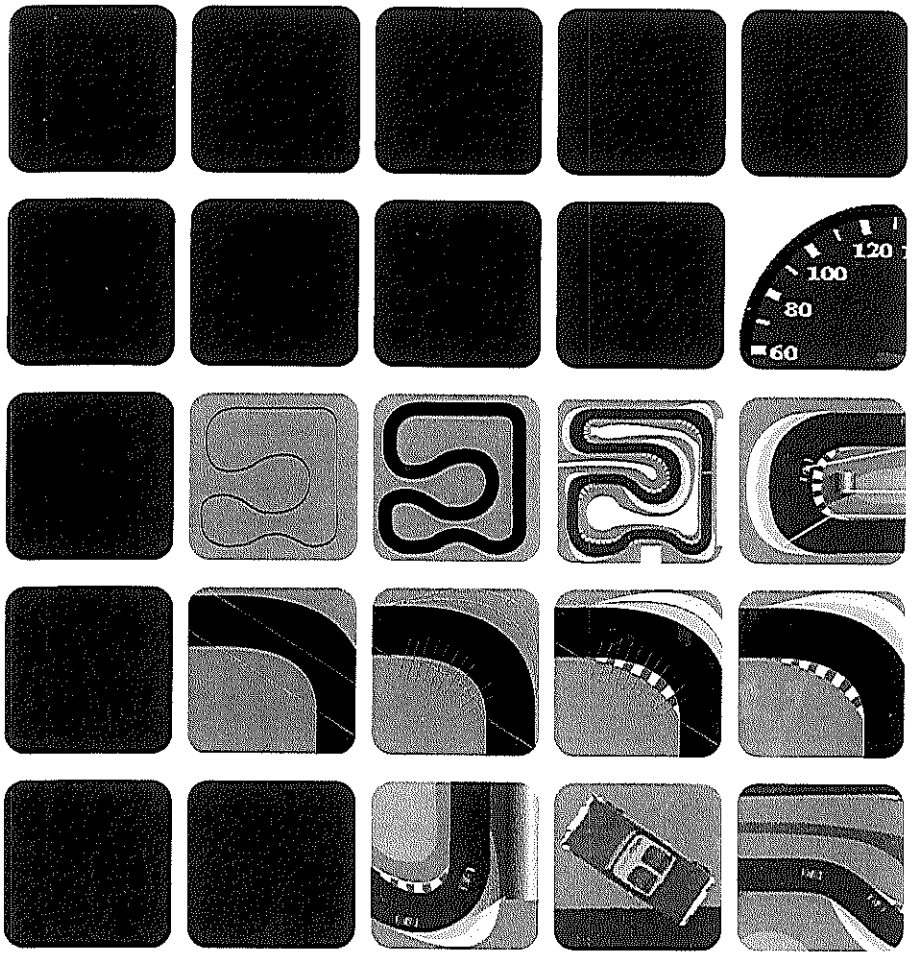
Anthony Eden

From an early age, Anthony developed a love of interaction with mathematics and computational languages, along the way gaining an appreciation of any given natural environment and the ability to transform his environment into a digital construct. Inspiration for his latest project, `www.arseiam.com` (essentially an ActionScript anthology of his Flash work), is testament to this philosophy. The last decade has included commercial roles with Microsoft, Disney, Toyota, and Adobe, providing a sound framework in which to explore and diversify his project development life cycle skills. Spare time? If he's not thinking about it, he's doing it!



RACING CARS

Throughout the course of this chapter, you'll learn the fundamentals of designing and developing a racing game engine. You'll also discover how to implement simple, yet innovating scrolling and scaling techniques to achieve an arcade-style look and feel. Although the examples in this chapter are based around the development of a car racing game, among other things we cover some AI path-following techniques that you can apply to a variety of games styles.

The racing game that we develop is a top-view game in which the car is positioned in the middle of the play area and the track moves and scales relative to the car's angle and velocity. We cover user control creation, collision detection, and computer-controlled opponents.

Open the file `race_final.swf` and familiarize yourself with the game play, the car physics, and the different types of objects used to build the track.

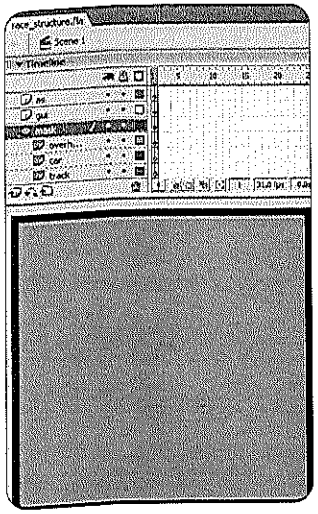
Getting Started

Before you start building the game you should establish the structure and details of the Flash file. If you're feeling lazy, then you can just open `race_structure.fla` and skip to the following instructions:

1. Create a new Flash document with the dimensions 550x400, a background of dark gray (#333333 should suffice), and a frame rate of 21.

> *Science has proven that humans need a frame rate between 21 and 23 to convince the eye that the motion is fluid and not made up of frames. Thus, a frame rate of 21 is the lowest frame rate required to give the feel of fluid, not frame-based, motion.*

2. Create five new layers (you need six layers in total) and name them as, gui, mask, overhead, car, and track from top to bottom.



3. On layer mask, create a square (any fill color, but without a line) 350x350 pixels and place it at the coordinates 10,10. Convert layer mask into a mask and make all layers beneath it masked.

When you first create the mask, only the layer immediately beneath the mask turns into a masked layer. To turn the other layers into masked layers, double-click the layer icon to open the Layer Properties dialog box, and then choose Masked from the options. Alternatively, you can drag each layer to the bottom of the stack of masked layers. A small rectangle appearing below the icon of the previous masked layer should indicate that dropping the dragged layer will also make it masked.

Once you've completed these steps, your final layer structure should look like this:

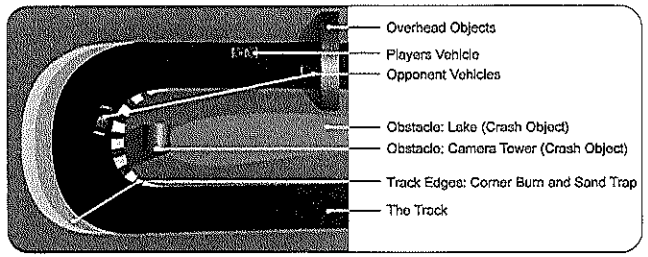
Now that we've covered the file structure, let's take a quick look at the steps required to build the game. We'll take a more detailed look at each of these steps in the subsequent sections of this chapter.

- **Building the racetrack:** Here we'll look at ways to create and detail the track and its elements.
- **Creating the race car:** Once the track is built, you'll be ready to put the wheels into motion by creating a race car and adding some user controls.
- **Moving the track:** As the car is stationary, you need to use the user controls to move the track beneath the car. If you do so, the user is able to move around the track.
- **Handling the vehicle when it leaves the track:** Well, there's no point having a racetrack on which you can drive off the road and into buildings unscathed. This section explains how to make sure the right things happen when the driver does the wrong things.
- **Scaling the racetrack:** To achieve an arcade look and feel, we establish an easy-to-learn method for scaling the track while keeping the center of scale beneath the vehicle.
- **Moving the computer-controlled cars:** AI made easy. This section covers a simple, yet effective path-following technique.
- **Creating the graphical user interface (GUI):** Here we look at adding detail to the user experience by creating a user interface that gives real-time feedback on laps and velocity.
- **Adding audio:** Here we look at adding sound to your game. Music tracks, event effects, and ambient sound can add great depth to the game play.
- **Adding customization and enhancements:** Once you have the game up and running, we discuss some of the endless possibilities for increasing the game functionality and user experience.

Building the Racetrack

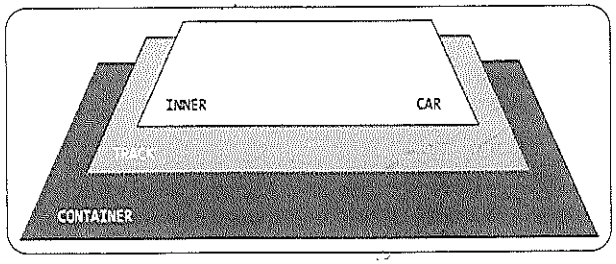
The track is essentially broken down into four main sections:

1. **The track:** The area you drive on (or are supposed to).
2. **The track edges:** The corner burns and grass/dirt areas that slow the car down if you fail to keep it on the track.
3. **Obstacles:** Buildings, walls, spectator stands, and trees will all cause you to crash the car if you're unfortunate enough to drive into them.
4. **Overhead objects:** The tops of bridges and shadows cast by obstacles in general have no effect on your vehicle and only serve an aesthetic purpose.



