A Racing Game Tutorial

A second tutorial for Unity designed to allow the reader to create a complete racing game. This tutorial will introduce more advanced concepts and techniques in Unity, and focus on scripting advanced functions within the game. We will gradually build a complete game following easy step-by-step instructions and a progression of concepts.

Contents

I – Introduction

Welcome to the **second** tutorial created for Unity! The idea behind this second tutorial is to provide users the opportunity to create a full, working game. This is a more advanced tutorial and will focus more on the general concepts of programming and scripting a complete game within the Unity environment. It will introduce you to advanced physics and modeling, camera views and styles that work for the racing genre, the basics of enemy AI, triggers, setting up more complex GUIs for a game, scoring and end-game goals. This tutorial assumes that you have a decent familiarity with the Unity interface and workflow, and also some familiarity with the first tutorial, as we are going to skip some basic concepts to quickly get in-depth into the game.

The tutorial will lead you through a series of steps that will build on one another, with detailed explanations so that you have an excellent working knowledge of the technical aspects of the finished racing game. There will also be some detailed explanations of basic programming questions and concepts. When complete, you will be able to take away scripts and techniques applicable to many other types of games that can be used in your own projects.

Theses scripts include such things as: player car/object scripts, camera scripts, enemy and AI scripts, goal trigger scripts, creating waypoints, timer scripts, and game startup and GUI scripts. Because of the complexity of this tutorial, it will be broken down into several main sections, some of which will have a number of specific subsections, as shown in the *Contents* at left.

The Goals Of This Tutorial

This tutorial was intended to be a script tutorial that really focuses solely on scripting; it is meant to be an introductory tutorial on scripting in Unity for beginners. For this reason, the models are kept as simple as possible, and in the scripts we chose to use the simplest possible solutions to get a functional game. The car, for example can be tweaked much more than we did in the tutorial. This tutorial is not meant to create a *fun* game, but rather it is meant as a scripting tutorial to show the complete process of creating a game from the scripting side.

Thanks

Thanks To Those Who Helped!

I was thrilled when the guys at OTEE allowed me to help create a second, more complete, complex tutorial for the Unity 1.1 release.

Joachim and Nicholas at OTEE were, as always, especially helpful and extremely responsive and professional. David and Keli at OTEE also provided a great deal of help, ideas, direction and support for this tutorial.

The Unity forums are an active community of involved, fun, resourceful users of Unity and I'd strongly recommend visiting the forums to contribute and help others create cool things with Unity. The forums are online at http://ww.otee.dk/forum/. Thanks to all the regulars there who help out!

Unity 1.1 brings a whole bunch of very cool updates and improvements to the application. Among these: the ability to build standalone Windows applications, easier level loading, updates to hinges and a new raycast collider, water textures, and many more.

Unity version 1.5 has introduced specialized wheel colliders. The new wheel collider has support for slip curves, a specialized friction model for cars, dampers, and allows you to model anything from a dune buggy through to a F1 racing car game.
See page 42 for details.

Any errors in this tutorial are the authors', and not OTEEs. 3 September 2006. Version 1.2.

A Note About The Files

The most up-to-date versions of the files for this tutorial are always available for download at the following URLs:

http://www.otee.dk/tutorials/car_tutorial.zip

http://www.otee.dk/tutorials/car_tutorial_complete.zip

There are two versions of the project files supplied with this tutorial. The first version provides all of the assets (scripts, objects, models, textures, physic materials etc) but assumes that the reader will **assemble these into working form** and create the working scenes by reading through this tutorial and actually doing all of the steps as outlined in the tutorial. This "bare bones" version starts with a single scene containing only a track and non-functioning player car. This is the *recommended* starting point and goal for readers when starting working on this tutorial.

The purpose of this method is that at the end of each of the tutorial's subsections and sections, the reader will have a completed, functioning scene with which to build the next scene with.

The second version (car_tutorial_complete) is a completed and fully working version of the tutorial and all of the scenes to be used as a reference should a part of the tutorial not make sense or the reader wants to see a section of the tutorial fully functioning.

Thanks!

Most thanks must go to the guys at OTEE who helped create this joint-effort tutorial. Their help with creating the scripts and explaining them to me for the tutorial was excellent. The tutorial certainly wouldn't exist without especially Joachim's patience, support, and desire to help me get it packaged like this. The great community on the Unity forums also have contributed a great deal to the development and exciting nature of Unity. Thanks, guys! Thanks also to fellow WidgetMonkey, Ron Letkeman, the artist and modeler who always come up with fun ideas for games and the art to make them come to life. And finally, hello to my kids, Cal and Luke.

Tutorial Assets

Racing Game Tutorial Assets

The latest versions of the two files to work through this tutorial are found online as explained on **page 2**. One file is the tutorial with assets set up for the reader to work through creating the full game, while the second is a completed version of the project for reference purposes.

I've assumed that readers will want to use this tutorial as a basis to build larger games, so the game assets have been structured into folders that are based on the asset's **function** rather than type, the reasons for which were explained in the first tutorial (mainly ease of finding assets in complex game projects).

Remember, this tutorial was created using the latest version of Unity as of this writing, version 1.1.

The 2 Split Layout

But Mine Doesn't Look Like That!

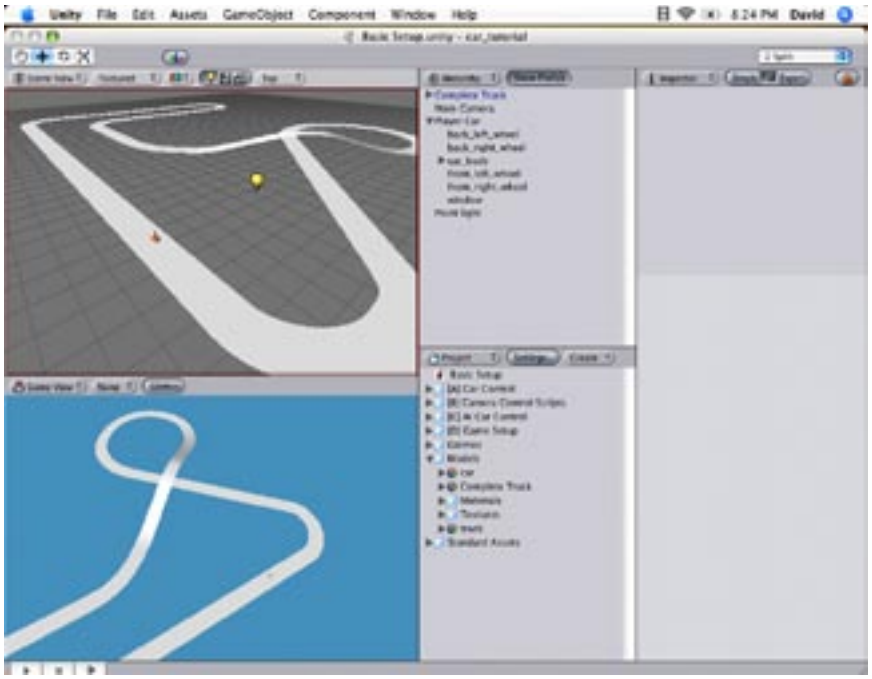
I'm using the 2 Split layout for this tutorial because I find it the most basic and intuitive of the UI layouts that come with Unity. Feel free though to use the layout that works best for you while working on the tutorial. But note that if you do use a different screen interface, your screens will vary from the ones shown here.

II – Setting the Scene

In this section, we're going to quickly set up the physical objects and assets that will be necessary in our racing game. These include the players and enemy cars, the racing track, scripts for each of the tutorial sections, and pre-made scenes that explain the tutorial's concepts. We'll start by downloading the pre-made assets and creating our project by opening them.

Begin by downloading the *Tutorial Asset* files and then unzipping the files to create the "car_tutorial" and "car_tutorial_complete" project folders. Both folders contain all of the required assets for the tutorial separated into logically arranged subfolders.

Move these folders where you prefer on your hard drive and then open Unity. Select File -> Open Project and locate and open the "car_tutorial" project. Unity will restart, update the assets, and then open the standard tutorial game world. Open the Basic Setup scene. We'll make two minor modifications now by changing a couple of the values in Edit -> Project Settings -> Player. Change the Default Size of the build to 800x600, and change the Default Graphics Quality to 4 (1024 x 768 is now the default setting for new projects). If you work in the 2 Split layout your screen will look like this:



We'll use the provided assets ([A] Car Control through [D] Game Setup) to work our way through the sections and concepts in the tutorial. At the end of the tutorial, we will have learned enough to create a small series of complete scenes that incorporate all of the tutorial concepts, and create a complete racing game. To begin, we've opened the Basic Setup scene and will start work on the first working section, III - Controlling the Player's Car.

What is Code?

Showing Code In The Tutorial

I will be showing all examples of scripts in a Courier typeface and inside a faint blue box, as shown below, so that code is easily picked out from the rest of the tutorial content.

```
var power = 3.0;

function FixedUpdate () {

// When you press space

// we move up the box

if (Input.GetButton
("Jump")) {

rigidbody.AddForce (Vector3.
up * power);

    }

}
```

III – Controlling the Player's Car

This is the longest section of the tutorial, so it's been broken into five major subsections. Each of the subsections builds upon the knowledge learned in the previous subsection so that at the end of Section III, you'll have a thorough knowledge of many important aspects of Unity and will have saved several scenes of your work.

Controlling a player's car in a game requires an understanding of physics in game engines, with a main focus on motion of a vehicle, over time, through a specific game world. It also involves the mechanics of collisions (and collision detection), and will introduce the concept of variables that will be required for this sort of player control of objects.

Many Unity users will be familiar with much of what is discussed in this section. This section will provide a complete understanding of all of these concepts to the beginning user and, for the advanced user, I hope to illustrate how Unity uniquely handles all of these physical and mechanical concepts, so that they can be used as a basis and help you develop your own titles.

We'll start with the very basics in the first subsection – moving a player's car forward through the world

1. Moving Forward

Objects in a scene are moved by using "Transform" and an "Update" function that is "called" (runs) every frame. Transforms take care of all the movement of every object in Unity. Before we get to the code, let's review some very basic coding stuff.

