Fundamentals of Game Design

ERNEST ADAMS

ANDREW ROLLINGS

PEARSON
Prentice
Hall

Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data

Adams, Ernest.
Fundamentals of game design/Ernest Adams and Andrew Rollings.
p. cm.
ISBN 0-13-168747-6 (alk. paper)
1. Computer games—Design. 2. Computer games—Programming. 3. Video games—Design.
I. Rollings, Andrew, 1972- II. Title.
QA76.76.C672A322 2006
794.8'1536—dc22

2006023805

Vice President and Publisher: Natalie E. Anderson
Associate VP/Executive Acquisitions Editor, Print: Stephanie Wall
Executive Acquisitions Editor, Media: Richard Keaveny
Executive Acquisitions Editor: Chris Katsaropoulos
Product Development Manager: Eileen Bien Calabro
Editorial Supervisor: Brian Hoehl
Editorial Assistants: Rebecca Knauer, Kaitlin O'Shaughnessy
Executive Producer: Lisa Strite
Content Development Manager: Cathi Profitko
Senior Media Project Manager: Steve Gagliostro
Project Manager, Media: Alana Meyers
Director of Marketing: Margaret Waples
Senior Marketing Manager: Jason Sakos
Marketing Assistant: Ann Baranov
Senior Sales Associate: Joseph Pascale
Managing Editor: Lynda J. Castillo
Production Project Manager: Lynne Breitfeller
Manufacturing Buyer: Chip Poakeart
Production/Editorial Assistant: Sandra K. Bernales
Design Manager: Maria Lange
Art Director/Interior Design/Cover Design: Blair Brown
Cover Illustration/Photo: Northern Electrics
Composition: Integra
Project Management: BookMasters, Inc.
Cover Printer: RR Donnelley/Harrisonburg
Printer/Binder: RR Donnelley/Harrisonburg

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text.

Pearson Education LTD.
Pearson Education Singapore, Pte. Ltd
Pearson Education, Canada, Ltd
Pearson Education–Japan
Pearson Education Australia PTY, Limited
Pearson Education North Asia Ltd
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education Malaysia, Pte. Ltd

PEARSON
Prentice Hall

10 9 8
ISBN 0-13-168747-6

Chapter 8

Creating the User Experience

Chapter Objectives

After reading this chapter and completing the exercises, you will be able to do the following:

- Explain how a game's user interface mediates between the player and the game's core mechanics to create the user experience.
- Discuss how principles of player-centric interface design can answer questions about what the player needs to know and wants to do.
- Know the basic steps required to design a game's user interface.
- List options that can help to control a game's complexity.
- Describe the five well-known interaction models.
- List the most commonly used game perspectives and discuss their advantages and disadvantages.
- Describe how visual elements such as the main view and feedback elements supply information a player needs to know to succeed in the game.
- Explain how audio elements such as sound effects and music affect the user experience.
- Know the types of one-dimensional and two-dimensional input devices and discuss how they affect the game experience.
- List the most commonly used navigation systems and explain how each system controls action in a game.

Introduction

The user interface brings the game to the player, taking the game from inside the computer and making it visible, audible, and playable. It creates the player's experience and, as such, has an enormous effect on whether she perceives the game as satisfying or disappointing, elegant or graceless, fun or frustrating.

In this chapter, we first discuss the general principles of user interface design and propose a process for designing your interface, along with some observations about how to manage its complexity. Then we define two key concepts related to game interfaces: *interaction models* and *perspectives.* We then delve into specifics, examining some of the most widely used visual and audio elements in video games and analyzing the functionality of various types of input devices. Because the overwhelming majority of video games include some notion of moving characters or vehicles around the game world, we turn next to consider a variety of navigation mechanisms as they are implemented in different perspectives and with different input devices. The chapter concludes with a few observations on how to make your game customizable.

What Is the User Experience?

What works is better than what looks good. The looks good *can change, but what works, works.*

—Ray Kaiser Eames, Designer and Architect

We've used the term *user experience* in the title of this chapter to emphasize the fact that a game's user interface (UI) actually presents the entertainment experience to the player. (For this reason, the user interface is often also called the *presentation layer.*) As we showed you in Figure 2.1, the user interface lies between the player and the internals of the game—that is, the core mechanics and the storytelling engine, which together contain all the data about the game's story, its rules, and its current state. The core mechanics and the storytelling engine should know nothing about the input and output devices of the machine—that way the internals of the game are independent of the hardware and can be more easily ported to another machine. The user interface's role is to take a portion of the internal data and present it to the player in visible and audible forms. The UI also takes the player's button-presses in the real world and interprets them as actions in the game world, passing on those actions to the core mechanics.

A game's user interface plays a more complex role than does the UI of most other kinds of programs. Most computer programs are tools, so their user interfaces allow the user to enter and create data, to control processes, and to clearly see the results. As part of a tool, the user interface must offer the user maximum control, information, and flexibility. A video game, on the other hand, exists to entertain, and while its user interface must be easy to learn and use, it doesn't tell the player everything that's happening inside the game, nor does it give the player maximum control over the game. It mediates between the internals and the player, creating an experience for the player that feels to him like gameplay and storytelling.

The user interface also implements two important mechanisms we introduced in Chapter 1, "Games and Video Games": the *interaction model,* which determines how the player interacts with the game world, and the *perspective,* which determines how the player sees the game world (that is, how the game's camera behaves).

In a more concrete sense, the user interface produces the game software's outputs on the machine's output devices and accepts the game software's inputs from the machine's input devices. The outputs traditionally consist of pictures and sound; the inputs, of button-presses and the movements of a joystick or mouse. In this chapter, we refer to the outputs as the *visual elements* and *audio elements* of the user interface and to the inputs as the *control elements.* When the game gives important information to the player about his activities, the state of the game world, or the state of his avatar (such as health or money remaining), we say that it gives ***feedback*** to the player—that is, it informs him of the effects of his actions. The visual and audio elements of the user interface that provide this information, we call ***feedback elements.***

FYI *Terminology Issues*

The term *button* is unfortunately overloaded, as it sometimes refers to a physical button on an input device that the player can press and at other times refers to a visual element on the screen, drawn to look like a button, which the player can click with the mouse. In order to disambiguate the two, in this chapter we will always refer to physical buttons on an input device as *controller buttons* and those on the screen, triggered by the mouse, as *screen buttons. Keys* refers to keys on a computer keyboard. We use the term *key* interchangeably with *controller buttons* because they both transmit the same type of data.

Menus and screen buttons both appear on the screen as visual elements, but clicking on them with the mouse sends a message to the internals of the game, which makes them control elements as well. Furthermore, the appearance of a screen button may change in response to a click, making it a mechanism for giving information as well as for exercising control. We rely on your experience with computers to understand from context what we're talking about when we write of these things.

Any discussion of user interface design runs into a chicken-and-egg problem: We can't tell you how to design a good user interface without referring to common visual elements such as power bars and gauges, and we can't introduce the common visual elements without making references to how they're used. In order to address the most critical information first, we've chosen to start with the principles of interface design. If you encounter a reference to an interface element you've never heard of, see the section *Visual Elements* later in this chapter for an explanation.

Dozens, perhaps hundreds, of published books address user interface design, and we do not try to duplicate all that material here. We concentrate specifically on user interfaces for games, how they interact with the game's mechanics, and how they create the entertainment experience for the player. To read more about user interfaces in general, see *The Elements of User Experience* by Jesse James Garrett (Garrett, 2003).

Player-Centric Interface Design

The player-centric approach taught in this book applies to user interface design, as it does to all aspects of designing a game. Therefore, we keep discussion tightly focused on what the player needs to play the game well and how to create as smooth and enjoyable an experience as possible.

About Innovation

While we encourage innovation in almost all aspects of game design—theme, game worlds, storytelling, art, sound, and of course gameplay—we urge caution about innovation in user interface design. Although you will want your player to be impressed by the originality of your gameplay, the player will almost certainly prefer a familiar UI. Over the years, most genres have evolved a practical set of feedback elements and control mechanisms suited to their gameplay. We encourage you to research these systems by playing other games in your chosen genre and to adopt whichever of them you find appropriate for your game. If you force the player to learn an unfamiliar user interface when a perfectly good one already exists, you frustrate the player and reduce his enjoyment of the game.

If you do choose to offer a new user interface for a familiar problem, be sure to allow the player to customize it in case he doesn't like it. We address this further in *Allowing for Customization,* near the end of this chapter.

Some General Principles

The following general principles for user interface design apply to all games regardless of genre:

- **Be consistent.** This applies to both aesthetic and functional issues; your game should be stylistically as well as operationally consistent. If you offer the same action in several different gameplay modes, assign that action to the same controller button or menu item in each mode. The names for things that appear in indicators, menus, and the main view should be identical in each location. Your use of color, capitalization, typeface, and layout should be consistent throughout related areas of the game.

- **Give good feedback.** When the player interacts with the game, he will expect the game to react—at least with an acknowledgment—immediately. When the player presses any screen button, the game should produce an audible response even if the button is inactive at the time. An active button's appearance should change either momentarily or permanently to acknowledge the player's click.
- **Remember that the player is the one in control.** Players want to feel in charge of the game—at least in regard to control of their avatars. Don't seize control of the avatar and make him do something the player may not want. The player can accept random, uncontrollable events that you may want to create in the game world or as part of the behavior of nonplayer characters, but don't make the avatar do random things the user didn't ask him to do.
- **Limit the number of steps required to take an action.** Set a maximum of three controller-button presses to initiate any special move unless you need combo moves for a fighting game (see Chapter 13, "Action Games"). The casual gamer's twitch ability tops out at about three presses. Similarly, don't require the player to go through menu after menu to find a commonly used command. (See *Depth versus Breadth* later in the chapter for further discussion.)
- **Permit easy reversal of actions.** If a player makes a mistake, allow him to undo the action unless that would affect the game balance adversely. Puzzle games that involve manipulating items such as cards or tiles should keep an undo/redo list and let the player go backward and forward through it, though you can set a limit on how many moves backward and forward the game permits.
- **Minimize physical stress.** Video games famously cause tired thumbs, and unfortunately, repetitive stress injuries from overused hands can seriously debilitate players. Assign common and rapid actions to the most easily accessible controller buttons. Not only do you reduce the chance of injuring your player, but you allow him to play longer and to enjoy it more.
- **Don't strain the player's short-term memory.** Don't require the player to remember too many things at once; provide a way for him to look up information that he needs. Display information that he needs constantly in a permanent feedback element on the screen.
- **Group related screen-based controls and feedback mechanisms on the screen.** That way, the player can take in the information he needs in a single glance rather than having to look all over the screen to gather the information to make a decision.

- **Provide shortcuts for experienced players.** Once players become experienced with your game, they won't want to go through multiple layers of menus to find the command they need. Provide shortcut keys to perform the most commonly used actions from the game's menus, and include a key reassignment feature. See the section *Allowing for Customization* at the end of the chapter.

What the Player Needs to Know

Players naturally need to know what's happening in the game world, but they also need to know what they should do next, and most critically, they need information about whether their efforts are succeeding or failing, taking them closer to victory or closer to defeat. In this section, we look at the information that the game must present *to* the player to enable her to play the game. Notice that in keeping with a player-centric view of game design, we discuss this mostly in terms of questions the player would ask.

Where am I? Provide the player with a view of the game world. We call this visual element the *main view.* If she can't see the whole world at one time (as she usually can't), also give her a map or a mini-map that enables her to orient herself with respect to parts of the world that she can't currently see. You should also provide audio feedback from the world: ambient sounds that tell her something about her environment.

What am I actually doing right now? To tell the player what she's doing, show her her avatar, party, units, or whatever she's controlling in the game world, so she can see it (or them) moving, fighting, resting, and so on. If the game uses a first-person perspective, you can't show the player's avatar, so show her something from which she can infer what her avatar is doing: If her avatar climbs a ladder, the player sees the ladder moving downward as she goes up. Here again, give audio feedback: Riding a horse should produce a clop-clop sound; walking or running should produce footsteps at an appropriate pace. Less concrete activities, such as designating an area in which a building will be constructed, should also produce visible and audible effects: Display a glow on the ground and play a definitive *clunk* or similar sound.

What challenges am I facing? Challenges such as puzzles, combat, or steering a vehicle can be shown directly in the main view of the game world; display the corridor of the maze, the onrushing barbarians, the road ahead, and so on. Some challenges make noise: Monsters roar and boxers grunt. To show conceptual or economic challenges, you may need text to explain the challenge; for example, "You must assemble all the clues and solve the mystery by midnight."

Did my action succeed or fail? Show animations and indicators that display the consequences of actions: The player punches the bad guy and the bad guy falls down; the player sells a building and the money appears in her inventory. Accompany these consequences with suitable audio feedback for both success and failure: a whack sound if the player's punch lands and a whiff sound if the player's punch misses; a ka-ching! when the money comes in.

Do I have what I need to play successfully? The player must know what resources she can control and expend. Display indicators for each: ammunition, money, mana, and so on.