Initially you'll focus on creating the container movie clip that contains (nested within) the track elements required for collision detection and opponent vehicles. The container movie clip is structured to optimize the way in which you manage collision detection and scale the track. Here's an outline of the movie clip structure that we describe throughout this section:

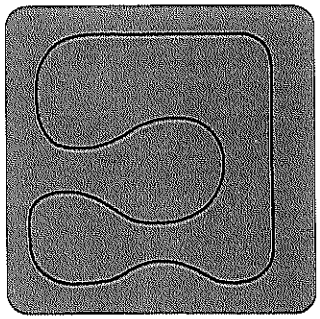
- **Container:** This movie clip holds all track layers.
- **Crash zone:** This movie clip contains the elements that cause the player's car to crash.
- **Track:** This movie clip contains the track data, opponent vehicles, and the movie clip track inner.
- **Track/track inner:** This movie clip is used by the opponent vehicles to ensure they stay on the track.



The Track

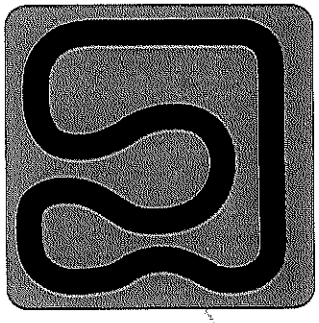
1. Open the FLA file you created earlier (or refer to the downloaded file `race_structure.fla`). On layer track create a rectangle 1000 pixels wide and 1000 pixels high anywhere on the stage. Convert it to a movie clip named `container`—this will be the base of your track, so there's no need for an outline. It should be an earthy solid green.

2. Edit `container` and, using the Pencil tool in Smooth pencil mode, create a dark gray line color with a thickness of 10. Draw the shape of the racetrack on top of the green base, so you have a track that looks something like this image:

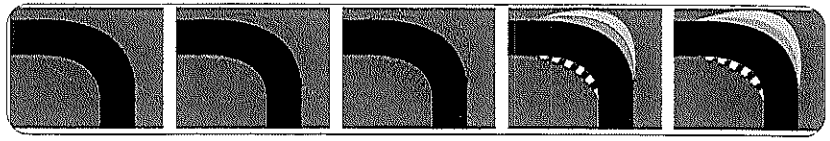


3. Next, using Modify ➤ Shape ➤ Smooth and Modify ➤ Shape ➤ Optimize, reduce the line to the minimum number of points possible while retaining the basic shape of your track.

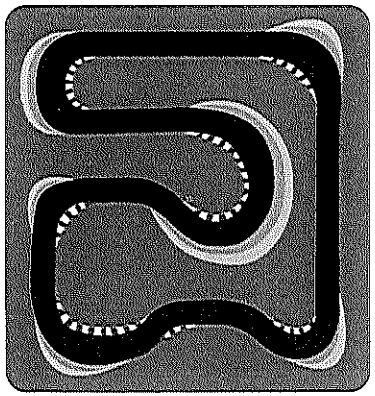
4. Once you're happy with the shape of your track, convert the lines to fills (Modify ➤ Shape ➤ Convert Lines to Fills) and expand (Modify ➤ Shape ➤ Expand Fill) to a distance of 75 pixels. You've now built your basic track:

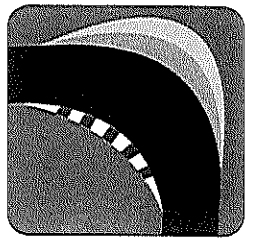


5. Fill out the track by adding burns and sand traps at the corners. It's easiest to draw a line across the track from the start of the corner to the end of the corner and then adjust the curve of the line to create the outline of traps and burns. You can then add detail for a more realistic effect. We recommend that you widen the track on corners to make cornering and overtaking easier:

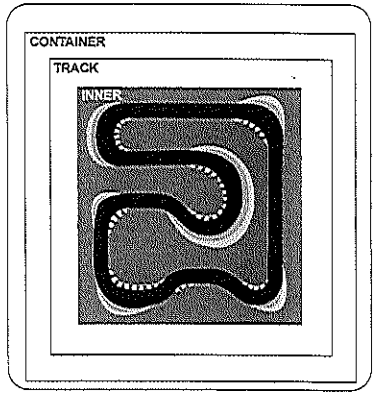


6. This process can be a little tedious and time consuming, but the more effort you put into creating these elements, the better the track will look. Continue to process all corners in this way until the entire track is complete:





7. Now select the area of the track that the cars drive on and delete it. You're deleting the track because you're going to use the shape of everything else for your collision detection. Finally, select all your shapes and convert them into an movie clip named `track inner`, and then convert that movie clip into another and name it `track` so that you now have three levels of nested movie clips: `container/track/inner`.



8. Make sure to name the instances of each of these movie clips `container`, `track`, and `inner`, respectively, and place the container movie clip on the track layer.

Open the file `race1.fla` to load the track used in this chapter.

Building the Crash Zone

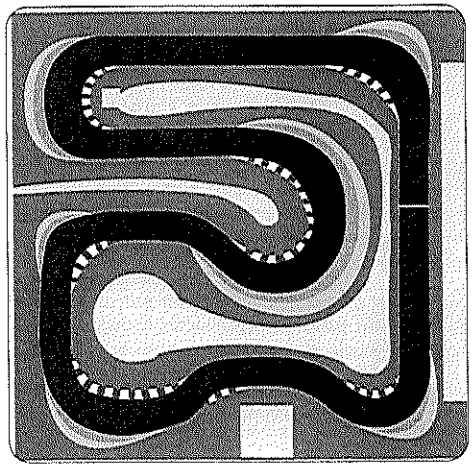
A crash zone is needed for when the user drives the car off the track and collides with something that would normally make the vehicle crash. Stadiums, trees, lakes, and so on are all types of objects that could cause a nasty crash. The crash zone is also handy in terms of preventing players from cheating and taking shortcuts through nontrack areas. Open the file `race_2.fla` to view a sample track with a crash zone included.

1. To begin making your crash cone layer, edit the container movie clip. If you haven't already done so, rename the layer containing the track movie clip to Race Track and create a new layer above called Crash Zone.

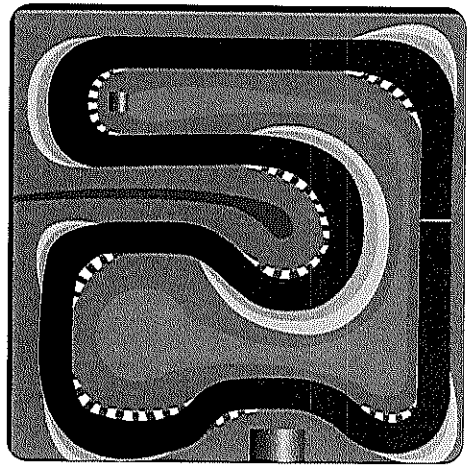
2. Make a copy of the instance of track and place it at the exactly the same coordinates (Edit ➤ Paste in Place) on layer Crash Zone. You should now have two instances of track on two separate layers with the top instance directly over the top of the bottom instance. If you're starting to get a little lost, just refer to `race_2.fla`.

3. You're now going to use this new instance of track as your guide for creating a crash zone. To start, name the new instance of the track movie clip crash, and while it's still selected choose Swap from the Property inspector. You should have track selected. Now make a duplicate of track (click the Duplicate Symbol button in the Swap Symbol dialog box), call it crash zone, and select it to replace the current selected instance. Now edit crash zone in place, and create a new layer above any layers you may have in there. On this layer you can start drawing the areas that you want to have as your crash zone.