A function is a piece of JavaScript code that is defined once, but then can be called many times by the program. A JavaScript function can be passed parameters that specify the values that a function will operate on. Functions can also return values back to the program. Functions are defined in JavaScript like this:

```
function square (x) {
   return x * x;
}
```

Once a function is defined you can invoke it by following the function's name with an optional comma-separate list of arguments within parentheses. The following four lines of code would print the number 9 to the console:

```
function square (x) {
   return x * x;
}
print (square(3));
```

Functions

A Complete List

You can find a complete list of all of the functions that are automatically called by Unity in OTEE's online script reference documentation.

You Comment, Right?

What's This Section Of Code Do?!

In my day job, I code CSS websites. Big, big websites with lots of CSS to make sure everything is accessible and compliant. And the only way to make sure that I can remember what I did six months ago, or make sure that the guy in the office next to me understands what he's looking at, is to comment my CSS code really, really well.

I'd strongly recommend, even if you work alone, that you comment your code well enough that anyone looking at it can understand what they're looking at and what does what. In JavaScript, comments are those helpful lines of code like this ...

```
// fade screen to black
```

lines that you will thank yourself for later. It's just good coding practice.

Some functions are automatically called on certain events. An important and very useful function in Unity that is called every frame is the "Update" function.

To do anything interesting in your game, you'll have to call some of Unity's built in functions. A few paragraphs ago I mentioned that objects in a scene are moved by using the class "Transform" and an "Update" function that is called every frame. Transforms take care of all the movement of every object in Unity. A complete list of all function's supported by Transform can be found online.

To call a function on an object, you refer to the object, followed by a period and the function to call. In the following example, we call the Transform's Translate function. It has 3 parameters that are used to define how much to transform (move) an object along it's x, y and z axis.

```
// Move the car forward along it's z-axis
function Update () {
   transform.Translate (0, 0, 1);
}
```

In the tutorial, this is the MoveForward.js script in the [A] Car Control -> 1 Move Forward folder, and has the comment line at the beginning to remind us and anyone else who looks at our code what this does (see the *You Comment, Right?* note at left). Attach this script to the Player Car object using drag and drop so that the car will simply move forward one unit along it's z axis each frame. The z axis is the blue arrow when you select the car. Run your scene to see this script in action. Do a Save as... and save the scene as Move Forward in the 1 Move Forward. Play with the values of the script to make the car go up or sideways instead of forward, or to go faster forward.

2. Moving Left and Right

We're now going to use Transform and Input to move the car left and right. Open up the Basic Setup scene again and attach the [A] Car Control -> 2 Move And Turn -> MoveAndTurn.js script to the Player Car. Run the scene and use the arrow keys to move the player car. The motion is very simplistic but we're now using input from the player to modify an object within the game world. Here's the code (MoveAndTurn.js) that's making this happen:

```
// The Update function is called once every frame
function Update () {

   // Apply motion along the z axis of the car
   transform.Translate (0, 0, Input.GetAxis
   ("Vertical"));

   // Apply motion along the y axis of the object
   transform.Rotate (0, Input.GetAxis
   ("Horizontal"), 0);

}
```

We're still using the Update function to make the car move, but only every frame it receives player input. Instead of declaring a number of units to move along either the z or y axis as in our first section, we're using a new function to make movement happen: Input.GetAxis; and one new function called transform.Rotate to make the car turn. These are basic movement functions in Unity.

Input.GetAxis is a function that looks for mouse or keyboard input from the user and who's values range from 1 to -1 (for keyboard). You can change many of the settings/variables associated with the Input.GetAxis using the Edit -> Project Settings -> Input menu and find more information in the online script reference documentation.

transform.Rotate is similar to transform.Translate, except that instead of moving ("translating") an object along an axis, it rotates the object around the specified axis. In our example, it will continue to rotate the object around the specified axis as long as one of the "Horizontal" input keys (the left and right arrows) are held down. I mentioned that the motion of the player car is still simplistic. But before we add some more physics to the car's motion, I want to review to concept of variables in Unity. Save as... and save this completed scene in 2 Move And Turn as Move Forward.

Framerate Independent Movement

I'd like to take a second to go off track (pardon the pun) here for a few minutes and talk about something called "framerate indepedent movement." Notice how old computer games play really quickly on your new computer? Or how one frame of game action can take a different amount of time depending on how much motion or how many explosions were taking place?

That's the result of framerate *dependent* movement.

```
// Move the object along it's z-axis
function Update () {
   transform.Translate (0, 0, 1);
}
```

In the above code, we move an object 1m along it's z axis **per frame**, but this will move the object a different distance along the z axis **per second** on different machines.

To work around this issue, you will most always want to change values dependent on **time**. In our simple example, you want to move a car 1 meter per second. In code this translates to:

```
// Move the car along it's z-axis 1m/second
function Update () {
   transform.Translate (0, 0, 1 * Time.deltaTime);
}
```

Time.deltaTime is the amount of time the last frame took. By multiplying with Time.deltaTime, you make your objects move 1 meter per second instead of 1 meter per frame. When should you multiply by Time.deltaTime? You want to multiply by delta time when **continously changing a value every frame**. Most of the time this is when dealing with input, as shown in the following two simple code examples:

```
function FixedUpdate () {
   transform.position.x += Input.GetAxis
   ("Horizontal") * Time.deltaTime;
}
```

```
function Update () {
   light.range += 0.1 * Time.deltaTime;
}
```

There is one notable exception when dealing with input: **Mouse delta**, which is always absolute. It is how far the mouse moved. Thus you should not multiply by delta time. A simple code example of this is shown below:

```
function Update () {
   transform.Rotate (0, Input.GetAxis ("Mouse X"), 0);
}
```

In the next subsection, An Introduction To Variables, we will apply this knowledge about framerate independent movement.

Case Sensitivity

JavaScript Is Case Sensitive

You will need to remember that JavaScript is a case-sensitive language. This means that language keywords, variables, function names and other identifiers have to typed with consistent capitalization of letters.

For example, the function keyword has to written "function", and not "Function" or "FUNCTION".

3. An Introduction to Variables

Many Unity users have some programming experience and will be able to simply skim this section. But since this tutorial has been designed as an introduction to Unity's programming functionality, we should explore a cornerstone of computer programming known as variables for newer users. The concept of variables will come into play in the next section of the tutorial when I will add some advanced physics and behvaiours to the player's car.

At its most basic, variables are used to store and manipulate data. A variable has a unique name and stores a value (numeric, text etc) that can be used by the program to do something.

The first line of code in the following example creates a variable named "i" and assigns the value "3" to it, while the second line of code creates a variable named "sum" whose value is equal to our first variable "i" multiplied by "2". In this case, "sum" would be equal to "6".

```
var i = 3;
var sum = i * 2;
```

Variables also have something called "scope;" most easily understood as being the area of your program in which it is defined and used. A "public" variable is defined outside any functions and available for use everywhere inside of a script, while a "local" variable is defined only inside a specific function where it is created, as shown here:

```
var publicVar = 1;
function Foo () {
   var localVar = 1;
}
```

Function parameters are also local; in the following example, `t` is a parameter and thus considered a local variable:

```
function Square (t) {
   return t*t;
}
```

Any public variables in your script will show up in the inspector when you attach a script to a game object. In the inspector you can then change the value. How is this important? The value that is showing in the Inspector can be changed in the Inspector and overrides the value used in the script. This means that ...

```
var i = 5;
print (i);
```

doesn't always print 5, because you can change the value of `i` in the Inspector to say `7`.

We can summarize what we now know about variables so far by way of a short example script (shown below): playerLives is a public variable, while carSpeed is a local variable.

```
// A public variable needs to be defined outside of
// any function blocks as shown with playerLives

var playerLives = 3;

function Update () {
   // Every frame we decrease playerLives by 1.
   // This means playerLives will become
   // smaller and smaller every frame.

 playerLives -= 1;

// Car speed is declared inside the Update function
// so it is a local variable. It can not be
// used by other functions in the script.
// At the end of the Update function it will
// always equal 3.

   var carSpeed = 2.0;
   carSpeed += 1;
}
```

That was pretty easy, huh?

So let's get back to the game now and combine some of this coding stuff into making something fun happen to the player's car. Our next subsection, Moving Left And Right With Variables, will make the player car move left and right using variables in a new script.

Important Note!

Unity-specific Terminology

I have put comments in the previous two code examples that state:

```
// The speed variables can be changed in the
// inspector without changing any scripts.
```

In Unity, component variables are known as "properties". This means that the term "public variable" actually means, and is the same as, a "component property".

Moving Left and Right (With Variables)

Open up the Basic Setup scene again and attach the [A] Car Control -> 3 Variables -> Variables.js script to the Player Car. Run the scene and use the arrow keys to move the player car. The motion has now dramatically changed when compared to the previous scene. It's now the Variables.js script in this asset folder making the player's car move. Here it is in detail:

```
// Variables defined outside of functions appear
// in the inspector and can be changed there
// without having to change the script

var speed = 10.0;
var rotationSpeed = 100.0;

function Update () {

   // The update function is called every frame.
   // Read user input and store it in a
   // translation and rotation variable

   var translation = Input.GetAxis ("Vertical");
   var rotation = Input.GetAxis ("Horizontal");

   // Multiply speed with the input.
   // The speed variable can be changed in the
   // inspector without changing any scripts

   translation = speed * translation;
   rotation = rotationSpeed * rotation;

   // To make the car move frame rate independent,
   // we multiply how quickly we rotate and move
   // with Time.deltaTime.

   translation *= Time.deltaTime;
   rotation *= Time.deltaTime;

   // Apply translation along z axis of the car
   transform.Translate (0, 0, translation);

   // Rotate around the y-axis
   transform.Rotate (0, rotation, 0);

}
```

Notice that `var speed = 10.0;` and `var rotationSpeed = 100.0;` are public variables that are outside the functions and will appear, and can be edited, in the Inspector panel. feel free to experiment with them. Don'f forget to now Save as... and save this completed scene in 3 Variables as Variables.

Transform

Transform Is An Important Class

Transform is one of the main classes in Unity. Most game object manipulations are done either through the game object's Transform and/or Rigidbody.

