

The Game Programmer's Guide to Torque

The Game Programmer's Guide to Torque

Under the Hood of the
Torque Game Engine

A GarageGames Book

Edward F. Maurina III



A K Peters, Ltd.
Wellesley, Massachusetts

Editorial, Sales, and Customer Service Office
A K Peters, Ltd.
888 Worcester Street, Suite 230
Wellesley, MA 02482
www.akpeters.com

Copyright ©2006 by GarageGames, Inc.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

Set in ITC Slimbach and ITC Eras by Erica Schultz for A K Peters, Ltd.

Cover image and art in the Advanced Maze Runner prototype by Christophe Canon.

Library of Congress Cataloging-in-Publication Data

Maurina, Edward F., III., 1969–

The game programmer's guide to Torque: under the hood of the Torque Game Engine / Edward F. Maurina III.

p. cm.

“GarageGames book.”

Includes index.

ISBN 1-56881-284-1 (pbk. : alk. paper)

1. Computer games—Programming. I. Title.

QA76.76.C672M36 2006

794.8'1526—dc22

2005056630

Printed in the United States of America

09 08 07 06 10 9 8 7 6 5 4 3 2

This book is dedicated to my wife Teresa, for her encouragement, her advice, and most of all for her tolerance of the odd hours I kept while locked away in my office writing this book.

I must give special thanks to Jerry for acting as an idea bouncing-board and for listening patiently as I discussed chapter ideas over, and over, and

Of course, I must also thank the many members of the GarageGames community for their unfailing interest in the guide and their encouragement.

Lastly, I would like to thank the GarageGames staff for making the publication of this book possible, giving specific thanks to the “draft reviewers”—Josh Williams, Matt Fairfax, Ben Garney, Matt Langley, and Justin Dujardin.

Contents

Preface	ix
I Introduction	
1 Introduction	3
II Engine Overview	
2 Torque from 10,000 Feet	13
3 Torque Tools	35
4 Introduction to TorqueScript	97
III Game Elements	
5 Torque Core Classes	143
6 Basic Game Classes	157
7 Gameplay Classes	201
8 Mission Objects	263
9 Game Setup Scripting	347
10 Gameplay Scripting	383
11 Special Effects	419
12 Standard Torque Game Engine GUI Controls	455
13 Game Interfaces	539
IV Making the Game	
14 Putting it All Together	571
Index	599

Preface

So, you want to make a game? You may be standing in a bookstore holding this book in your hands, or you may be reading this online. Whatever the case may be, some or all of the following thoughts and questions are probably running through your mind:

- **I want to make a game, but can I do it on my own or with a small team?** Making a game is great fun, and a very rewarding experience. You can definitely make a game alone or with a small team as long as you have the right tools available to you. One of those tools is the Torque Game Engine (TGE) and the other is *Game Programmer's Guide to Torque* (GPGT). Using TGE and GPGT, you can create any game that your imagination can encompass and that your skills will allow.
- **TGE sounds good, but will GPGT tell me what I need to know to make my particular game?** TGE is a powerful and flexible game engine that can be used to make any number of different and unique games. You may choose to make single-player or multiplayer games. The game can be a shooter, an adventures, or a role-playing-game, to name just a few. *Game Programmer's Guide to Torque* will teach you the Torque skills you need to create these game types. (See section 1.1, "About the Torque Game Engine," and section 1.2, "What This Guide Contains," to learn more.)
- **Can I get up to speed fast enough to make my game?** Like any other complex and powerful piece of software, Torque can be hard or easy to learn. Everything depends on your approach to the task and whether you have the right resources available to you. With *Game Programmer's Guide to Torque*, with the hundreds of samples that come on the accompanying disk, and with the experience of making the sample game we write while reading this book, you will be able to ramp up very quickly and to move on to your goal—namely, making your own game.

Having been down the path you are just now starting upon, I know how hard it can be to get started and how hard it is to stay motivated in the face of the many challenges involved with learning to use Torque along with the other skills you will need to acquire. I decided to write this guide so that others would not have to struggle to learn Torque.

In closing, this guide is the result of my own need for a better reference and my desire to help other learn about the powerful and flexible Torque Game Engine. It is the culmination of my own game-writing and Torque-using journey. I sincerely hope that it provides you a pleasant beginning to your own game-making adventures.

Introduction

Part I

Chapter 1

Introduction

1.1 About the Torque Game Engine

1.1.1 What Is Torque?

The Torque Game Engine (TGE) is a AAA 3D game engine made available to the indie games community by GarageGames. It is the product of many years of dedicated work and interactive design and development by the staff of Dynamix, a well-known game development company which the founders of GarageGames previously started. As Dynamix made games, they would reuse and refine, taking the best parts of their work to the next generation of the engine. With this engine, they produced games like *Earthsiege*, *Starsiege*, *Tribes*, and eventually *Tribes 2*. All in all, it is safe to say that the code in this engine has its roots in code written as far back as 1995 and perhaps even earlier.

In summary, the Torque Game Engine is a product with man-centuries of development done by proven experts who time and time again used this engine to produce stellar titles. As far as I know, there is no other game engine like this on the market *at any price*.

1.1.2 Why Should I Use Torque?

Educational: One of the best ways to learn programming is to read code written by other developers. If you are going to read code, you might as well have fun and read game code and learn a few tricks in the process.

Resume Building: Mod (modify) the engine to show off your skills to future employers.

MOD Makers: How many times have you gotten stuck trying to mod other engines because they did not support feature X? Now you have the source and can easily add any features you want and truly differentiate your mod from the rest.

To Make Great Games! That's what we all live for, so do it. This is an unprecedented opportunity to build your game using an industry-proven game engine that rocks!

—GarageGames Site

One of the beauties of the Torque Game Engine is that you don't have to use it to make games. "What's that, you say?" I repeat, you do not have to use the Torque Game Engine to make games. With the features included in this engine, you can just as easily make a variety of professional, educational, or "your category here" products.

Introduction

Of course, you must abide by the end user license agreement (EULA), but once you have licensed the engine, the terms of the agreement are pretty free about what you can create. The only real limitation is your own imagination.

1.1.3 Not Just First-Person Shooters

Some people, examining the Torque Game Engine for the first time, may be under the impression that it is only for making first-person shooters (FPS). Nothing could be further from the truth. Yes, it is well suited to the FPS genre, but it can and has been used to make a variety of different game types.

Current Titles

Action Games	
 <p><i>Marble Blast GOLD</i></p>	 <p><i>Think Tanks</i></p>
 <p><i>Lore</i></p>	 <p><i>Orbz</i></p>

1.8.1 Icons Legend: Warnings, Notes, and Expert Tips

Throughout this guide, you will be presented with side notes of various forms. Some of these will be warnings of odd or misleading behavior, others will be notes on interesting bits or facts, and some will be expert tips for those who want to explore the edges of Torque's behaviors. You will be able to recognize these side notes by looking for the following icons.



Warning



Note



Expert Tip

1.8.2 Game-Building Lessons

Throughout the guide, you will find sections marked as one of the following:

1. **Maze Runner Lesson #123 (90 Percent Step)**. If you intend to make the game at the end of the guide, you must complete these lessons. They construct game elements without which the game will not function.
2. **Maze Runner Lesson #123 (10 Percent Step)**. These lessons are considered optional when making the initial version of the game. If you should choose to skip them, the game will still be playable but may be a bit rough around the edges.

These lessons will be largely independent of each other, but if a lesson depends on another lesson, the numeric ID of the lesson, as well as the chapter it is in, will be referenced.

Combined Lessons Appendix

For those who want the entire lesson set in one place, all of the lessons from the printed chapters, up to but not including Chapter 14, are included in the "Combined Lessons" electronic appendix.

Skip Ahead!

To learn about the motivation for the above lesson titles, and to learn what the game will be, please skip ahead to Chapter 14. There, you should read Section 14.1, "Maze Runner: A Simple Single-Player Game," which includes the following.