   Start by roughly tracing out the area you want to use (in this case, the areas are indicated in yellow):

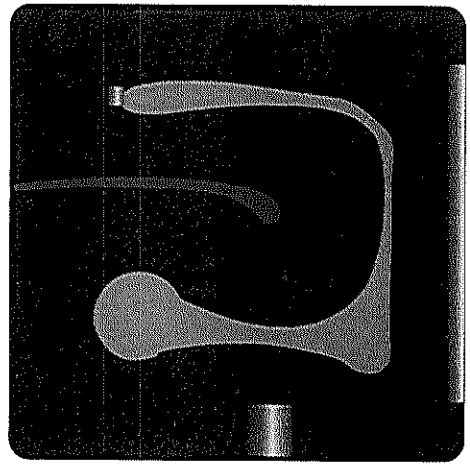


4. Now gradually color and detail the area to look like water, trees, buildings, or whatever you want:

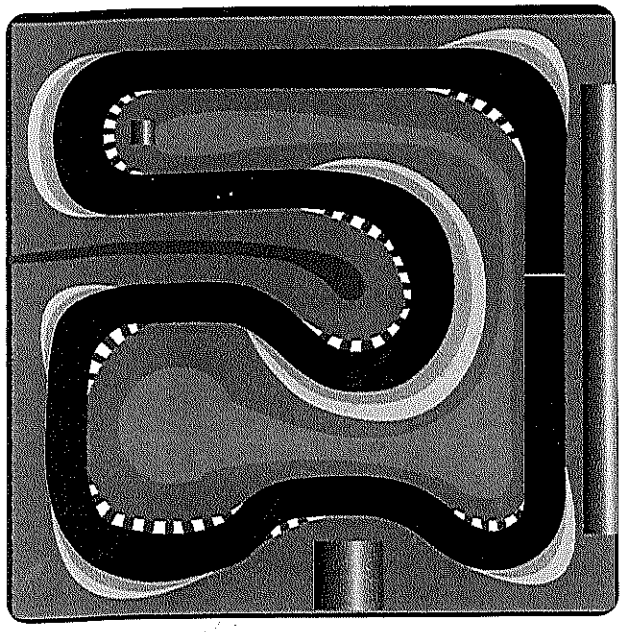


Notice the thin black outline around the bounding box of the crash zone in the preceding image? This prevents players from leaving the track completely. In a commercial release of this game, I left a gap in the outside of the track to allow the player to drive off and explore the game production credits—these kind of Easter egg features can be real bonuses!

5. Once you're happy with the design and rendering of the crash zone, you can delete any layers that belong to the track, leaving just your crash zone details:



6. Check how the crash zone looks with the track by viewing the container movie clip, which should look exactly the same as it did two steps ago:



Adding Track Overheads

You're probably itching to get down and dirty with some ActionScript by now. Well, if you really want to you can skip this section and come back to it later (alternatively, you can just take a glance at the file race_3.fla), as it only affects the game's aesthetics and doesn't involve any actual functionality. However, paying attention here will make your game look even cooler.

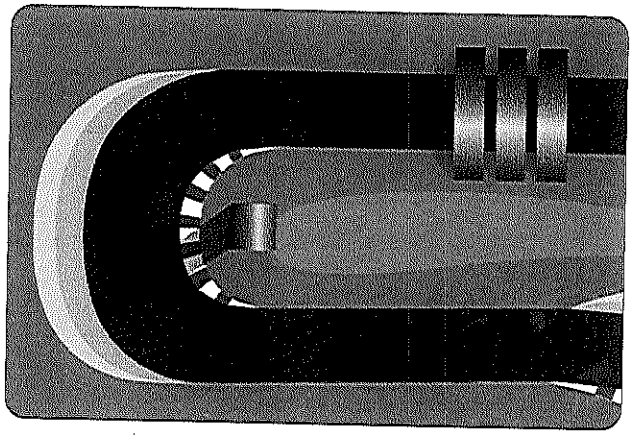
The overhead layer allows you to add shadows, bridges, and any other object that the cars can drive under safely. It works in a very similar way to the track layer, except there's no hit detection required (you'll learn about this shortly).

1. To begin with, go to your root timeline, copy the container movie clip, and paste (Edit ➤ Paste in Place) another copy of it into your overhead layer. In a similar process to the construction of the crash zone, you're going to use previously created movie clips as a guide to creating your overhead layer. This also needs to be nested inside a container movie clip in the same way as the track movie clip.

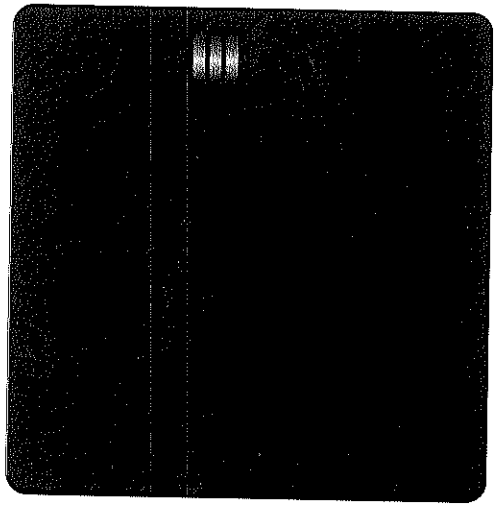
2. Now that you have a new instance of container, go to the Property inspector and select Swap. Make a duplicate of container, name it container_over, and select it for use. Name this instance container_over.

3. Edit in place container_over, select all (*Ctrl/Cmd+A*), and convert both selected layers into a movie clip called overhead_inner. Name its instance inner. You've just made a copy of container and prepared it for use in the overhead layer by adding overhead_inner as a nested movie clip that contains all the track/crash details.

4. Now edit overhead_inner and create a new layer above the layer containing your track and crash movie clips. On this layer, create shadows falling from the buildings and add a little bridge to one of the straights on your track.



5. The preceding image illustrates the addition of a bridge with shadows as well as a shadow for the camera tower on the hairpin turn that we've added. Once you're happy with the details, delete the track and crash movie clips, which leaves you with only the overhead details, like this (admittedly, this isn't such an exciting image!):



Creating the Race Car

Create a car using your preferred techniques—either draw something in Flash or import a bitmap and manually trace the car details (or use the movie clip in the file race_4.fla). For the best results, make sure your car is made completely of optimized shapes (using bitmaps causes undesirable effects when they're scaled and is more processor intensive). Make sure that the car is facing downward, convert it into a movie clip (named car), and also name the instance car. Place car on layer car and align it to the center of the mask.

> *Important note: When you design the car, keep in mind that the car must point downward and have a width of approximately 15 pixels and a height of approximately 35 pixels.*

User Controls

Now that you've set up your track, its various components, and the car, you can start to define the way in which the different objects react to user input and Flash player events.

1. To start, you need to set some basic variables that will be used to control the velocity and rotation of the vehicle. You're going to consolidate all of the core code into one central point to make changes easier. Place the following script on the first frame of the layer as:

```
// this is optional but the game will be more
// responsive on lower speed processors.
_quality = "low";
// the maximum velocity of the vehicle (pixels per frame)
max_vel = 10;
// the maximum velocity of opponent vehicles (pixels per frame)
opponent_max_vel = 10;
// the minimum velocity of the vehicle (pixels per frame)
min_vel = 0;
// the acceleration of the vehicle (pixels per frame per frame)
acceleration = 0.4;
// the acceleration of opponent vehicles
// (pixels per frame per frame)
opponent_acceleration = 0.4;
// the deceleration of the vehicle when not accelerating
// (pixels per frame per frame)
deceleration = 0.98;
// the deceleration of the vehicle when braking
// (pixels per frame)
brake = -1;
// the acceleration of the vehicle when in reverse
// (pixels per frame)
reverse = -0.1;
// the maximum speed of the vehicle when in reverse
// (pixels per frame)
max_rev_vel = -3;
// the amount the car turns (_rotation per frame)
turn = 4;
// the amount the opponent car turns (_rotation per frame)
opponent_turn = 10;
// the rate at which the car slows down if it leaves the track
slow = 0.85;
// the rate at which the opponent car slows down
// when it detects the track
opponent_slow = 0.85;
// the starting lap number
lap = 1;
// the number of laps per race
totallaps = 5;
// rate at which car is moving
vel = 0;
// a function that starts the race
startRace();
```

2. Let's tidy all that up a little bit by adding a stop() function and making an init (short for "initialize") function like so:

```
stop();
function init() {
  _quality = "low";
  max_vel = 10;
  opponent_max_vel = 10;
  min_vel = 0;
  acceleration = 0.4;
  opponent_acceleration = 0.4;
  deceleration = 0.98;
  brake = -1;
  reverse = -0.1;
  max_rev_vel = -3;
  turn = 4;
  opponent_turn = 10;
  slow = 0.85;
  opponent_slow = 0.85;
  lap = 1;
  totallaps = 5;
  vel = 0;
  startRace();
}
init();
```

3. Now that you have an init function, you should create a function for the race start and race end. Most of your action happens during the race and needs to be updated every frame. To do this, you need to define what the movie is going to do at the onEnterFrame event handler, and in this case, you're going to define the onEnterFrame event handler for the car movie clip. After your init function, write this:

```
function startRace() {
  car.onEnterFrame = function(){
    // code goes here
  };
}
```

4. And, for the case of ending the race, you need to remove that onEnterFrame event handler:

```
function endRace() {
  delete car.onEnterFrame;
}
```

5. For the moment you don't need to do anything further with the endRace function, so let's focus on startRace. First, you need to capture some keyboard interaction by adding code to the event handlers for the spacebar and cursor keys:

```
function startRace() {
  car.onEnterFrame = function(){
    if (Key.isDown(39)){
    } else if (Key.isDown(37)){
    }
    if (Key.isDown(38)){
    } else if (Key.isDown(40)){
    }
    if (Key.isDown(32)){
    }
  };
}
```

Note the use of ASCII characters for the right cursor (39), left cursor (37), up cursor (38), down cursor (40), and spacebar (32). You could create a new object and use Key.addListener, though this method is somewhat simpler and in most cases at least one of the keys will be used during every frame.

