    if(guessY >= alienY && guessY <= alienY + 20)
    {
      //It's in both the X and Y range, so it's a hit!
```

```
        gameWon = true;
        endGame();
      }
    }
    else
    {
      output.innerHTML = "Miss!" + gameState;

      //Check for the end of the game
      if (shotsRemaining < 1)
      {
        endGame();
      }
    }

    //Update the alien's position if the
    //game hasn't yet been won

    if(!gameWon)
    {
      //Update the alien's X position
      alienX = Math.floor(Math.random() * 280);

      //Add 30 to the new Y position so that
      //the alien moves down toward earth
      alienY += 30;
    }

    //Render the new game state
    render();
    console.log("X: " + alienX);
    console.log("Y: " + alienY);
  }

  function endGame()
  {
    if(gameWon)
    {
      output.innerHTML
        = "Hit! You saved the earth!" + "<br>"
        + "It only took you " + shotsMade + " shots.";
    }
    else
    {
      output.innerHTML
        = "You lost!" + "<br>"
        + "The earth has been invaded!";
    }
  }
  </script>
```

Setting up the game

The project folder uses the same format as in the previous example. The HTML file is in the src folder. The game uses images that are in an images folder. Figure 3-22 illustrates this.

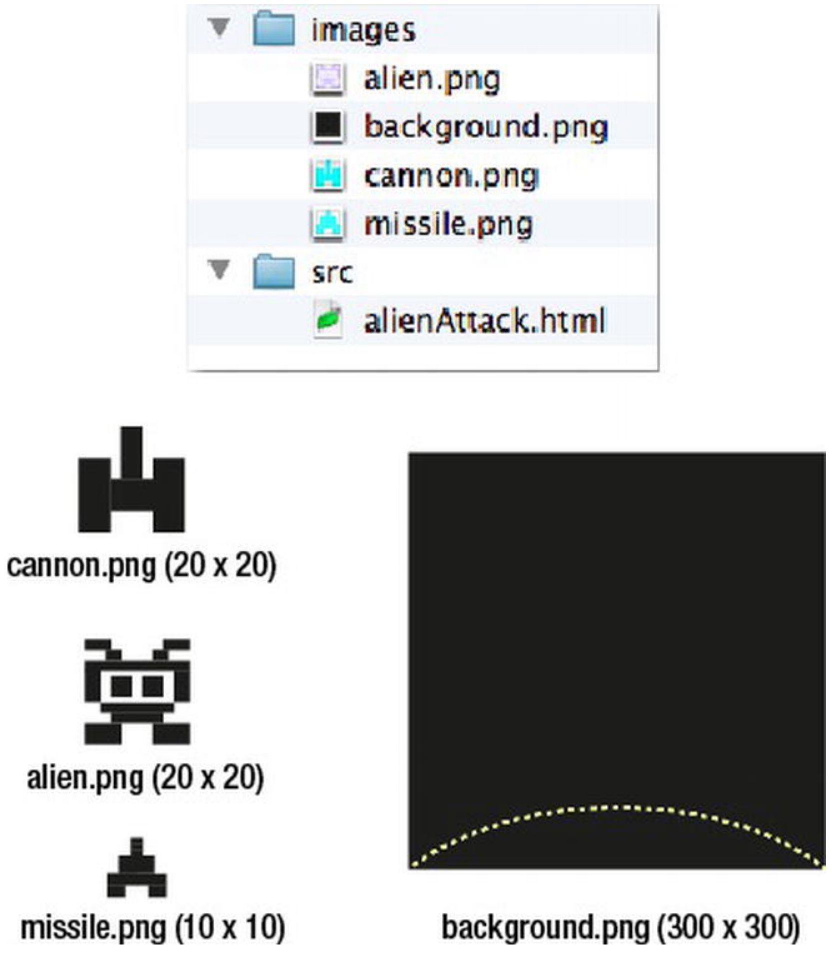


Figure 3-22. The Alien Attack! project folder

The alien and cannon are each 20 x 20 pixels. The missile is 10 x 10 pixels. The background is 300 x 300 pixels. These are all PNG images that I've designed to make them look like they are from a retro video game. But you can use any images of any sizes that you like.

You'll see that the format for this program follows exactly the format of the previous example. The HMTL code creates the visual game elements inside the stage:

```
<div id="stage">
  <div id="background"></div>
  <div id="cannon"></div>
  <div id="missile"></div>
  <div id="alien"></div>
</div>
```

The order in which you add these is important. The first image you add will be behind the images you add next. That means you have to add the background first if you want it to appear behind the other images. If the background is added last, it will cover up the other images. The images are stacked in **layers**, just as they are with graphic design software like Photoshop or Illustrator.

Note: You can use the CSS z-index property to change the stacking order of elements. Give elements z-index numbers such as 1, 2, 3, or 4. Elements with lower numbers will be stacked below elements with higher numbers. Here's an example:

```
#elementOne

{

  z-index: 2;

}

#elementTwo

{

  z-index: 1;

}
```

elementOne will appear above elementTwo because it has a higher z-index number.

The z-index properly only works with elements that have been positioned with CSS property.

The game has two input fields. `inputX` lets you enter the X position of the missile, and `inputY` lets you enter the Y position.

```
<p id="output">Enter the X and Y position (0-300), then click fire.</p>
<input id="inputX" type="text" placeholder="X...">
<input id="inputY" type="text" placeholder="Y...">
<button>fire!</button>
```

The CSS code loads the images and gives all the objects absolute positions relative to the stage's top left corner. The dimensions of the stage are as large as the largest thing it contains, which in this case is the background.

```
<style>

#stage
{
  width: 300px;
  height: 300px;
  position: relative;
}

#background
{
  width: 300px;
  height: 300px;
  position: absolute;
  top: 0px;
  left: 0px;
  background-image: url(../images/background.png);
}

#cannon
{
  width: 20px;
  height: 20px;
  position: absolute;
  top: 270px;
  left: 140px;
  background-image: url(../images/cannon.png);
}

#alien
{
  width: 20px;
  height: 20px;
  position: absolute;
  top: 20px;
  left: 80px;
  background-image: url(../images/alien.png);
}
```

```
#missile
{
  width: 10px;
  height: 10px;
  position: absolute;
  top: 240px;
  left: 145px;
  background-image: url(../images/missile.png);
}

</style>
```

These are the positions of the objects when the game first loads.

The code then initializes all the variables it needs so that it can access and modify these objects. This should all be pretty familiar to you by now.

```
//Game variables
var alienX = 80;
var alienY = 20;
var guessX = 0;
var guessY = 0;
var shotsRemaining = 8;
var shotsMade = 0;
var gameState = "";
var gameWon = false;

//The game objects
var cannon = document.querySelector("#cannon");
var alien = document.querySelector("#alien");
var missile = document.querySelector("#missile");

//The input and output fields
var inputX = document.querySelector("#inputX");
var inputY = document.querySelector("#inputY");
var output = document.querySelector("#output");

//The button
var button = document.querySelector("button");
button.style.cursor = "pointer";
button.addEventListener("click", clickHandler, false);
```

There are two variables that will store the player's X and Y position guesses:

```
var guessX = 0;
var guessY = 0;
```

The `guessesRemaining` and `guessesMade` variables from the number-guessing game have been changed to match our new space-war theme.

```
var shotsRemaining = 8;
var shotsMade = 0;
```

Their jobs are the same.

The alienX and alienY variables store the position of the alien on the stage. Together they describe its starting position. alienX is initialized to 80, which is 80 pixels left of the stage. alienY is initialized to 20, which is 20 pixels below the top of the stage.

```
var alienX = 80;
var alienY = 20;
```

These two variables have been set to the same left and top values that the alien was assigned in the CSS code. It's the CSS code that sets the alien's initial position, not these variables. However, when the game plays we're going to use these variables to change the CSS values. That's what will make the alien move. Giving these variables the same values as the CSS values is a good starting point.

Figuring out if the alien has been hit

When the player clicks the button, the playGame function is called. It first calculates the shotsRemaining and shotsMade, and it updates the gameState. It also copies the inputX and inputY values from the textfields into the guessX and guessY variables.