Introduction

- **Game Elements.** Here, we will briefly discuss the concept of a *game element*.
- **Game Goals, Rules, and Mechanics.** Next, we will explore the motivation for planning a game's goals, rules, and mechanics before we write the game. Then, we will do this planning for our game.
- **Setting up our workspace.** Before we can start working on the lessons, we need to set up a workspace. In this section, I will instruct you on what steps are required to prepare for the lessons.
- **90 Percent or 10 Percent?** Lastly, I will give you an overview of the 90 percent versus the 10 percent steps and why these ideas matter.

So, skip ahead; it's OK. When you're done, you can come back and start learning about Torque!

Engine Overview

Part II

Chapter 2

Torque from 10,000 Feet

The Torque Game Engine (TGE) has a long legacy. In its various incarnations, it has been used to make both non-networked single-player games and networked multiplayer games. Today, TGE has the following features.

- **Single-player and multiplayer ready.** TGE is based on a standard client-server architecture and is fully scalable to 128 players and beyond.
- **Raster-based graphics.** TGE is not shader based but has the capability to incorporate any features you desire (you have the source code). Furthermore, it is the predecessor to the Torque Shader Engine (TSE), and thus most things learned using TGE will apply to TSE.
- **Event-driven simulation.** TGE is designed around an event-driven simulator. It utilizes separate client and server event loops. Additionally, most game logic and GUI logic is driven by an event system.
- **Memory and network bandwidth efficient.** TGE is designed to have a reduced memory footprint and an accompanying low-bandwidth requirement per connection. It utilizes static datablocks for common information and network compression plus transmission-reduction algorithms.
- **Broad functionality.** Because of its long heritage, TGE comes ready with most of the methods and functions required for standard game calculations, actions, and responses.
- **Fully integrated.** TGE incorporates all the code required to render/play/capture all game elements, including GUIs, sound, 3D graphics, and other I/O (input/output). It also includes a large and expanding set of content creation and debugging tools out of the box.

2.1 TGE Terms and Concepts

When you first start working with TGE, you will come across terms like *interior*, *shape*, *datablock*, *portal*, *IFL*, *image*, etc. Some of these words have TGE specific meanings, others are industry-standard terms, and a small set are hybrid terms with meanings in both worlds. Either way, if you are not very experienced, just trying to figure out what these terms are may be a big challenge. To help ease this transition, we will run through some of the more confusing terms and concepts you will encounter while working with TGE. For a more extensive list of terms, see the “Glossary Of Terms” appendix.

2.1.1 Shapes and DTSs (TGE Term)

A shape, also known as a DTS object, is a model created using a polygon (or equivalent) editor. Such models may have

- skeletal animations (see Section 2.1.8, "Animations: Blended vs. Non-Blended"),
- multiple skins (textures),
- animated skins,
- visibility animations,
- multiple levels of detail (see Section 2.1.5, "Level of Detail"),
- translucent and/or transparent components,
- multiple collision boxes (see Section 2.1.6, "Collision Detection"),
- and much more.

This is the first of two model categories used by TGE. DTS, which stands for the Dynamix Threespace Shape, is both the shorthand notation for this concept and the file extension (e.g., player.dts). Shapes are generally used to represent nonstructural entities such as players, power-ups, trees, and vehicles. Shapes can be created with 3ds Max, MilkShape, or Caligari's gameSpace/trueSpace, to name just a few possible content-creation tools. See the GarageGames website to learn how this is done and to find the proper exporter for your content tool(s).

Non-DTS Renderers?

Some users have complained that they would rather use an alternate format instead of being "forced" to use the DTS format. This is entirely possible. Users have already produced alternate mesh renderers to include such formats as 3DS and MS3D. If you have a favorite format and are familiar with how it works, you can simply pick up one of the previously mentioned mesh renderers and modify it for your own format.

Shapes in Our Game

In the prototype for our game, we will need just a few shapes: a player, coins, maze blocks, and fireballs.

- **An avatar or player.** The lesson kit comes with Joe Maruschak's "Blue Guy" (Figure 2.1, left), but we will not be using him beyond a quick introduction. Why? In order to demonstrate the minimum set of animations that need to be included to make the shape work with the Player class, we will make the "Simplest Player" (Figure 2.1, right), a simple geometric shape.
- **Pick-ups, maze blocks, and fireball blocks.** In our game, we will also require shapes to represent coins that we can pick up. Also, we will need

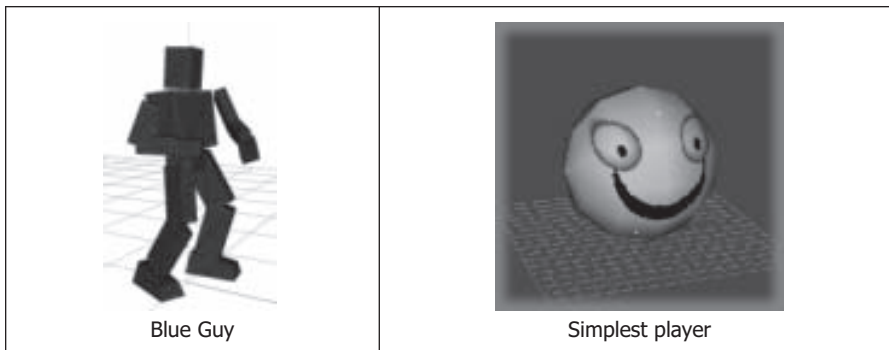


Figure 2.1.
Simple Player shapes.

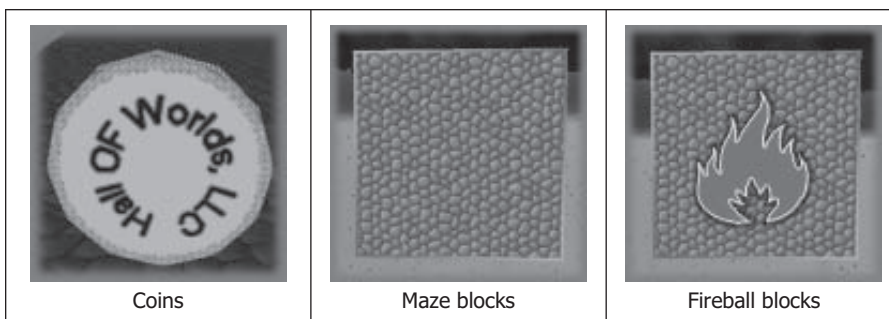


Figure 2.2.
Required shapes and blocks.

a variety of blocks and obstacles (fireball blocks) to build our mazes from (see Figure 2.2).

2.1.2 Interiors and DIFs (TGE Term)

Interiors are models created using convex (see Section 2.1.3, “Convex vs. Concave”) brushes.

The InteriorInstance class, frequently referred to simply as Interior(s), is used to display models that represent any structural object, to include such things as buildings, bridges, walls, and other large structures. The motivation for this name comes from the fact that these objects can have an actual inside, i.e., interior.

This modeling technique is used to solve a few technical issues associated with creating large and geometrically complex models that are intended to be entered by other models (or the camera). Some of the biggest technical problems solved by this technique are the following.

- **Efficient collision detection.** Binary space partitioning (BSP) trees are generated and used for detecting collisions against Interior objects. BSP trees provide a very efficient way of determining object collision, one of the most CPU-intensive processes a real-time application performs.

Engine Overview

- **Visibility culling.** This technique also provides numerous shortcuts for culling of visibility through the use of portals (see Section 2.1.7, “Portals”) so that rooms and terrain that the player can’t see don’t get sent to the graphics card for rendering. This is a lot harder to do, from a mathematical standpoint, than a nonprogrammer might imagine.
- **Efficient lighting.** Finally, this technique “regularizes” (to abuse the English language a bit) the process of calculating lighting and shading as affected by the presence of the model in the game world.

DIF, which stands for Dynamix Interior Format, is both a shorthand notation for the same concept and the extension for these files (e.g., myBuilding.dif).

Interiors can be created with QuArK, Worldcraft/Hammer, 3ds Max, MilkShape (not advised), or Caligari’s gameSpace/trueSpace. See the GarageGames website to learn how this is done and to find the proper exporter for your content tool(s).

2.1.3 Convex vs. Concave (Industry Terms)

In TGE, all collision meshes must be convex, not concave. The trouble is, many people either do not know what these terms are or cannot remember how to identify a convex or concave mesh.