6. Now you'll give those keypresses some functionality. The right and left cursor keys are used to rotate the car right and left, respectively. To do this you simply increase the car's _rotation by the amount defined by the variable turn:

```
function startRace() {
  car.onEnterFrame = function(){
    if (Key.isDown(39)){
      this._rotation += turn;
    }
    if (Key.isDown(37)){
      this._rotation -= turn;
    } else if (Key.isDown(38)){
    }
    if (Key.isDown(40)){
    } else if (Key.isDown(32)){
    }
    if (Key.isDown(32)){
    }
  };
}
```

7. The up cursor key is used to increase the velocity of the car (this is bit of a misnomer as the car is stationary and turns on the spot—velocity is actually used to define the speed that the track moves underneath the car and not the movement of the car itself):

```
function startRace() {
  car.onEnterFrame = function(){
    if (Key.isDown(39)){
      this._rotation += turn;
    } else if (Key.isDown(37)) {
      this._rotation -= turn;
    }
    if (Key.isDown(38)){
      if (vel < max_vel){
        vel += acceleration;
      }
    } else if (Key.isDown(40)){
    } else {
      vel*=deceleration;
    }
    if (Key.isDown(32)){
    }
  };
}
```

In this case, the car accelerates whenever the up arrow key is pressed, and the car decelerates when the key isn't pressed. Note that you only allow the car to continue accelerating so long as its velocity is less than the maximum velocity you allowed earlier.

8. The reverse functions in a similar manner, but you're making sure that the car's velocity only changes if it's greater than the maximum reverse speed:

```
function startRace() {
  car.onEnterFrame = function(){
    if (Key.isDown(39)){
      this._rotation += turn;
    } else if (Key.isDown(37)){
      this._rotation -= turn;
    }
    if (Key.isDown(38)){
      if (vel < max_vel){
        vel += acceleration;
      }
    } else if (Key.isDown(40)){
      if (vel > max_rev_vel){
        vel += reverse;
      }
    } else {
      vel*=deceleration;
    }
    if (Key.isDown(32)){
    }
  };
}
```

9. The brake follows the same process as the acceleration and reverse, except you ensure that on the last iteration of slowing down if the velocity is less than the minimum speed you force it to be equal to the minimum velocity. Generally, the minimum velocity is 0, and if you allow even the slightest negative velocity, the car will continue to roll backward.

```
function startRace() {
  car.onEnterFrame = function(){
    if (Key.isDown(39)){
      this._rotation += turn;
    } else if (Key.isDown(37)){
      this._rotation -= turn;
    }
    if (Key.isDown(38)){
      if (vel < max_vel){
        vel += acceleration;
      }
    } else if (Key.isDown(40)){
      if (vel > max_rev_vel){
        vel += reverse;
      }
    } else {
      vel*=deceleration;
    }
    if (Key.isDown(32)){
      if (vel>min_vel){
        vel += brake;
      } else {
        vel = 0;
      }
    }
  };
}
```

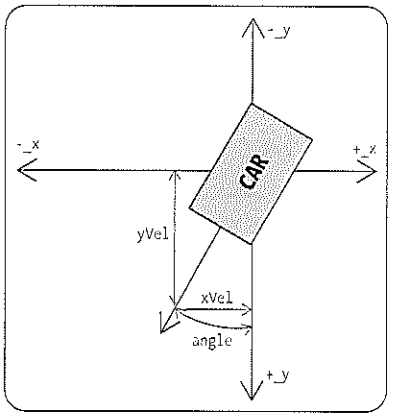
Moving the Track

Let's put things into motion now:

1. First of all, you need to make sure that everything is aligned properly. Ensure that the center of container, container_over, and the car is aligned to the center of the mask. It's essential that all these elements be aligned correctly. Refer to the file race5a.fla if you're unsure.

2. Having established the car's velocity and rotation, you can use a little trigonometry to control the way in which the track actually moves. Fortunately, the trigonometry required in this race engine is fairly simple, so the following crash course on the basics should cover things nicely.

   As the car rotates, you need to calculate the _x and _y components of its trajectory. The following illustration shows the angles involved when the car is rotated relative to the stage:



If you aren't familiar with the basics of trigonometry, then I suggest you check out some of the excellent online tutorials. In any case, I like to use the old trigonometry rule I learned in the early days of school: **SOH CAH TOA** (sounds like a famous extinct volcano!).

SOH: Sine (angle) = opposite / hypotenuse

CAH: Cosine (angle) = adjacent / hypotenuse

TOA: Tan (angle) = opposite / adjacent

3. In this case, in order to calculate the x component of velocity (xVel), you have the angle and hypotenuse (vel), so you need to use this:

   Sin(angle) = xVel/vel

   or, more usefully, this:

   *XVel = vel * Sin(angle)*

4. Before you get ahead of yourself, you need to convert the angle (this._rotation, where this refers to the car movie clip) from degrees to radians (because Flash likes it that way):

   *xVel = vel * Math.sin(this._rotation * (Math.PI/180))*

   As xVel is the _x component that you need to move the track (which is located inside of container), you can remove the use of xVel and use the track position directly. Put this line into the car's onEnterFrame handler in the startRace function, right after the key code you entered in the last step:

   ```
   container.track._x += vel * Math.sin(this._rotation * (Math.PI/180))
   ```

5. Finally, you use the same process to calculate the _y component (only this time, you use Cosine(angle) = Opposite / Hypotenuse), and you simplify the code a little by setting a variable (toRadians) equal to your degrees-to-radians conversion (Math.Pi/180), leaving you with the following:

   ```
   toRadians = Math.PI/180;
   container.track._x += vel*Math.sin(this._rotation * toRadians);
   container.track._y -= vel*Math.cos(this._rotation * toRadians);
   ```

6. Don't forget that you're using multiple layers of movie clips: track, crash zone, and container over. All three need to be moved in the same way as the track; therefore, you add a little more code to ensure this happens, referencing the clips by their instance names:

   ```
   toRadians = Math.PI/180;
   container.track._x += vel*Math.sin(this._rotation * toRadians);
   container.track._y -= vel*Math.cos(this._rotation * toRadians);
   container.crash._x = container_over.inner._x = container.track._x;
   container.crash._y = container_over.inner._y = container.track._y;
   ```

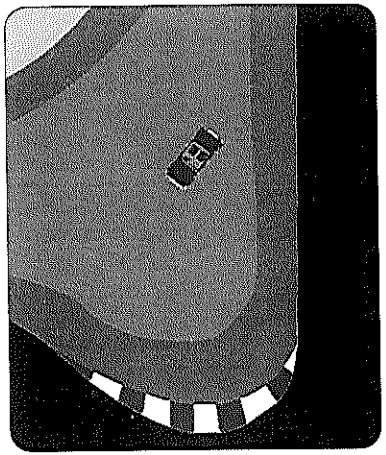
Because you need to move these layers every frame while the game is playing, you place this code within the onEnterFrame event handler inside the startRace function. Your code should now look like this:

```
function startRace() {
  car.onEnterFrame = function() {
    // user controls code described in the previous section...

    // move
    toRadians = Math.PI/180;
    container.track._x += vel*Math.sin(this._rotation*toRadians);
    container.track._y -= vel*Math.cos(this._rotation*toRadians);
    container.crash._x = container_over.inner._x=container.track._x;
    container.crash._y = container_over.inner._y=container.track._y;
  };
}
```

You can now test the movie. You should be able to drive the car around and have the track move relatively beneath it. If you're having trouble getting this far, refer to the file race_5b.fla. (You might consider moving the toRadians variable up into your init function. It really needs to be calculated only once, not every frame, but we've included it here to make this section more understandable.)