Am I in danger of losing the game? Show indicators for health points, power, time remaining before a timed challenge ends, or any other resource that must not be allowed to reach zero. Use audio signals—alarms or vocal warnings—to alert the player when one of these commodities nears a critical level.

Am I making progress? Show indicators for the score, the percentage of a task completed, or the fact that a player passed a checkpoint.

What should I do next? Unless your game provides only a sandbox-type game world in which the player can run around and do anything she likes in any order, players need guidance about what to do. You don't need to hold their hands every step of the way, but you do need to make sure they always have an idea of what the next action could or should be. Adventure games sometimes maintain a list of people for the avatar to talk to or subjects to ask NPCs about. Race courses over unfamiliar territory often include signs warning of curves ahead.

How did I do? Give the player emotional rewards for success and (to a lesser extent) disincentives for failure through text messages, animations, and sounds. Tell her clearly when she's doing well or badly and when she has won or lost. When she completes a level, give her a debriefing: a score screen, a summary of her activities, or some narrative.

Do Not Taunt the Player

A few designers think it's funny to taunt or insult the player for losing. This is mean-spirited and violates a central principle of player-centric game design—the duty to empathize. The player will feel bad about losing anyway. Don't make it worse.

What the Player Wants to Do

Just as the player needs to know things, the player wants to do things. You can offer him many things to do depending upon the game's genre and the current state of the game, but some actions crop up so commonly as to seem almost universal. We list some extremely common actions here.

Move. The vast majority of video games include travel through the game world as a basic player action. How you implement movement depends on your chosen interaction model and perspective. You have so many different options that we devote a section, *Navigation Mechanisms,* to movement later in this chapter.

Look around. In most games, the player cannot see the whole game world at one time. In addition to moving through the world, he needs a way of adjusting his view of the world. In avatar-based games, he can do this through the navigation mechanism (see *Navigation Mechanisms*). In games using multipresent and other interaction models that provide aerial perspectives, give him a set of controls that allow him to move the virtual camera to see different parts of the world.

Interact physically with nonplayer characters. In games involving combat, this usually means attacking nonplayer characters, but interaction can also mean giving them items from the inventory, carrying or healing them, and many other kinds of interactions.

Pick portable objects up and put them down. If your game includes portable objects, implement a mechanism for picking them up and putting them down. This can mean anything from picking up a chess piece and putting it down elsewhere on the board to a full-blown inventory system in a role-playing game in which the player can pick up objects in the environment, add them to the inventory, give them to other characters, buy them, sell them, or discard them again. Be sure to include checks to prevent items from being put down in inappropriate places (such as making an illegal move in chess). Some games do not permit players to put objects down, in order to prevent the players from leaving critical objects behind.

Manipulate fixed objects. Many objects in the environment can be manipulated in place but not picked up, such as light switches and doors. For an avatar-based game, design a mechanism that works whenever the avatar is close enough to the object to press it, turn it, or whatever might be necessary. In other interaction models, let the player interact more directly with fixed objects by clicking on them. You can simplify this process by giving fixed objects a limited number of states through which they may be rotated: a light switch is on or off; curtains are fully open, halfway open, or closed.

Construct and demolish objects. Any game that allows the player to build things needs suitable control mechanisms for choosing something to build or materials to build with, selecting a place to build, and demolishing or disassembling already-built objects. It also requires feedback mechanisms to indicate where the player may and may not build, what materials he has available, and if appropriate, what it will cost. You should also include controls for allowing him to see the structure in progress from a variety of angles. For further discussion of construction mechanisms, see Chapter 18, "Construction and Management Simulations."

Give orders to units or characters. Players need to give orders to units or characters in many types of games. Typically this requires a two- or three-step process: designating the unit to receive the order, giving the order, and optionally giving the object of the order, or *target*. Orders take the form of verbs, such as *attack, hug, open,* or *unload,* and targets take the form of direct objects for the verbs, such as *thug, dog, crate,* or *truck,* indicating what the unit should attack, hug, open, or unload.

Conduct conversations with nonplayer characters. Video games almost always implement dialog with NPCs as *scripted conversations* conducted through a series of menus on the screen. See *Scripted Conversations and Dialog Trees* in Chapter 7, "Storytelling and Narrative."

Customize a character or vehicle. If your game permits the player to customize his character or vehicle, you will have to provide a suitable gameplay mode or shell menu for it. The player may want to customize visible attributes of avatar characters, such as hair, clothing, and body type, as well as invisible ones, such as dexterity. Players like to specify the color of the vehicles they drive, and they need a means of adjusting a racing car's mechanical attributes because this directly affects its performance.

Talk to friends in networked multiplayer games. Multiplayer online games must give players opportunities to socialize. Build these mechanisms though chat systems and online bulletin boards or forums.

Pause the game. With the exception of arcade games, any single-player game must allow the player to pause the action temporarily.

Set game options. Outside the game world, the player may want to set the game's difficulty level, customize the control assignments (see *Allowing for Customization* later in this chapter), or adjust other features such as the behavior of the camera. Build shell menus to allow the player to do this.

Save the game. All but the shortest games must give the player a way of stopping the game and continuing from the same point when the player next starts up the game software. See *Saving the Game* in Chapter 9, "Gameplay."

End the game. Don't forget to include a way to quit!

The Design Process

You will recall that in Chapter 2, "Design Components and Processes," we divided the game design process into three stages: concept, elaboration, and tuning. Designing the user interface takes place early during the elaboration stage. There's no point in designing it any earlier; if you do so before the end of the concept phase, the overall design may change dramatically and your early UI work will be wasted.

In this section, we give the steps of the UI design process. Remember that definitions for many of the components you will use to design your game's UI can be found later in this chapter.

Define the Gameplay Modes First

A gameplay mode consists of a camera perspective, an interaction model, and the gameplay (challenges and actions) available. A lead designer generally defines the gameplay modes and, in a commercial development team, would then hand them off to a UI designer. Still, because the UI designer must implement the lead designer's ideas for camera perspectives and interaction models, the task of gameplay mode definition cannot be entirely separated from that of UI design.

In any case, the first job in the elaboration stage will be to design the *primary* gameplay mode, the one in which the player spends the majority of her time. We discuss gameplay in Chapter 9, "Gameplay." See *Interaction Models* and *Perspectives* later in this chapter for details about each of them. Once you have chosen (or been given) the perspective, interaction model, and gameplay for the primary gameplay mode, you can begin to create the details of the user interface for that mode.

When you have designed the primary gameplay mode, move on to the others that you think your game will need. Plan the structure of the game using a flowboard, as described in Chapter 2, "Design Components and Processes." In addition to gameplay activities, don't forget story-related activities. Design modes for delivering narrative content and engaging in dialog if your game supports it. Be sure to include a means to interrupt narrative and get back to gameplay.

If your game provides a small number of gameplay modes (say, five or fewer), you can start work on the user interfaces as soon as you decide what purpose each mode serves and what the player will do there. However, if the game provides a large number of modes, then you should wait until *after* you have planned the structure of the game and you understand how the game moves from mode to mode. Gameplay modes do not typically use completely different user interfaces but share a number of UI features, so it's best to define all of the modes before beginning UI work.

Once you have the list of gameplay modes, start to think about what visual elements and controls will be needed in each. Using graph paper or a diagramming tool such as Microsoft Visio, make a flowchart of the progression of menus, dialog boxes, and other user interface elements that you intend to use in each mode. Also document what the input devices will do in each.

You will usually need to define a different user interface for each gameplay mode your game offers. Occasionally gameplay modes can share a single UI when the modes differ only in the challenges they offer. If you want to allow the player to control the change from one mode to another, your user interface must offer commands to accomplish these mode changes.

Steps in designing a game's user interface include, for each mode, designing a screen layout, selecting the visual elements that will tell the player what she needs to know, and defining the inputs to make the game do what she wants to do. We'll take up these topics in turn. Throughout the remainder of this discussion, we'll assume that you're working on the user interface for the most important mode—the primary gameplay mode—although our advice applies equally to any mode.

Build a Prototype UI

Experienced designers always build and test a prototype of their user interfaces before designing the final specifications for the real thing. When you have the names and functions of your UI elements for a mode worked out, you can begin to build a prototype using placeholder artwork and sounds so that you can see how your design functions. Don't spend a lot of time creating artwork or audio on the assumption that you'll use it in the final product; you may have to throw it away if your plans change. Plenty of good tools allow interface prototyping including graphics and sound with minimal programming required. You can make very simple prototypes in Microsoft PowerPoint using the hyperlink feature to switch between slides. Macromedia Flash offers more power, and if you can do a little programming, other game-making tools such as Blitz Basic **(www.blitzbasic.com)** will let you construct a prototype interface.

Your prototype won't be a playable game but will only display menus and screen buttons and react to signals from input devices. It should respond to these as accurately as possible given that no actual game software supports it. If a menu item should cause a switch to a new gameplay mode, build that in. If a controller button should shoot a laser, build the prototype so that at least it makes a *zap* noise to acknowledge the button press.

As you work and add additional gameplay modes to the prototype, keep testing to see if it does what you want. Don't try to build it all at once; build a little at a time, test, tune, and add some more. The finished prototype will be invaluable to the programming and art teams that will build the real interface.

Choosing a Screen Layout

Once you have a clear understanding of what the player does in the primary (or any) mode and you've chosen an interaction model and a perspective, you must then choose the general screen layout and the visual elements that it will include.

All on-screen UI features will be oriented on and around the main view of the game world. The main view will be the largest visual element on the screen, but you must decide whether it will occupy a subset of the screen—a window—or whether it will occupy the entire screen and be partially obscured by overlays. See *Main View,* later in the chapter, for more information about your options.

During our research for this chapter, we examined more than 2,000 screen captures taken from games produced over the past 25 years. Nine designs occurred especially frequently (counting all symmetric variations as one design).

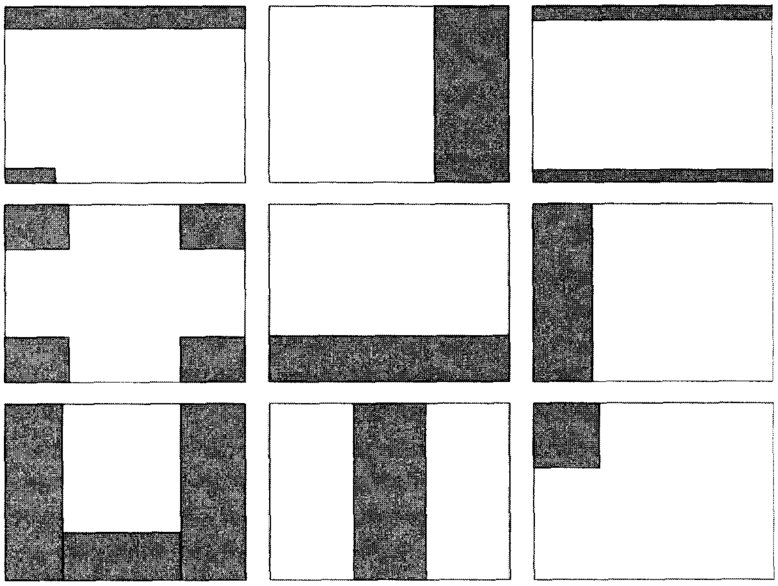

FIGURE 8.1 Common screen layouts.

Figure 8.1 shows how they look. White areas indicate the main view of the game world, and gray areas indicate feedback elements and onscreen controls. Any one of these would make a good start for the layout of your user interface.

You will need to find a balance between the amount of screen space that you devote to the main view and the amount that you devote to feedback elements and onscreen controls. Fortunately this seldom presents a problem in personal computer and console games, which use high-resolution screens. It remains a serious challenge for handheld devices and a very serious one indeed for mobile phones, which do not yet have standardized screen sizes and shapes.

Telling the Player What He Needs to Know

What, apart from the current view of the game world, does the player need to see or to know about? What critical resources does he need to be aware of at all times, and what's the best way to make that information available to him? Select the data from your core mechanics that you want to show, and choose the feedback elements most suited to display those kinds of data using the list in *Feedback Elements,* later in this chapter, as a guide. Also ask yourself what warnings the player may need and then decide how to give both visual and audible cues. Use the general list we gave you in *What the Player Needs to Know* earlier in this chapter, but remember that the gameplay you offer might dictate a slightly different list and that your game may include unique elements that have never been used before.

Once you have defined the critical information, move on to the optional information. What additional data might the player request? A map? A different viewpoint on the game world? Think about what feedback elements would best help him obtain these things and how to organize access to such features.

Throughout this process, keep the general principles of good user interface design in mind; test your design against the general principles listed in *Some General Principles* earlier in this chapter.

Letting the Player Do What She Wants to Do

Now you can begin devising an appropriate control mechanism to initiate every action the player can take that affects the game (whether within the game world or outside of it, such as saving the game). Refer to the list provided earlier in *What the Player Wants to Do* to get started.

What key actions will the player take to overcome challenges? Refer to the genre chapters in this book, as these mention special UI concerns for each genre. What other actions unrelated to challenges might she need: moving the camera, participating in the story, expressing herself, or talking to other players online? Create visual and audible feedback for the actions to let the player know if these succeeded or failed.