```
function playGame()
{
  shotsRemaining = shotsRemaining - 1;
  shotsMade = shotsMade + 1;
  gameState = " Shots: " + shotsMade + ", Remaining: " + shotsRemaining;

  guessX = parseInt(inputX.value);
  guessY = parseInt(inputY.value);
  //...
```

All of this is similar to what happened in the number-guessing game.

What's new is that now that the game has to figure out whether the player's guesses are correct. How does it do this? Here's the code in the playGame function that figures out if the alien has been hit. It uses a nested if statement to do this:

```
if(guessX >= alienX && guessX <= alienX + 20)
{
  //Yes, it's within the X range, so now let's
  //check the Y range

  if(guessY >= alienY && guessY <= alienY + 20)
  {
    //It's in both the X and Y range, so it's a hit!
    gameWon = true;
    endGame();
  }
}
```

This code is checking whether the guesses are within an X and Y range occupied by the alien. It first checks `guessX`. If `guessX` is a number that's greater or equal to the alien's X position *and* less than its width (`alienX + 20`), then the guess is correct. That means there might be a hit on the horizontal axis. But it's only a hit if `guessY` is also correct, which is what the second if statement checks. It checks whether `guessY` is a number that's greater or equal to the alien's Y position *and* less than its height (`alienY + 20`). If both of these are true, then the guesses are in a range occupied by the alien, and the player wins.

This logic is pretty condensed, but if you walk though it carefully you'll see that it really makes a lot of sense. Let's break it down into more bite-size chunks.

Imagine that the alien is at an X position of 50 and a Y position of 30. The player guesses that the X position is 55. Remember that these positions refer to the alien's *top left corner*. Figure 3-23 illustrates this.

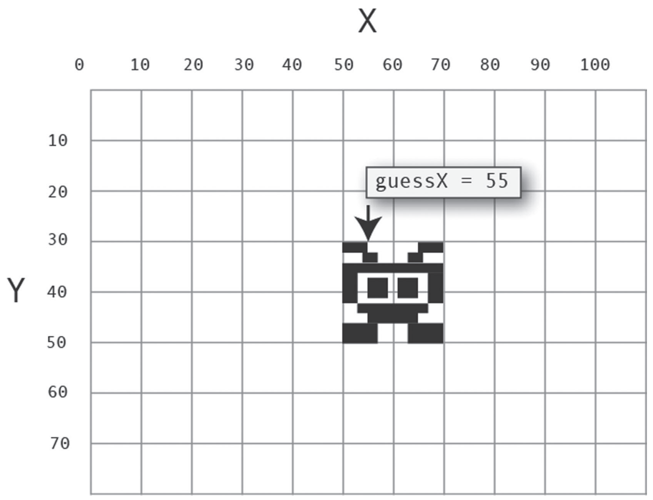


Figure 3-23. Did the player guess an X position that's within the range occupied by the alien?

We need to figure out if the player's guess, 55, is within the range occupied by the alien on the X axis. We know that the alien's top left corner is at position 50. And the alien is 20 pixels wide, which means it occupies a space between 50 and 70. Is 55 between 50 and 70? It sure is! That's what the first if statement tells us. Here's what the if statement looks like with these actual numbers:

```
if(55 >= 50 && 55 <= 70)
{
  //Yes, it's within the X range
```

So we know that the player is at least half-right. The alien might have been hit. But we'll only know for sure if we also check the player's Y position guess.

Let's imagine the player guesses 45. Is this within the range occupied by the alien? Figure 3-24 shows that it is.

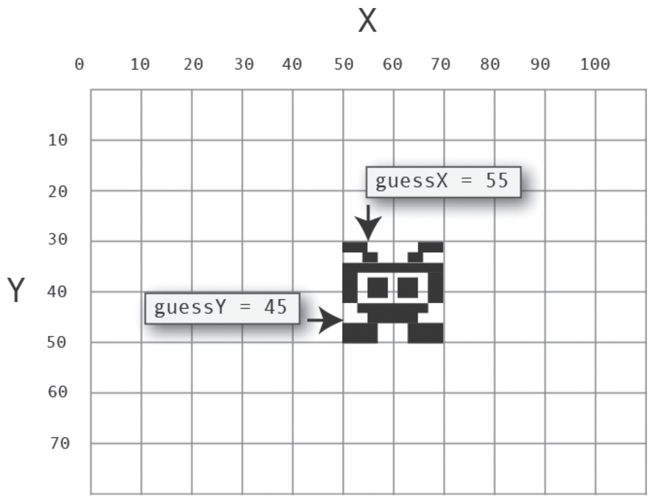


Figure 3-24. Is *guessY* withn the alien's range?