Handling the Vehicle When It Leaves the Track

Up until now, you've been able to drive the vehicle in any direction and over anything—you could take a quick spin over the lake, for instance.



Because you want the car to slow down whenever it leaves the track, you're going to use a simple hitTest to determine if the center of the car is over the track. But first let us remind you of the variable slow that defines how much the car will slow down:

```
slow = 0.85;
```

slow is the multiplication factor that the track has on the car's velocity. Set it to 0 and the car will come to a dead stop; set it to 1 and the track has no effect. Keep in mind that if it's set too low, then the car may not be able to get back onto the track.

1. Let's see what the hitTest looks like within the startRace function:

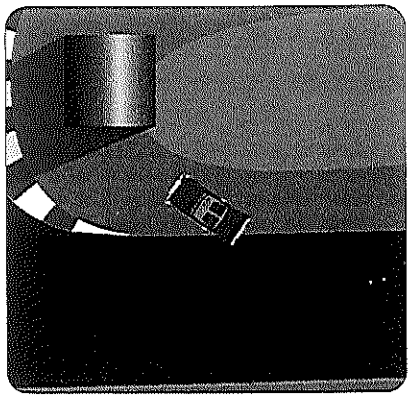
```
function startRace() {
  car.onEnterFrame = function(){
    // user controls code
    // track move code
    // is the car off the track?
    if (container.hitTest(this._x, this._y, true)) {
      vel *= slow;
    }
  };
}
```

Here you're detecting if the center of the car (_x and _y) is over any of the elements inside of the track movie clip. Remember earlier when you removed the actual track? Had you not removed it, the car would always be on top of the track and would never be able to accelerate properly. Setting the shapeflag argument to true in the hitTest call, you can test whether the car is over any actual movie clip inside of container (as opposed to using container's bounding box).

2. Similarly, you can use your crash clip to see if the car has crashed into anything (and ended the race). The crash movie clip hitTest is used in the same way (see also race_6.fla):

```
function startRace() {
  car.onEnterFrame = function(){
    // user controls code
    // track move code
    // is car over the track?
    if (container.hitTest(this._x, this._y, true)) {
      vel *= slow;
    }
    // is car over the crash area?
    if (container.crash.hitTest(this._x, this._y, true)) {
      endRace();
      attachMovie("game_over", "game_over", 1);
      game_over.onRelease = function() {
        this.removeMovieClip();
        gotoAndStop(1);
      };
      game_over._x = 180;
      game_over._y = 200;
    }
  };
}
```

Here you're testing to see if the car is over the crash movie clip and, if so, you execute the function endRace, which removes the onEnterFrame event handler used to control the car and track motion. You also add an onRelease event handler to the movie clip so that if the user wants to play again, she can simply click the game over movie clip. There are a number of ways you can end the game—in this case you're attaching a movie clip to the main timeline that states the game is over.



For the game over movie clip, we simply created a rectangle (200x100) with the words Game Over – click to start again in it and converted it into a movie clip named game over. Next, we set its linkage properties to Export for ActionScript, Export in first frame, and gave it a linkage identifier of game_over. Later in the chapter you'll look at ways of improving the end-of-game experience through the implementation of high scores and play-again functionality.

You now have the basic elements that allow the user to drive a car around a track and to have the off-track areas slow the vehicle or cause it to crash.

Scaling the Racetrack

As the car moves throughout the course, the track and the car itself scale according to the velocity of the car. For the track to scale properly, the center of container needs to be aligned to the center of the mask (where the car should currently be). This ensures that the center of scale is always in the center of the game screen and underneath the player's car.

> *To place your car at its starting point on the track, you'll have to realign the* track *movie clip rather than the* container *clip. You can do this by editing* container *and making sure that the point at which you want your car to start is also aligned to the center of* container.

1. To get the track scaling, you need to add the following code to the onEnterFrame defined in the first frame of layer as:

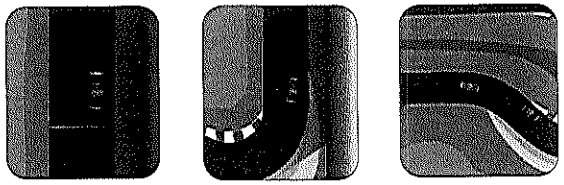
```
function startRace() {
  car.onEnterFrame = function(){
    // user controls code
    // track move code
    // is car over the track?
    // is car over the crash area?

    // scale track code
    scale_factor = 200-vel*10;
    container._yscale = container._xscale = scale_factor;
    container_over._yscale = container_over._xscale = scale_factor;
    this._yscale = this._xscale = scale_factor;
  };
}
```

The first new line of ActionScript here defines the way in which the track and car scale relative to the velocity of the car, and it's better represented by this:

```
scale_factor = Maximum Scale - Car Velocity * Scale Factor;
```

In the case of your script, you have a maximum scale of 200% and a scale factor of 10. When the car is stationary (vel = 0), the track scales at 200%. When the car is at maximum velocity (vel = 10 as defined by the variable max_vel), the track scales at 200 - 10*10 or 100%. You can change these values to suit the amount of track scaling you desire (consider making a variable out of the maximum scale value as well to make things even easier to update).

The remaining three lines of script scale the track container, the overhead container, and the car to the amount of scale_factor. Test the movie and experiment with different scale values. The following image shows the effect of scale on the track, ranging from 200% through 50%:



2. You may have found that the scaling is a little jerky. To resolve this, you can implement a simple ease algorithm. **Easing** is the process by which you set a target, establish how far away the target is, and then move a percentage toward that target, constantly reducing the distance between your current position and your target position. Think of a frog trying to jump out of a well—every time it jumps it covers half of the remaining distance. As it gets closer to the top of the well, its movements become shorter and it eases toward jumps of infinitely smaller length (and therefore it never actually manages to escape).

```
position = position + (target position - current position) * factor;
current position = position;
```

Or in the case of the scale_factor (see race_7.fla):

```
function startRace() {
  car.onEnterFrame = function(){
    // user controls code
    // track move code
    // is car over the track?
    // is car over the crash area?

    // scale track code
    rate = 0.1;
    scale_factor = 200-_root.vel*10;
    scaleSmooth = scaleSmooth+(scale_factor-container._yscale)*rate;
    container._yscale = container._xscale=scaleSmooth;
    container_over._yscale = container_over._xscale=scaleSmooth;
    this._yscale = this._xscale=scaleSmooth;
  };
}
```

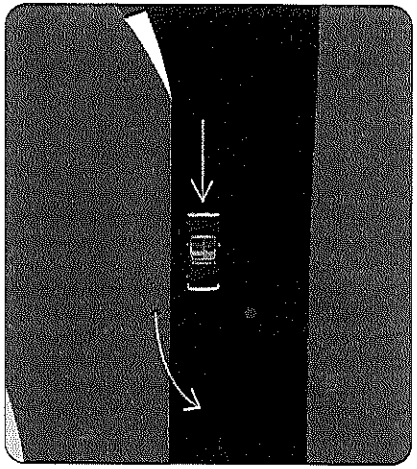
Of course, to get this to work, you need to declare an initial value for scaleSmooth when the game begins (this is a new concern with coding in Flash MX 2004). So jump back up to the init function and add this line:

```
scaleSmooth = 0;
```

Test the movie to see the effect. Experiment with different values of rate to get the desired speed of scale.

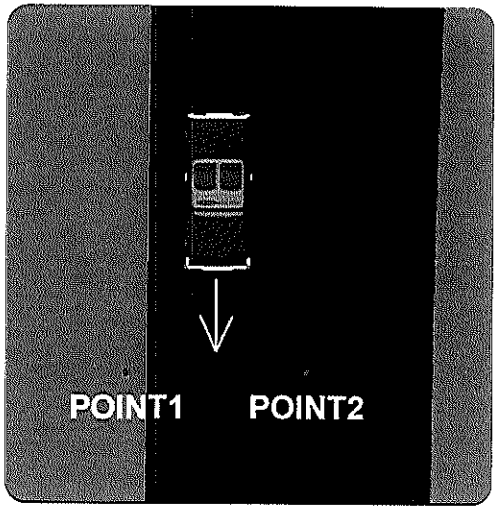
Moving the Computer-Controlled Cars

The following method for computer-controlled opponents is surprisingly straightforward. This technique is a simple form of artificial intelligence (AI), which breaks the stigma that all AI involves lots of nasty code and high-level mathematics. It also offers a wide scope for modification and inclusion into all sorts of motion and game development techniques.