You'll need to map the input devices to the player's actions in the game based on the interaction model you have chosen (see *Interaction Models* later in this chapter). Games vary too much for us to tell you exactly how to achieve a good mapping; study other games in the same genre to see how they use onscreen buttons and menus or the physical buttons, joysticks, and other gadgets on control devices. Use the latter for player actions for which you want to give the player the feeling that she's acting directly in the game without mediation by menus. Whenever possible, borrow tried-and-true techniques to keep it all as familiar as possible.

Work one gameplay mode at a time, and every time you move to a new gameplay mode, be sure to note the actions it has in common with other modes and keep the control mechanisms consistent.

Shell Menus

Shell menus allow the player to start, configure, and otherwise manage the operation of the game before and after play. The screens and menus of the shell interface should allow the player to configure the video and audio settings and the game controls (see *Allowing for Customization* later in the chapter), to join in multiplayer games over a network, to save and load games, and to shut down the game software.

The player should not have to spend much time in the shell menus. Provide a means to let players get right into the action by one or two clicks of a button.

A surprising number of games include awkward and ugly shell menus because designers assumed that creating these screens could wait until the last minute. Remember, the shell interface is the first thing your player will see when he starts up the game. You don't want to make a bad impression before the player even gets into the game world.

Managing Complexity

As game machines become more powerful, games themselves become increasingly complex with correspondingly complex user interfaces. Without a scheme for managing this complexity, you can end up with a game that players find extremely difficult to play—either because no one can remember all the options (as with some flight simulators) or because so many icons and controls crammed onto the screen (as in some badly designed strategy games) leave little room for the main view of the game world. Here we discuss some options for managing your game's complexity.

Simplify the Game

This option should be your first resort. If your game is too complex, make it simpler. You may do this in two ways: *abstraction* and *automation*.

Abstraction When you ***abstract*** some aspect of a complicated system, you remove a more accurate and detailed version of that aspect or function and replace it with a less accurate and detailed version or no version at all. This makes the game less realistic but easier to play. If the abstracted feature required UI control or feedback mechanisms, you may save yourself the trouble of designing them.

Many driving games don't simulate fuel consumption; the developers abstracted this idea out of the game. They don't pretend that the car runs by magic—the player can still hear the engine—but they just don't address the question. Consequently, the user interface needs no fuel gauge and no means of putting fuel in the car. The player doesn't have to think about these things, which makes the game easier to play.

Automation When you *automate* a process, you remove it from the player's control and let the computer handle it for her. When the game requires a choice of action, the computer chooses, thus simplifying the game. Note that this isn't the same as abstraction because the underlying process remains part of the core mechanics; you just don't bother the player about it. The computer can take over the process entirely, in which case, again, you can save the time you would have spent on designing UI, or you can build the manual controls into the game but keep them hidden unless the player chooses (usually through an option in a shell menu) to take over manual control.

If you let the player choose between an automated or manual control over a game feature, you can refer to the two options as *beginner's mode* and *expert mode* in the menu where she makes the choice. You might want to reward the player for choosing the more complex task, such as choosing to shift the gears of a racing car manually, by making the manual task slightly more efficient than the automated one once the player has learned to do it well. If the automated task is perfectly efficient, the player has no incentive to learn the manual task.

Depth Versus Breadth

The more options you offer the player at one time, the more you risk scaring off a player who finds complex user interfaces intimidating. A UI that provides a large number of options simultaneously is said to be a *broad* interface. If you offer only a few options at once and require the player to make several selections in a row to get to the one he wants, the user interface is said to be *deep*.

Broad interfaces permit the player to search the whole interface by looking for what he wants, but finding the one item of current interest in that broad array takes time. Once the player learns where to find the buttons or dials, he can usually find them again quickly. Players who invest the (sometimes considerable) training time find using a broad interface to be efficient; they can quickly issue the commands they want. The cockpit of a commercial passenger aircraft qualifies as an enormously broad interface; with such a huge array of instruments, the pilot can place his hand on any button he needs almost instantly, which makes flying safer. On the other hand, pilots must train for years to learn them all.

Deep interfaces normally offer all their choices through a hierarchical series of menus or dialog boxes. The user can quickly see what each menu offers. He can't know in advance what sequence of menu choices he must make to find the option he wants, so the menus must be named and organized coherently to guide him. Even once he learns to find a particular option, he still has to go through the sequence of menus to get to it each time. On the other hand, using a well-designed deep interface takes almost no training.

It's a good idea to offer both a deep and a broad interface at the same time; deep for the new players, broad for the experienced ones. You can do this on the PC by assigning shortcut keys to frequently used functions. The large number of keys on a PC keyboard enables you to construct a broad interface easily. Console machines, with fewer controller buttons available and no mouse for pointing to screen elements, offer fewer options for creating broad interfaces.

If you can only offer one interface, we recommend that you make the breadth and depth of your interface roughly equal; but try not to make anything more than three or four levels deep if you can help it. When deciding how to structure menus, categorize the options by frequency of access. The most frequently accessed elements should be one or two steps away from the player at most. The least frequently accessed elements can be farther down the hierarchy.

Context-Sensitive Interfaces

A context-sensitive interface reduces complexity by showing the player only the options that she may actually use at the moment. Menu options that make no sense in the current context simply do not display. Microsoft Windows takes a middle path, continuing to show unavailable menu options in gray, while active menu items display in black. This reduces the user's confusion somewhat because she doesn't wonder why an option that she saw a few minutes ago has disappeared.

Graphic adventures, role-playing games, and other mouse-controlled games often use a context-sensitive pointer. The pointer changes form when pointed at an object with which it can interact. When pointing to a tree, for instance, it may change to the shape of an axe to indicate that pressing the mouse button will cause the tree to be cut down. The player learns the various things the mouse can do by pointing it at different objects in the game world and seeing how it changes.

Avoiding Obscurity

A user interface can function correctly and be pretty to look at, but when the player can't actually tell what the buttons and menus do, it is *obscure*. Several factors in the UI design process tend to produce obscurity, and you should be on the lookout for them:

- **Artistic overenthusiasm.** Artists naturally want to make a user interface as pleasing and harmonious as they can. Unfortunately, they sometimes produce UI elements that, while attractive, convey no meaning.
- **The pressure to reduce UI screen usage.** Using an icon instead of a text label on a screen button saves space, and so does using a small icon instead of a large one. But icons can't convey complicated messages as well as text can, and small simple icons are necessarily less visually distinctive than large complex ones. When you reduce the amount of space required by your UI, be sure you don't do so to the point of making its functions obscure.
- **Developer familiarity with the material.** *You* know what your icons mean and how they work—you created them. That means you're not the best judge of how clear they will be to others. Always test your UI on someone unfamiliar with your game. See whether your test subjects can figure out for themselves how things work. If it requires a lot of experimentation, your UI is too obscure.

Interaction Models

In Chapter 2, "Design Components and Processes," we defined the *interaction model* as the relationship between the player's inputs via the input devices and the resulting actions in the game world. You create the game's interaction model by deciding how the player's controller-button presses and other real-world actions will be interpreted as game world activities by the core mechanics. The functional capabilities of the various input devices available will influence your decisions, and we discuss input devices at length in *Input Devices* later in the chapter. We do not have room here to discuss button assignments in detail, so you should play other games in your genre to find examples that work well.

In practice, interaction models fall into several well-known types:

- ***Avatar-based,*** in which the player's actions consist mostly of controlling a single character—his avatar—in the game world. The player acts upon the world *through* the avatar and, more important, generally can influence only the region of the game world that the avatar currently inhabits. An avatar is analogous to the body of a human being: To do something in our world, we have to physically take our bodies to the place where we want to do it. That doesn't mean an avatar must be human or even humanoid; a vehicle can be an avatar. To implement this mode, therefore, many of your button-assignment decisions will center on navigation, which we discuss in *Navigation Mechanisms* later in the chapter.
- ***Multipresent*** (or *omnipresent*), in which the player can act upon several different parts of the game world at a time. In order to do so, you must give him a perspective that permits him to see the various areas that he can change; typically, an aerial perspective. Chess uses a multipresent interaction model; the player may ordinarily move any of his pieces (which can legally move) on any turn. Your decisions to implement this mode will concentrate on providing ways for the player to select and pick up, or give orders to, objects or units in the environment.
- The ***party-based*** interaction model, in which small groups of characters generally remain together as a group. This model is most commonly found in role-playing games. In this model, you will probably want to use point-and-click navigation and an aerial perspective.
- The ***contestant*** model, in which the player answers questions and makes decisions, as if a contestant in a TV game show. Navigation will not be necessary; you will simply have to assign different decision options to different buttons.
- The ***desktop*** model mimics a computer (or a real) desktop and is ordinarily found only in games that represent some kind of office activity, such as business simulations.

A coherent design that follows common industry practice will probably fit into one of these familiar models. You can create others if your game really requires them, but if you do so, you may need to design more detailed tutorial levels to teach your player the controls.

Perspectives

Old video games, especially those for personal computers, used to treat the game screen as if it were a game board in a tabletop game. Today we use a cinematic analogy and talk about the main view on the screen as if it displayed the output of a movie camera looking at the game world. In our discussion of perspectives, we make numerous references to the game's ***virtual camera.***

Like the interaction model, the game's *perspective* grows out of a cluster of design decisions about how you want the player to view the game world, what the camera focuses on, and how the camera behaves. Certain perspectives work best with particular interaction models, so as we introduce the most common game perspectives, we will discuss the appropriate interaction models for them at the same time.

A note on terminology: We use terms adopted from filmmaking to describe certain kinds of camera movements. When a camera moves forward or back through the environment, it is said to ***dolly,*** as in *the camera dollies to follow the avatar.* When it moves laterally, as it would to keep the avatar in view in a side-scrolling game, it ***trucks.*** When it moves vertically, it ***cranes.*** When a camera swivels about its vertical axis but does not move, it ***pans.*** When it swivels to look up or down, it ***tilts.*** When it rotates around an imaginary axis running lengthwise through the lens, it is said to ***roll.*** Games almost never roll their cameras except in flight simulators; as in movies, the player normally expects the horizon to be level.

The 3D Versus 2D Question

Before we discuss perspectives in detail, we address the question of when games should use a 3D graphics display engine and when they should stick to 2D graphics technology. If a game uses 2D graphics, the first-person and third-person perspectives will not be available; those perspectives require a 3D engine.

Virtually all large standalone games running on powerful game hardware such as a personal computer or home game console employ 3D. (Small games and those played within a Web browser often still use 2D graphics.) With modern hardware now standard, you should use 3D graphics *provided* that you have the tools, the skills, and the time to do it well. If you do *not* have the more complex tools and the specialized skills to get good results, you should not try it. Good-looking 2D graphics are always preferable to bad-looking 3D graphics.

This question becomes a critical issue on mobile phones and personal digital assistants. With no 3D graphics acceleration hardware, if these devices display 3D graphics, they must do it with software rendering—a complex task and one that taxes the slow processors that run these gadgets. Think twice before committing yourself (and your programming team) to providing 3D graphics on such platforms.

While it may take the player a while to detect weak AI or bad writing in a game, bad graphics show up from the first moment he starts to play. Here, above all, heed the principle that if you cannot do it well, don't do it at all.

First-Person Perspective

In the ***first-person perspective,*** used only in avatar-based gameplay modes, the camera takes the position of the avatar's own eyes. Therefore, the player doesn't usually see the avatar's body, though the game may display handheld weapons, if any, and occasionally the avatar's hands. The first-person perspective also works well to display the point of view of the driver of a vehicle; it shows the terrain ahead as well as the vehicle's instrument panel but not the driver herself. It conveys an impression of speed and helps immerse the player in the game world. First-person perspective also removes any need for the player to adjust the camera and, therefore, any need for you to design UI for camera adjustment. To look around, the player simply moves the avatar.

Advantages of the First-Person Perspective Note the following benefits of the first-person perspective compared with the third-person perspective:

- Your game doesn't display the avatar routinely, so the artists don't have to develop a large number of animations, or possibly any image at all, of the avatar. This can cut development costs significantly because you need animations only for those rare situations in which the player can see the avatar: cut-scenes, or if the avatar steps in front of a mirror.
- You won't need to design AI to control the camera. The camera looks exactly where the player tells it to look.
- The players find it easier to aim ranged weapons at approaching enemies in the first-person perspective for two reasons. First, the avatar's body does not block the player's view; second, the player's viewpoint corresponds exactly with the avatar's, and therefore, the player does not have to correct for differences between his own perspective and the avatar's.
- The players may find interacting with the environment easier. Many games require the player to maneuver the avatar precisely before allowing him to climb stairs, pick up objects, go through doorways, and so forth. The first-person perspective makes it easy for the player to position the avatar accurately with respect to objects.

Disadvantages of the First-Person Perspective Some of the disadvantages of the first-person perspective (as compared with third-person) include:

- Because the player cannot see the avatar, the player doesn't have the pleasure of watching her or customizing her clothing or gear, both of which form a large part of the entertainment in many games. Players enjoy discovering a new animation as the avatar performs an action for the first time.