Finding the parts of a mesh that are concave (making it a bad collision mesh) can be frustrating at best. Therefore, you can follow this simple rule when making collision meshes:

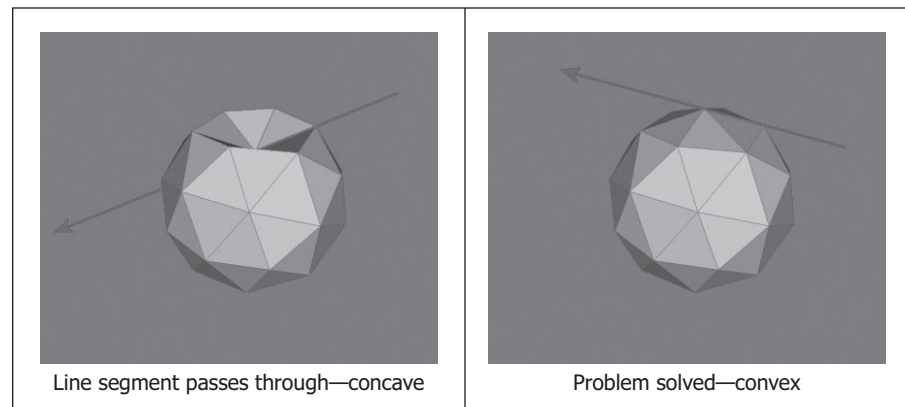
If any **line segment** on the mesh, when extended infinitely in both directions, **passes through** the **interior of** your mesh, the **collision mesh** is concave and therefore **bad**.

Or the shorter version:

Line segment passes through interior of collision mesh ... bad (Figure 2.3).

Figure 2.3.

Using line segments to discover concavity.



ActionMaps

ActionMaps are a special class designed to capture and redirect inputs. There are two kinds of ActionMap. There is the GlobalActionMap and the normal ActionMap. The main differences between these are:

- **GlobalActionMap.** This is the daddy of input processors and supersedes all other processing methods. This action map should not be popped from the processing stack (see below).
- **ActionMap.** This is a generic action map. It takes lower priority than all other processing methods. These action maps can be pushed and popped from the processing stack as the game's requirements change.

ActionMaps in Our Game

Our game will require some kind of mapping between keyboard and mouse inputs to player movements and behaviors. We will stop briefly and show what these mappings are and discuss how they are attached (indirectly) to the player.

Processing Stack

What the heck is a processing stack, you ask? TGE implements an event queue, which is used to collect all user inputs and various other events. These events are then processed by the engine. The ActionMap is one consumer of these events. Because ActionMaps can be stacked and because they process events on the input queue, I refer to this as the processing stack.

In short, an ActionMap not on the processing stack is not catching and therefore not processing input events.

2.4.2 TGE File I/O

TGE has a file manager that maintains a working list of all the files found in the game directory and all subdirectories. This list is created on start-up. Subsequently, the file manager will locate new files that you add and then attempt to load from the console or via scripts. It will also notice when files have been modified and recompile and load them when requested to do so.

In short, with TGE you can easily add new files and modify existing content without having to restart the engine. This is a huge timesaver when creating new content and while debugging.

It is worth mentioning that finding new files without restarting is a new feature (introduced in version 1.4). If you are currently using 1.3 or a prior version, you may use the `setModpaths()` function to find new files. This isn't as nice as an automatic find, but you can still work without restarting.

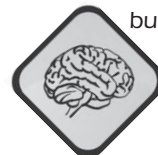


Table 3.1 (continued).

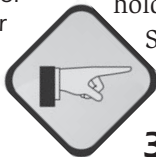
Tools	Start Tool	Description
Mission Area Editor (Area Editor)	F5	This tool allows you to adjust the boundaries of the current mission and provides a means to mirror the current terrain.
Terrain Editor	F6	This tool provides the ability to directly manipulate the terrain using the mouse as a multi-operation brush.
Terrain Terraform Editor (Terraformer)	F7	In addition to providing all the capabilities of the Terrain Editor, this editor allows you to load images as terrain files and to apply various algorithmic generators and filters to the terrain.
Terrain Texture Editor	F8	In addition to providing all the capabilities of the Terrain Editor, this tool allows you to select any number of textures and apply them using a set of algorithms to determine blending and placement.
Terrain Texture Painter (Terrain Painter)	Window Menu → Terrain Texture Painter	In addition to providing all the capabilities of the Terrain Editor, this tool allows you to select and subsequently to apply up to six different textures to the terrain.

3.3 The World Editor Tools

Let us tackle the World Editor toolset first, as it has the most components and is the most likely place to start when creating a simple mod (modification) or a new game.

As we investigate and learn how to use each of the World Editor tools, please use the GPGT Lesson Kit (provided on the accompanying CD) and run the “World Editor Training” mission.

Please note that, while you are editing in the World Editor, you can get help simply by pressing F1. This will bring up a help dialog with descriptions of the tools and their features.



3.3.1 World Editor Basics

Before leaping into the World Editor tools, let us review some things that hold true for all of the tools. First, we will review the user interface devices.

Subsequently, we will discuss the mechanics of movement and viewpoint control, as well as object selection, translation, rotation, and scaling.

3.3.2 World Editor Devices

In this guide, the cursors, menus, and other graphical elements that you encounter in the editors are referred to as devices. Simply stated, these devices provide meaningful feedback to you regarding what action can or should be taken. The terms below are mostly of my own invention, with the exclusion of the appropriately named *gizmo*.

3.3.3 Cursors

Table 3.2 explains what each cursor image means.





Device	Description
 No-Select Cursor	When the cursor looks like this, it means that the cursor is not over a selectable object. In other words, you are pointing to an empty space.
 Select Cursor	When the cursor looks like this, it means that the cursor is over a selectable object. In other words, you are pointing to an object that can be selected.
 Grab Cursor	When the cursor looks like this, it means you have successfully selected an object's gizmo axis in translation mode. In other words, you can move the object around by clicking and dragging when this cursor device appears.
 Rotate/Scale Cursor	When the cursor looks like this, it means you have successfully selected an object's gizmo axis in either rotation or scaling mode. It also appears when you have successfully selected a bounding box face for scaling or rotation.

Table 3.2.

Descriptions of cursors.

3.3.4 The Gizmo and Gizmo Scales

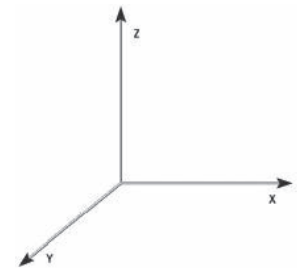
The graphic in Figure 3.1 represents the gizmo. The gizmo is a device that is activated when you select one or more objects. It displays the three traditional x - y - z axes. Individual axes are selectable and afford the ability to translate, rotate, and scale.

By default, a gizmo axis is dark cyan when not selected and light cyan when the cursor is over it or when it has been “grabbed.” Additionally, when a selected gizmo is used for an operation, one of three scales will be shown: the gizmo translation, rotation, or scaling scale.

This scale shows the current position of the object's centroid when you use the gizmo to translate an object.	x: -51.024, y: -127.829, z: 226.473 Gizmo Translation Scale
This scale shows the current degrees of rotation around the selected axis when you use the gizmo to rotate an object.	x: 0.000, y: 0.000, z: 1.000, a: 52.519 Gizmo Rotation Scale
This scale shows the current height, width, and depth of an object when you use the gizmo to scale it. <w,h,d> correspond to the x,y,z axes of the gizmo.	w: 1.2000, h: 1.2000, d: 2.144 Gizmo Scaling Scale

Figure 3.1.

The axis gizmo.



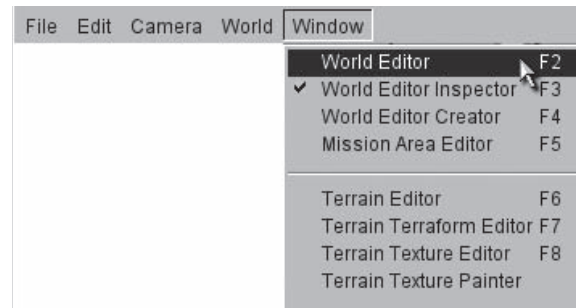
3.3.5 Menus and Windows

The World Editor provides a set of traditional menus for selecting the current tool as well as other features (see Figure 3.2).