Imagine that each of the opponent cars has two points ahead of it: one forward and to the left, and one forward and to the right. If the point on the left hits the track, then the car turns to the right. If the point to the right hits the track, then the car turns to the left. The following illustration shows a car traveling downward, and the red point on the car's right has hit the grass area off the track. When this hit is detected, the car is rotated a negative amount, forcing it to steer away from the detected grass area.

Easy enough! It would also be possible to add a third point directly in front of the car. Then, if the car was to hit the point directly in front of it, it would turn to the direction in which it turned last (that is, if it last turned right, then when the forward point hits the track it will continue to turn right). In other words, the three points detect the track and turn the car away to avoid running over it and thus stay on the track course. Note that the forward point isn't necessary for curved tracks because it's mostly used to avoid colliding with objects when approaching them at a perpendicular angle, so we omit it from this exercise.



1. Make a duplicate of the user's car and change its color or design to visually differentiate it from the user's car. Call this duplicate opponent and place a copy of it inside the track movie clip at a position on the track from which you want the computer-controlled car to start.

2. Edit track, select your opponent vehicle, and name its instance opponent1. Edit opponent1 and place a small red circle (about 2 pixels wide) in front of the car, convert it into a movie clip named point, and name its instance point1. Make a copy of point1 and name this second instance point2. Open your Info panel and move point1 to coordinates –20, 40 and move point2 to coordinates 20, 40.

3. You're now ready to start adding the computer AI code. The car needs to move, so let's define an onEnterFrame event handler. Add the following script to your startRace function after the car's onEnterFrame handler (alternatively, check out our version in race_8.fla):

```
container.track.opponent1.vel = 0;
container.track.opponent1.onEnterFrame = function() {
  if (this.vel <= opponent_max_vel) {
    this.vel += opponent_acceleration;
  }
};
```

Take note that you're defining the onEnterFrame handler for the opponent1 vehicle, so on each frame the variable vel (which you initialize at 0) will increase at a rate of opponent_acceleration until it reaches a maximum value, in this case opponent_max_vel (that is, the car accelerates to its top speed).

4. Now to put this acceleration to use, you need to add a few more lines of ActionScript:

```
container.track.opponent1.onEnterFrame = function() {
  var rot = this._rotation;
  if (this.vel <= opponent_max_vel) {
    this.vel += opponent_acceleration;
  }
  this._x -= Math.sin(rot*toRadians)*this.vel;
  this._y += Math.cos(rot*toRadians)*this.vel;
};
```

You're now establishing a variable rot and making it equal to the _rotation of your opponent vehicle. Using a combination of this rotational value and velocity you can then use some trigonometry similar to what you used earlier when moving the track:

```
this._x -= Math.sin(rot*toRadians)*this.vel;
this._y += Math.cos(rot*toRadians)*this.vel;
```

Notice how you now subtract the _x component and add the _y component, whereas with the track you did the opposite? Previously, you were moving the track in the opposite direction of the car. Now you're moving the car itself, thus the _x and _y component values need to change accordingly.

Test the movie and the car should drive down the bottom of your screen. Try rotating opponent1 and then testing the movie. Your car should now drive off in whatever direction it is pointing.

5. Back to those two points you created in opponent1 earlier. Place the following code after the previous chunk:

```
container.track.opponent1.point1.onEnterFrame = function() {
  var myPoint = new Object();
  myPoint.x = this._x;
  myPoint.y = this._y;
  this._parent.localToGlobal(myPoint);
  if (this._parent._parent.inner.hitTest(myPoint.x, myPoint.y, true)) {
    this._parent._rotation -= opponent_turn;
    this._parent.vel *= opponent_slow;
  }
};
```

Here you're using new Object to convert the _x and _y coordinates of point1 from local space (relative to its parent movie clip) to global space (relative to the stage). This code makes good use of the localToGlobal() coordinate-conversion method by testing to see if myPoint is over inner (your track) and, if so, it makes the car rotate away from it. The car is also partially slowed down (_parent.vel is decreased proportionally) to simulate the vehicle slowing down for corners.

6. The next batch of script is required for point2 and is exactly the same, with the exception of the direction in which the car is rotated:

```
container.track.opponent1.point2.onEnterFrame = function() {
  var myPoint = new Object();
  myPoint.x = this._x;
  myPoint.y = this._y;
  this._parent.localToGlobal(myPoint);
  if (this._parent._parent.inner.hitTest(myPoint.x, myPoint.y, true)) {
    this._parent._rotation += opponent_turn;
    this._parent.vel *= opponent_slow;
  }
};
```


By testing the movie, you should now see the car driving along the track, avoiding the edges and slowing for corners. If your car has a tendency to drive off the track or zigzag along it, then experiment with the _root.opponent_turn value or the distance in which point1 and point2 lay relative to the car. Generally, the farther away the points are from the car along the x axis, the more the car will zigzag, and the closer the points are to the car on the y axis, the less likely it is that the car will turn in time to avoid running off the track. Also be aware that the car's capability to stick to the track will be affected by its maximum velocity and rate of turn. Should you continue to have trouble, please refer to the file race_8a.fla.

Note that because the previous two onEnterFrame handlers refer to almost identical functions, you may want to consolidate your code by making a single function that both points may reference. For instance, you could replace the two previous sections of code with the following lines:

```
function checkTurn() {
  var myPoint = new Object();
  myPoint.x = this._x;
  myPoint.y = this._y;
  this._parent.localToGlobal(myPoint);
  if (this._parent._parent.inner.hitTest(myPoint.x, myPoint.y, true)) {
    this._parent._rotation += (opponent_turn*this.direction);
    this._parent.vel *= opponent_slow;
  }
};
container.track.opponent1.point1.direction = -1;
container.track.opponent1.point2.direction = 1;
container.track.opponent1.point1.onEnterFrame = checkTurn;
container.track.opponent1.point2.onEnterFrame = checkTurn;
```

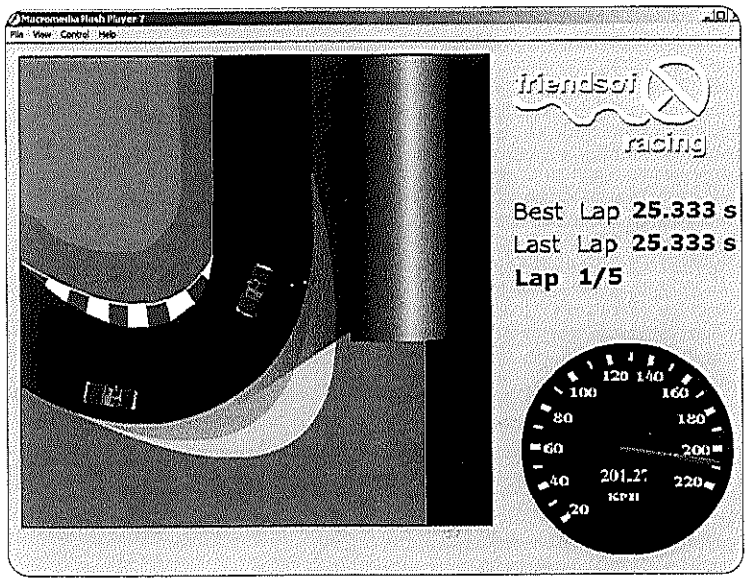
Generally, it's a good idea to abstract functionality, when you can, to keep your code more modular and easier to debug.

7. Finally, a little bit of housekeeping. When the game ends, you should turn off the event handlers used to control the opponent vehicle, so add the following script to your endRace function:

```
function endRace() {
  delete car.onEnterFrame;
  delete container.track.opponent1.point2.onEnterFrame;
  delete container.track.opponent1.point1.onEnterFrame;
  delete container.track.opponent1.onEnterFrame;
}
```

Creating the GUI

Ultimately, it's up to you as to how you'd like to lay out the different GUI items. The following is the layout that we've chosen for this demonstration:



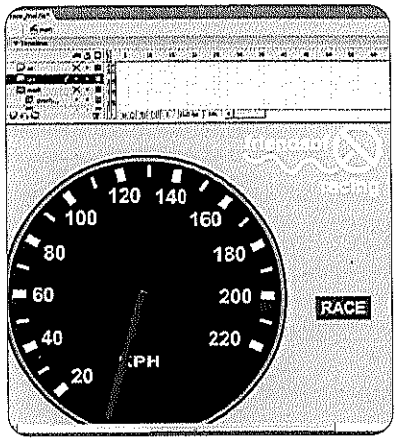
Before we demonstrate the addition of the game play elements that are tracked through the user interface, you'll need to add some further functionality to the game engine itself.