- Being unable to see the avatar's body language and facial expressions (puzzlement, fear, caution, aggression, and so on) reduces the player's sense of her as a distinct character with a personality and a current mood. The avatar's personality must be expressed in other ways, through scripted interactions with other characters, hints to the player, or talking to herself.
- The first-person perspective denies the designer the opportunity to use cinematic camera angles for dramatic effect. Camera angles create visual interest for the player, and some games rely on them heavily: *Resident Evil,* for example, and *Grim Fandango.*
- The first-person perspective makes certain types of gymnastic moves more difficult. A player trying to jump across a chasm by running up to its edge and pressing the jump button at the last instant finds it much easier to judge the timing if the avatar is visible on screen. In the first person, the edge of the chasm disappears off the bottom of the screen during the approach, making it difficult to know exactly when the player should press the button.
- Rapid movements, especially turning or rhythmic rising and falling motions, can create motion sickness in viewers. A few games tried to simulate the motion of walking by swaying the camera as the avatar moves; this also tends to induce motion sickness.

Third-Person Perspective

Games with avatar-based interaction models can also use the ***third-person perspective.*** The most common perspective in modern 3D action and action-adventure games with strongly characterized avatars, it has the great advantage of letting the player see the avatar. The camera normally follows the avatar at a fixed distance, remaining behind and slightly above her as she runs around in the world so as to allow the player to see some way beyond the avatar into the distance.

The standard third-person perspective depends on an assumption that threats to the avatar will come from in front of her. Some games now include fighting in the style of martial-arts movies, in which the avatar can be surrounded by enemies; consider recent games in the *Prince of Persia* series. To permit the player to see both the avatar and the enemies, the camera must crane up and tilt down to show the fight from a raised perspective.

Designing the camera behavior for the third-person perspective poses a number of challenges, as we discuss next.

Camera Behavior When the Avatar Turns So long as the avatar moves forward, away from the camera, the camera dollies to follow; you should find this behavior easy to implement. When the avatar turns, however, you have several options:

- The camera keeps itself continuously oriented behind the avatar, as in the *chase* view in flight simulators (see Chapter 17, "Vehicle Simulations," for further discussion). Using this option, the camera always points in the direction in which the avatar looks. This arrangement allows the player to always see where the avatar is going, which is useful in high-speed or high-threat environments. Unfortunately, the player never sees the avatar's side or front, only her back, which takes some of the fun out of watching the avatar. Also, a human avatar can change directions rapidly (unlike a vehicle), and the camera must sweep around quickly in order to always remain behind her, which can produce motion sickness in the player.
- The camera reorients itself behind the avatar somewhat more slowly, beginning a few seconds after the avatar makes her turn. This option enables the player to see the avatar's side for a few seconds until the camera reorients itself. Because the camera moves more slowly in this maneuver, fewer players will find the images dizzying.
- The camera reorients itself behind the avatar only after she stops moving. The least-intrusive way to reorient the camera, this does mean that if the player instructs the moving avatar to turn and run back the way she came, she runs directly toward the camera, which does not reorient itself; instead it simply dollies away from her to keep her in view. The player cannot see any obstacles or enemies in the avatar's way because they appear to be behind the camera (until the instant before she runs into them). *Toy Story 2: Buzz Lightyear to the Rescue!* used this option; the effect, while somewhat peculiar, worked well in the game's largely nonthreatening environment.

If you plan for the camera to automatically reorient itself, you can give the player control over how quickly the reorientation occurs, a switch known as *active camera mode* or *passive camera mode.* This adjustment determines the length of time before the camera reorients itself so as to take up a position behind the avatar's back. In active mode, the camera either remains oriented behind the avatar at all times or reorients itself quickly; in passive mode, it either orients itself slowly or only when the avatar stops moving. This setting helps players affected by vertigo.

Intruding Landscape Objects What happens when the player maneuvers the avatar to stand with her back to a wall? The camera cannot retain its normal distance from the avatar; if it did, it would take up a position on the other side of the wall. Many kinds of objects in the landscape can intrude between the avatar and the camera, blocking the player's view of her and everything else.

If you choose a third-person perspective, consider one of the following solutions:

- Place the camera as normal but render the wall (and any other object in the landscape that may come between the camera and the avatar) semitransparent. This allows the player to see the world from his usual position but makes him aware of the presence of the intruding object.
- Place the camera immediately behind the avatar, between her and the wall, but crane it upward somewhat and tilt it down, so the player sees the area immediately in front of the avatar from a raised point of view.
- Orient the camera immediately behind the avatar's head and render her head semitransparent until she moves so as to permit a normal camera position. The player remains aware of her position but can still see what is in front of her.

When the player moves the avatar such that an object no longer intrudes, return the camera smoothly to its normal orientation and make the object suitably opaque again, as appropriate.

Player Adjustments to the Camera In third-person games, players occasionally need to adjust the position of the camera manually to get a better look at the game world without moving the avatar. If you want to implement this, assign two buttons, usually on the left and right sides of the controller, to control manual camera movement. The buttons should circle the camera around the avatar to the left or right, keeping her in focus in the middle of the screen. This enables the player to see the environment around the avatar and also to see the avatar herself from different angles.

Toy Story 2: Buzz Lightyear to the Rescue! used a different adjustment: The left and right buttons caused the avatar to pivot in position while the camera swept around to remain behind his back. This changed the direction the avatar faced as well as moving the camera and proved to be helpful for lining the avatar up for jumps.

Allowing the player to adjust the camera can help with the problem of intruding landscape items, but only as a stopgap, not a real solution.

Aerial Perspectives

Games with party-based or multipresent interaction models use an aerial perspective to allow the player to see a large part of the game world and several different characters or units at once. These perspectives give priority to the game world in general rather than to one particular character.

In games with multipresent interaction models, provide a means for the player to scroll the main game view around to see any part of the world that he wants (although parts of it may be hidden by the *fog of war;* see Chapter 14, "Strategy Games," for a discussion of the fog of war). With party-based interaction models, you may reasonably restrict the player's ability to move the camera to the region of the game world where the party is.

Top-Down Perspective Designers now usually reserve the top-down perspective, once standard for the main game view, to show maps in computer and console games. Easily implemented using 2D graphics, the top-down perspective remains in common use on smaller devices.

The ***top-down perspective*** shows the game world from directly overhead with the camera pointing straight down. In this respect, it resembles a map, so players find the display familiar. Its easy implementation using 2D graphics keeps it in common use on smaller devices, but its many disadvantages have led designers to use other methods on more powerful machines.

For one thing, the top-down point of view enables the player to see only the roofs of buildings and the tops of people's heads. To give a slightly better sense of what a building looks like, artists often draw them *cheated*—that is, at a slight angle even though that is not how a building should appear from directly above. (See Figure 18.1 for a top-down view with cheated buildings.)

The top-down perspective also distances the player from the events below. He feels remote from the action and less attached to its outcomes. It makes a game world feel like a simulation rather than a place that could be real.

Isometric Perspective The isometric perspective solves some of the problems presented by the top-down perspective, and because draftsmen developed isometric projection for two-dimensional display of 3D objects well before the computer age, it works well on systems without 3D graphics. An isometric projection shows the game world from an angle such that all three dimensions can be seen at once. If the game world is rectilinear (as they almost always are in the isometric perspective) and oriented on the cardinal compass points with north at the top of the map, then the isometric perspective shows the world from the southwest, looking toward the northeast, with north at the upper-left corner of the screen. This perspective requires an elevated camera position but not the extreme elevation of a top-down projection. See Figure 4.5 for a typical example of an isometric perspective.

An isometric projection distorts reality in that faraway objects don't get smaller as they recede into the distance. That's not the way we normally see the world, but because the camera does not display much of the landscape at one time, players don't mind the slight distortion. Because the isometric perspective is normally drawn by the 2D display engine using interchangeable tiles of a fixed size, the player can truck or dolly the camera above the landscape but cannot pan, tilt, or roll it. Some versions allow the player to shift the camera orientation from facing northeast to one of the other ordinal points of the compass in order to see other sides of objects in the game world. Doing this requires the artists to draw four sets of tiles, one for each possible camera orientation. Some also include multiple sets of tiles drawn at different scales, to let the player choose an altitude from which to view the world.

The isometric perspective brings the player closer to the action than the top-down perspective and allows him to see the sides of buildings as well as the roofs, so the player feels more involved with the world. It also enables him to see the bodies of people more clearly. Real-time strategy games and construction and management simulations, both of which normally use multipresent interaction models, routinely display either the isometric perspective or its modern 3D equivalent, the free-roaming camera. Some role-playing games that use a party-based interaction model still employ isometric perspective (see Figure 15.4).

Free-Roaming Camera For aerial perspectives today, designers favor the ***free-roaming camera,*** a 3D perspective that evolved from the isometric perspective and is made possible by modern 3D graphics engines. It allows the player considerably more control over the camera; she can crane it to choose a wide or a close-in view; she can tilt and pan in any direction at any angle, unlike the fixed camera angle of the isometric perspective. The free-roaming camera also displays the world in true perspective: Objects farther away seem smaller. The biggest disadvantage of the free-roaming camera is that you have to implement all the controls for moving the camera and teach the player how to use them.

Context-Sensitive Perspectives Context-sensitive perspectives require 3D graphics and are normally used with avatar-based or party-based interaction models. In a ***context-sensitive perspective,*** the camera moves intelligently to follow the action, displaying it from whatever angle best suits the action at any time. You must define the behavior of the camera for each location in the game world and for each possible situation in which the avatar or party may find themselves.

ICO, an action-adventure game, implemented context-sensitive perspective, using different camera positions in different regions of the world to show off the landscape and the action to the best advantage. This made *ICO* an unusually beautiful game (see Figure 13.12). Context-sensitive perspectives allow the designer to act as a cinematographer to create a rich visual experience for the player. Seeing game events this way feels a bit like watching a movie because the designer intentionally composed the view for each location.

This approach brings with it two disadvantages. First, composing a view for each location requires a great deal more effort on the part of designer and programmers than implementation of other perspectives. Second, a camera that moves of its own accord can be disorienting in high-speed action situations. When the player tries to control events at speed, he needs a predictable viewpoint from which to do so. The context-sensitive perspective suits slower-moving games quite well, and frenetic ones less well. Some games, such as those in the survival horror series *Silent Hill,* use a context-sensitive perspective when the avatar explores but switch to a third-person or other more fixed perspective when she gets into fights.

Other 2D Display Options

For the sake of completeness, we briefly mention a number of approaches to 2D displays now seldom used in large commercial games on PCs and consoles but still widely found in Web-based games and on smaller devices. Modern games that intentionally opt for a retro feel, such as *Alien Hominid* and *Strange Adventures in Infinite Space,* also use 2D approaches.

- **Single-screen.** The display shows the entire world on one screen, normally from a top-down perspective with cheated objects. The camera never moves. *Robotron: 2084* provides a classic example. (See the left side of Figure 13.1.)
- **Side-scrolling.** The world of a side-scroller—familiar from an entire generation of games—consists of a long 2D strip in which the avatar moves forward and backward, with a limited ability to move up and down. The player sees the game world from the side as the camera tracks the avatar.
- **Top-scrolling.** In this variant of the top-down perspective, the landscape scrolls beneath the avatar (often a flying vehicle), sometimes at a fixed rate that the player cannot change. This forces the player to continually face new challenges as they appear at the top of the screen.
- **Painted backgrounds.** Older graphical adventure games displayed the game world in a series of 2D painted backgrounds rather like a stage set. The avatar and other characters appeared in front of the backgrounds. The artists could paint these backgrounds from a variety of viewpoints, making such games more visually interesting than side-scrolling and top-scrolling games, constrained only by the fact that the same avatar graphics and animations had to look right in all of them. (See Figure 19.9 for an example.)

Visual Elements

Having introduced the major interaction models and perspectives you may wish to offer in your game, we now turn to the visual elements that you can use to supply information that the player needs to know.

Main View

The player's main view of the game world, from whatever perspective you choose, should be the largest element on the screen. You must decide whether the main view will appear in a window within the screen with other user interface elements around it, or whether the view will occupy the whole screen and the other user interface elements will appear on top of it. We address these options next. (See also *Choosing a Screen Layout,* earlier in the chapter.)

Windowed Views In a windowed view, the oldest and easiest design choice, the main view takes up only part of the screen, with the rest of the screen showing panels displaying feedback and control mechanisms. You'll find this view most frequently used by games with complicated user interfaces such as construction and management simulations, role-playing games, and strategy games, because they require so many on-screen controls (see Figure 15.4 for a typical example). Using a windowed view does not mean that feedback elements *never* obscure the main view, only that they need to do so less often because most of them are around the edges.

The windowed view really does make the player feel as if she's observing the game world through a window, so it harms immersion somewhat. It looks rather like a computer desktop user interface, and you see this approach more often in PC games than in console games. The loss of immersion, undesirable for high-speed games in which the majority of the player's actions take place in the window itself, matters less when the game requires a great deal of control over a complex internal economy and the player needs access to all those controls at all times.