Every object in a scene has a Transform. It's used to store and manipulate the position, rotation and scale of the object. Transform has a number of variables and functions associated with it, and a complete list of all of these are online in OTEE's script reference web page.

Every Transform can have a parent, which allows you to apply position, rotation and scale hierachically. This is the hierarchy seen in the Hierarchy pane. They also support enumerators so you can loop through children.

The script makes use of everything we discussed in the Introduction to Variables section. The player car now slows gradually after the player stops holding the arrow keys.This is a bit weird at first because our code actually just moves the car forward with a constant velocity. But the Input.GetAxis function automatically smoothes keyboard input and because we multiply the input with velocity the velocity is automatically smoothed. You can modify the smoothness of keyboard input, setup joysticks etc. via the input manager. Edit -> Project Settings -> Input, and see the Input reference online for more information. With these basic in mind, we can now bring other physics into our racing game; specifically, RigidBody and Colliders.

4. Basic Physics

In this scene, we want to start to affect the player's car through the use of forces and torques (angular forces) rather than position and rotation directly. This has several advantages including more realistic motion, but more importantly, we **don't** have to multiply movements by Time.deltaTime when working with physics because forces are already time independent. Once again, start by opening up the Basic Setup scene, then click on [A] Car Control -> 4 Physics -> SimplePhysics.js and examine it in the Inspector panel.

The SimplePhysics.js script for this is not any more complex than the script for Moving Left and Right With Variables, shown here:

```
// We are now working with forces and torques
// (Angular forces) instead of position/rotation-
var speed = 30.0;
var rotationSpeed = 10.0;

function FixedUpdate () {

   // Multiply speed with the input.
   // var speed can be changed in the inspector
   // without changing any scripts
   var force = Input.GetAxis ("Vertical");
   force = speed * force;

   // Don't multiply by Time.deltaTime
   // since forces are already time independent.

   // Apply force along the z axis of the object
   rigidbody.AddRelativeForce (0, 0, force);
   var torque = Input.GetAxis ("Horizontal");
   torque = rotationSpeed * torque;

   // Rotate around the world y-axis
   rigidbody.AddTorque (0, torque, 0);

}
```

So now let's create a working version of this scene.

Raycast Colliders

Collision Detection Using Rays

A raycast collider simply uses a ray for collision detection instead of a volume shape such as a sphere, box, or capsule.

Raycast colliders are useful when modelling cars because any volume based colliders can have small numerical imprecisions, which makes the motion more jerky. Raycast colliders are, therefore, a little more robust. When dealing with fast moving cars this is very important.

In this section we will build the first version of a raycast car. A raycast car is a car which **slides** over the ground. It's wheels don't spin as at drives forward; in this respect it is actually more like a snowmobile than a car. Every frame we apply a force to move it forward. It doesn't move sideways because we will use a Physic Material with *anisotropic friction*. Anisotropic friction is a necsssary part of a game involving motion because it allows you to have different friction values for forward or sideways motion. When moving sideways we will use a much higher friction value than forward. This will make the car tend to slide forward instead of sideways and follow the front wheels when they are rotated. We'll use a simple, three-step process to build the first version of a raycast car for the player to control.

First, we want to create colliders for the car: add a box collider to the car_body of Player Car, then add raycast colliders to each of the four wheels. These are added to an object by selecting the object in the Scene view or the Hierarchy panel, and then selecting Component -> Dynamics from the menubar. A raycast collider simply uses a ray for collision detection instead of a volume shape; i.e., a sphere. Also attach a RigidBody with a mass of 10 to the Player Car.

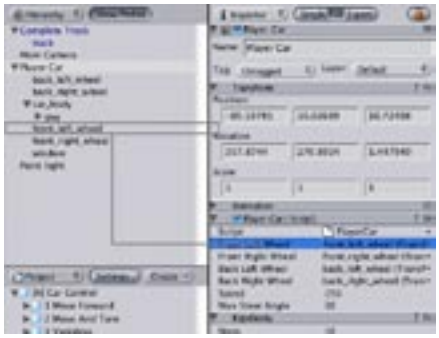
Second, create a Physic Material and set it up as shown in the following screenshot. Rename it WheelMaterial and move it to the 4 Physics directory. Assign this new material to all the 4 wheels using drag and drop.

Above: The Physic Material settings for the next section, Advanced Physics. these apply to both the front and back wheels. See the middle of page 13 for an explanation.

Connecting Variables

Public Variables In The Inspector

As mentioned much earlier, variables in a script appear in the Inspector panel and will need to be "associated" with an object. In this case, our four variables (frontLeftWheel, frontRightWheel etc) appear linked to "None" when we first click on the Play Car in the Hierarchy panel. You link a variable to an object by simply dragging the object onto the variable in the Inspector as shown in this screenshot.



Private Variables

Public vs. Private Variables

In Section 3 we introduced public variables. Public variables are declared like this `var i = 5;`.

It is also possible to declare a variable as private as shown here:

```
private var i = 5;
```

Declaring a variable as private is useful for storing state that should not be accessed from other scripts. Private variables are also not visible in the inspector. This is quite useful to unclutter the inspector from unneccessary variables.

We will be using private variables in the Physics.js script on the following pages.

You create a new Physic Material by clicking on the "Create" button in your project panel and selecting Physic Material. These materials contain all the friction and bounciness parameters that are needed to handle and enhance collisions between objects. Physic Material also has a setting to combine/multiply these effects to increase these two forces.

The third and final step is to attach the SimplePhysics script located in your Project panel 4 Simple Physics to the car. If you play with the value of the variables of the Simple Physics script in the Inspector make sure that Speed is at least 100 and Rotation Speed is at least 80.

We can now Run this scene and test the motion of our car. Save the scene as Simple Physics in the 4 Simple Physics directory.

5. Advanced Physics

In this section we're going to improve the raycast car. The main change we'll make is that we will now rotate the wheels instead of applying a torque to the entire car to make it rotate. Start by opening up the Basic Setup scene, and adding a box collider, raycast colliders, and a RigidBody to the various Player Car parts as we did on page 12 in the previous section. Save as... the scene now in the 5 More Physics directory naming it simply Physics.

Now create and save two new Physic Materials – one for the back wheel named BackWheel, and one for the front wheel named FrontWheel. These will give us more control over the car handling, because we can tweak and fiddle with the friction values for the front and back wheels as separate entities in the Inspector. Set these up using the values as shown in the smaller Physic Materials setting screen shot in the *left sidebar* on page 12.

Notice how we refined the values we've assigned the Physic Materials by adding a springy contact to the wheel's physics material to make it soft. This is done by enabling the use Spring flag, and setting spring to 25 and damper to 5. Feel free to play around with these values to get the dampers in your car right. We can now Run this scene and watch the improved motion of our car.

Now let's tackle the script associated with this scene: PlayerCar.js. There's two things in this script that will give the car better physics in this section. **1)** In the script we will now tweak the center of mass and inertia tensor so that the car has a lower center of mass (to prevent it from flipping over too easily). **2)** We remove the torque from the car because we'll modify the rotation of the wheels based on the input horizontal axis instead of torque.

When we attach this script to the player car, the four declared variables will have to be connected in the Inspector so that the script knows which variable is associated with which wheel (*shown at top left*). The script is shown in all it's gory detail on the next page:

Dynamic/Static Typing

Unity Uses Static Typing

Netscape's JavaScript implementation is a dynamically typed language. For example you can assign a number to a variable and later assign a string to it, for example:

```
i = 5;
i = "Hello World";
```

In most cases writing the above is an oversight in the script and not intentional. Statically typed languages usually require you to specify the type of the variable when declaring it like this:

```
var i : int = 5;
```

Unity's Javascript implementation, however, is statically typed but doesn't require you to declare the type yourself. How is this possible? Instead of having to specify the type every time, the type is automatically inferred when assigning a value to a variable. For example `i` is automatically inferred to the type `int` in this example: `i = 5;`

If you write:

```
i = 5;
i = "Hello World";
```

The compiler will give you an error because the types don't match. This is a big advantage of statically typed languages – they give you compile errors early on, telling you when you do something wrong.

Common Mistakes

Inferring the type can sometimes cause small errors that are hard to track down because you don't think about what the type was inferred to. For example, in the following script:

```
// These transforms need to be connected in the
// Inspector so the script can identify the wheels

var frontLeftWheel : Transform;
var frontRightWheel : Transform;
var backLeftWheel : Transform;
var backRightWheel : Transform;

// Speed is a multiplier of how much force
// we add to the wheels every frame
var speed = 150;

// The maximum steering angle of the wheels
var maxSteerAngle = 30;

// This is used to track if
private var hasBackWheelContact = false;

// Tweak the center of mass. You'd want a
// low center of mass, a bit towards the front
// of the model on a long but not very tall car

rigidbody.centerOfMass = Vector3 (0, 0, 0);
rigidbody.inertiaTensorRotation = Quaternion.identity;
rigidbody.inertiaTensor = Vector3 (1, 1, 2) * rigid-
body.mass;

function FixedUpdate () {
   var motorForce = speed * Input.GetAxis ("Vertical");

   // Do the wheels touch the ground?
   // Apply force along the z axis of the object
   if (hasBackWheelContact) {
     rigidbody.AddRelativeForce (0, 0, motorForce)
   }

   // The horizontal is in the range [-1 ... 1]
   // The maximum steer
   var rotation = Input.GetAxis ("Horizontal") * maxSteerAngle;

   // Set rotation around the y-axis of both wheels

   frontLeftWheel.localEulerAngles = Vector3 (0, rotation, 0);
   frontRightWheel.localEulerAngles = Vector3 (0, rotation, 0);

   // This tracks if the back wheels are grounded
   // Every frame we reset the variable
   // OnCollisionStay will then enable the variabl
   // if the wheel is grounded.
   hasBackWheelContact = false;

}
```

1

2

3

4

```
// countdown is inferred
to an integer value

var countdown = 5;

function Update () {

// countdown will never
decrease!!!

countdown -= Time.del-
taTime;

// When subtracing
// deltaTime from countdown
// deltaTime is rounded to
// an int, because
// deltaTime is normally
// less than 0.5 we will
// always subtract zero
}
```

Instead we have to make countdown a float:

```
// countdown is inferred
to be a float value
var countdown = 5.0;

function Update () {

// This works perfectly
fine, since countdown now
is a float.

countdown -= Time.del-
taTime;
}
```

```
// Called every frame if car collides with something.
// Used to calculate if the wheels touch the ground.
function OnCollisionStay (collision : Collision)
{
   for (var p : ContactPoint in collision.contacts)
   {

     // Enable hasBackWheelContact if we are
     // touching the ground

     if (p.thisCollider.transform == backLeftWheel)
         hasBackWheelContact = true;
     if (p.thisCollider.transform == backRightWheel)
         hasBackWheelContact = true;
   }
}
```


There are several important parts to this script that I will quickly review for the reader.