Splash Screen

You need to make room for the splash screen on the main timeline, so move all frame 1 layers to frame 2. Place a `stop()` function in the first frame on the as layer. On frame 1 of the gui layer, place any designs and instructions that you require for your game.

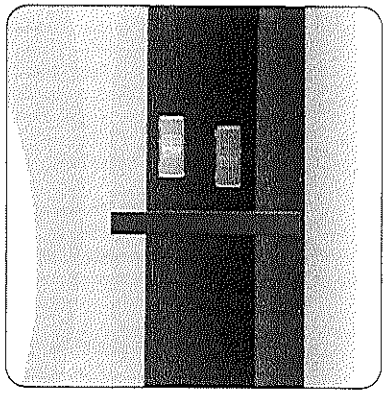
The only element that you need is a button to start the race. It requires that you place the following code on frame 1 of the as layer (refer to `race_final.fla`):

```
start_btn.onRelease = function() {
  play();
};
stop();
```



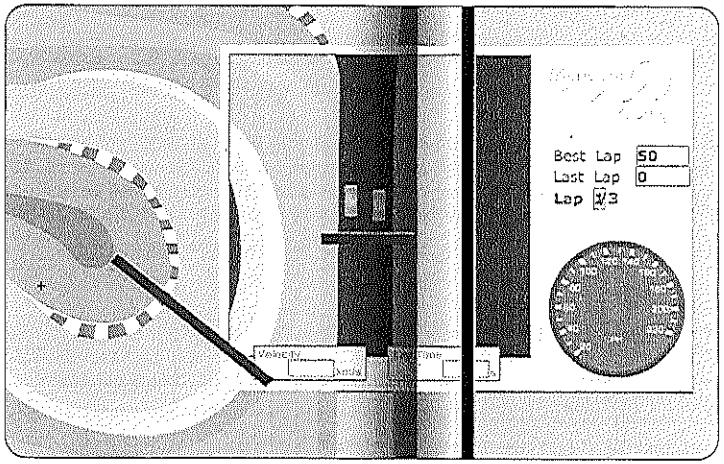
Laps

To calculate laps, you need to determine whether the starting line has been crossed and, if so, whether the player managed to go all the way around the course (rather than starting and then reversing back over the lap line). To do this, you need to add two new movie clips to the overhead layer (you add them here because you don't want them to be treated as track components).



1. On the main timeline, edit the `container_over` movie clip in place and then edit `overheads_inner` in place. On the place where you want your cars to start the race, create a small rectangle that crosses the track and is approximately 10 pixels tall (don't worry about its color—you're going to make it invisible):

2. Turn your rectangle into a movie clip named `lapstart`, name its instance `lapstart`, and place a second instance of it across the course approximately halfway down the track (and name its instance `laphalf`):



3. You've now created locations for your starting line and halfway point. So, in your `startRace` function, add the following code, which defines `onEnterFrame` for the starting line:

```
// finish line
container_over.inner.lapstart.onEnterFrame = function() {
  if (this.hitTest(car._x, car._y)) {
    if (halfway) {
      gui_lastlap = (getTimer()-startTime)/1000 + " s";
      if (gui_lastlap < gui_bestlap) {
        gui_bestlap = gui_lastlap;
      }
      if (lap == totallaps) {
        endRace();
        attachMovie("congratulations", "congrats", 100);
        congrats.onRelease = function() {
          this.removeMovieClip();
          gotoAndStop(1);
        };
        congrats._x = 180;
        congrats._y = 200;
        delete car.onEnterFrame;
      } else {
        startTime = getTimer();
        lap++;
        halfway = false;
      }
    }
  }
};
```

Let's take a moment to break this code down and ensure that it's perfectly clear. First, you determine if the player's car has hit the lap movie clip:

```
if (this.hitTest(car._x, car._y)) {
```

You then determine if the player has made it to the halfway mark (more on that in a moment):

```
if (halfway) {
```

So the player has made it to the halfway mark and back to the start again (that is, the player has finished a lap). Using the `getTimer()` function, you can determine how long it has taken the player to complete the lap, and using an `if` statement, you can determine if it was the player's fastest lap (we cover the use of the `startTime` variable shortly):

```
gui_lastlap = (getTimer()-startTime)/1000 + " s";
if (gui_lastlap < gui_bestlap) {
  gui_bestlap = gui_lastlap;
}
```

Now you test to see if the current lap is equal to the total number of laps (that is, is this lap the last one?):

```
if (lap == totallaps) {
```

If this is the last lap, you execute the `endRace` function, display the race `congratulations` movie clip, and position it accordingly:

```
if (lap ==totallaps) {
  endRace();
  attachMovie("congratulations", "congrats", 100);
  congrats.onRelease = function() {
    this.removeMovieClip();
    gotoAndStop(1);
  };
  congrats._x = 180;
  congrats._y = 200;
  delete car.onEnterFrame;
}
```

For the `congratulations` movie clip, create a rectangle (200x100) with the words You've finished the race in it, convert it into a movie clip named `congratulations`, and set its linkage properties to Export for ActionScript, Export in first frame, and its linkage ID to `congratulations`.

If this isn't the last lap, you increment the lap count and switch the `halfway` lap flag off:

```
} else {
  startTime = getTimer();
  lap++;
  halfway = false;
}
```

Notice that you also set (or in the case of lap 2 and onward, reset) the variable you're using to determine the starting time of the lap:

```
startTime = getTimer();
```

4. Of course, in Flash MX 2004, you need to initialize `startTime` so that when you evaluate it at the first lap, the mathematical operation doesn't return `NaN` (an undefined variable in previous versions of Flash defaulted to 0, but in Flash MX 2004 it defaults to `undefined`). Add the following line to your `init` function:

```
startTime = getTimer();
```

5. For your second movie clip, `laphalf` (the one placed halfway around the track), you need to add the following code:

```
// halfway line
container_over.inner.laphalf.onEnterFrame = function() {
  if (this.hitTest(car._x, car._y)) {
    halfway = true;
  }
};
```

This one is much simpler in that you determine if the car has hit the `halfway` movie clip. If so, you set the `halfway` flag to `true`.

6. You can now set the `_alpha` for both movie clips to 0 (via the Color options in the Property inspector's color list menu), thus making them invisible to the user but still available for lap counting. You're now

successfully counting laps (so that there are no cheaters!) and triggering a congratulations event when the finish line is reached.

7. Once again, you need to finish by turning off your handlers used for lap counting at the end, so add the following lines of script to the endRace function:

```
function endRace() {
  delete car.onEnterFrame;
  delete container.track.opponent1.point2.onEnterFrame;
  delete container.track.opponent1.point1.onEnterFrame;
  delete container.track.opponent1.onEnterFrame;
  delete container_over.inner.lapstart.onEnterFrame;
  delete container_over.inner.laphalf.onEnterFrame;
}
```

Displaying Variables

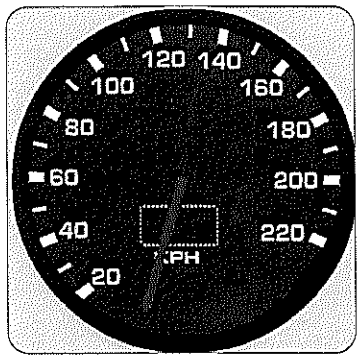
Displaying game variables is a simple process and adds greatly to the overall game experience. You can display variables such as current lap (lap), total laps (totallaps), last lap (gui_lastlap), and best lap (gui_bestlap) by placing dynamic text fields onto the stage and in their var field by placing the name (or path) of the variable you want to display.

1. In the case of adding the car's speed to the GUI, place the following code in the onEnterFrame portion of the startRace function:

```
// gui calculations
gui_vel = vel*20;
gui_needle._rotation = gui_vel*1.3;
```

   This code calculates the display velocity and rotates a needle on a tachometer. The multiplication factor is up to you—it relates to how fast you think the cars are going in terms of a real-world value.

2. To display these values, design a tachometer that looks something like this:



3. Turn the needle into a movie clip and name its instance gui_needle. Adjust it so that the needle's center of rotation is centered to the movie clip. Somewhere on the tachometer (in this case, above the KPH label), place a dynamic text field and name its instance gui_vel. Test the game and adjust the gui_vel and gui_needle._rotation multiplication factors to your satisfaction.