Opaque Overlays If you want to create a greater sense of immersion than the windowed view offers, you can have the main view fill all or almost all of the screen and superimpose graphical elements on it in *overlays,* small windows that appear and disappear in response to player commands. The most common type, the *opaque overlay,* entirely obscures everything behind it (see Figure 18.5 for an example). Opaque overlays carve a chunk out of the main view, but when they're gone the player can see more of the game world than in a windowed view, and she doesn't feel as if she's looking through a window.

Action games that don't need a lot of UI elements on the screen often use *borderless* opaque overlays—overlays that don't appear in a box. Compare the rather old-fashioned windowed view on the left side of Figure 13.2 with the borderless opaque overlays on the right side. The overlays obscure only a small part of the main view, which otherwise runs edge-to-edge.

Semitransparent Overlays Semitransparent overlays let the player see partially through them. See Figure 16.3 for an example. Semitransparent overlays feel less intrusive than opaque ones and work well for things such as instruments in the *cockpit-removed view* in a flight simulator, described in Chapter 17, "Vehicle Simulations." However, the bleed-through of graphic material from behind these overlays can confuse the information that the overlay presents. You can barely read the semitransparent overlay in the upper left corner of Figure 17.5 because it consists of light colors with a light sky behind.

We suggest that you use semitransparent overlays only for graphical information such as the baseball diamond mini-map in Figure 16.3, not for text. Players find it irritating to read text with graphics underneath it, especially moving graphics.

Feedback Elements

Feedback elements communicate details about the game's inner states—its core mechanics—to the player. They tell the player what is going on, how she is doing, what options she has selected, and what activities she has set in motion.

Indicators ***Indicators*** inform the player about the status of a resource, graphically and at a glance. We will use common examples from everyday life as illustrations. The meaning of an indicator's readout comes from labels or from context; the indicator itself provides a value for anything you like. Still, some indicators suit certain types of data better than others, and where appropriate, we will mention this. Choose indicators that fit the theme of your game and ones that don't introduce anachronisms; a digital readout or an analog clock face would both be shockingly out of place in a medieval fantasy.

Indicators fall into three categories: general numeric, for large numbers or numbers with fractional values; small-integer numeric, for integers from 0 to 5; and symbolic, for binary, tristate, and other symbolic values. Here are some of the most common kinds of indicators, with their types.

- **Digits.** General numeric. (A car's odometer.) Unambiguous and space-efficient, a digital readout can display large numbers in a small screen area. Digits can't be read easily at a glance, however; *171* can look a lot like *111* if you have only a tenth of a second to check the display during an attack. Worse, many types of data the player needs—health, mana, and armor strength—can't be appropriately communicated to the player by a number; no one actually thinks, "I feel exactly 37 points strong at the moment." Use digits to display the player's score and amounts of things for which you would normally use digits in the real world: money, ammunition, volumes of supplies, and so on. Don't use digits for quantities that should feel imprecise, such as popularity.
- **Needle gauge.** General numeric. (A car's speedometer.) Vehicle simulations use duplicates of the real thing—speedometers, tachometers, oil pressure levels, and so on—but few other games require needle gauges. Generally easy to read at a glance, they take up a large amount of screen space to deliver a small amount of information. You can put two

needles on the same gauge if you make them different colors or different lengths and they both reflect data of the same kind; an analog clock is a two-needle gauge (or a three-needle gauge if another hand indicates seconds). Use needle gauges in mechanical contexts.

- **Power bar.** General numeric. (On an analog thermometer, the column of colored fluid indicating temperature.) A power bar is a long, narrow colored rectangle that becomes shorter or longer as the value that it represents changes, usually to indicate the health of a character or time remaining in a timed task. (The name is conventional; power bars are not limited to displaying power). When the value reaches zero, the bar disappears (though a framework around the bar may remain). If shown horizontally, by convention zero is at the left and the maximum at the right; if shown vertically, zero is at the bottom and maximum is at the top. The chief benefit of power bars is that the player can read the approximate level of the value at a glance. Unlike a thermometer, they rarely carry gradations. You can superimpose a second semitransparent bar of a different color on top of the first one if you need to show two numbers in the same space. Many power bars are drawn in green when full and change color to yellow and red as the value indicated reaches critically low levels to help warn the player. Power bars are moderately space efficient and, being thematically neutral, appear in all sorts of contexts. You can make themed power bars; a medieval fantasy game might measure time with a graduated candle or an hourglass.

- **Small multiples.** Small-integer numeric. (On a mobile phone, the bars indicating signal strength.) A small picture, repeated multiple times, can indicate the number of something available or remaining. ***Small multiples*** have long been used for lives remaining in action games, which often employed an image or silhouette of the avatar. Nowadays designers use them for things the avatar can carry, such as grenades or healing potions, although you should limit the maximum number to about 5; beyond that the player can't take in the number of objects at a glance and must stop to count the pictures. To make this method thematically appropriate for your game, simply choose an appropriate small picture.

- **Colored lights.** Symbolic. (In a car, various lights on the instrument panel.) Lights are highly space efficient, taking up just a few pixels, but can't display much data, normally indicating binary (on/off) values with two colors, or tristate values with three (off/low/high). Above three values, players tend to forget what the individual colors mean, and bright colors are not thematically appropriate in some contexts. Use a suitable palette of colors.

- **Icons.** Symbolic. (In a car, the symbols indicating the heating and air conditioning status.) Icons convey information in a small space, but you must make them obvious and unambiguous. Don't use them for numerical quantities but for symbolic data that record a small number of possible options. For example, you can indicate the current season with a snowflake, a flower, the sun, and a dried leaf. This will be clear to people living in the temperate parts of the world where these symbols are well known, but it would work less well in cultures where snow is never seen. The player can quickly identify icons once she learns what they mean, and you can help her learn by using a *tooltip,* a small balloon of text that appears momentarily when the mouse pointer touches an icon for a few seconds without clicking it. Don't use icons if you need large numbers of them (players forget what they mean) or if they refer to abstract ideas not easily represented by pictures. In those cases, use them with text alongside, or use text instead. Make your icons thematically appropriate by drawing pictures that look as if they belong in your game world. The icons in *Populous: The Beginning,* set in a Stone Age fantasy world, were excellent (see Figure 4.6).

- **Text indicators.** Symbolic. Text represents abstract ideas well, an advantage over other kinds of indicators. In *Civilization III,* for example, an advisor character can offer the suggestion, "I recommend researching Nationalism." Finding an icon to represent nationalism or feudalism or communism, also options in the game, poses a problem. On the other hand, some people find text boring, and two words can look alike if they're both rendered in the same color on the same color background. The worst problem with text, however, is that it must be localized for each language that you want to support. (See *Text* later in this chapter.)

We strongly encourage you to read the books of Dr. Edward Tufte for more information on conveying data to the player efficiently and readably, particularly *The Visual Display of Quantitative Information* (Tufte, 2001).

Mini-Maps A *mini-map,* also sometimes called a *radar screen,* displays a miniature version of the game world, or a portion of it, from a top-down perspective. The mini-map shows an area larger than that shown by the main view, so the player can orient himself with respect to the rest of the world. To help him do this, designers generally use one of two display conventions: *world-oriented* or *character-oriented* mini-maps.

- The world-oriented map displays the entire game world with north at the top, just like a paper map, regardless of the main view's current orientation. An indicator within the mini-map marks that part of the game world currently visible in the main view (see Figure 4.2). In a multipresent game, you can use the world-oriented map as a camera control device: If the player clicks on the map, the camera jumps to the location clicked.

- The character-oriented map displays the game world around the avatar, placing him at the center of the map facing the top of the screen. If the player turns the avatar to face in a new direction in the game world, the landscape, rather than the avatar, rotates in the map. These mini-maps don't show the whole game world, only a limited area around the avatar, and as the avatar moves, they change accordingly. They're often round and for this reason are sometimes called radar screens. Because the landscape rotates in the map, character-oriented mini-maps sometimes include an indicator pointing north, making the map double as a compass.

Because the mini-map must be small (usually 5 to 10 percent of the screen area), it shows only major geographic features and minimal non–mission-critical data. Key characters or buildings typically appear as colored dots. Areas of the game world hidden by the fog of war appear hidden in the mini-map also.

A mini-map helps the player orient himself and warns him of challenges not visible in the main view, such as nearby enemies in a strategy or action game or a problem developing in a construction and management simulation. Mini-maps typically show up in a corner of the screen. You can find them in virtually any game that uses aerial perspectives and many others as well. Figures 4.2, 4.5, 4.6, and 4.8 all contain mini-maps.

Use of Color You can always double the amount of data shown in a numeric indicator by having the color of the indicator itself represent a second value. You might, for example, represent the speed of an engine with a needle gauge, and the temperature of that engine by changing the color of the needle from black to red as it gets hotter. Colors work best to display information that falls into broad categories and doesn't require precision within those categories. Consider the green/yellow/red spectrum used for safety/caution/danger: It doesn't display a precise level of safety but conveys the general level at a glance. (Note our warnings about color-blind players in Appendix A, "Designing to Appeal to Particular Groups," on the Companion Website.)

Companion Website

Colors are also very useful for differentiating groups of opponents, however, and you can apply them to uniforms and other insignia. This is especially handy if the shapes or images of the actual units are identical regardless of which side they're on . . . as any chess player knows!

You can also use color as a feedback element by placing a transparent color filter over the entire screen. Some first-person shooters turn the whole screen reddish for a few frames to indicate that the avatar has been hit.

Character Portraits

A character portrait, normally appearing in a small window, displays the face of someone in the game world—either the avatar, a member of the player's party in a party-based game, or a character the player speaks to. If the main view uses an aerial perspective, it's hard for the player to see the faces of characters in the game, so a character portrait gives the player a better idea of the person he's dealing with. Use character portraits to build identification between your player and his avatar or party members and to convey more about the personalities of nonplayer characters. An animated portrait can also function as a feedback element to give the player information; *Doom* famously used a portrait of the avatar as a feedback element, signaling declining health by appearing bloodier and bloodier. This portrait also allowed the player to see his avatar even though playing a first-person shooter.

Screen Buttons and Menus

Screen buttons and menus enable the player to control processes too complex to be managed with controller buttons alone. They work best with the mouse as a pointing device but can also be used with a D-pad or joystick. Because a console doesn't have a mouse, console games make less use of screen buttons and menus than do PC games, one of several reasons console games tend to be less complex than PC games.

Screen buttons and menus should be so familiar to you from personal computers that we do not discuss them in detail here, though we do note a couple of key issues to keep in mind. First, an overabundance of buttons and menus on the screen confuses players and makes your game less accessible to casual players (see *Managing Complexity,* earlier in the chapter). Second, unless you use the desktop perspective, try to avoid making your buttons and menus look too much like an ordinary personal computer interface. The more your game looks like any other Windows or Macintosh application, the more it harms the player's immersion in the game. Make your screen-based controls fit your overall visual theme.

Text Most games contain a fair amount of text, even action games in which the player doesn't normally expect to do much reading. Text appears as a feedback element in its own right, or as a label for menu items, screen buttons, and to indicate the meaning of other kinds of feedback elements (a needle gauge might be labeled *Voltage,* for example). You may also use text for narration, dialog (including subtitles), a journal kept by the avatar, detailed information about items such as weapons and vehicles, shell menus, and as part of the game world itself, on posters and billboards.

Localization ***Localization*** refers to the process of preparing a game for sale in a country other than the one for which you originally designed the game. Localization often requires a great many changes to the software and content of the game, including translating all the text in the game into the target market's preferred language. In order to make the game easily localizable, you should store all the game's text in text files and never embed text in a picture. Editing a text file is trivial; editing a picture is not.

Keep Text Separate from Other Content

Never have the programmers build text into the program code. *Never* build text that the player is expected to read into an image such as a texture or a shell screen background. Store all text in one or more text files.

The only exception to this rule applies to text used purely as decoration when you don't expect the player to read it or understand what it says. A billboard seen in a game set in New York should be in English and remain in English even after localization *if* the billboard text doesn't constitute a crucial clue.

Note that a word and its translation may differ in length in different languages, so that a very short menu item in English can turn into a very long menu item in, say, German. When you design your user interface, don't crowd the text elements too close together; the translations may require the extra space.

Typefaces and Formatting Make your text easily readable. The minimum size for text displayed on a screen should be about 12 pixels; if you make the characters any smaller, they became less legible. If the game will be localized to display non-Roman text such as Japanese, 12 pixels is the bare minimum, and 16 pixels is distinctly preferable.

If you're going to display a lot of text, learn the rules of good typesetting. Use mixed uppercase and lowercase letters for any block of text more than three or four words long. Players find text set entirely in uppercase letters difficult to read; besides, it looks like SHOUTING, creating an inappropriate sense of urgency you might not want. (On the other hand, in situations that *do* require urgency, such as a warning message reading *DANGER,* uppercase letters work well.)