```
if(45 >= 30 && 45 <= 50)
{
  //It's within both the X and Y range, so it's a hit!
  gameWon = true;
  endGame();
}
```

You can see in Figure 3-25 that `guessX` and `guessY` are actually pointing to X and Y positions inside the alien.

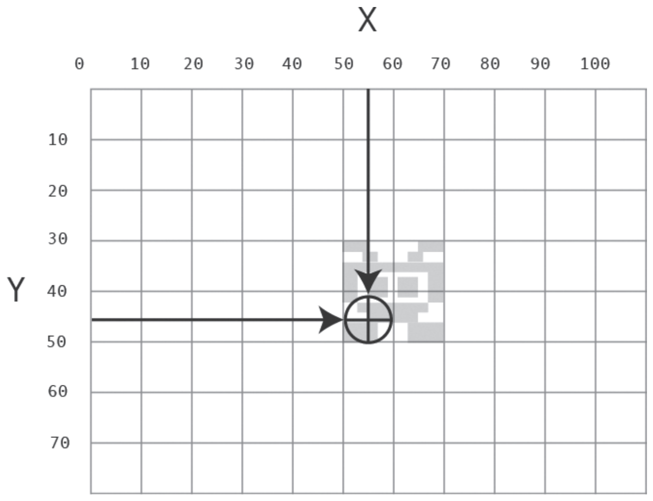


Figure 3-25. It's a hit!

So math is actually useful for shooting down aliens, and that's cool with us!

Take a bit of time to understand how it's working; it's worth spending a bit of brainpower on this until it clicks. This is actually a sneaky introduction to an important game design topic called **collision detection**. You'll learn about it in Chapter 8.

Moving the game objects

If the player misses the alien, the playGame function finds the alien's next new position. It gives it a random X position between 0 and 280 and then adds 30 to its current Y position.

```
if(!gameWon)
{
  //Update the alien's X position
  alienX = Math.floor(Math.random() * 281);

  //Add 30 to the new Y position so that
  //the alien moves down toward earth
  alienY += 30;
}
```

The alien is 20 pixels wide, so giving it a random range of between 0 and 280 means that its right side will never cross the right side of the background. This keeps it within the playing field.

Remember, this code doesn't move the alien yet. It's just the game information. It just figures out where the alien should go when it's rendered. The job of moving the image of the alien to this position is done by calling the render function:

```
render();
```

Here's the render function that moves all the game objects based on the game information:

```
function render()
{
  //Position the alien
  alien.style.left = alienX + "px";
  alien.style.top = alienY + "px";

  //Position the cannon
  cannon.style.left = guessX + "px";

  //Position the missile
  missile.style.left = guessX + "px";
  missile.style.top = guessY + "px";
}
```

The alien is moved to a new position based on the new `alienX` and `alienY` values that the game just figured out:

```
alien.style.left = alienX + "px";
alien.style.top = alienY + "px";
```

This code is changing the alien's CSS position values to actually move it to a real new position in the browser.

The cannon is moved to the `guessX` position:

```
cannon.style.left = guessX + "px";
```

And the missile is moved to the precise X and Y positions that the player guessed:

```
missile.style.left = guessX + "px";
missile.style.top = guessY + "px";
```

It's important to remember that these points represent the top left corners of the objects. For now, that will do, but often you'll want to move objects relative to their center points. That involves a little more simple math, and you'll find out how to do this in Chapter 8.

Making the game better

I've kept this game as simple as possible so that you can see how the underlying programming logic works. There's nothing stopping you from making it better by using some of the many other techniques you've learned in this chapter and adding some of your own ideas. Here are some suggestions of improvements you could make.

Adding an explosion

Create an explosion when the alien is hit. Figure 3-26 shows what this could look like.

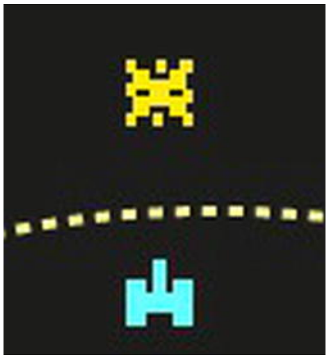


Figure 3-26. Add an explosion when the alien is hit

The explosion should appear in the same place as the alien. The alien and the missile should disappear.

You're smart enough to do this on your own, but here are some hints:

- Add a new `<div>` element with the id `explosion`.
- In the CSS code, set the explosion's `display` property to `none` so that it's not visible when the game first starts.
- Create a variable that references the explosion using `document.querySelector`.
- In the `render` function, add an if statement that checks whether the game has been won. If it has, your code should do these things:
    - Make the explosion visible by setting its CSS `display` property to `block`.
    - Give the explosion the same X and Y positions as the alien.
    - Make the alien and missile invisible by setting their display properties to `none`.

If you get stuck, take a look at the `alienExplosion.html` file in the chapter's source files for a working example.

Validating the new input numbers

Prevent the player from entering anything except numbers. Also, prevent players from entering numbers greater than 300. You'll need to do this in the `validateInput` function, which might look like this:

```
function validateInput()
{
  guessX = parseInt(inputX.value);
  guessY = parseInt(inputY.value);

  if(isNaN(guessX) || isNaN(guessY))
  {
    output.innerHTML = "Please enter a number.";
  }
  else if(guessX > 300 || guessY > 300)
  {
    output.innerHTML = "Please enter a number less than 300.";
  }
  else
  {
    playGame();
  }
}
```

Adding some more HTML and CSS styling

Use what you've learned so far to improve how the game looks and plays. Here are some obvious things:

- Create a custom Fire button.
- Apply all the polishing-up techniques you learned from the final number-guessing game: let the player use the Enter key; add focus to the first input field; disable input fields and buttons at the end of the game.
- Finally, try and use what you know about HTML and CSS to make the game look much better. You could add a heading with an embedded font, enclose the whole game in its own box, and use some gradient or drop-shadow effects.

Figure 3-27 shows what your finished game might look like, and you'll find a working example with all these modifications in the alienAttackFinished folder in the chapter's source files.

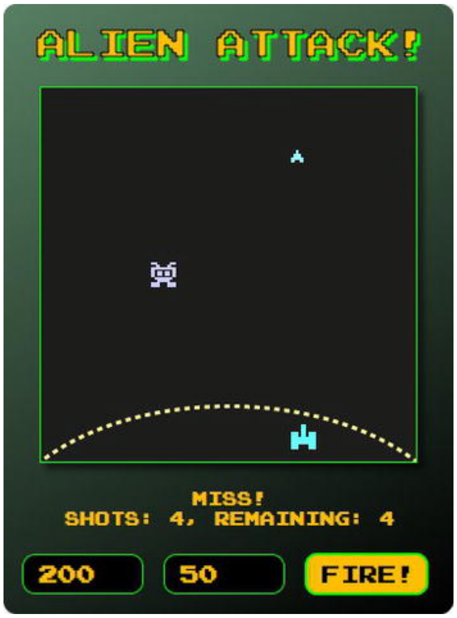


Figure 3-27. Add some finishing touches to your game

Let's take a quick look at how this finished game was put together, and I'll show you a few new HTML and CSS tricks you might want to use.

Figure 3-28 illustrates how the project files and folders have been organized.

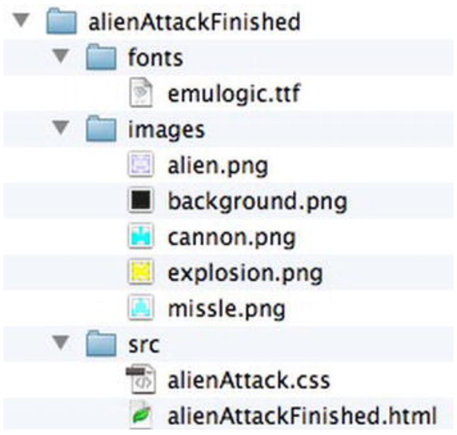

```
alienAttackFinished
  fonts
    emulogic.ttf
  images
    alien.png
    background.png
    cannon.png
    explosion.png
    missle.png
  src
    alienAttack.css
    alienAttackFinished.html
```

Figure 3-28. The organization of the final project folder

There's a lot of CSS code in this project, so I've put it in its own separate CSS file and linked it to the HTML with the `<link>` element.