You can find all of these GUI additions in the file race_final.fla.

Adding Audio

Music tracks and event effects add great depth to the game play, as does ambient sound. There are so many options to consider that it's not possible to cover them all in this chapter. Therefore we focus on a method to add audio dynamically from your Library, as well as a nice little trick that allows you to alter the pitch of a sound.

Sound is best controlled through the sound object as opposed to attaching sounds to frames. You can do this by importing a sound and giving it a linkage name—preparing sound assets is very much the same as preparing movie clip assets. Once you've prepared your sound, you attach and play it by creating an instance of the sound object, attaching a sound to it, and telling it to play, just like this:

```
mySound = new Sound();
mySound.attachSound("mySoundAsset");
mySound.start();
```

In the context of this game, you can create multiple instances of the sound object (with names such as mySound, mySound2, carSound, and so on) by placing the first two lines of script in the init function. You then trigger them by placing them throughout the rest of the ActionScript. For example, you could make the sound of the car crashing like this:

```
crashSound = new Sound();
crashSound.attachSound("crash");
```

And then you could trigger this sound, for example, as part of the relevant crash script:

```
if (container.crash.hitTest(this._x, this._y, true)) {
  endRace();
  attachMovie("game_over", "game_over", 1);
  game_over.onRelease = function() {
    removeMovieClip(this);
    gotoAndStop(1);
  };
  game_over._x = 180;
  game_over._y = 200;
  crashSound.start();
}
```

So far we've been looking at sound at a high level. For more details on using sound in games, see the "Sound for Games" chapter. Let's move on to a neat little trick that demonstrates how you can use the sound object. For this example, we created the sound of an engine revving increasingly faster. If you can't source or create a similar sound, then just use the audio samples supplied within race_final.fla.

1. Once you have your audio file imported into your Library, edit its linkage properties, select Export for ActionScript, and make engine its identifier. Make an instance of the sound object by placing the following script inside the init function of your racing car game:

```
engineSnd = new Sound();
engineSnd.attachSound("engine");
```

2. You're now going to add a function that plays the audio but does it in a way that is probably very different from how you've thought about the sound object so far. Place this function at the end of your code:

```
function engine() {
  var enginePercent = vel/max_vel;
  engineSnd.stop();
  engineSnd.start((engineSnd.duration)*enginePercent/1300);
}
engineInt = setInterval(engine, 100);
```

This function combined with setInterval makes the audio file play from different spots (depending on the car's velocity) in intervals of a hundredth of a second. The result is the sound of an engine that changes pitch relative to the car's speed and its change in speed. Let's walk through it and have a look at what's going on:

```
var enginePercent = vel/max_vel;
```

This line determines the current velocity as a fraction of the maximum velocity (resulting in a number between 0 and 1).

```
engineSnd.stop();
```

This line is fairly obvious—it makes the engine sound stop.

```
engineSnd.start((engineSnd.duration)*enginePercent/1300);
```

This line makes the sound play again, only this time from a different starting point. At first you discovered the percentage of maximum velocity that the car was traveling out; now you want to play the sound at that percentage value. You determine this starting point by multiplying the duration of the sound (which is in milliseconds) by the percentage. Your starting point needs to be in seconds, so you should divide by 1,000. But you don't want the audio to start right at the end (because there would be nothing left to play), so you divide by a number greater than 1,000 (in this case, 1,300. Although there's a mathematical rule of thumb here, we suggest trial and error, as slight changes can have significant effects on the sound of the engine at maximum velocity).

```
engineInt = setInterval(engine, 100);
```

Finally, you use setInterval to run the function independent of the timeline. In this case, you're playing (well, replaying) the audio file every hundredth of a second. You can change this value to get different results. You set the interval ID to enginInt, which allows you to clear the interval when the game is completed.

> *Note that Flash doesn't handle audio all that well, so this technique can result in undesirable audio artifacts depending on your sound card and machine setup. If you encounter undesirable sound, try changing the audio or* `setInterval` *rate, or increasing the division value used in* `engineSnd.start()`.

3. When the game is finished (whether through completion of the laps or by running into an obstacle), you need to stop the engine roar by stopping the sound. You do this by simply adding two lines to your endRace function:

```
function endRace() {
  delete car.onEnterFrame;
  delete container.track.opponent1.point2.onEnterFrame;
  delete container.track.opponent1.point1.onEnterFrame;
  delete container.track.opponent1.onEnterFrame;
  delete container_over.inner.lapstart.onEnterFrame;
  delete container_over.inner.laphalf.onEnterFrame;
  clearInterval(engineInt);
  stopAllSounds();
}
```

These lines stop the interval from running and stop all sounds currently playing in the movie. If you had more sounds playing in the game (such as music), you might want to simply stop the engine with `engineSnd.stop()`, but this method works fine for the current game.

Adding Customization and Enhancements

So far we've discussed the development of a basic car racing game and we've covered the fundamental aspects. You can expand these and create your own game-play enhancements and customized functionality. Here are a few suggestions and samples to get you started on building the best darn racing game ever made in Flash.

Power-Ups, Bonuses, and Special Items

Previously we discussed the creation of track and crash layers. Try adding a third layer just for power-ups. Whenever the user drives over a power-up, the user's maximum velocity is temporarily (or permanently) increased. Using the same detection methods, you could allow the computer-controlled cars to also gain power-ups, making the game a struggle to reach the special items before the opponents.

Different Race Surfaces

So far, you have a slowdown area and a collision area in the game, but you can also add many other areas, such as a speed-up area (oil slicks), an area with an increased rotation angle (ice), and so on, simulating other racing surfaces.

Adding Sound Effects

You can easily add sound effects to the movie and to specific events that occur during the race—we've included a skidding sound clip (`skid.wav`) in `race_final.fla` for you to play around with. You can also try adding a background soundtrack and sound effects to the key events to add greater depth to the game play.

Track Selection

Instead of sticking to the same track, why not create multiple tracks and display them by changing the contents of the `track` and `overheads` movie clips?

Adding Animation

In this current game version, you start the race as soon as you go to frame 2. By adding a starting lights animation that plays through and then calls `startRace`, you can offset the time before the player starts the actual race. Keep in mind that unless you add a race-started flag for the opponent vehicles, they'll start before you get the chance to. Why not animate the scenery, such as the water and tree, or better yet, add some parallax effects with clouds, birds, and airships that scroll at a quicker speed compared to the track in order to add a greater feeling of depth?

Collision Detection

The clip that represents the set of collision areas can be dynamic; you can include obstacles (such as randomly rolling barrels, animals, or broken-down race cars). The opponent vehicle is also part of the collision area, but because collision is detected based on a single point, you could add more advanced collision detection such as rectangle or even circular collision.

Finish Line

Using a similar technique to the lap counter mechanism, you can count laps for the opponent cars as well. By implementing this, you can determine the final finishing order for the play and computer-controlled cars.

Damage

Whenever the car leaves the track and collides with something on the crash layer, the car is destroyed and the game is over. Alternatively, if the car leaves the track but doesn't crash, it is, instead, slowed down. You could add a variable that increments whenever the car hits the track layer so that when it hits a certain amount, the car is destroyed and you play the crash sequence.

Difficulty

You can achieve different difficulty levels by adding additional opponent cars, increasing the cars' acceleration and velocity, adding power-ups and power-downs, and increasing the player's damage sensitivity. Increase the difficulty progressively by adding tracks and obstacles that become increasingly tricky, or change the functionality so that the game is time-trial based and the tracks have to be completed in a progressively decreasing amount of time.

Multiuser

Through the use of XML socket technology, or indeed the Flash Communication Server, you're able to re-create the game engine, which allows multiple users to access and play on the same track.

Summary

We explored many different techniques in this chapter that cover a wide range of different purposes. We looked at

- Drawing techniques in the Flash authoring environment
- Using keyboard input with ActionScript
- Using basic trigonometry
- Moving and scaling movie clips
- Detecting collisions
- Implementing path-following AI
- Creating a user interface
- Adding events sound and manipulating sound dynamically

By breaking down these techniques into modules and applying them systematically, you can now see how you can use some simple ActionScript to great effect. This knowledge, combined with what you've learned in the other chapters of this book, will allow you to create high-quality, engaging games.

The most important thing to remember is now that you've learned the basics of racing game development, you're ready to add your own creativity and personal flair to the creation of bigger and better games.

For further information and more code samples, please refer to `www.arseiam.com`.