1. These are the four transform variables that we needed to connect to specific wheels in the Inspector panel as demonstrated on page 13.

2. This is where we change the player car's centre of mass based on our car model. In our case, the car is long and not too tall so we've moved the centre of mass slightly down and toward the front of the car. The center of mass is relative to the transform's origin. If you don't set the center of mass from a script like we are doing here, it will be calculated automatically from all colliders attached to the rigidbody. `InertiaTensorRotation` and `InertiaTensor` are the rotation of the inertia tensor and the diagonal inertia tensor of mass relative to the center of mass respectively. The inertia tensor is rotated by the inertiaTensorRotation.

3. This function checks to see if the car's back wheels are in contact with the ground/track surface, then applies a relative force along the car's z axis to move the car either forwards or backwards based on the player pushing the up or down arrows. This part also sets the angle of the front wheels turning based on the player pressing the left or right (horizontal) keys.

4 and 5. These parts of the script track if the wheels are still in contact with the ground. At the end of every frame we set `hasBackWheel-Contact` to `false`. `OnCollisionStay` is called every frame if any of the colliders of the car are colliding with other objects. Inside `OnCol-lisionStay` we check if the collider is a backwheel. If it has, we enable `hasBackWheelContact` again. This is all repeated every frame, thus `hasBackWheelContact` will track if the car is grounded or not.

Now that we have a fully functional player car, it's time for us to set up a camera to follow the player's progress in the race.

IV – The Game Camera

The goal of this shorter section is to create and manage a camera to follow the player's car as it races around the track. In a racing game, one of the most common camera views is from a third-person perspective; that is, behind and slightly above the player. This tutorial will show you how to create one of these cameras and then to move it along in a fluid manner with the player car. In more advanced games you might wish to set up numerous cameras to capture the action from different angles. This more complex camera work is done by enabling and disabling various cameras based on user input.

1. Basic Camera

Let's start by first simply getting the main camera positioned above and behind the player car. Open our previous scene's Physics.unity scene and do a Save as... in [B] Camera Control Scripts -> 1 Basic Follow Camera and name the file Camera. This scene contains all of the elements and scripts from the previous section of the tutorial. Now attach the Camera.js script to the main camera.

The Camera.js script has a distance and height variable that we will be able to modify from the Inspector, but also a "target" variable that we will need to assign to the player's car. Connect this variable to the player car object the same way we connected the wheel control variables to the wheel objects of the car object back on page 12.

```
// The target shows up in the inspector and has to
// be assigned to the object it should follow

var target : Transform;                                    1
var distance = 10.0;
var height = 10.0;

// LateUpdate is like Update but always called
// after all Update functions on all scripts
// are called.
// This allows you to order script execution
// in the same frame.
// Cameras should always be updated last.

function LateUpdate () {                                   2

   // Early out if we don't have a target
     if (!target)
     return;

   // Get the forward direction of the target
      forward = target.TransformDirection (Vector3.
      forward);                                            3

   // Get the target position
   targetPosition = target.position;
```

Lerp Function

What Is Unity's Lerp Function?

A Lerp function is simply an interpolation between two values. In the Smooth camera.js script we use it for both angles and height. The Lerp function can be used on many different types of values including colours, float, quaternion and vector3.

```
    // Place the camera distance behind the target
    transform.position = targetPosition - forward *
distance;                                                    4

    // And move the camera a bit up
    transform.position.y += height;

    // Always look at the target                             5
    transform.LookAt (target);
}
```

The Camera.js script is very simple.

1. We set up variables for height, distance and target.

2. We use the function LateUpdate in this script to make sure that this particular script runs after any other scripts that are run during a frame.

3. It gets the position and direction of the car to position itself.

4. It places itself a specific distance and height behind the target using variables. In this instance we've specified in the script that behind means along the negative z-axis of the target. Then we move the camera upwards by height, so the camera looks down on the car.

5. We rotate the camera to always look at the target. Run the scene and drive the player car to see the camera in action. Neat!

But this is a very simple solution. What we'd really like to see is the camera reacting better to the motion of the player car. We need to create and use a camera script that smoothes out the motion of the camera as it follows the car.

2. Smooth Camera

Start again by opening the Physics.unity scene and do a Save as... in [B] Camera Control Scripts -> 2 Smooth Camera and name the file Smooth Camera. This time, attach the Smooth Camera script to the main camera. Connect the target variable to the car object as we did in the previous scene and Run the scene to see the camera react more smoothly to the car's movement.

What's now happening is that this camera script smoothes out rotation around the y-axis and height, while still maintaining a static horizontal distance. This method gives you a lot of control over how the camera behaves because you can tweak lots of the variables. For every of those smoothed values we calculate a wanted value and the current value. Then we smooth it using the *Lerp* function.

Here is the Smooth Camera.js script in detail.

```
// The target we are following
var target : Transform;

// The distance in the x-z plane to the target
var distance = 10.0;

// Height we want the camera above the target
var height = 5.0;

// How much we want to dampen the camera's motion
var heightDamping = 2.0;
var rotationDamping = 3.0;

function LateUpdate () {

// Early out if we don't have a target
   if (!target)
   return;

// Calculate the current rotation angles
wantedRotationAngle = target.eulerAngles.y;
wantedHeight = target.position.y + height;

currentRotationAngle = transform.eulerAngles.y;
currentHeight = transform.position.y;

// Damp the rotation around the y-axis

currentRotationAngle = Mathf.LerpAngle (currentRo-
tationAngle, wantedRotationAngle, rotationDamping
* Time.deltaTime);

// Damp the height

currentHeight = Mathf.Lerp (currentHeight, wanted-
Height, heightDamping * Time.deltaTime);

// Convert the angle into a rotation. The
// quaternion interface uses radians not degrees
// so we need to convert from degrees to radians

currentRotation = Quaternion.EulerAngles (0, cur-
rentRotationAngle * Mathf.Deg2Rad, 0);

// Set the position of camera on the x-z plane to:
// distance meters behind the target

transform.position = target.position;
transform.position -= currentRotation * Vector3.
forward * distance;
```

1

2

3

4

5

```
// Set the height of the camera
transform.position.y = currentHeight;

// Always look at the target
transform.LookAt (target);
}
```

The Smooth Camera.js script is very simple.

1. We set up variables for height, distance and additionally for damping.

2. We calculate both current and wanted rotation and height for the camera.

3. We dampen the height and rotation by using the *Lerp* function (explained below).

4. We convert our rotation calculation from degrees into radians so that the Quaternion interface understands it.

5. In this part we finally position the camera where we want it and point the camera to always look at the target.

Run the scene and drive the player car to see the improved camera in action. Notice that we're using specifically the `Mathf.LerpAngle` to damp the rotation around the player car's vertical (y) axis, and using `Mathf.Lerp` to damp the height. It also uses some other basic functions built into Unity such as EulerAngles, Time.deltaTime and others. Most of the rest of the script uses basic variables and functions to move the camera with the car. It's now time to add opponent vehicles and program them to race around the track against the player. Otherwise known as the "cool stuff."

V – The Other Cars and AI

This section is where the fun begins. Up until this point we've been focusing on getting a car under the control of the player and also to set up a simple camera that will smoothly follow the car along a race track. But this is a "racing" game – we certainly need some other cars on the track that will race around the course trying to beat us. It will be the most difficult to date as the scripts in this section are more advanced than anything we've looked at so far, mainly because we need to create a method for opponent cars to drive themselves around the track. To begin the section we'll start by creating a car that drives itself around the race track. This car will be duplicated in the scene to create multiple opponent cars.

AI Cars

Start by opening the Physics.unity scene and do a Save as... in [C] Camera Control Scripts -> 1 AI Car and Waypoints and name the file Waypoints. Create a new Physic Material called AI Car Wheels and set it up as shown *at left*, then remove the previous wheel physic materials from the wheels in the Inspector by replacing it with our new AI Car Wheels material via a drag and drop.

Basic AI

How Basic AI Cars Function

AI cars generally work with a simple, two-step process like this: **1.** Calculate where we should drive towards; this is handled mainly by setting up "waypoints" and using the Waypoint script. **2.** Calculate how to get there. Suprisingly, with a car this is really simple – we just rotate the wheels towards the waypoint targets and drive forward.

Remove the Playercar script component from the AI Car in the Inspector and attach the AICar script, making sure to connect the four wheel variables to their associated wheel objects. Finally, rename the player car to AI Car. Save the scene at this point. I'll explain how to create waypoints after we we examine our AICar script in detail.

In our AICar.js script we have the `UpdateWithTargetPosition` function which is the meat of the AICarScript. It rotates the wheels, accelerates forward and slows down in sharp turns. The `UpdateWithTargetPosition` function is called from inside `FixedUpdate`. In `FixedUpdate` we also calculate where we should drive towards, and calculating where we drive towards is simple and goes like this:

We have a current waypoint as a target and we switch to the next waypoint whenever we enter our current waypoint trigger. So we simply drive an AI-controlled car towards the current waypoint and if we get *close* to the waypoint, we let the car drive towards the waypoint after that. This improves the AI car quite a bit because the car will otherwise have to drive exactly through the waypoint and will do sharp turns shortly before hitting the waypoint trigger.