Choose your typefaces (fonts) with care so that they harmonize both with the theme of your game and with each other. Avoid using too many different typefaces, which looks amateurish. Be aware of the difference between *display fonts* (intended for headlines) and ordinary *serif* and *sans serif* fonts (intended for blocks of text) such as Times or Arial, respectively.

Avoid *monospaced* or *fixed width* fonts such as Courier, in favor of proportional fonts, such as Times, unless you need to display a table in which letters must line up in columns. For other uses, fixed width fonts waste space and look old-fashioned and unattractive.

Audio Elements

We have already spoken a little about sound in our discussion of player feedback. Here we address sound in more detail, looking at several different areas: sound effects, ambient sounds, music, and dialog and voiceover narration.

Always include a facility that allows the player to adjust the volume levels of the music and of all the other effects independently—including turning one or the other off completely. Many players tire of hearing the music but still want to hear the sound effects and ambient sounds. Bear in mind that not all your players will have perfect hearing, and the more control you can give them, the better. See *Accessibility Issues* in Appendix A, "Designing to Appeal to Particular Groups."

Sound Effects

The most common use of sound in a game is for sound effects. These sounds correspond to the actions and events of the game world—for example, a burst of gunfire or the tight squealing of tires as a car slides around a corner. In the real world, sound often presents the first warning of approaching danger, so use sound as an indicator that something needs the player's attention. Suspense movies do this well, and you can borrow techniques from them: Play the sound of footsteps or the sound of a gun being cocked before the player can see it. You can also use sound to provide feedback about aspects of the game under the player's control, such as judging when to change gears in a racing game by listening to the pitch of the engine.

You should also include sound effects as audible feedback in your user interface, not just in the game world. At the very minimum, screen buttons should make an audible *click* when pressed, but try to find interface effects that harmonize with the theme of your game world (as long as they're not corny). Be sure to support audio feedback from the UI with visual feedback too, so that when players hear a click or beep or buzz, the visual feedback directs them to the issue that generated the audible signal. We interpret events that we can see more easily than by audio alone.

Ambient Sounds

Just as the main view gives the player visual feedback about where she is, ambient sounds give her aural feedback. Traffic sounds tell her that she's in an urban street; cries of monkeys and exotic birds suggest a jungle. Anything that ordinarily makes distinctive sounds in the real world, such as a fountain or a jackhammer, should make the same sound in your game.

A first- or third-person game should definitely use *positional audio* if the platform's audio devices support it. Positional audio refers to a system in which different speakers present sounds at different volume levels, allowing you to position the point sources of sound in the three-dimensional space of the world. Some personal computers support as many as seven speakers, but even two-speaker stereo can help a player detect where a sound is coming from. Correctly positioning sound sources in the 3D space helps the player to orient himself and to find things that he may be searching for, such as a river, an animal, or another member of the party.

Don't overuse ambient sounds, especially in games that mostly feature mental challenges. A cacophonous environment isn't conducive to thought. Your ambient sounds must also work with the music you choose, which we address next. You may also be limited by the capabilities of your audio hardware, because some machines support only a small number of channels for simultaneous playback; when playing all the sound effects, you may not have channels left to use for ambient sounds.

Music

Music helps to set the tone and establish the pace of your game. Think about what kind of music will harmonize with the world and the gameplay that you're planning. Music sends strong cultural messages, and those too must fit thematically with the rest of the game. A pentatonic scale composition for the shamisen (a traditional Japanese lute) might work well in a medieval Japanese adventure game but would certainly sound out of place in a futuristic hi-tech game. You will probably collaborate with the audio director to choose or compose music for the game. Many larger commercial games now use licensed music from famous bands.

The music doesn't have to support the game world at every moment; you can choose music to create a contrasting effect at times. The introductory movie for *StarCraft* used classical opera as its theme, set against scenes in which admirals calmly discussed the war situation as they prepared to abandon the men on the planet below to their fate. The choice of music accentuated the contrast between the opulence and calm of the admiral's bridge and the hell of war on the surface. In a simpler example, the tempo of the music featured in certain levels of *Sonic the Hedgehog* was out of sync with the pace of the level, which, in a subtle way, made the game harder to play. We encourage you not to overuse these techniques, however; the rarer they are, the more effective they are.

In the real world, few pieces of music last as long as an hour, but players may hear the same music for several hours at a stretch in a game. Whatever you choose, be sure it can tolerate repetition. Avoid background music with a wide dynamic range; the louder parts will become intrusive and remind the player that the music repeats itself.

For some years, the game industry has experimented with the difficult problem of writing music that changes dynamically in response to current game situations, a technique called *adaptive music*. Developers would like to provide music that correctly anticipates the player's actions and upcoming game events so that the music further enhances the mood of the game, as movie music does. Movies are not interactive, however, so the composer knows what will be happening at every moment. Adaptive music must follow and even anticipate unpredictable situations. Creating adaptive music remains an experimental technique for the moment.

On the other hand, game musicians have become extraordinarily skilled at *layering*—writing separate but harmonizing pieces of music that the audio engine delivers simultaneously by mixing them together at different levels of volume. The engine determines which piece should be most clearly heard depending on what happens in the game.

If you would like to know more, we suggest you read *Audio for Games: Planning, Process, and Production,* by Alexander Brandon (Brandon, 2004) and *The Fat Man on Game Audio: Tasty Morsels of Sonic Goodness* by George "The Fat Man" Sanger (Sanger, 2003).

Dialog and Voiceover Narration

Many games with strong storylines, particularly adventure and role-playing games, make great use of recorded dialog and voiceover narration. Sports games use recorded audio to produce play-by-play and color commentary. Other kinds of games play back smaller bits of dialog to build atmosphere and provide feedback. Real-time strategy games often use spoken words as feedback to indicate that units received their orders or completed a task and need new orders. Finally, just about any kind of game can use spoken material to provide information, from mission briefings to the care and feeding of your virtual pet.

From a user interface standpoint, you should be aware of two key things that set spoken words apart from other forms of audio feedback. First, the human brain rapidly tires of hearing the same words played back repeatedly, and the longer the sentence, the worse the problem. If you plan to play voice material in response to in-game events or player actions that occur repeatedly, you *must* record multiple variants and mix them up at random when you play them back. You may frequently repeat short clips such as "Aye aye, sir" and "Strike three!" (though we still recommend you record several variants), but if you want to deliver a warning using a longer sentence, such as, "Sire, your peasants are revolting!" you must either have a large number of variants available or, better yet, play the sentence only once when the problem first occurs and then use visual feedback for as long as the problem continues. If the kingdom suffers a *second* peasant revolt later, you can play the sentence again.

Second, we cannot emphasize enough that *writing and acting must be good.* The quality of writing in the vast majority of games ranges from terrible to barely passable, and the voice acting is frequently worse than the writing. Players tolerate a sound effect that's not quite right, but an actor who can't act instantly destroys immersion. Don't use actors whose voices don't work thematically with the material, either. You wouldn't use the voice of an eighteenth-century English fop in a game set in the Old West, so don't use an American in a game set in medieval times. The American accent didn't exist then. Don't try to get an actor to fake a foreign accent, either; hire a native speaker.

For more information on writing for games, we suggest you read *Game Writing: Narrative Skills for Videogames,* edited by Chris Bateman (Bateman, 2006).

Input Devices

In this book, we have placed little emphasis on the game machine's hardware, because the variety of processors, display screens, data storage, and audio devices makes it impossible to address them comprehensively. We have expected you to design with your machine's processing facilities and output devices in mind. In the case of input devices, however, certain standards have evolved that we can address. We also feel that it is critically important that you understand the capabilities, strengths, and weaknesses of the various devices because they constitute the means by which your player will actually project his intentions and desires into the game. Designing for them well makes the difference between a seamless experience and a frustrating one.

We concentrate on the most common types of input devices for handheld, PC, and console games, the sorts normally shipped with the machine and to which you can expect any player to have access. The new motion-sensitive controllers from Nintendo and Sony show promise for the future, but as yet no standards for using them have evolved.

We don't address extra-cost items such as flight control yokes, steering wheels, rudder pedals, dance mats, fishing rods, bongo drums, cameras, and microphones. If you build a game that cannot be played without these items, you limit the size of your market to a specialist audience, and we cannot address such issues in a work on general game design. We encourage you to design for the default control devices shipped with a machine if at all possible. Only support extra-cost devices if using them significantly enhances the player's experience, or if you are intentionally designing a technology-driven game to exploit the device.

For most of their history, input devices for personal computers differed greatly from those of game consoles, so the two were best discussed separately. Console games never used analog joysticks; PC games never used D-pads. Now, both machines can use either, so we discuss the various input devices independently of the platforms.

Terminology

The discussion below uses the game industry's standard terminology for the kinds of data that control devices send to the processor as the result of player inputs. You may find some familiar terms that nevertheless require explanation, because the game industry uses those terms in ways that may differ from what you're used to.

Most input devices—the mouse being a notable exception—default to a *neutral position*. To send a signal to the game, the user must push, pull, grasp, or press the physical device to deflect it from this neutral position, and a spring-loaded mechanism returns it to the neutral position when the player releases the device. Joysticks and D-pads return to center; buttons and keys return to the off state.

A device that can return only two specific signals and no other we call a binary device, the signals generally being interpreted as off and on. The other common kind of input device transmits a value from a range of many possible values and the industry, for historical reasons, calls these analog devices. Any game control device can be classed as either analog or binary, though all of the technology is digital.

Don't confuse the type of data (binary or analog) with the *dimensionality* of the device. A one-dimensional device transmits one datum, and a two-dimensional device transmits two data, regardless of whether they transmit binary or analog data.

A device that returns data about its current position as measured from the neutral position provides *absolute* values. Such a device—a joystick, for example—can travel only a limited distance in any direction and so transmits values in a range from zero to its maximum.

Other devices offer virtually unlimited travel and have no neutral position. These return *relative* values, that is, the relative distance that the device has traveled from its previous position. Mouse wheels and track balls are examples; the player may rotate them indefinitely.

Two-Dimensional Input Devices

Two-dimensional input devices allow the player to send two data to the game at one time from a single device.

Directional Pads (D-Pads) We begin our discussion with directional pads (D-pads), the most familiar form of directional control mechanism on game machines, still offered by many smaller handheld machines as the only two-dimensional input device. Console and PC controllers often supply a D-pad in addition to a joystick to provide backward compatibility with older software.

A D-pad is a circular or cross-shaped input device on a game controller constructed with binary switches at the top, bottom, left, and right edges. The D-pad rocks slightly about its central point and, when pressed at any edge, turns on either one switch or, if the player presses between two adjacent switches, two. It can, therefore, send directional information to the game in eight possible directions: up, down, left, and right with each of the individual sensors, and

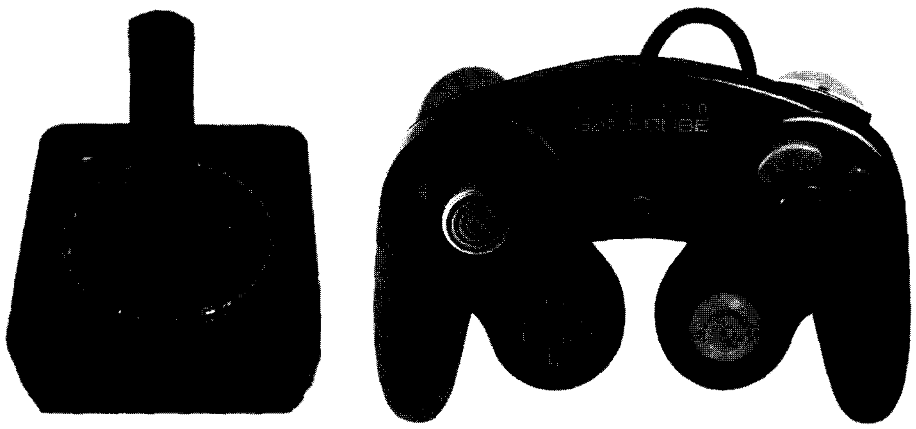

FIGURE 8.2 The original Atari 8-way joystick and the Nintendo GameCube controller.

upper-left, upper-right, lower-left, and lower-right when the player triggers two sensors together. (See Figure 8.2; the cross-shaped device at the lower left of the controller on the right is a D-pad.)

The D-pad gives the player a crude level of control over a vehicle or avatar; she is able to make the vehicle move in any of the eight major directions but not in any other. You should use D-pads for directional control only if you have no better device available. D-pads do remain useful alongside a joystick; you can assign to the D-pad functions requiring less subtle control, such as scrolling the main view window in one of the eight directions, leaving the joystick free for such tasks as avatar navigation control.

Joysticks Joysticks developed from an early aircraft flight control device (still used in fighter planes but replaced by the control yoke in modern civilian aircraft). It is a single vertical stick anchored at the bottom that can be tilted a limited amount in any direction. When the game software checks the position of the joystick, it returns two absolute, analog data: an X-value indicating tilt to the left or right, and a Y-value indicating the tilt forward or back.