Please note that all of the menu options will be covered in Section 3.5.3, “World Editor Menus.”

Figure 3.2.

World Editor menus.



Several of the tools have windows that appear on the right side of the screen (see Figure 3.3). Although these windows have many similarities, it will be better to explain them individually in the respective tool sections below.

3.3.6 Selection Boxes

When selecting a previously unselected object, the selection cursor lets you know when you can select something, and the green selection box (see Figure 3.4) shows which previously unselected object will be selected.

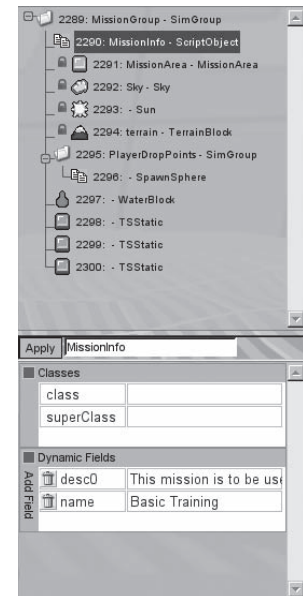
Once you have successfully selected an object, the object will be shown with both a red selection box and a yellow selection box (see Figure 3.5). The red box is object aligned, while the yellow box is world aligned.

The purpose of the yellow box is to show which objects are selected as a group and will therefore be affected by any actions you take. The red boxes are to show which individual objects in the group selection box are actually part of the selection. Notice that, in Figure 3.5, the leftmost and rightmost characters are selected, while the middle character is not.

Once you have successfully selected an object, the selection box will turn blue if your cursor passes over it (see Figure 3.6). Please note that this is not true for drag-select.

Figure 3.3.

Tool windows.



Chapter 14

Putting It All Together

14.1 Maze Runner: A Simple Single-Player Game

Maze Runner is a simple platform game brought into the 3D realm. It isn't based on a specific game, but it is inspired by games I have played. My purpose for this game was not to create a new blockbuster but rather to provide an easy-to-understand game idea upon which we could hang examples as we worked through the guide.

A 60-second summary of this game would read something like the following.

In this game, you run around a maze and pick up coins. Your goal is to pick up all the coins while avoiding various obstacles. Mazes will vary in size and in scope. They may run along one level, or have multiple levels. Along the way, as you hunt for all of the coins, you will need to avoid disappearing bridges that may drop you to a lower level or into a fiery cauldron below. You will be blocked by fireballs and impassable chasms. To get around these obstacles, you will have to use your ingenuity and the occasional teleport station. Timing, awareness of your surroundings, agility, and a little luck are all required for winning. You will start with three lives and gain a new life for each level you complete. To continue the game, pick up all of the coins and move on to the next level. Get the highest score and win the admiration of your peers! Good luck.

14.2 Game Elements

Let's stop for a moment and define the term *game element*. This is a term that I am using to describe any and all of the pieces that are used to create a game. For example, all of the following listed items are game elements:

- **The game view.** This general term incorporates point of view, field of view, and other view-related concepts and describes the end view of our game. We discuss this in Chapter 7, "Gameplay Classes."
- **Interfaces and HUDs.** However much we might wish to ignore it, all games require some GUI work and will have a variety of interfaces (splash screens, main menus, play GUIs, etc.) and some HUDs (counters, indicator bars, etc.).
- **Players and opponents.** Although we could certainly have a game with no directly identifiable players or opponents, 3D games generally do have at least one model representing the player and other models opposing this player in some fashion.
- **Weapons.** This seems pretty straightforward, but what I really mean here is weapons *and weapon analogues*. The analogue, in this case, is something that functions like a weapon but may not necessarily do damage.

Making the Game

- **The world.** This is a rather large game element and is in fact composed of a multitude of subelements, including terrain, water, the sky, environmental objects (trees, rocks, grass, etc.), environmental effects (rain, wind, lightning, the sun(s) and planets, etc.), structures (buildings, fences, bridges, etc.), sounds, and so on.
- **Power-ups and pickups.** These are items that are often at the core of a game and are meant to be interacted with. Sample items in this category would be coins, gems, weapons, ammunition, health packs, etc.
- **Special effects.** Here we are talking about eye and ear candy. These do have a place in gameplay, but they are often not directly tied to interaction, which is where we should focus our attention first.
- **Miscellaneous elements.** This last category is a grab bag for elements that don't fit anywhere specifically. Some examples are inventory systems, collision detection and response, damage and energy, and general scripting tasks.

Now, armed with an idea of what a game element is, let's list the game elements in our game.

14.2.1 Maze Runner: Game Elements

The finished game has the following elements and attributes.

- **Interfaces.** Splash screen GUI, main menu GUI, credits GUI, and play GUI.
- **Game view.** The game can be played in 3rd POV only.
- **Player.** The initial player will be the Blue Guy that comes with the FPS Starter Kit. We will later design our own player. This player will be an example of the simplest possible player that can be used in a game.
- **Opponents.** There are no opponents in this game, but some suggestions will be provided for adding them if you wish to expand on this game later.
- **The world.** The game world is a simple cauldron-shaped pit. This pit will contain a lake of lava. Our maze will consist of individual shapes that we place using scripts and level-definition files. We will place some environmental objects to spruce the place up. Additionally, there will be a sky box, celestial bodies, clouds, wind, rain, and even lightning. We're going all out on special effects to show how to use as many Torque features as is reasonable.
- **Obstacles.** There are two types of active obstacles and three static obstacles. The active obstacles include level blocks (individual and grouped) that fade, disappear, and reappear over time. There are also blocks that shoot fireballs in any of eight fixed compass directions (N, NE, E, SE, S, SW, W, NW), or down, or any of the prior directions, but randomly. The static obstacles are open horizontal spaces between blocks, vertical spaces between blocks, and blocks themselves.

- **Getting around.** To get around the maze, the player will run and jump. Also, there can be up to three distinct teleport stations; that is, teleport stations can be grouped in sets, and there can be up to three distinct sets of teleport stations in a level. Additionally, if any set contains more than two stations, entering one station will randomly send the player to any one of the other stations in the set.
- **Pickups and power-ups.** The only pickup in the game is the coin. Picking up all coins is the primary goal. A HUD will show the total coins picked up and the number of coins remaining for the level.
- **Inventory system.** We will use the “Simple Inventory” system that comes with this guide and is described in Chapter 7, “Gameplay Classes.” It will provide all the mechanics necessary to pick up coins and remove them from the game world.
- **Miscellaneous “glue” scripts.** We will end up writing quite a few scripts to tie the game together, to track the score and our lives count, as well as to load the levels.

14.3 Game Goals, Rules, and Mechanics

Great! Now we know generally what the game is about and what elements it has. The last thing we need to do is describe how the individual game elements interact.

The goal of this game is very simple: score as high as possible by finishing as many levels as possible before losing all of your lives.

The rules and mechanics for this game are as follows.

- **Pick up all the coins.** Picking up all coins on a level ends the level and takes the player to the next level.
- **Stay alive.** Falling into the lava below or getting hit by a fireball kills the player.
- **Gain lives.** To gain more lives, simply complete a level. One new life is gained for each level completed.
- **Teleporting.** We can place up to three sets of teleport stations. Each set may have two or more stations. If there are only two stations in a set, the stations will teleport back and forth between each other. If a set has three or more stations, the spawn point will be randomly selected. Teleporting occurs by running over a station. The destination station will be temporarily disabled to avoid infinite teleport loops. It will not operate again until you walk off the station. Teleporting is not instantaneous, so be careful about fireballs that cross stations, as you are temporarily unable to move when teleporting.
- **Respawning.** When the player is killed, it will respawn in the location where it was first dropped into the game.

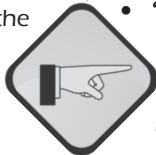
- **Level loading.** To make this game easily maintainable, tunable, and modifiable by players, all level loading is controlled by a text file (the level file). Players can add new levels and redefine levels to their hearts' content.

14.4 Setting Up Our Workspace

Before we can work on any lessons, we must first set up a work area. Everything that you need to do this is supplied on the CD that comes with this guide. If you examine the CD, you will find the following directories.