With the basic in mind, we can now examine the AICar.js script functions in detail, like we did with some of our previous scripts.

```
// Variables defined outside of functions appear
// in the inspector and can be changed there
// without having to change the script
var frontLeftWheel : Transform;
var frontRightWheel : Transform;
var backLeftWheel : Transform;
var backRightWheel : Transform;
var wheelForce = 120.0;

// All these variables are only used internally,
// thus we make them private
private var hasWheelContact = false;
private var steerMaxAngle =  40.0;
private var activeWayPoint : WayPoint;

function Start () {
   // Initialize the waypoint we drive towards!
   activeWayPoint = WayPoint.start;

   // Tweak the center of mass.
   // - Low center of mass a bit towards the front
   // - model a long long and not very high car
   rigidbody.centerOfMass = Vector3 (0, 0, 0);
   rigidbody.inertiaTensorRotation = Quaternion.identity;
   rigidbody.inertiaTensor = Vector3 (1, 1, 2) * rigid-
   body.mass;
}
```

1

```
function UpdateWithTargetPosition (target : Vector3) {

// Calculate the target position relative to the
// target this transforms coordinate system.
// eg. a positive x value means the target is to
// to the right of the car, a positive z means
// the target is in front of the car
relativeTarget = transform.InverseTransformPoint
(target);

// Calculate the target angle for the wheels,
// so they point towards the target
targetAngle = Mathf.Atan2 (relativeTarget.x, rela-
tiveTarget.z);

// Atan returns the angle in radians, convert to degrees
targetAngle *= Mathf.Rad2Deg;

// The wheels should have a maximum rotation angle
targetAngle = Mathf.Clamp (targetAngle, -steerMax-
Angle, steerMaxAngle);

// Apply the rotation to the wheels
// We want the wheels to rotate around the y-axis
// The rotation has to be relative to the car,
// which is the transform parent of the wheels
frontLeftWheel.localEulerAngles = Vector3 (0, tar-
getAngle, 0);
frontRightWheel.localEulerAngles = Vector3 (0,
targetAngle, 0);

rigidbody.drag = 0;
if (hasWheelContact)
{
   // Accelerate ...
   // force = maxSpeed * force;
   rigidbody.AddRelativeForce (0, 0, wheelForce);

   // Too fast? Need to turn too much? Slow down!
   if (Mathf.Abs (targetAngle) > 15 && rigidbody.
   velocity.magnitude > 10) {

     // We are too fast
     rigidbody.drag = 4;
     }
}
```

2A

2B

Debugging

Debugging Scripts In Unity

Unity provides a couple of useful tools to debug scripts, for example, the Debug.Log function.

```
function Update () {

Debug.Log ("Inside Update", gameObject);

}
```

When you run this script, "Inside Update" will popup in the status bar; when you single click on it, it will popup the console and show the connection to the gameObject we passed as the second parameter of Debug.Log.

This can be used for any reference to another Object. It is extremely helpful when you have some bug where you don't know exactly in which object the problem occurrs.

Another very useful tool is Debug.DrawLine. It will simply draw a line. I have used this Debug.DrawLine when developing the AI car for example.

It was very useful to show:
1) where the car calculated it wants to drive to; and, 2) where the wheels where actually pointing towards

```
function Update() {

Debug.DrawLine (transform.position, Vector3.forward, Color.green);

}
```

Also if you don't know already, private variables can be viewed but not edited in the Inspector by clicking on the Expert button.

```
   // This is handy for debug visualizing where
   // we actually want to drive
   // Debug.DrawLine (transform.position, target);

   // This is reset every frame.
   // OnCollisionStay enables it again.
   hasWheelContact = false;
}

function FixedUpdate () {
   // Calculate position the AI car should drive towards
   targetPosition = activeWayPoint.CalculateTargetPosition (transform.position);

   // Apply forces, steer the wheels
   UpdateWithTargetPosition (targetPosition);
}

// Whenever we hit a waypoint we have to
// skip forward to the next way point
function OnTriggerEnter (triggerWaypoint : Collider) {
   if (activeWayPoint.collider == triggerWaypoint) {
       activeWayPoint = activeWayPoint.next;
   }
}

// Track if we the wheels are grounded

function OnCollisionStay (collision : Collision) {
   for (var p : ContactPoint in collision.contacts) {
     if (p.thisCollider.transform == frontLeftWheel)
           hasWheelContact = true;
     if (p.thisCollider.transform == frontRightWheel)
           hasWheelContact = true;
     if (p.thisCollider.transform == backLeftWheel)
           hasWheelContact = true;
     if (p.thisCollider.transform == backRightWheel)
           hasWheelContact = true;
   }
}
```

3

4

5

There are several important functions of our AICar script that I will review in more detail for the reader now.

1. `Start`. The Start function initializes the center of mass like we did on page 15. And also intializes the `activeWaypoint` to the start waypoint. The active waypoint is used in `FixedUpdate` to find where we want to drive towards.

2A. `UpdateWithTargetPosition`. Rotate the wheels towards the target. We use `transform.InverseTransformPoint` to get the position relative to the local coordinate system. Then we calculate the angle using `Atan2`. We clamp the angle, because in real life wheels

have a maximum steering angle. Then we apply the rotation to the wheel transforms.

2B. If the wheel has contact (also previously discussed on page 15) we move the car forward. But that makes the car drive over the edge way too much, so we slow down if we have a large turning angle and are driving fast. We slow the AI car down by simply giving the rigidbody a high drag value. This is obviously not a very *realistic* way of doing it, but it is very **easy** and **effective** for our purposes.

3. `FixedUpdate`. In `FixedUpdate` we talk to the waypoint system to calculate the target position the car should drive towards this frame. Then we pass the target position onto the `UpdateWithTargetPosition` function, which carries out the movement of the car.

4. `OnTriggerEnter`. In `OnTriggerEnter` we register when we hit a new waypoint. When we hit a waypoint we first check if it is the next waypoint – so that the player can't skip one waypoint or go backwards – then make the active waypoint the next waypoint.

5. `OnCollisionStay`. This function is the same as shown in points 4 and 5 on page 15 that calculates wheels being grounded.

With the AI car now functioning, it's time to look at setting up and programming waypoints around our race track for the cars to drive towards.

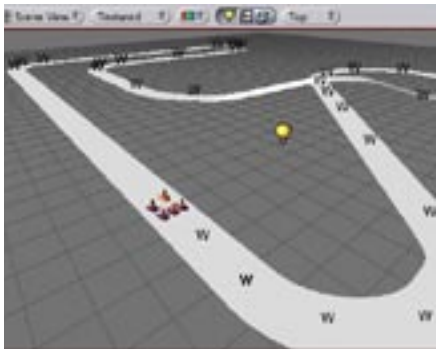
Waypoints

To set up waypoints you add an empty game object to our scene, add a box collider to it, and set the isTrigger property to true, then attach the WayPoint script to it (this also draws the W texture linked in the Gizmos folder). The waypoints need to be manually connected by setting the next variable of every waypoint. The screenshot *at left* shows our completed track with 30 waypoints set up in the Scene view. The first waypoint on the track should be named "Start" in the Inspector; the others should be called "Waypoint".

The waypoint script is used by the AI cars and is rather simple – the scripts contain a varible to the next waypoint. This variable needs to be setup manually in the Inspector for every waypoint. Then there is a `CalculateTargetPosition` function which calculates where the car should drive towards. Here's the script:

```
// The start waypoint, this is initialized in Awake.
// This variable is static thus all instances
// of the waypoint script share it.
static var start : WayPoint;

// The next waypoint, this variable needs to be
// assigned in the inspector.
// You can select all waypoints to see the
// full waypoint path.
var next : WayPoint;
```



Prefabs

A Perfect Use For 30(!) Waypoints

I'd strongly suggest after setting up one waypoint with it's box collider set up and the Waypoint.js script attached to create a prefab waypoint object from it. Drag more prefab waypoints into your scene to create the rest of the waypoints. That way, any changes made to the prefab will be reflected in all of the waypoints created using the prefab object.

I'd also recommend creating an empty game object that holds all of the waypoints you'll want to create in one scene. The finished version of the tutorial uses 30 waypoints.

Static Variables

Static Variables Are Shared Variables

What if we want to share (or not share) a variable between all "instances" of the script? Sometimes you **want** a variable to be shared between all instances of the script (e.g., is a player dead, setting a standard reset value for a timer, etc). An instance of an script occurs when you attach a script to a game object. The script with its value shows up in the inspector. All variables become instanced and can be modified for this particular instance. You use the static keyword for this as shown in the following code example:

```
static var count = 0;

function Awake () {
   count++;
   print (count);
}
```

If you attach this script to more than one game object, count will increase for every one of them. So if you attach it to 3 game objects and hit play, it will print 1, 2, and 3 to the console. If you leave out the static keyword (i.e., `var count = 0;`) it will print 1, 1 and 1.

```
// This determines where the start waypoint is.
var isStart = false;

// Returns where the AI should drive towards.
// position is the current position of the car.
function CalculateTargetPosition (position : Vector3) {

   // If we are getting close to the waypoint,
   // we return the next waypoint.
   // This gives us better car behaviour when
   // cars don't exactly hit the waypoint

   if (Vector3.Distance (transform.position, posi-
tion) < 6) {
     return next.transform.position;
   }

   // We are still far away from the next waypoint,
   // just return the waypoints position

   else {
     return transform.position;
   }
}

// This initializes the start and goal static variables.
// We have to inside Awake because the waypoints need
// to be initialized before the AI scripts use it
// All Awake function are always called before all
// Start functions.
function Awake () {
   if (!next)

      Debug.Log ("This waypoint is not connected,
      you need to set the next waypoint!", this);

   if (isStart)
       start = this;
}

// Draw the waypoint pickable gizmo
function OnDrawGizmos () {
   Gizmos.DrawIcon (transform.position, "Waypoint.tif");
}

// Draw the waypoint lines only when you select
// one of the waypoints
function OnDrawGizmosSelected () {
   if (next) {
     Gizmos.color = Color.green;
     Gizmos.DrawLine (transform.position, next.trans-
form.position);
   }
}
```

1

2

3

4

In this script we are using something called a static variable for the first time. The static start variable is of type waypoint. On `Awake`, we set the start waypoint to the waypoint, which has the isStart flag enabled.

Static variables are shared between all instances of the script and can also be accessed by other scripts by using the name of the script followed by a period and the function or variable name, as shown in this example:

SomeScript.js has the following static variable in it:

```
static var staticVar = 5;
```

Another script called YetAnotherScript.js has this:

```
print (SomeScript.staticVar);
// will print 5
```

The static `start` variable was used by the car script to find the first waypoint like this: `activeWayPoint = WayPoint.start;`. Also the game controller in the next chapter will use it to find out when we cross the finish line. Let's look at the functions in the waypoint script.