```
<link rel="stylesheet" href="alienAttack.css">
```

Look back to Chapter 1 if you need a review how to link external CSS files.

The game also uses an embedded font called `emulogic.ttf` for that old-school video-game vibe. It's in a fonts folder inside the project folder. Here's the CSS code that loads and embeds it.

```
@font-face
{
  font-family: emulogic;
  src: url("../fonts/emulogic.ttf");
}
```

> *Note: If you are using Firefox, embedded fonts can only be loaded if they are in the same folder as the CSS file.*

I want all the elements to use this font, including the button and input fields, so I've used the **universal selector**, an asterisk, to force all the elements to use it.

```
*
{
  font-family: emulogic;
  padding: 0px;
  margin: 0px;
}
```

Any properties you set with the universal selector will apply to *all* your CSS elements.

I've also used the universal selector to give all the elements a padding and margin of zero. This overrides the automatic padding and margins that the web browser gives some elements. Building up your paddings and margins up from zero will usually be the easiest and clearest way for you to start laying out elements.

You can see in Figure 3-27 that the whole stage and the user interface are surrounded by a box. I did this by creating a special `<section id="game">` element that surrounds all the other HTML code.

Do you see where the `<section>` begins and ends? That's the box that encloses the game.

```
<!doctype html>
<meta charset="utf-8">
<title>Alien explosion</title>
<link rel="stylesheet" href="alienAttack.css">

<section id="game">

<h1>Alien Attack!</h1>

<div id="stage">
  <div id="background"></div>
  <div id="cannon"></div>
  <div id="missile"></div>
  <div id="alien"></div>
  <div id="explosion"></div>
</div>

<p id="output">Enter the X and Y position (0-300), then click fire.</p>
<input id="inputX" type="text" placeholder="X..." autofocus>
<input id="inputY" type="text" placeholder="Y...">
<button>fire!</button>

</section>
```

You can use a `<section>` element to enclose any group of HTML code that you want to keep together for some reason. The `<section id="game">` element has rounded corners, a background gradient, a drop shadow, and some padding, and it is large enough to comfortably contain all the other elements.

```
#game
{
  margin: 0px auto;
  width: 330px;
  height: auto;
  padding: 15px;
  border: black;
  background: linear-gradient(top, #588063, #000);
  box-shadow: 5px 5px 5px rgba(0, 0, 0, 0.5);
  border-radius: 10px;
}
```

You'll also notice that it is centered in the browser window, which, as you learned in Chapter 1, is thanks to the `margin` property:

```
margin: 0px auto;
```

This gives the element equal, automatic left and right margins, which centers it in the browser. The stage element, which contains the game images, is also centered inside the game element with the same line of code:

```
#stage
{
  margin: 0px auto;
  ...
}
```

This works for block elements but not for text. Remember that if you want to center text inside another element, use the CSS `text-align` property and give it the value `center`:

```
text-align: center;
```

Using a text outline and shadows

The game title, Alien Attack!, has a black outline and a lime green drop shadow. To outline text, use the CSS `text-stroke` property. It needs two values: the thickness of the line in pixels, and the color:

```
-webkit-text-stroke: 1px #000;
-moz-text-stroke: 1px #000;
text-stroke: 1px #000;
```

This outlines the text with a 1-pixel-wide black line.

Note: At the time of writing, the text-stroke property only works on Webkit-based browsers like Chrome and Safari.

Use the `text-shadow` property to create a drop shadow for the text. It needs three values: the x offset, the y offset, and the color. This is the code that creates a green drop shadow in the example:

```
text-shadow: 3px 3px lime;
```

The green shadow is offset by 3 pixels down and to the right. The color can be a preset color, like lime, or any hex, RGB/RGBA, or HSL/HSLA color.

You can also add an additional optional value, which is the blur amount. And you can also use an RGBA color value for the shadow, which will make it semitransparent. This bit of code creates a semitransparent shadow, offset by 5 pixels, with a 3 pixel blur:

```
text-shadow: 5px 5px 3px rgba(0, 0, 0, 0.5);
```

This is a nice effect because the semitransparency allows any images under it to be visible.