- “**\Appendices**”. This directory contains the GPGT appendices.
- “**\Base**”. This directory contains data and scripts that are used in the lessons and can also be used later to make new games. Please see the “Lesson Kit Assets” appendix for additional information about the contents of this directory.
- “**\GPGT LessonKit**”. This directory contains the GPGT lesson kit. For more information about it, please read the “Lesson Kit User’s Guide” appendix.
- “**\MazeRunner**”. Excluding the data and scripts in “\Base” and some content we will copy from the TGE demo that you should install using one of the installers found in “\TorqueDemoInstallers”, this directory contains all of the unique resources and scripts required to build the MazeRunner prototype.
- “**\MazeRunnerAdvanced**”. This directory contains a completed version of MazeRunner with several additional features as suggested in Section 14.10, “Improving The Game”.
- “**\TorqueDemoInstallers**”. This directory contains installers for TGE.

If you are a Linux user, I must apologize. At the time this book went to print, version 1.4 of TGE for Linux was still being worked on. Please check the GarageGames website to see if it is ready and, if so, download the demo kit. Otherwise, I suggest using one of the other versions of the engine in the interim.



At this time, if you do not have the demo installed on your machine, please do so by running the appropriate installer (based on your computer and operating system type). Once you have finished, please continue reading.

14.4.1 Starting from Torque Demo

First, be sure to install a copy of the TGE demo using one of the installers found in “\TorqueDemoInstallers”. Feel free to install this anywhere you please. While writing our game, we will be copying files out of the installed demo to a working directory.

Second, let’s make a new (working) directory named “MazeRunner” and place it on a drive with at least 100 MB of free space. We’ll want some elbow room while we work. Please note, while we are writing our game (reading the numbered lessons), this is the directory we will be working in. We will be copying materials from the CD to this directory and editing them in some places. Do not confuse this with the GPGT Lesson Kit which is also included on the CD. The GPGT Lesson Kit is a separate application containing several

Player::loseALife()

The easiest way to handle removing lives is to make a method scoped to the Player class (so it can be called on the Player object) that handles all of the bookkeeping. This simplifies things greatly. Yes, right now only two things can kill the player, but later you might add more, and having killing code all over the place would be very bad.

Here is the code (located in “mazerunnerplayer.cs”).

```
function Player::loseALife( %player ) {
    // 1
    %player.lives--;

    // 2
    if( %player.lives <= 0 ) {
        schedule( 0 , 0 , endGame );
        return;
    }

    // 3
    %player.setVelocity("0 0 0");
    %player.setTransform(%player.spawnPointTransform);
}
```

This code does the following.

1. It decrements the player’s life counter. (Yes, we haven’t talked about this yet. It’s coming up soon.)
2. It checks to see if all of our lives are gone and then schedules a call to `endGame()` (in “game.cs”) to unload the mission, destroy the player, disconnect the client from the server, and get us back into the main menu.

Why not call `endGame()` directly?

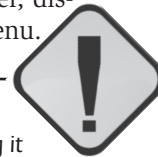
You may wonder why we schedule a call to `endGame()` instead of calling it directly.

The reason we do this is that, when we call `endGame()`, we indirectly cause the player to be deleted.

However, the player is the object that the `loseALife()` method was called on, so when the engine tries to return from the call to `endGame()`, it will not have anywhere to return to. **This will crash the engine.**

The lesson here is to never delete the current object in a method that is called on that object. Always defer that deletion by using a call to `schedule()`.

Calling `schedule()` with a time of 0 milliseconds tells the engine to run the function as soon as possible after returning from all nested function calls. In practice, this will always be on the next processing cycle or later.



Making the Game

3. If the game is not over, the player is moved back to its last spawn point. This information is stored in the player by `playerDrop()` in the file “levelloader.cs”:

```
$Game::Player.spawnPointTransform = (%actX SPC %actY SPC
                                     $CurrentElevation);
```

Initial Lives

In order to take away lives, we must have lives to take. The best place to add initial lives to the player is either in its `onAdd()` method or at the location where we create it. I chose the `onAdd()` method (in “mazerunnerplayer.cs”; bold lines are new code):

```
function MazeRunner::onAdd( %DB , %Obj ) {
    Parent::onAdd( %DB , %Obj );
    %Obj.lives = 3;
}
```

Fireballs

OK, we got a little off topic there, but we’re back now. The next question is: how do fireballs kill?

The projectile object has an `onCollision()` callback that is called for collisions with any world object. So, if we write a version of this callback in the namespace of our projectile, we can have that callback check to see if the player was hit and call `loseALife()`.

```
function FireBallProjectile::onCollision( %projectileDB ,
                                         %projectileObj ,
                                         %collidedObj ,
                                         %fade , %vec ,
                                         %speed ) {
    if ( %collidedObj.getClassName() $= "Player" ) {
        %collidedObj.loseALife();
    }
}
```

In the above callback (located in “fireballs.cs”), the engine is asked to get the class name for the collided-with object. It then compares this to “Player”. If the comparison returns `true`, `loseALife()` is called on the collided-with object.

Alternate Solution #1

There is an alternate way to write this code that would actually work in more cases (i.e., for `Player` and `aiPlayer`).

```
// Alternate implementation
function FireBallProjectile::onCollision( %projectileDB ,
                                         %projectileObj , %collidedObj ,
                                         %fade , %vec , %speed ) {
    if ( %collidedObj.getType() $= $TypeMasks::PlayerObjectType ) {
        %collidedObj.loseALife();
    }
}
```

This alternate implementation uses the `getType()` method to get a bitmask for the collided-with object. The bitmask contains bit settings for all classes from which the object is derived as well as for the class itself. So, as I alluded to, if the collision occurred against an `aiPlayer` (which is derived from `Player`), this comparison would still work, whereas the prior code would not. In this game, we don't have that worry, so let's leave it as is.

Alternate Solution #2

Originally, as I wrote this code for the book, I was using a bleeding-edge version of the engine (version 1.4 before release), and I ran into a bug (that has since been fixed) where `%collidedObj` was always getting "1". For a moment, I thought I was stuck. Then, it occurred to me that there are other ways to solve the identification problem, and I wrote the following code.

```
%Offset = vectorSub( %vec , $Game::Player.getWorldBoxCenter() );
%Len = vectorLen( %offset );
if( %len < 1.7 ) {
    $Game::Player.loseALife();
}
```

This code uses the position of the projectile's collision and then compares it to the position of the player's centroid. If the distance between them is small (1.7 world units or less), in all likelihood the object that was hit is the player, and I call `loseALife()`. This solved my temporary problem, and in the occasional instance when the player wasn't hit but was just close to the collision point, the difference was not noticeable.

The lesson here is that TGE is very flexible, and you can often solve the same problem in many ways. So, don't let one problem stop you.

Out of Lives

At some time, after all this losing of lives, the player will be out of lives. According to our initial rules list, this means the game is up, time to go home. As we have already seen (above) the `loseALife()` method handles this case and ends the game for us.

14.8.5 Moving On

The last things we need to fix with regard to gameplay are moving on to the next level and getting our extra life.

Last Coin

Our design rules stated that, when the last coin is picked up, the current level should be unloaded and the next level should be loaded. So, how do we do this?

If you recall, the inventory system has a callback called `onPickup()`. When we discussed this callback, I said that you might want to override it to implement special behaviors. This is one of those times.

If you will look in “coins.cs”, you will find the following implementation of `onPickup()`.

```
function Coin::onPickup( %pickupDB , %pickupObj ,
                        %ownerObj ) {
    // 1
    %status = Parent::onPickup( %pickupDB , %pickupObj ,
                               %ownerObj );

    // 2
    if (CoinsGroup.getCount() == 0 ) {
        buildLevel( $Game::NextLevelMap );
        $Game::Player.lives++;
    }

    // 3
    return %status;
}
```

This callback does the following.

1. It takes advantage of the prewritten pickup code by calling the `Parent::` version.
2. It then checks to see if the `SimGroup CoinsGroup` is empty. In the case that it is empty, `buildLevel()` is called with the stored numeric ID of the next level, and a new life is added to our player.