1. `CalculateTargetPosition`. This function simply takes the current position of the car and returns the point that the car should drive towards. This returns either the position of the waypoint, or the next waypoint beyond that one if we are close to the waypoint already. This is important so that the car doesn't drive exactly to the center of the waypoint when it doesn't exactly hit a waypoint in the middle.

2. `Awake`. This function initializes the start variable to the waypoint which has the isStart flag set in the inspector. We also do error checking here to check if all the next variables are setup.

3. `OnDrawGizmos`. This draws the waypoint icon so waypoints are easily pickable. Specific information about Gizmos can be found on OTEE's scripting reference site.

4. `OnDrawGizmoSelected`. This function neatly draws the line connecting one waypoint to the next waypoint. You can select all waypoints to visualize any setup errors that might exist in your track in the Scene view.

Now that we have the script written, we can actually go about setting up waypoints on our track, placing a test car in the scene, and attaching the AICar script to it to watch it drive itself around the track. I've shown the track from the completed files below. In other words, to set things up, you simply need to place a few waypoints in the scene – remember, one of them has to have the isStart flag

Future Unity Feature

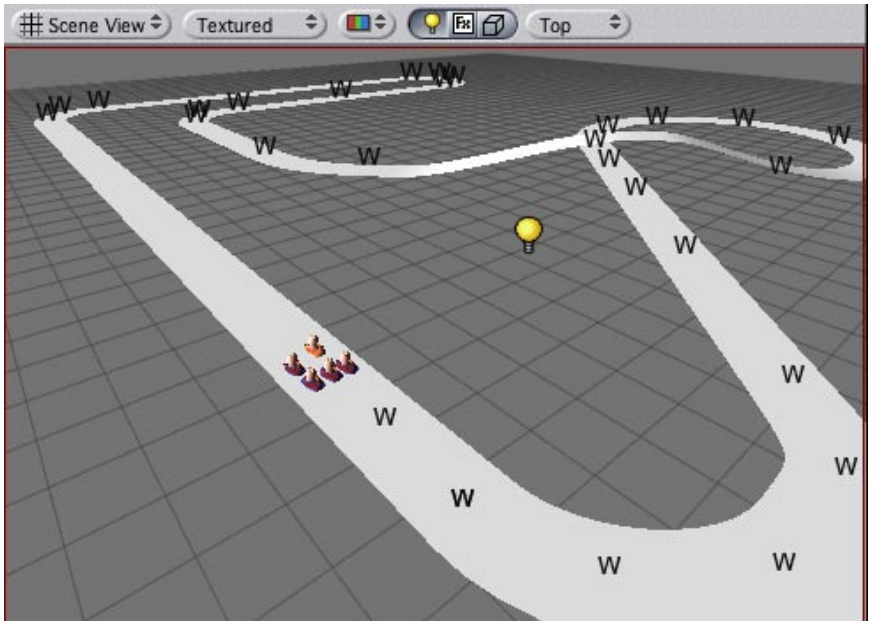
Dedicated Wheel Components?

Version 1.2 of Unity should introduce dedicated wheel components that will allow for very realistic cars.

enabled, so we know where to start. All of them have to be connected together by setting the next property in the Inspector of each waypoint. This scene uses a total of 30 waypoints.

Create duplicates of the AI Car we assembled on page 19 and place them in the scene. Now Run the scene. While the scene Runs, our AI cars will simply drive towards the start waypoint. When a car hits it, the car will start driving towards the next waypoint in turn, and then keep driving from waypoint to waypoint around the track.

To get ready for the final chapter, Save this scene in [D] Game Setup -> 1 Finishing the Racing level and name it Track1.unity.



VI – Start Your Engines!

This section is where we finish the game by adding some necessary things to polish the game. These include things commonly found in many genres of games and what we learn here will certainly be applicable to your own games in the future. They include things like (race) timer, the game's graphic user interface (GUI), text objects, a countdown timer, and simple scene transitions. Let's begin by finishing the racing part of the tutorial in a basic way.

1. Finishing The Racing Level

Start by opening the Track1.unity file we saved above. This should contain four copies of the AI opponent car and our finished player car from the Physics.unity scene. We need to add a script to all of the cars, both player and AI, that tracks the cars as they drive through the correct waypoints (CarStats.js). And we need to create a simple text object that will display a countdown to begin the race by using a simple script (StartGame.js). Let's begin by creating the GUI object.

Create a text object by selecting GameObject -> Create Other -> Text in the menubar and position and scale it so it is in the centre of the screen. You can leave the default text it is displaying or delete it to leave it blank, in the Inspector, once you are happy with it's position and scale. Attach the StartGame.js script to the text object.

The camera will always display the Text object in front of all other objects in the game because the camera contains a GUILayer component. The camera's GUILayer displays objects that have a GUIText component and GUITextures always in the front specifically to make interfaces for games. Text objects have a GUIText component included, visible in the Inspector panel by default. GUI objects remain in the same position in relation to the camera even if we move the camera because they are, for al intents and purposes, attached to the camera.

The StartGame.js script is used to temporarily disable all the cars in the race until a "ready, set, go" series of words flashes on the screen for the player, then enables all the cars so that they can begin racing. It then will "hide" the text object that displays the words. This script also introduces a simple, but very useful kind of function called coroutines.

Coroutines are functions that can be paused at any point and wait for a specific event to occur, after which they should continue to the next instruction. For example, while a function was running, we could wait for the next frame or wait for 5 seconds. Coroutines make your life a lot easier when dealing with a sequence of events. For example, if you want to display some text for 2 seconds and then want to remove it again you can do it by simply attaching the following script to an object in your scene; assuming the object has a GuiText component:

```
guiText.text = "Hello";
yield WaitForSeconds (2);
guiText.text = "";
```

For more information about coroutines also see documentation about the yield statement can be found on the OTEE scripting reference documentation website.

Now that we understand basic coroutines the StartGame.js script, shown in detail on the following page, will be very easy to understand.

```
function Start ()
{

   // Disable AI Cars
   aiCars = FindObjectsOfType (AICar);
   for (var car : AICar in aiCars)
      car.enabled = false;

   // Disable Player Cars
   playerCars = FindObjectsOfType (PlayerCar);
   for (var car : PlayerCar in playerCars)
      car.enabled = false;

   // Display get ready, wait
   guiText.text = "Get Ready";
   yield WaitForSeconds (.75);

   // Display Set, wait
   guiText.text = "Set";
   yield WaitForSeconds (.75);

   // Display go and enable all cars
   guiText.text = "Go";
   aiCars = FindObjectsOfType (AICar);
   for (var car : AICar in aiCars)
      car.enabled = true;

   // Enable Player Cars
   playerCars = FindObjectsOfType (PlayerCar);
   for (var car : PlayerCar in playerCars)
      car.enabled = true;

   // Wait and Hide text
   yield WaitForSeconds (.75);
   guiText.text = "";

}
```

1

2

3

4

1. We "disable" all the cars in the race, including the AI cars and the player's car. We find all the AI cars by calling `FindObjectsOfType`. This returns a list of all active loaded objects of Type *type*. It will return no assets such as meshes, textures, prefabs or inactive objects. More information about calling FindObjectsOfType can be found online. We give it the type of the class we are looking for, i.e., the AICar, and get back all instances of that script. We go through all the script instances we found and set enabled to false. So what happens if a script is enabled or disabled?

The answer is simple: the `Update` function and `FixedUpdate` function is **only** called when the script is enabled. By disabling the update function we prevent the car from being able to drive.

The enabled checkbox is also visible for an object or its components in the Inspector (the small checkbox next to the title).

2. Then we display the words "Get Ready" in our Text object and then wait for 3/4 of a second using yield. Then we display the word "Set" and wait for 3/4 of a second using yield.

3. Then we display the word "Go" and enable all car scripts similar to how we disabled them.

4. The final step is to wait for another 3/4 second and then remove the Text object, in a manner of speaking, by putting nothing ("") in the GUIText, and then the race can begin.

But *before* the cars will actually go, we have to make sure to have attached the CarStats.js script to **all** of the cars (player car and AI cars). Make sure that has been done, and then save the current scene so we can examine that script.

The CarStats script simply tracks that we drive through the correct waypoints. When we hit the start waypoint, it will disable the car like we did in the StartGame.js script, since the car has completed the race and it no longer should drive around the track. If it is the player's car that has completed the race, then we'll need to tell the GameController about it. This "GameController" will be introduced in 2. GameController and handle a number of tasks in our racing game such as showing the highscore etc. Here's the CarStats script in detail:

```
private var activeWayPoint : WayPoint;

function Start () {
   activeWayPoint = WayPoint.start.next;
}

// Keeps track of when the player reaches the goal

function OnTriggerEnter (triggerWaypoint : Collider) {

   // We allow the player to go through the waypoints
   // only one after another so he can't skip any

   if (activeWayPoint.collider == triggerWaypoint) {
     // When we reach the game might be finished!
     if (activeWayPoint == WayPoint.start)
           ReachedGoal();
     activeWayPoint = activeWayPoint.next;
   }
}
```

1

2

```
function ReachedGoal () {

   // Stop driving if an AI car has completed the race!
   var aicar : AICar = GetComponent(AICar);
   if (aicar != null)
      aicar.enabled = false;

   // The player has finished
   var playerCar : PlayerCar = GetComponent(PlayerCar);
   if (playerCar != null) {
      // Stop driving
      playerCar.enabled = false;

     // Tell game controller that race is complete
     var controller = FindObjectOfType(GameController);
     if (controller)
           controller.SendMessage("CompletedRace");
   }
}
```


1. On `Start` we set the activeWaypoint to the waypoint after the start so that we don't complete the game immediately. We discussed this also in the previous chapter of the tutorial.

2. In `OnTriggerEnter` we register when we hit a new waypoint. When we hit a waypoint we first check if it is the next waypoint so that the player can't skip one waypoint or go backwards, and then make the active waypoint the next waypoint. When we hit the `startWaypoint` we have completed the game and call the `ReachedGoal` function.