A joystick offers a finer degree of control than a D-pad does. The Nintendo GameCube controller on the right in Figure 8.2 features two small joysticks, one to the upper left of the D-pad and one marked with C at the lower center. (Early video game machines shipped with an 8-way joystick: a device that looked like an analog joystick but actually functioned as a D-pad. The joystick on the left in Figure 8.2 is an 8-way joystick.)

Modern joysticks built for use with combat flight simulators may include a large number of other controller buttons as well: one or more fire buttons, used to fire weapons; a *hat switch* on the top of the stick, functionally equivalent to a D-pad; a slider that can be used as a throttle; and switches triggered by twisting the stick to the left or right. All of these ultimately amount either to controller buttons or sliders. Here we concern ourselves only with the tilting action of the basic device.

Joysticks make ideal steering controls for vehicles. To return to a default—flying straight and level, for instance—the player only has to allow the joystick to return to the neutral position. Since joysticks may travel only a limited amount in any direction, they allow the player to set a direction and a *rate* of movement. The UI interprets the degree of tilt as indicating the rate. For instance, moving a joystick to the left causes an airplane to roll to the left; moving it farther left causes the airplane to roll faster.

Joysticks don't work well for precise pointing; when the player lets go, the joystick returns to center, which naturally causes it to point somewhere else. To allow the player to point a cursor at an object *and leave it there* while she does something else, use a mouse. Efforts to port mouse-based games to console machines, substituting a joystick for the mouse, have an extremely poor success rate.

The Mouse (or Trackball) We're all familiar with mice from personal computers. A mouse returns two data that consist of X and Y values, but these are *relative* data, indicating how far the player moved the mouse relative to its previous location. A mouse offers more precise positioning than a joystick and unlimited travel in any direction on the two-dimensional plane in which it operates. This unlimited relative movement makes a mouse ideal for controlling things that can rotate indefinitely in place, so first-person PC games virtually always use mice to control the direction in which the avatar looks. Because it stays where it is put, a mouse is invaluable for interfaces in which the player needs to let go of the pointing device from time to time to do something else.

Note that when a mouse is used specifically to control a cursor on the screen, the driver software converts the mouse's native relative data into absolute data for the cursor position. This choice of either absolute or relative modes lends the mouse great flexibility.

A mouse wheel constitutes a separate knob with unlimited movement that also functions as a controller button when pressed. Not all mice come with mouse wheels, however, and you cannot count on them. If you support the mouse wheel, supply alternative controls.

The mouse's lack of a neutral position makes it weak as a steering mechanism for vehicles that need a default behavior—driving straight or flying straight and level. The player must find the vehicle's straight or level position herself rather than allowing the device to snap back into neutral. You may want to designate an extra controller button that returns the vehicle to its default state if the mouse will be your primary control option.

Designers find mice generally more flexible input devices than joysticks, but players find them more tiring to use for long periods.

Touch-Sensitive Devices Personal digital assistants (PDAs) and the Nintendo DS machine offer the player a touch-sensitive screen, and laptop personal computers usually come with a touch pad below the keyboard. These devices return absolute analog X and Y positions indicating where they are touched, as a mouse cursor does. Unlike a mouse, you can make a touch-sensitive device's cursor return to a neutral position whenever you detect that the player has stopped touching the device. Touch-sensitive screens may be manipulated by the fingers or a stylus; touch pads usually cannot detect a stylus, and must be touched with the fingers, which tends to make fingers sore after long use.

One-Dimensional Input Devices

One-dimensional input devices send a single value to the game. Ordinary controller buttons and keys send binary values; knobs, sliders, and pressure-sensitive buttons send analog values.

Controller Buttons and Keys A controller button or a keyboard key sends a single binary value at a time: on when pressed and off when released. Despite this simplicity, buttons and keys may be used in a variety of ways:

- **One-shot actions.** Treat the on signal as a trigger, a message to the game to perform some action immediately (ignoring the off signal). The action occurs only once, when the player presses the button; to perform it twice, he must press the button again. You might use this to let players fire a handgun, firing once each time they hit the button.
- **Repeating actions.** The on signal tells the game to begin some action and to repeat it until it receives the off signal from the same button, at a repetition rate determined by the software. You could let the players fire a machine gun continuously from button press to button release.
- **Continuous actions.** The button's on signal initiates a continuous action, and its off signal ends it. Golf games use this to give a player control over how hard the golfer swings the club; the player presses the button to start the golfer's backswing and releases it to begin the swing itself; the longer the backswing, the harder the golfer will hit the ball. Some football games allow the player to tap the button quickly to throw a short pass or to hold it down for a moment before release to throw a long pass, with the length of time between button's on and off signals determining the distance thrown.

Console game controllers feature anywhere from one to about ten buttons. Buttons on the top face of the controller, to be pressed with the thumbs, are known as *face buttons.* Others, known as *shoulder buttons,* appear on the part of the controller facing away from the player, under the index fingers. Faced with large numbers of buttons, the player can find it quite difficult to remember what they all do. Here, as elsewhere, be sure to maintain consistency from one gameplay mode to another, and if an industry standard has evolved for your game's genre, do not depart from it without good reason.

Personal computer keyboards have 110 keys, allowing for very broad user interfaces indeed. Players can assign keys to functions that might otherwise require selecting options on the screen with the mouse, provided that you allow for it. Be sure to assign actions to keys in such a way that the letter printed on the key becomes a mnemonic for the action, for example, F for flaps, B for brakes, or other similarities.

Knobs, Sliders, and Pressure-Sensitive Buttons You rarely find knobs (also sometimes called *Pong paddles* for historical reasons) nowadays, although the mouse wheel functions as a knob. Limited-travel knobs can move only so far, like a volume knob on a stereo, and return an absolute value. Unlimited-travel knobs, including the mouse wheel, may be spun continuously and return relative data. Knobs are generally not self-centering; they stay where the user puts them. Knobs, especially large ones, offer fine unidimensional control.

Converting a game designed for a knob to a joystick seldom produces good results, for the same reason that a mouse-based game does not convert well to a joystick: The joystick's combination of limited travel and self-centering contradicts the game's original design. (Also, while the game ignores the second value from the 2D joystick, the additional freedom of movement confuses the player somewhat.) The arcade game *Tempest* used a large, heavy knob that could be spun continuously; when ported to a console machine with a joystick, players enjoyed the game less despite the improved graphics.

A slider is a small handle that moves along a slot in the controller, which constrains its travel. It returns an absolute position and stays where the player puts it. You find sliders usually used as adjuncts to joysticks for flight simulators; the slider controls the throttle for the engine, letting the player set his speed and leave it there.

A few controllers, such as the Nintendo GameCube controller on the right in Figure 8.2, include analog pressure-sensitive buttons that, instead of transmitting a binary on or off value, send a number that indicates how hard the player presses. This gives the player a finer degree of control than an ordinary binary controller button.

Navigation Mechanisms

Navigation mechanisms allow the player to tell an avatar, vehicle, or other mobile unit how to move. When a player gives movement commands, the avatar must respond in a consistent and predictable way.

We will use the term *avatar* to refer to the general case of a character or object that can be made to move under player control. Designers usually find creating vehicle navigation systems easier than creating ones for characters because input devices more closely resemble a vehicle's controls than they do an avatar's body.

Terminology

When the player uses a joystick, a D-pad, or a mouse and keyboard to control an avatar that walks over a landscape, we refer to the action as *steering*. We assume that players steer using a joystick except where otherwise indicated; for most purposes, you may consider a joystick interchangeable with a D-pad but offering finer control. We don't address steering wheels for cars or control yokes for aircraft here as they should be self-explanatory.

If a vehicle or character can move freely in three dimensions, such as an aircraft or spacecraft, it *flies*.

If the player designates a point in the landscape and the character or vehicle moves to that target without further player control, the game uses *point-and-click navigation.*

Screen-Oriented Steering

In screen-oriented steering, when the player moves the joystick toward the top of the screen, the avatar moves toward the top of the screen. Implementation details vary somewhat depending on the perspective; we briefly document several major variants.

Top-Down and Isometric Perspectives In a top-down or isometric perspective in which the player sees the avatar from above, moving the joystick in the direction of one edge of the screen causes the avatar to instantly turn and face that edge of the screen, then move in that direction. Classic arcade games that used a top-down perspective, such as *Gauntlet,* used this simplest of all steering methods.

2D Side-Scrolling Games In traditional side-scrollers, the joystick controls left and right movement as it does for the top-down perspective. The player controls the avatar's vertical jumps to platforms using a separate controller button. Moving the joystick up can augment the effect of the jump button; moving the joystick down may be left undefined; and because the game world is 2D, the avatar cannot move away from or toward the player.

3D Games Three-dimensional games usually use avatar-oriented rather than screen-oriented steering to provide a consistent set of controls regardless of camera angle, but rare exceptions do exist. *Crash Bandicoot* provides the best-known example. When the player pushes the joystick up, the avatar moves toward the top of the screen, which is also forward into the 3D environment, away from the player. Moving the joystick down makes the avatar turn to face the player and move toward him through the 3D environment. Pushing the joystick left or right makes the avatar turn to face and then move in that direction.

Unlike avatar-oriented steering, in this model left and right cause the avatar to *move* in those directions with the camera continuing to face forward and to show the avatar from the side. In this respect, *Crash Bandicoot* feels rather like a side-scroller with an additional dimension. In avatar-oriented steering, addressed next, left and right cause the avatar to *turn and face* in those directions *but not to move* while the camera swings around to remain behind him.

Avatar-Oriented Steering

In avatar-oriented steering, the only suitable model for first-person games, pushing the joystick up causes the avatar to move forward in whatever direction she currently faces, regardless of her orientation to the screen. However, implementation of avatar-oriented steering varies somewhat from one device to another, so we discuss devices individually in the following sections.

Avatar-oriented steering remains consistent regardless of the perspective. It presents a slight disadvantage in games using aerial perspectives: Avatar-oriented steering can be rather disorienting when the avatar faces the bottom of the screen, yet the player must push the joystick up to make it walk down to the bottom of the screen.

Joystick and D-Pad Controls As stated above, pushing the joystick up makes the avatar move forward in whatever direction she faces. Pushing the joystick down makes the avatar move backward away from the direction she faces, while continuing to face the original direction; that is, she walks backward. In some vehicle simulators, *down* applies the brakes rather than reversing the direction of movement, and the player must press a separate controller button to put the vehicle in reverse. Pushing the joystick to the left or right makes the avatar turn to face toward the left or right or turns the wheels of a vehicle. The avatar does not move in the environment if the joystick moves directly to left or right; the player must push the joystick diagonally to get forward (or backward) motion in addition to a change of direction. This feels more natural with vehicles than it does with characters.

Mouse-Based Control With mouse-based navigation, now standard for first-person PC games, the mouse only controls the direction in which the avatar faces, and the player uses the keyboard to make the avatar move. Moving the mouse left or right causes the avatar to turn in place, to the left or the right, and to a degree in proportion to the distance the mouse moves. Up and down mouse movements tilt the camera up or down, which becomes important if the player wants the avatar to climb or descend, but these commands do not move the avatar. Considerably more flexible than a joystick-based system, mouse-based navigation allows the player to look around without moving the character.

Keys on the PC's keyboard control movement. The standard arrangement for players who use their right hands for the mouse and left hands for the keyboard uses W to produce forward movement in the direction the avatar currently faces; movement continues as long as the player holds the key down. S works similarly for moving backward (or applying the brakes). A and D produce movement at right angles to the direction the avatar faces, left or right respectively, thus producing the feeling of sliding sideways while facing forwards. This sideways movement is often called *strafing*.

Flying

Flying presents a further complication because it involves moving through three dimensions while a two-dimensional input device such as a joystick offers control in only two. Control over movement in the third dimension must be handled by a separate mechanism, either extra controller buttons or an additional joystick. How you implement this depends on the nature of the aircraft itself, generally taking as your model the mechanisms in real aircraft. Navigational controls in modern flying games are almost always intended for use in the first-person perspective from inside the cockpit. (See Chapter 17, "Vehicle Simulations," for further details.)

Fixed-Wing Aircraft The player maneuvers the aircraft using the joystick to *pitch* (the equivalent of a camera's tilt) or roll, and the engine pulls the plane in the direction the nose faces. A throttle control, generally a slider or keys that increase and decrease the engine speed by fixed increments, sets the rate of forward movement. When flying straight and level, forward on the joystick pushes the nose down, producing descent, and back pulls the nose up, causing it to climb. Left on the joystick causes the plane to roll to the left while remaining on the same course; right rolls it to the right in the same manner. To turn in the horizontal plane, the pilot rolls the aircraft in the desired direction and pulls back at the same time, so the nose follows the direction of the roll, producing a banked turn. When the joystick returns to center, the plane should fly straight and level at a speed determined by the throttle.

Helicopters Game user interfaces typically simplify helicopter navigation, which is more complicated than flying fixed-wing aircraft. The joystick controls turning and forward or backward movement, and a slider control or keys cause the helicopter to ascend or descend. Left on the joystick causes the helicopter to turn counterclockwise about its vertical axis but not to actually go in that direction unless also moving forward. Right causes the equivalent rotation to the right. Forward propels the helicopter forward, and back the reverse.