Index

A

- ActionMaps 33, 356
 - actions 359
 - defining 357
 - devices 359
 - moveMap 222
 - unbinding 361
 - vehicle ActionMaps 235
- add parent 89
- alarmMode 197
- animation 169
 - blended 20
 - cyclic 169
 - direction 170
 - non-blended 20
 - pausing 170
 - playing 169
- animation sequences
 - activateBack 230, 231
 - activateBot 230
 - back 225
 - brakelight 230
 - Damage Animations 171
 - fall 225
 - jump 225
 - land 225
 - maintainBack 230, 231
 - maintainBot 230, 231
 - root 225
 - run 225
 - side 225
 - spring0 .. spring7 230
 - standjump 225
 - steering 230, 231
 - Vehicle 228
- Atlas* 268
- Audio Emitters 296

B

- Blue Guy 16, 17, 223
- brushes
 - brush hardness 61, 62

- brush mode 59, 60
- editing actions 60
- selection and < Radius > 62
- selection mode 59, 60, 62
- bump mapping 265

C

- callbacks 21, 355, 383
- Canvas 456
- classes
 - animating 329
 - animations 196
 - as control object 208
 - AudioDescription 448
 - AudioEnvironment 448
 - AudioProfile 448
 - AudioSampleEnvironment 448
 - bouncy 178
 - Camera 169, 201
 - CameraData 201
 - collisions 196
 - controlling 221
 - Debris 419
 - DebrisData 419
 - DecalData 426
 - ExplosionData 427
 - field of view (FOV) 205
 - FileObject 369
 - friction 179
 - GameBase 31, 143, 155
 - GameBaseData 143, 155
 - gravity 179
 - GuiControl 470
 - HoverVehicle 240
 - HoverVehicleData 240
 - InteriorInstance 17, 31, 197
 - Item 157, 175
 - ItemData 175
 - movement 217
 - namespaces 353
 - networking 356
 - pitch 208

Index

- Player 213
- PlayerData 213
- POV Cookbook 210
- Projectile 437
- ProjectileData 438
- restricting POV 208
- rotating 177
- SceneObject 31, 143, 151
- ScriptGroup 31, 352
- ScriptObject 31, 352
- Selecting Node 208
- ShapeBase 31, 158
- ShapeBaseData 31, 158
- ShapeBaseImageData 157, 189
- SimDataBlock 143, 148
- SimGroup 31, 350
- SimObject 31, 143
- SimSet 31, 347
- static 177
- StaticShape 157, 183
- StaticShapeData 183
- sticky 178
- TStatic 31, 187
- Vehicle 231
- VehicleData 231
- WheeledVehicle 236
- WheeledVehicleData 236
- WheeledVehicleSpring 238
- WheeledVehicleTire 237
- yaw 209
- client-server architecture. *See* networking, client-server
- cloaking 160
- clouds 281
 - storm 284
- collision detection (COLDET) 17, 19, 153, 220
 - collision meshes 18
 - collision timeout 180
 - onCollision() 21, 234, 249, 385
 - ShapeBaseImageData 196
 - TStatic 187
- concave 18
- console callbacks
 - applyDamage() 163
 - click() 494, 508, 544, 545
 - doDismount() 235
 - eval() 413, 414
 - exec() 133, 181, 185, 223, 226, 256, 257, 315, 364, 372, 380, 576
 - onAction() 518
 - onClearSelected() 486
 - onCollision() 21, 135, 233, 234, 248, 249, 252, 385, 584, 585
 - onEnterTrigger() 339, 342, 386, 582
 - onInputEvent() 525
 - onInspect() 535
 - onLeaveTrigger() 339, 342, 343, 386
 - onMount() 234
 - onPickup() 21
 - onRightMouseDown() 535
 - onSelectPath() 524
 - onSleep() 385, 459, 544
 - onTabComplete() 503
 - onTabSelected() 486
 - onTickTrigger() 339, 340
 - onTrigger() 340
 - onTriggerTick() 340
 - onURL() 497
 - onWake() 385, 459, 491, 544, 551, 552
- console functions 114
 - activatePackage() 123, 124, 125
 - addMaterialMapping() 217
 - calcExplosionCoverage() 434
 - call() 414
 - cancel() 390, 518
 - commandToClient() 416, 417
 - commandToServer() 250, 251, 364, 415, 416, 418
 - compile() 379, 380
 - containerRayCast() 401
 - detag() 107
 - echo() 100, 144, 145, 146, 154, 155, 156, 168, 177, 179, 184, 341, 348, 349, 350, 351, 353, 354, 355, 358, 365, 366, 369, 384, 388, 390, 393, 394, 395, 396, 397, 398, 399, 400, 404, 405, 406, 411, 412, 413, 414, 434, 473, 474, 475, 494, 495, 505, 506, 507, 520, 527, 532, 551
 - error() 405
 - eval() 413, 414
 - exec() 133, 181, 185, 223, 226, 256, 257, 315, 364, 372, 380, 576
 - expandFilename() 367, 369, 490, 492, 551, 556

console functions (*continued*)

fileBase() 368
 fileExt() 368
 fileName() 367, 368, 369, 490, 492, 551, 556
 filePath() 367
 findFirstFile() 364, 365, 366
 findNextFile() 364, 365, 366
 firstWord() 392, 393
 getBoxCenter() 404
 getEventTimeLeft() 389
 getFieldCount() 395
 getFields() 135, 395
 getFileCount() 366
 getFileCRC() 366
 getRandom() 405, 409
 getRandomSeed() 405
 getRealTime() 390, 391
 getRecord() 394, 395
 getRecordCount() 394, 395
 getRecords() 394
 getScheduleDuration() 390
 getSubStr() 396, 397, 557
 getTimeSinceStart() 389
 getWord() 153, 392, 472, 562, 566
 getWordCount() 392, 393
 getWords() 153, 243, 343, 392
 isEventPending() 389
 isFile() 368
 isObject() 258, 344, 375, 409, 412, 413, 417, 528, 550
 ltrim() 399
 mAbs() 400, 402
 mAcos() 402, 565
 mAsin() 402
 mAtan() 402
 MatrixCreate() 403
 MatrixMulPoint() 400, 401, 403
 MatrixMultiply() 403
 mCeil() 400, 402
 mCos() 402
 mDegToRad() 402
 mFloatLength() 405, 406
 mFloor() 400, 402, 560
 mLog() 402
 mPow() 400, 402
 mRadToDeg() 402
 mSin() 402
 mSolveCubic() 403, 404
 mSolveQuadratic() 403, 404
 mSqrt() 400, 402
 mTan() 402
 NextToken() 393, 394
 quit() 90, 547
 removeField() 395
 removeRecord() 394, 395
 removeWord() 392, 393
 restWords() 392, 393
 rtrim() 399
 schedule() 148, 161, 171, 377, 387, 388, 389, 390, 391, 392, 406, 407, 409, 491, 561
 setDefaultFov() 203, 205, 206
 setField() 395
 setFov() 203, 205, 206
 setRandomSeed() 405
 setRecord() 394, 395
 setWord() 392, 393
 setZoomSpeed() 203, 206
 strchr() 396, 398
 strcmp() 398
 stricmp() 398
 stripChars() 399
 StripMLControlChars() 399
 stripTrailingSpaces() 399
 strlen() 396, 397, 556
 strlwr() 396
 strpos() 397
 strreplace() 397, 398
 strstr() 397
 strupr() 396
 trim() 399
 VectorCross() 402
 VectorDist() 402
 VectorDot() 402, 565
 VectorLen() 179, 243, 402, 585
 VectorNormalize() 402, 447, 565
 VectorOrthoBasis() 402
 VectorScale() 168, 402, 444, 447
 VectorSub() 243, 341, 402, 528, 585
 console methods 118, 120
 activateLight() 197
 add() 110, 135, 153, 163, 167, 177, 181, 182, 257, 258, 259, 334, 342, 347, 348, 350, 351, 355, 375, 383, 384, 402, 444, 447, 458, 494, 517, 543, 580, 582, 584, 589

Index

console methods (*continued*)