3. The `ReachedGoal` function uses something called GetComponent. This returns the component of Type *type* if the game object has one attached, null if it doesn't. More information about accessing other components can be found in the Script Reference and the Get Component information online.

Remember that the CarStats script was attached to the PlayerCar and AICar? We will need to handle these two cases differently because there are two different results if: a) we have a player finishing the race, or b) if we have an AI car finishing the race. If the AI car crosses over the start line, it should simply be disabled so it stops driving. If the player crosses the start line it should be also be disabled but we also need to tell the GameController about it so it can display highscore etc. So we use `GetComponent(AICar);`.

This will return `null` if there is no AICar attached to the game object. In that case we simply ignore it. If we do have an AICar then we disable it. We do the same check for the `PlayerCar` and then disable the player car and send a message to the GameController.

Think Of It As ...

GameControllers Are Like Includes

Anyone familiar with web design and coding might recognize in this "GameController" object many aspects of something known as a server-side include ... and they'd be right.

An include is a bit of code on a website that exists on numerous pages, i.e., like a right column menu. Since it's the same on all of the pages, it makes sense from a website maintenance standpoint to create a single version of the right column menu on its own, and then use a single line of code on all of the pages to bring it (Include it) in the page when the page loads. That way, when you need to change the right column menu on 50 pages, you only need to change it in one file and then all of the pages are updated.

A GameController object that is set to not be destroyed when a different scene loads can be used for a myriad of purposes like this, as the GameController script illustrates.

2. The GameController

We've mentioned this thing called "GameController" a great deal in the last few pages. What we want to create is an object that will exist throughout the entire game, in all of its scenes, and act as a central place that we can use to, well, control aspects of our game.

Create a new scene (File -> New) in [D] Game Setup -> 2 GameController and save it as GameStartup . In this scene create a GUIText object displaying the name of our game; i.e., Racing Car Tutorial. Now create another new scene in the same asset directory called MainMenu. There are two scripts in the [D] Game Setup -> 2 GameController folder; a Button script (*see page 34*) and the game controller. The button script implements a simple mouse over effect and simply forwards mouse clicks to the GameController. Reopen the GameController scene and create an empty object in the scene and attach the GamerController.js script to it. Let's take a quick look at the first part of the script.

```
function Start () {

   // Make sure that the gamecontroller
   // always survives level loads
   DontDestroyOnLoad (this);

   // Wait until any key is pressed
   while (!Input.anyKeyDown)
      yield;

   // Load the main menu
   Application.LoadLevel ("MainMenu");
}
```

This first part of the script does three simple things: it makes sure the GameController script and the object it's attached to are going to be available in every scene; it waits until a key is pressed; and then it starts our game by loading the MainMenu scene.

The GameController is scripted to be not destroyed when loading a new level. This is done using the function call: `DontDestroyOnLoad (this);` in the `start` function of GameController. Why do we need this? Because, normally, when a new level is loaded, all game objects from the previously loaded level are destroyed. `DontDestroyOnLoad` allows you to prevent this by calling the function on the objects you don't want to be destroyed.

This is very useful in many programs: for example, to keep playing music while loading a level, or when you need to maintain some state from one level to another level. And since GameController is the only game object we keep alive when loading levels, this makes it the ideal place to perform **all** the menu commands; for example,

`DisplayHighscore ()`, which loads the highscore level, waits until user pressed a button, then loads the main menu again. This functionality is only possible if the object is marked `DontDestroyOnLoad`.

What's going to happen with the GameController script now is this: When a GUI button is pressed in the MainMenu scene, we find the GameController instance and send it a message. The message sent can be changed in the Inspector, however, so when we add new menu commands that are associated with new buttons in our scene, we don't need to modify the Button script. Instead we change the *action string* in the Inspector and add a new function to GameController.

A GameController that never gets destroyed when moving from scene to scene is the ideal way to handle **all** the menu commands in a game because we implement the menu commands in it, rather than individual Button scripts for every menu command. As I mentioned earlier, the button script can then simply be used to create a simple mouse over effect and forward mouse clicks to the GameController. Here's the rest of the GameController script:

```
// - Display highscore and let user enter his name
// - keep displaying until user pressed any key
// - Display the main menu again

function CompletedRace () {
   // How long did it take the player to finish
   // The race starts only 0.75 seconds
   // after the level is loaded
   finishTime = Time.timeSinceLevelLoad - 0.75;

   // Display "Race completed".
   // We reuse the gui text from the
   // Start game display script.
   FindObjectOfType (StartGame).guiText.text =
   "Race Complete!";

   yield WaitForSeconds (2);

   // Load the high score scene
   Application.LoadLevel ("HighScore");

   // Wait one frame, because LoadLevel is
   // delayed until the end of the frame.
   yield;

   // Wait until user has entered the highscore
   var highscore = FindObjectOfType (HighscoreTa-
ble);
   yield highscore.StartCoroutine ("EnterHigh-
Score", finishTime);
```

1

```
   // Go back to the main menu
   Application.LoadLevel ("MainMenu");
}

// Shows the high score and hides it again
// after the user pressed a key
// Display high score is called from the main menu
// when the Highscore text is clicked.
function DisplayHighscore () {

   // Load the high score scene
   Application.LoadLevel ("HighScore");

   // Wait until no key is pressed anymore
   while (Input.anyKey)
      yield;

   // Wait until any key is pressed
   while (!Input.anyKey)
      yield;

   // Go back to the main menu
   Application.LoadLevel ("MainMenu");
}

function Quit () {
   Application.Quit ();
}

function NewGame () {
   Application.LoadLevel ("Track1");
}
```

2

3

1. `CompletedRace` is called from the CarStats script when the player has finished the race and displays a familiar "race over" message to the player. The game then waits a bit and loads the highscore. The calculated highscore is the time it took the player to complete the race. We call `EnterHighScore` with the score and wait for `EnterHighScore` to complete using yield. `EnterHighscore` will complete once the user has finished entering his name. Then we go back to the main menu.

2. `DisplayHighscore` is called when Highscore button is clicked. The High Score button referred to in the script is actually located in the MainMenu scene (not GameStartup).

3. `Quit` simply quits the application and the `NewGame` button loads the first track, which in turn starts our game.

Save the scene and reopen the [D] Game Setup -> 2 GameController -> MainMenu scene now. Create three text objects (we'll use

as buttons) named "HighScore", "Quit" and "New Game" in the centre of the screen. Attach the single Button script to these objects. Let's take a look at this simple script:

```
var action = "The name of the action";

// Change the color of the text when the mouse enters

function OnMouseEnter () {
   if (audio)
      audio.Play ();
   guiText.material.color =
   Color.yellow;
}

// Change the color of the text back to white
// when the mouse exits

function OnMouseExit () {
   guiText.material.color =
   Color.white;
}

// We simply forward all mouse down events
// to the GameController.

// The function that will be called is defined
// by the action string.

// The action string is setup in the inspector.

function OnMouseDown () {

   var controller : GameController = FindObjec-
tOfType (GameController);

   controller.SendMessage
  (action);
}
```

1

2

3

1. This Button script simply changes the colour of the Text GUI object when it's rolled over and plays a sound if there is an audio source attached to the button.

2. It changes the text colour back to white when the cursor exits the button object.

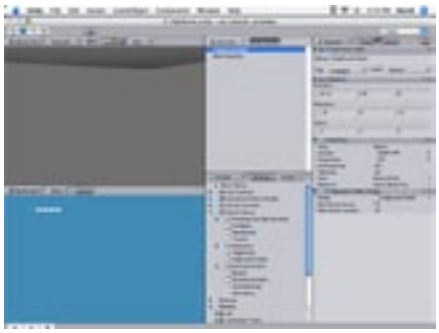
3. And finally, when the text object is clicked, it send a message to the GameController object so that a specific function is run.

This, by the way, is an example of why we used a script and object that contain `DontDestroyOnLoad` in the GameStartup scene. The GameController object will continue to exist in the MainMenu scene after we've loaded this scene from the GameStartup scene.

The function that is being called in the GameController script is the string called "action" in the Inspector. The Button script is passing a message to the GameController script so that the script knows which function to run. So in this case, `DisplayHighscore` loads the highscore, then waits for the user to press a key, then goes back to the main menu. All of the buttons in the MainMenu scene work in the same way and that is why we needed only one Button script.

Technically, the MainMenu scene doesn't do anything on its own. It only forwards messages to the GameController. Because the game controller is stored our games splash/opening scene that we called Gamestartup, you will need to open the GameStartup scene and hit Play from that scene to see how our game will work, rather than opening the MainMenu scene, which on its own, does not contain the GameController script.

You will find that this technique will make creating your game GUIs much easier. By separating the splash screen from the game options screen, and putting a game controller-type script with a `DontDestroyOnLoad` function in it, in the same scene as the splash screen, we don't end up with multiple instances of a game controller for each time we load (return to) the main menu level of our game.



3. High Scores

We're almost done with the racing game tutorial and only need to create a high score display that will record the top 10 player's names and times. Unfortunately, this little requirement happens to be the *hardest* part of the tutorial. It was left until the end so that you'd have well-grounded introduction into programming before tackling it. But, no racing game would be complete without a way for the player to record his name after the race. One of the many rewards players get out of both causal and hardcore computer games is when they finally get to see their names listed in a game's top scores. And we wouldn't want to leave that out, would we?