(Real helicopters can also slide sideways while facing forward; to implement this would require extra controls, which few games do.) When the joystick returns to center, the helicopter should gradually slow down through air friction until it remains hovering above a fixed point in the landscape. A separate key set or slider controls vertical movement.

Spacecraft Most designers treat spacecraft as they would fixed-wing aircraft, although in one variant left or right on the joystick causes the vehicle to *yaw* (the equivalent of panning a camera), turning about its vertical axis to face in a different direction, rather than rolling.

Point-and-Click Navigation

Aerial or context-sensitive perspectives in which the player can clearly see his avatar, party, or units as well as a good deal of the surrounding environment can use point-and-click navigation. In a game with a multipresent or party-based model, the player first chooses which unit or units should move (unnecessary in an avatar-based model), then in all cases the player selects a destination in the environment, and the unit or avatar moves to that location automatically using a ***pathfinding*** algorithm (an artificial intelligence technique to avoid obstacles). Typically the player can select one of two speeds: If he holds down a special key while selecting the location, the vehicle moves more quickly (a humanoid character runs rather than walks).

This technique is most often used in real-time strategy and party-based role-playing games in which many units may need to be given their own paths and the player does not have time to control them all precisely. If a unit cannot get to the location the player designated, that unit either goes as far as it can and then stops or, upon receipt of the command, warns the player that it cannot proceed to an inaccessible destination. Point-and-click also used to be common for adventure games but has begun to be replaced by avatar-oriented navigation.

Using point-and-click navigation, the player can indicate precisely where he wants the unit to end up without concerning himself about avoiding obstacles, a convenience in cluttered environments where the player may not clearly see which objects actually block the path. It is also helpful in context-sensitive perspectives because the player cannot always see clearly how the avatar should get from one place to another and often has no freedom to move the camera.

At times, it can be a disadvantage that the player cannot control the path that the unit takes, so point-and-click navigation frequently needs to include a *waypoint* mechanism. This system allows the player to designate intermediate points, called waypoints, that the unit must pass through one by one on its way to the final destination. Waypoints enable the player to plot a route for the unit and so exercise some control over how the units get to where they are going.

Allowing for Customization

One of the most useful, and at the same time easiest to design, features you can offer your player is to allow him to customize his input devices to suit himself. Normally you handle this via a shell menu, although a few PC games store the information in a text file that the player can edit. Two of the most commonly needed customizations are:

- **Swap left and right mouse buttons.** If the mouse has more than one button, left-handed and right-handed players will need different layouts. Providing a mirror image of your standard layout takes little trouble, so don't make players go through a function-reassignment process just for this; give them a feature that allows them to simply swap the current assignments.
- **Swap the up and down directions of the mouse or joystick in first-person 3D games.** Some players like to push the mouse or joystick up to make their avatar look up (an idea borrowed from screen-oriented steering in 2D games); others like to pull it down to look up (an idea borrowed from airplane joysticks). Either works just as well, and you might as well let the player play as she prefers—so let the players reorient these signals if they want.

The term ***degree of freedom*** refers to the number of possible dimensions that an input device can move through. An ordinary key or button has one degree of freedom: It can only move up and down. A joystick or mouse has two degrees of freedom: It can move up and down, left and right. If two devices, both binary or both analog, (see our discussion in *Input Devices* earlier) have the same degree of freedom, you can generally let the player interchange them, although there will be practical difficulties if one device is self-centering and the other is not or if one allows unlimited travel and the other does not. When exchanging assignments between two devices not identical in every way, some functionality or convenience is almost always lost.

Almost all games assign some of their player actions to particular keys or buttons. Your game should include a *key reassignment* shell menu that allows players to assign actions to the keys they prefer. If your game includes menus, also allow the player to assign menu items to keys so he can select them quickly without using the mouse. You may need to enforce some requirements: If the game requires that a particular action must be assigned to a key (for example, the fire-weapon action in a shooter game), don't let the player exit the shell menu if the action remains unassigned.

When implementing a shell menu for key reassignment, be sure to show *all* the current assignments, *all* the game features not currently assigned to keys, and *all* the currently unassigned keys on the same screen. Many games don't do this, so when the player wants to assign a feature to a key, he can't tell which keys already carry actions and which do not.

Be sure to save the player's customizations between games, so he doesn't have to set them up every time he plays. If you want to be especially helpful, let players save different setups in separate, named profiles so that each player can have his own set of customizations. Include a *Restore Defaults* option so the player can return his customizations to the original factory settings.

Summary

When game reviewers praise a game highly, they cite its user interface more often than any other aspect of the game as the feature that makes the game great. The gameplay may be innovative, the artwork breathtaking, and the story moving, but a smooth and intuitive user interface improves the player's perception of the game like nothing else.

In this chapter, we have introduced you to interaction models and perspectives, two concepts central to game user interfaces. We looked at ways to manage the complexity of an interface and a number of visual and audio elements that games use, and we examined input devices and navigation mechanisms in detail.

If you tune and polish your interface to the peak of perfection, your players will notice it immediately. Give it that effort, and your work will be well justified.

Test Your Skills

MULTIPLE CHOICE QUESTIONS

1. Which of the following questions reflects information that a player needs to know to be able to play a game?
   A. Am I making progress?
   B. How do I turn off the sound effects?
   C. Can my actions be easily reversed?
   D. Can I customize my character?

2. Steps in designing a game's user interface include all of the following *except*
   A. designing a screen layout.
   B. selecting visual elements that tell a player what she needs to know.
   C. creating narrative material.
   D. defining the inputs for the game.

3. An example of how you could use abstraction to simplify a pioneer adventure game would be to
   A. let the computer handle the feeding of your oxen.
   B. build in an expert mode option for advanced players.
   C. create both broad and deep interfaces.
   D. not model how much sleep your characters need.

4. A broad interface would most likely be used in a game that requires
   A. research through a series of menus and dialog boxes.
   B. easy access to a wide variety of controls.
   C. first-person perspective.
   D. two-dimensional input devices.

5. What interaction model requires you to provide ways for players to give orders to units in the game environment?
   A. Contestant model.
   B. Avatar-based model.
   C. Multipresent model.
   D. Desktop model.

6. When the game's virtual camera moves vertically, it is said to
   A. dolly.
   B. truck.
   C. pan.
   D. crane.

7. When a game uses first-person perspective, the player sees her avatar
   A. if she looks in a pool of water.
   B. from a side view.
   C. straight on from the front.
   D. from a point over the avatar's left shoulder.

8. When using a top-down perspective, artists can improve the look of the environment by
   A. making the tops of the buildings transparent.
   B. making the people as big as the buildings.
   C. cheating the depiction of the buildings.
   D. craning the virtual camera.

9. You are creating a game that gives the player the experience of riding a bicycle in the Tour de France. You would most likely choose to put the game's controls
   A. on the input device only.
   B. around the edges of a windowed view.
   C. on semitransparent overlays.
   D. on context-sensitive overlays.

10. To allow the player to keep track of his progress around France in the Tour de France game, include a
    A. needle gauge showing current health.
    B. small multiple indicator of the days spent riding.
    C. colored overlay of areas that are especially challenging.
    D. mini-map showing the route.

11. If a game's chosen platform supports it, you can inform a player of the location of noisy objects by using
    A. quad audio.
    B. positional audio.
    C. surround audio.
    D. ambient audio.

12. Which of the following is *not* a two-dimensional input device?
    A. D-pad.
    B. Joystick.
    C. Slider.
    D. Mouse.

13. In screen-oriented steering, when the player moves the joystick toward the top of the screen, the avatar
    A. moves to the bottom of the screen.
    B. will not move until a button is pressed.
    C. moves to the next waypoint.
    D. moves to the top of the screen.

14. In avatar-oriented steering for a game that uses mouse control,
    A. the mouse controls the direction the avatar faces, and the keyboard makes the avatar move.
    B. the keyboard controls the direction the avatar faces, and clicking a mouse button makes the avatar move.
    C. the mouse controls both the direction the avatar faces and the speed that the avatar moves.
    D. the mouse controls the direction the avatar faces, and the mouse wheel controls avatar turns.

15. As a general rule, you can allow a player to make all of the following customizations *except*
    A. swapping left and right mouse buttons.
    B. swapping up and down directions of the mouse or joystick.
    C. exchanging key input for mouse input.
    D. changing the default key assigned to a specific action to another key.

EXERCISES

1. Design and draw one icon for each of the following functions in a game:
   - Build (makes a unit build a certain structure)
   - Repair (makes a unit repair a certain structure)
   - Attack (makes a unit attack a certain enemy unit)
   - Move (moves a unit to a certain position)
   - Hide (makes a unit hide to be less visible to enemy units)

   Briefly explain the design choices you made for each icon. All icons should be for the same game, so make them consistent to a game genre of your choice.

2. In this exercise, you will practice designing user interfaces for two different gameplay modes, each of which has different indicators. Using the descriptions of the modes below, decide how best to display the functions to the player and sketch a small screen mock-up showing how these indicators can be positioned on the screen. Briefly explain your design decisions.

   In the primary gameplay mode, the avatar can move around in the game world and do different things such as attacking, talking to NPCs, and so on. The mode is avatar-based in the third-person perspective.

Functions/indicators:

- Character's health
- Character's position in the game world
- Currently chosen weapon
- Waypoint to the next mission
- Character visibility to enemies (indicate that, if the character stands in shadows or in darkness, he is less visible to enemies)

In the secondary gameplay mode, the player will enter vehicle races that include shooting at other vehicles driven by nonplayer characters. The perspective is first person.

Functions/indicators:

- Vehicle health
- Vehicle speed
- Primary weapon ammo left
- Type of secondary weapon mounted, if present (if not present, so indicate)
- Position in race
- Laps remaining of race

3. In this exercise, you will design the same UI, once for breadth and once for depth. Make the broad UI no more than two levels deep at any point. Make the deep UI at least three levels deep at one point, offering no more than three options at the top level. Present the UIs by making flowcharts showing the different levels of interaction or how you group different functions. Include all the following functions. Briefly explain your design decisions.

Attack	Defend	Guard	Patrol	Move
Set waypoint	Choose weapon	Research	Build barracks	Build headquarters
Build hospital	Destroy	Repair	Harvest	Save current game
Load game	Quit game	Change video settings	Change sound settings	Change control settings

4. A game intended for a console needs to have its functions mapped to a game pad with a limited number of buttons. Make a button layout that supports all the actions in the primary gameplay mode (described below). Discuss the pros and cons of your button layout.

The game pad has the following button layout:

- A D-pad
- One analog joystick
- Four face buttons
- Two shoulder buttons

(Note: This is the button layout of the Sony PSP, excluding the start and select buttons.)

The main gameplay mode has the following actions:

Normal attack	Hard attack	High attack (attack upward)	Low attack (attack downward)	Block
Jump	Crouch	Move forward	Move backward	Strafe left
Strafe right	Rotate left	Rotate right	Choose weapon	Use health pack

DESIGN QUESTIONS

As you design the user interface for each mode in your game, consider the following questions:

1. Does the gameplay require a pointing or steering device? Should these be analog, or will a D-pad suffice? What do they actually do in the context of the game?
2. Does the function of one or more buttons on the controller change within a single gameplay mode? If so, what visual cues let the player know this is taking place?
3. If the player has an avatar (whether a person, creature, or vehicle), how do the movements and other behaviors of the avatar map to the machine's input devices? Define the steering mechanism.
4. How will the major elements of your screen be laid out? Will the game use a windowed view, opaque overlays, semitransparent overlays, or a combination?
5. What perspective will the main view use? What interaction model does the gameplay mode use? Is it one of the common ones or something new? How does the perspective support the interaction model?
6. Does the game's genre, if it has one, help to determine the user interface? What standards already exist that the player may be expecting the game to follow? Do you intend to break these expectations, and if so, how will you inform the player of that?

7. Does the game include menus? What is the menu structure? Is it broad and shallow (quick to use, but hard to learn) or narrow and deep (easy to learn, but slow to use)?
8. Does the game include text on the screen? If so, does it need provisions for localization?
9. What icons does the game use? Are they visually distinct from one another and quickly identifiable? Are they culturally universal?
10. Does the player need to know numeric values (score, speed, health)? Can these be presented through nonnumeric means (power bars, needle gauges, small multiples), or should they be shown as digits? If shown as digits, how can they be presented in such a way that they don't harm suspension of disbelief? Will you label the value and if so, how?
11. What symbolic values does the player need to know (safe/danger, locked/unlocked/open)? By what means will you convey both the value and its label?
12. Will it be possible for the player to control the game's perspective? Will it be necessary for the player to do so in order to play the game? What camera controls will be available? Will they be available at all times or from a separate menu or other mechanism?
13. What is the aesthetic style of the game? How do the interface elements blend in and support that style?
14. How will audio be used to support the player's interaction with the game? What audio cues will accompany player actions? Will the game include audio advice or dialog?
15. How does music support the user interface and the game generally? Does it create an emotional tone or set a pace? Can it adapt to changing circumstances?