- addColumn() 480
- addMenu() 513
- addPage() 486
- addRow() 480, 504
- addScheme() 516
- addSelection() 533, 534, 535
- addText() 498, 551
- applyDamage() 163
- applyImpulse() 168, 256
- applyRepair() 163
- attach() 496
- bind() 357, 360, 361, 364, 415, 418
- bindCmd() 250, 251, 358, 360, 361
- bringToFront() 349
- buildIconTable() 530
- clear() 350, 505, 518, 532
- clearMenuItems() 513
- clearMenus() 513
- clearSelection() 533
- close() 369, 370, 371, 551
- delete() 146, 147, 148, 258, 344, 348, 351, 369, 370, 371, 375, 384, 388, 389, 543, 551
- deleteLine() 496
- deleteSelection() 534, 535
- detach() 496, 497
- dump() 139, 147, 148, 247, 496
- echoTriggerableLights() 197
- findItemByName() 532
- findText() 517
- findTextIndex() 506
- forceOnAction() 518
- forceReflow() 498, 551
- get() 543
- getChild() 535
- getClassName() 145, 148, 149, 584
- getColumnCount() 480
- getColumnOffset() 480
- getControlObject() 205
- getCount() 348, 349, 350, 351, 377, 406, 409, 417, 586, 589, 590
- getCursorPos() 503
- getDamageLevel() 164
- getDataBlock() 145, 155, 163, 164, 168
- getExtent() 562, 566
- getEyePoint() 168
- getEyeTransform() 168
- getEyeVector() 168
- getForwardVector() 154
- getGroup() 148, 343, 528
- getId() 119, 144, 145, 148, 250, 251, 348, 349, 350, 413
- getItemText() 532
- getItemValue() 532
- getLineText() 495
- getMountNodeObject() 243
- getMuzzlePoint() 437, 444
- getMuzzleVector() 444
- getName() 145, 148, 255, 355, 384, 388
- getNextSibling() 535
- getNumDetailLevels() 198
- getObject() 348, 349, 417
- getObjectBox() 154, 401
- getParent() 535
- getPathId() 336
- getPosition() 334, 342, 434, 472, 566
- getPoweredState() 184
- getPrevSibling() 535
- getRowCount() 480
- getRowId() 505
- getRowNumById() 505
- getRowOffset() 480
- getRowText() 505
- getRowTextById() 505
- getScale() 152, 314
- getSelected() 517
- getSelectedFile() 524
- getSelectedId() 505, 506
- getSelectedPath() 524
- getSlotTransform() 243
- getState() 417, 582
- getText() 508, 517
- getTextById() 517
- getTransform() 153, 207, 401
- getType() 146, 148, 180, 585
- getValue() 519, 520
- getVelocity() 167, 444
- getWorldBox() 154
- getWorldBoxCenter() 154, 168, 243, 341, 447, 585
- identity() 519
- init() 425
- insertLine() 495
- isActive() 474

console methods (*continued*)

isAwake() 474
 isEOF() 369, 376, 551
 isRotating() 177
 isRowActive() 506
 isStatic() 177
 isVisible() 474, 561
 listObjects() 350
 makeFirstResponder() 461, 462, 473, 482
 mountImage() 174, 175
 mountObject() 173, 174
 moveSelection() 533, 535
 open() 531
 openForAppend() 371
 openForRead() 369, 551
 openForWrite() 370
 pauseThread() 170
 performClick() 508
 PhysicalZone() 128, 129, 334, 342
 playAudio() 172
 playThread() 169, 170, 171
 pop() 250, 362
 popBackLine() 496
 popDialog() 457
 popFrontLine() 496
 push() 362
 pushBackLine() 495
 pushDialog() 457
 pushFrontLine() 495
 pushToBack() 349
 readLine() 369, 551
 reload() 551, 552
 remove() 258, 344, 349, 355, 376, 383, 384, 459, 543
 removeColumn() 480
 removeMenu() 513
 removeRow() 480, 504
 removeRowById() 505
 replaceText() 517
 resize() 472, 497, 558, 562, 566
 rowCount() 480, 505
 save() 148, 361
 scrollToBottom() 482
 scrollToTag() 498
 scrollToTop() 482, 498
 scrollVisible() 506
 select() 516, 518, 535
 setActionThread() 415, 417
 setActive() 474
 setAlarmMode() 197
 setBitmap() 490, 491, 509, 557
 setCloaked() 160
 setCollapsed() 485
 setCollisionTimeout() 180
 setColumnOffset() 480
 setContent() 93, 456, 545, 547, 549
 setControlObject() 205, 207, 581
 setCursor() 521
 setCursorPosition() 503
 setDamageFlash() 165
 setDamageState() 164, 166
 setDataBlock() 155, 156
 setDetailLevel() 198
 setEnergyLevel() 167
 setFlyMode() 203, 207
 setHidden() 407
 setInvincibleMode() 164
 setMenuItemBitmap() 514
 setMenuItemChecked() 515
 setMenuItemEnable() 515
 setMenuItemText() 515
 setMenuItemVisible() 515
 setMenuText() 515
 setMenuVisible() 515
 setName() 148
 setOrbitMode() 203, 207
 setPath() 524
 setPoweredState() 184
 setProfile() 471
 setRechargeRate() 167
 setRepairRate() 163
 setRowActive() 506
 setRowById() 504
 setRowOffset() 480
 setScale() 152, 188, 314
 setSelectedById() 506
 setSelectedPath() 524
 setSelectedRow() 506
 setSkinName() 161, 185
 setText() 497, 498, 501, 508, 517
 setThreadDir() 170
 setTransform() 153, 188, 343, 583
 setValue() 490, 491, 519, 551
 setVelocity() 167, 583
 setVisible() 474, 561
 setWhiteOut() 165

Index

- console methods (*continued*)
 - size() 472, 497, 558, 562, 566
 - sort() 507, 518
 - startFade() 161, 407
 - stopAudio() 172
 - stopThread() 170, 171
 - stormClouds() 284
 - stormFog() 283
 - stormFogShow() 283
 - toggle() 80
 - writeLine() 370, 371
- console objects 115, 133
 - console methods 118
 - dynamic fields 119
 - fields 118
 - handles 118
 - names 118
- control statements 112
 - branching 112
 - for 113
 - if-then-else 112
 - switch 112
 - switch\$ 113
 - while 113
- conversion 396
- convex 17, 18, 19, 158
- CRC 175, 366
- D**
- damage flashes 165
- damaging 162, 163
 - Damage States 163
 - Invincibility 164
 - Visual Feedback 165
- datablocks 29, 127, 133, 149
 - accessing fields 132
 - creating objects with 129
 - declaring 130
- data types 106
 - arrays 109
 - Booleans 108
 - cleaning 399
 - comparisons 398
 - escape sequences 107
 - manipulating 392
 - metrics 396
 - numbers 106
 - searching and replacing 396
 - strings 106
 - string operators 107
 - vectors 110
- debugging
 - dump() 139, 147
 - tree() 139
- DecalManager 426
- Decals 425
- destroying 162
- dialogs
 - popping 457
 - pushing 457
- DIF. *See* Interiors
- disabling 162
- DML 279, 280, 281, 284, 285
- DTS. *See* shapes
- Dynamix 3, 16, 18
- E**
- Earthsiege* 3
- emitters
 - backwardJetEmitter 229
 - damageEmitter 230
 - damageEmitterOffset 230
 - dustEmitter 230
 - dustTrailEmitter 230
 - footPuffEmitter 215
 - forwardJetEmitter 229
 - numDmgEmitterAreas 230
 - particleEmitter 304, 305, 315, 316, 317, 341, 344, 430, 436, 440, 445, 446
 - splashEmitter 216
 - stateEmitter 192
 - stateEmitterNode 192
 - stateEmitterTime 192
 - tireEmitter 230
 - trailemitter 229, 230
 - useEmitterColors 306
- energy 166
- environmental mapping 160
- events 386
 - accuracy 390
 - cancelling 390
 - checking for 389
 - repeating 391
 - scheduling 387, 388
 - times 389

explosions 162, 166, 427
 eyeOffset 190
 eyeRotation 190

F

fields 395
 field of view (FOV) 205
 files
 appending to 371
 calculating CRC 366
 counting 366
 Dot (.) versus Slash (/) versus Tilde (~) 367
 expanding names 367
 extracting name 367
 extracting path 367
 extracting prefix 368
 extracting suffix 368
 filename wildcards 366
 locating 364
 overwriting 370
 reading 368, 369
 writing 368, 370
 file I/O 364
 firstPerson 190
 fog 282
 general 282
 layers 282
 forces and factors 217
 forward vector 154
 fxFoliageReplicator 318
 fxLight 335
 fxShapeReplicator 318
 fxSunLight 326

