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

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

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.

Part I Introduction

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.

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	
 *Marble Blast GOLD*	 *Think Tanks*
 *Lore*	 *Orbz*

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.

- **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

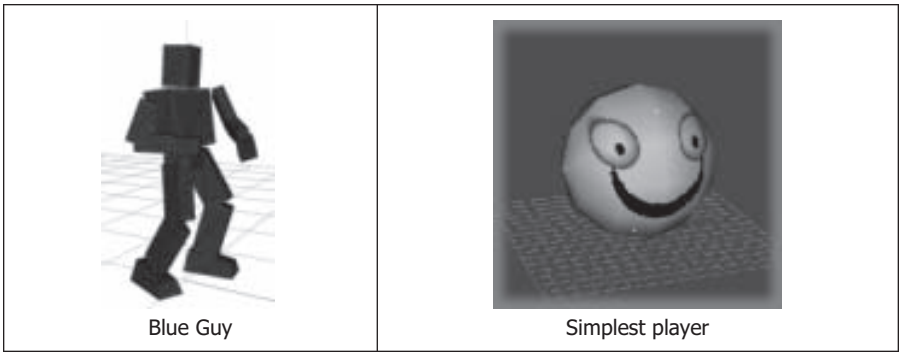


Figure 2.1.
Simple Player shapes.

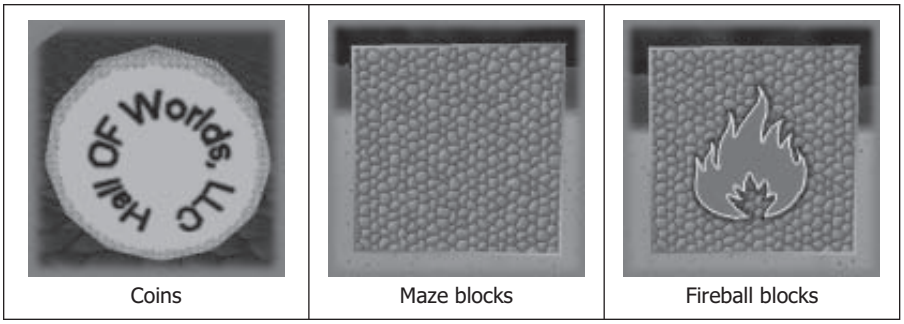


Figure 2.2.
Required shapes and blocks.

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.

- **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).

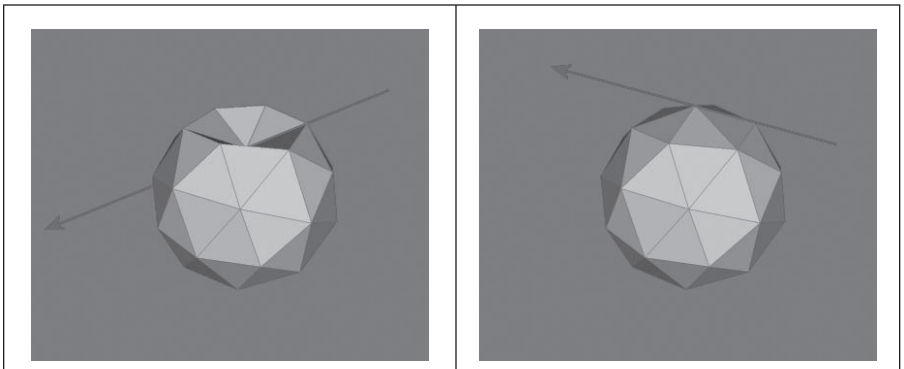


Figure 2.3.
Using line segments to discover concavity.

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).

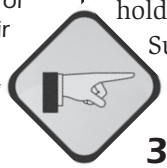
Tools	Start Tool	Description
Mission Area Editor (**Area Editor**)	**F5**	This tool allows you to adjust the boundaries of the current mission and provides a means to mirror the current terrain.
Terrain Editor	**F6**	This tool provides the ability to directly manipulate the terrain using the mouse as 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.

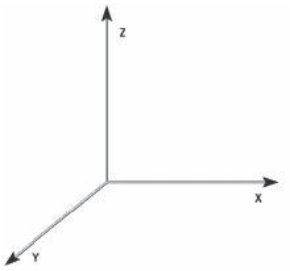
3.3.4 The Gizmo and Gizmo Scales

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.

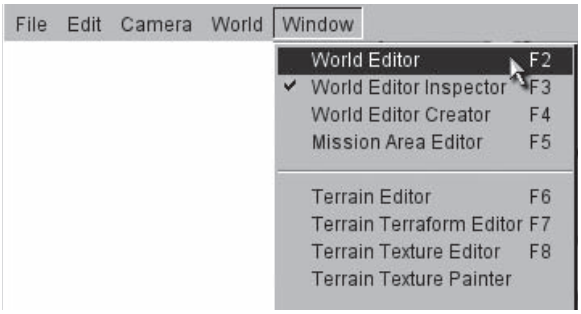
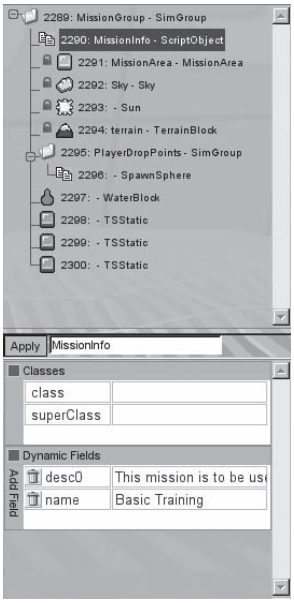


Figure 3.3.

Tool windows.



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.

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.

- **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.

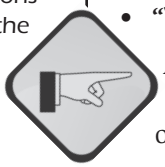
- **Level loading.** To make this game easily maintainable, tunable, and modifiable by players, all level loading is controlled by a text file (the level file). Players can add new levels and redefine levels to their hearts' content.

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.

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