Create an new scene in [D] Game Setup -> 3 Highscore and call it HighScore. Create a text object in the scene called HighScoreTable and attach the script of the same name to it. The HighScoreTable.js is a complex script fully shown on the following three pages. **Take your time** and review it function by function. Now that you've come this far in the tutorial, you'll be pleasantly surprised at how much you'll understand. Once again, I'll examine and explain the key parts of the script after you've read through it.

```
// This stores the score and name of the players
class Entry {
   var score = 0.0;
   var name = "";
}
private var entries = ArrayList ();

// How many high score entries are shown?
var maxEntryCount = 10;
// How long is the player's name allowed to be?
var maxNameLength = 10;

function Awake () {
   // Load the entries from prefs
   LoadEntries ();
   // And display high score table text.
   SetupHighscoreTableText ();
}


// This is a coroutine which runs until the user has entered his name.
// To enter a new highscore in the table do like this:
// highscoreTable.StartCoroutine ("EnterHighScore", 10);
function EnterHighScore (score : float) {

   // Setup the highscore table text
   SetupHighscoreTableText ();

   // Insert the entry, it might get rejected if the score is not high enough
   var entryIndex = InsertEntry (score);
   if (entryIndex == -1)
     return;

   // Check for the last name the user entered and reuse it
   var inputName = PlayerPrefs.GetString ("LastHighscoreName");

   while (true) {
     for (var c : char in Input.inputString) {
           // Backspace - Remove the last character
           if (c == "\b"[0]) {
                 if (inputName.Length != 0)
                       inputName = inputName.Substring(0, inputName.Length - 1);
           }

           // End of entry.
           else if (c == "\n"[0]) {
                 // But the user must have at least entered something
                 if (inputName.Length)
                 {
                       ChangeName (entryIndex, inputName);
                       SaveEntries ();
                       // Store the name the user entered as the last high score name,
                       // so next time the user doesn't have to enter it again
```

1

2

```
                        PlayerPrefs.SetString ("LastHighscoreName", inputName);
                        return;
                  }
            }
            // Normal text - just append
            else {
                  inputName += c;
            }
      }

      // Make sure the name doesn't grow above max entry length
      if (inputName.Length > maxNameLength)
            inputName = inputName.Substring (0, maxNameLength);

      // Add a "." as a blinking text marker.
      // Show the "." every .5 seconds
      blinkingName = inputName;
      var time = Mathf.Repeat (Time.time, 1.0);
      if (time > .5)
            blinkingName = inputName + ".";
      else
            blinkingName = inputName;

      // Change the name
      ChangeName (entryIndex, blinkingName);

      yield;
   }
}

// Insert a new entry
function InsertEntry (score : float) : int {
   entry = Entry ();
   entry.score = score;

   // In
   for (var i=0;i<entries.Count;i++) {
      if (entry.score < entries[i].score || entries[i].score == 0.0) {
            entries.Insert (i, entry);
            break;
      }
   }

   // Remove excess entries
   if (entries.Count > maxEntryCount)
      entries.RemoveRange (maxEntryCount, entries.Count - maxEntryCount);
```

```
// We changed the high score table, so we need to rebuild
   // the text we rneder
   SetupHighscoreTableText ();

   return entries.IndexOf (entry);
}

// Changes the name of the entry at index
function ChangeName (index : int, name : String) {
   var entry = entries[index];
   entry.name = name;
   SetupHighscoreTableText ();
}


// Loads all entries from the preferences
// Sorts them and removes excess high score entries
function LoadEntries ()
{
   entries.Clear ();
   // Load entries from preferences
   for (var i=0;i<maxEntryCount;i++)
   {
     var entry = Entry ();
     entry.name = PlayerPrefs.GetString ("HighScore Name " + i);
     entry.score = PlayerPrefs.GetFloat ("HighScore Score " + i);
     if (entry.score != 0)
           entries.Add (entry);
   }

   // Add empty entries to fill up the remaining highscore entries
   while (entries.Count < maxEntryCount) {
     entry = Entry ();
     entry.name = "Empty";
     entry.score = 0;
     entries.Add (entry);
   }

   if (entries.Count > maxEntryCount)
     entries.RemoveRange (maxEntryCount, entries.Count - maxEntryCount);
}

// Saves all high score entries to the preferences
function SaveEntries () {
   for (var i=0;i<entries.Count;i++) {
     PlayerPrefs.SetString ("HighScore Name " + i, entries[i].name);
     PlayerPrefs.SetFloat ("HighScore Score " + i, entries[i].score);
   }
}
```

4

5

6

```
function SetupHighscoreTableText () {
   text = "";
   count = 0;
   // Loop through all entries
   for (var entry : Entry in entries) {
     // Create one line of the entry text
     text += entry.name + "\t" + FormatScore (entry.score) + "\n";
     count++;
   }

   guiText.text = text;
}

function WipeoutPrefs () {
   for (var i=0;i<maxEntryCount;i++) {
     PlayerPrefs.SetString ("HighScore Name " + i, "");
     PlayerPrefs.SetFloat ("HighScore Score " + i, 0);
   }
   Awake ();
}

// We format score like a count down timer
function FormatScore (time : float) : String {
   var intTime : int = time;
   var minutes : int = intTime / 60;
   var seconds : int = intTime % 60;
   var fraction : int = time * 10;
   fraction = fraction % 10;

   // Build string with format 12[minutes]:34[seconds]
   var timeText : String = minutes + ":";

   if (seconds < 10)
     timeText = timeText + "0" + seconds;
   else
     timeText = timeText + seconds;
   timeText += "." + fraction;

   return timeText;
}
```

7

8

9

While it may seem complex at first glance, this whole script is actually only doing a small handful of tasks. These tasks include: setting up an array (think "table") and populating it with the current high scores; recording a time and allowing the player to enter his or her name; automatically adding or removing entries after a race and then rebuilding the array; saving the scores to the game's preferences; adding the ability to clear the high scores; and making sure the score is correctly formatted. The reason the script is so long is that some of these tasks are made up of a number of subtasks and functions. Let's go through the function in the script in details now.

1. In this part of the script we create a class that contains two variables named `score` and `name`. We will add these two variables to the `entries` array. For those readers who have never heard the term before, an *array* is simply a list, and is one of the most basic data structures in computer programming. Arrays hold some number of data elements, generally of the same data type. `Awake` is called when the script is first loaded. In this script, we call two functions: `LoadEntries`., which reads the entries from prefs; and `SetupHighscoreTableText`, which builds a string out of the highscore list and assigns it to the HighScoreTable text object we created in the scene.

2. OK, the `EnterHighScore` function is a big one but is made up of understandable sections. `EnterHighScore` is simply a coroutine (see page 27), which runs until the user has entered his or her name in the high score table. It's made up of several discrete steps. First we try to insert an highscore entry using `InsertEntry`. If the player raced to a score that did not make it into the top 10 best scores, we exit the coroutine at this point.

Then we loop waiting one frame between every iteration. A program loop is something that receives an event, handles the event, and then waits for the next event. An event loop usually does not end until the conditions set up in the loop are met. In our example, we do this using a `while` loop with a `yield` statement at the end. In the loop we update the user typed string. If the user hits enter or return ("\n") we simply update the highscore name one last time, write out the highscore table to the prefs, and then exit.

The last part of this section of script simply allows us to show a blinking dot if the user hasn't completed entering his name yet. We do this by adding a "." to the entered user string the first half of every second.

3. Insert Entry. `InsertEntry` enters an entry and returns the index or position. If the player's score is too low we return -1. We find the right place to put the player's score entry by looping through the highscore list. As soon as we find something with a higher score we `Insert` the score and break. Then we need to `Remove` any highscores that dropped out because they are not in the top 10 scores anymore. Then we update the guiText to reflect the changes. Fi-

nally, we return the index of the score. If no score was added `IndexOf` will return -1.

4. Change Name. `ChangeName` simply changes the entry at index to the given name and then updates the guiText.

5. The `LoadEntries` function does pretty much what it says it does: it loads the highscore from the prefs. We use the `PlayerPrefs` class to read and write from the preferences file. More information about PlayerPrefs can be found online.

All written highscore prefs follow a simple pattern: highscore's are keyed: "HighScore Score 0" 1, 2, 3 etc, and highscore names are keyed: "HighScore Name 0" 1, 2, 3 etc. When loading the entries we simply retrieve the score and name for the index then add it to the `entries` array. Afterwards we fill up the rest of the highscore with empty entries.

6. The `SaveEntries` function works like in a similar fashion to `LoadEntries`. This simple function saves out the high score entries in the same pattern as we loaded them by looping over the entries array writing every score and name using `PlayerPrefs.SetString` and `PlayerPrefs.SetValue`.

7. The `SetupHighscoreTableText` is used to build a string out of the `entries` array. We make the program go through all entries, tab separate a player's name and score, then add a newline. It repeats itself until all of the top 10 names and scores have been displayed.

8. `WipeoutPrefs` is a utility function that you can use to clear the highscore when debugging. It's sometimes useful to include functions in complex scripts to track down where problems might occur, to reset certain variables or events, etc. We used a similar sort of debugging function in our Waypoint script on page 25, when we discussed what the `OnDrawGizmoSelected` function did.

9. The `FormatScore` function is used to nicely format the score float. We want to split up the time float into minutes, seconds, and deciseconds integer values. The integer values are then concatenated (strung) together to form a nicely formatted time string.

And that's it! Now, that wasn't too hard was it? Really take your time to review the sections in this script. Many of the techniques used here are very simple and quite commonly used in many types of games. You'll be using many of these basic scripting ideas in your own game. Once you've read through them and start to use and alter them for your own games, they will become second nature and you'll be able to look at other scripts and better understand and utilize them for your own projects.

Updates (Unity 1.5)

The latest version of Unity (1.5) has provided users with a significant number of improvements and enhancements to the Unity engine. One example has been the introduction of a specialized wheel collider, and thats what users should now use for car wheels in this tutorial. An example project showing the new wheel collider can be downloaded from http://www.otee.dk/examples

The new wheel collider has support for slip curves, a specialized friction model for cars, dampers, and allows you to model anything from a dune buggy through to a F1 racing car game. The wheel collider is of very high quality, so you can create cars that just feel right, regardless of the type of vehcile or terrain your game models.

The new wheel collider has changed the way cars are setup from how we did it in this tutorial. The example project (at the link above) contains a description on how to set up cars with wheel colliders. many of the concepts like waypoints and AI can be applied to the new car model as well.

Where To Go From Here?

You will likely want to use your own artwork for the player car. When creating the artwork for the car you need to make sure it fits how the scripts were made. The z-axis of the car and all wheels have to be in the direction of travel. The x-axis of the car and all wheels have to be to the right when you view the car from behind. Try replacing the included racetrack with your own 3D race track. Experiment with some of the variable settings used; e.g., make the car accelerate faster or turn sharper, increase or decrease the number of AI opponents or stagger their starting position, or modify the Physic Material values. Most of all, experiment, have fun, and drop by the Unity forums to share questions, ideas and comments.