G

games
 3-D Language Spain 5
 dRacer 5
 Earthsiege 3
 Golden Fairway 5
 Lore 4
 Marble Blast GOLD 4
 Minions Of Mirth 6
 Orbz 4
 RocketBowl Plus 5
 Starsiege 3
 Think Tanks 4

Tribes 1 & 2 3, 64, 99, 197, 268, 273, 401
 getType() 146
 Type Masks 145
 globals
 \$Camera::movementSpeed 203, 207
 \$cameraFov 203, 204, 208
 \$movementSpeed 203, 207, 221
 \$mvBackwardAction 221
 \$mvDownAction 221
 \$mvForwardAction 221
 \$mvFreeLook 209
 \$mvLeftAction 221
 \$mvPitch 222
 \$mvPitchDownSpeed 222
 \$mvPitchUpSpeed 222
 \$mvRightAction 221
 \$mvTriggerCount0-\$mvTrigger-
 Count5 232, 236, 241
 \$mvUpAction 221
 \$mvYaw 222
 \$mvYawLeftSpeed 222
 \$mvYawRightSpeed 222
 \$pref::Decal::decalTimeout 426
 \$pref::Decal::maxNumDecals 426
 \$pref::decalsOn 426
 \$pref::Input::KeyboardTurn-
 Speed 222
 \$pref::Interior::detailAdjust 198
 \$pref::Net::PacketRateTo-
 Client 595
 \$pref::Net::PacketRateTo-
 Server 595
 \$pref::Net::PacketSize 595
 \$pref::Terrain::enableEmboss-
 Bumps 266
 \$thisControl 472
 gravity 179, 180, 182, 183, 241, 304, 305, 316, 333, 422, 439, 445
 GUI
 accelerators 472
 active 474
 autosizing 469
 awake 474
 background color 465
 bitmap arrays 463

Index

GUI (continued)

- borders 464
- commands 472
- cursors 464
- extent 471
- first responder 473
- fonts 465
- key and mouse attributes 469
- margins 481
- modifiers 527
- mouse events 525
- position 471
- profiles 470
- scrollbars 481
- size 471
- skinning 476, 482, 484, 487, 492,
509, 510, 512, 523
- text formatting 468
- Torque Markup Language (TorqueML)
499
- variables 473
- visibility 472, 474

I

I/O

- file 364
- images 189
- image file lists (IFLs) 21
- impulses 167
- interiors 17. *See also* Classes: Interior-
Instance
 - level of detail (LOD) 198
- inventories 243

L

Lessons

- #1—Terrain for Our Game 72
- #2—Loading Datablocks 132
- #3—Game Coins 181
- #4—Fade and Fireball Blocks 184
- #5—Maze Blocks 188
- #6—Simplest Player 223
- #7—Preparing Our Game Inventory
256
- #8—Lava in the Cauldron 278
- #9—Starry Night 284
- #10—Low Lighting 288
- #11—Stormy Weather 294
- #12—Teleport Station Effect 315

- #13—Celestial Bodies 332
- #14—Teleport Stopper 334
- #15—Teleport Triggers 340
- #16—MoveMap 363
- #17—Level Loader 371
- #18—Game Events 406
- #19—FireBall Explosion 434
- #20—The FireBall 444
- #21—Game Sounds 450
 - About 11
- level of detail (LOD) 19, 198
- lightning 288
- lights and lighting 191, 285
 - constantLight 176, 191
 - Interiors 197
 - lightColor 176, 191, 440
 - lightRadius 176, 191, 335, 440
 - lightTime 191
 - lightType 176, 182, 191
 - noLight 176, 182, 191
 - pulsingLight 176, 191

M

math 400

- absolute value 402
- addition 402
- ceiling 402
- centroids 404
- conversion
 - degrees to radians 402
 - radians to degrees 402
- cosine 402
- creation 403
- creation (from Euler angles) 403
- cross Product 402
- cubics 403
- distance (between) 402
- dot product 402
- floor 402
- inverse cosine 402
- inverse tangent 402
- length 402
- logarithm 402
- modifying mantissas 405
- multiplication 403
- normalization 402
- orthographic basis 402
- point multiplication 403
- power 402

- math (*continued*)
 - quadratics 403
 - random numbers 404
 - scaling 402
 - sine 402
 - square root 402
 - subtraction 402
 - tangent 402
- meshes
 - collision-0–collision-8 185, 228
 - LOSCol-9–LOSCol-16 228
- Mesh Nodes
 - cam 165, 204, 208, 210, 223, 224, 229
 - chassis 228, 236, 260
 - contrail0–contrail3 229, 230
 - eye 204, 208, 210, 223, 224, 229
 - hub0–hub7 229
 - JetNozzle0–JetNozzleX 229, 230
 - mount0–mount31 172, 173, 174, 229, 242
 - Tire 228, 237, 238, 239, 240, 260, 261
- mirrors 198, 199, 214
- missions 22
- mounting 172, 191
 - alternate positions (vehicles) 242
 - image-to-shape 174
 - mountPoint 191
 - nodes 172
 - offset 191
 - rotation 191
 - shape-to-shape 173
 - slots 172
 - vehicle 233
- movement 217, 221
- N**
 - namespaces 126, 133
 - building 149
 - chaining 149
 - inheritance 150
 - rules 149
 - scope 151
 - networking
 - client-server 24
 - communications 27
 - control object 28
 - division of labor 27
 - ghosts 28
 - scope 28
- O**
 - objects 28
 - objects (console) 115
 - operators
 - string comparisons 111
- P**
 - packages 122
 - particles 302
 - paths 336
 - performance
 - culling replicators 323
 - physical zones 333
 - portals 20
 - position 152
 - POV cookbook 210
 - precipitation 288
- R**
 - random numbers 404
 - records 394
 - render bans 280
 - repairing 162, 163
 - replicators 318
 - rotation 152
- S**
 - scale 152
 - scales
 - over vertex brush scale 58
 - selected brush scale 58
 - ShapeBaseImageData
 - animations 195, 196
 - running scripts 195
 - shapes 16, 157
 - skinning 487
 - skins (shape) 161
 - multi-skinning naming convention 161
 - sky
 - visibility 282
 - sky box 279, 280
 - sound 172
 - 2D 22, 297
 - 3D 22, 299
 - AudioDescription 448
 - AudioProfile 448
 - Audio Emitters 296
 - special effects 31

Index

- squareSize 267, 268, 270, 271
- Starsiege* 3
- state machines 192
 - defining 193
 - doing work 194
 - running animations 195
 - running scripts 195
 - transitioning 193
- strings
 - cleaning 399
 - comparisons 398
 - manipulating 392
 - metrics 396
 - searching and replacing 396
- Sun 285
- T**
- Terrain 263
- ticks 24
- tokens 393
- TorqueScript
 - built-in functions 103
- transforms 153, 168
 - getEyeTransform() 168
 - getEyeVector() 168
 - getForwardVector() 154
 - getPosition() 334, 342, 434, 472, 566
 - getScale() 152, 314
 - getTransform() 153, 207, 401
 - object boxes 151, 154, 401
 - setScale() 152, 188, 314
 - setTransform() 153, 188, 343, 583
 - world boxes 154
- Tribes 1 & 2* 3, 99, 197, 268, 273, 401
- triggers 338
 - group 340
- type masks 145
- U**
- Unicode 467, 468
- V**
- vehicles 227
 - animations 228
- velocity 167, 178
 - getVelocity() 167, 444
 - maxVelocity() 178, 179, 221, 290
 - setVelocity() 167, 583
- visibility 282
- W**
- water 269
 - flowing 274
 - reflections 276
 - shoreline 275
 - types 273
 - waves 272
- words 392
- Z**
- zooming 205