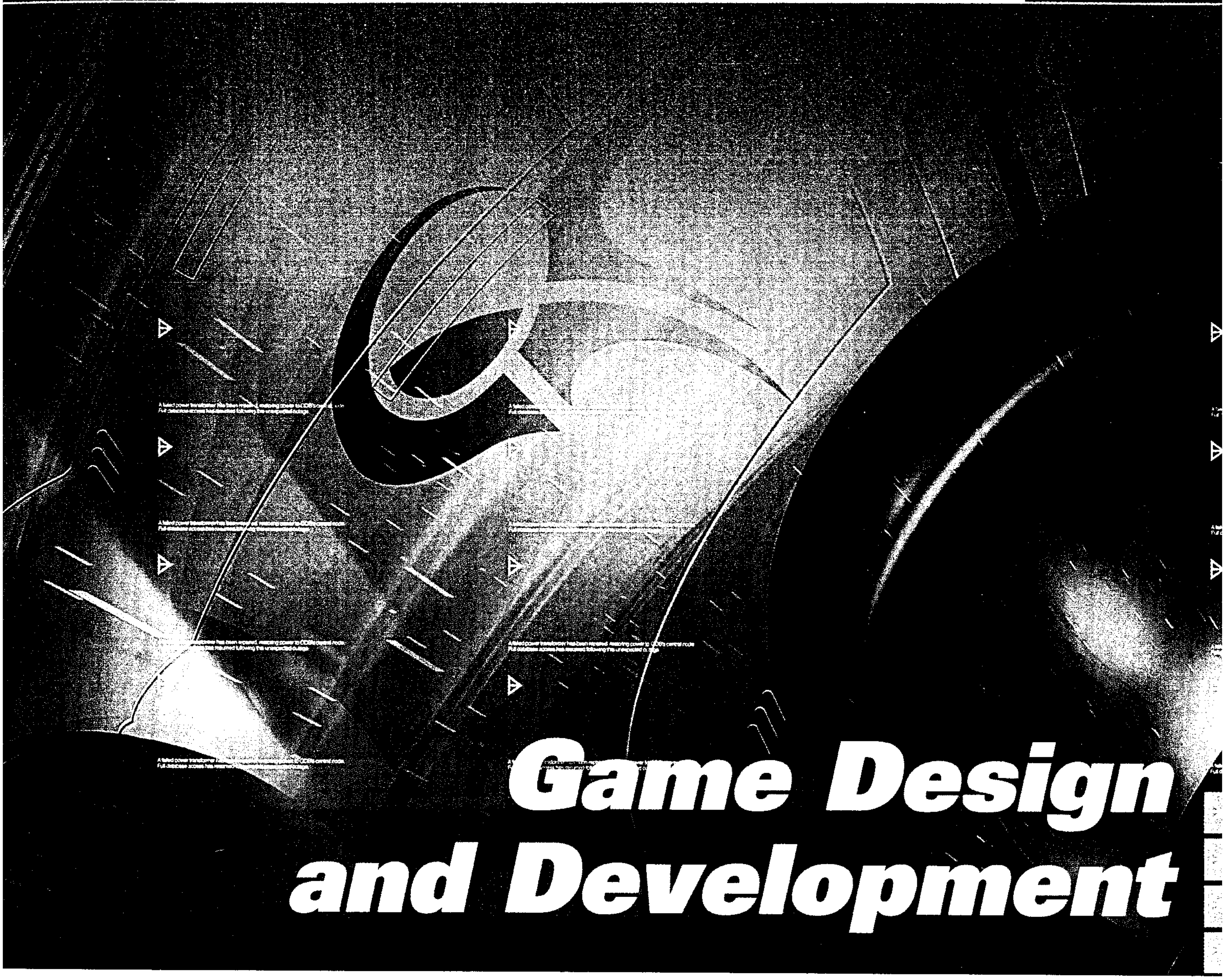
Creating the User Experience



Fundamentals of Game Design

Ernest Adams • Andrew Rollings

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 7 6 5 4 3 2 1
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.

COMMANDMENT: 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.

IN THE TRENCHES: 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).

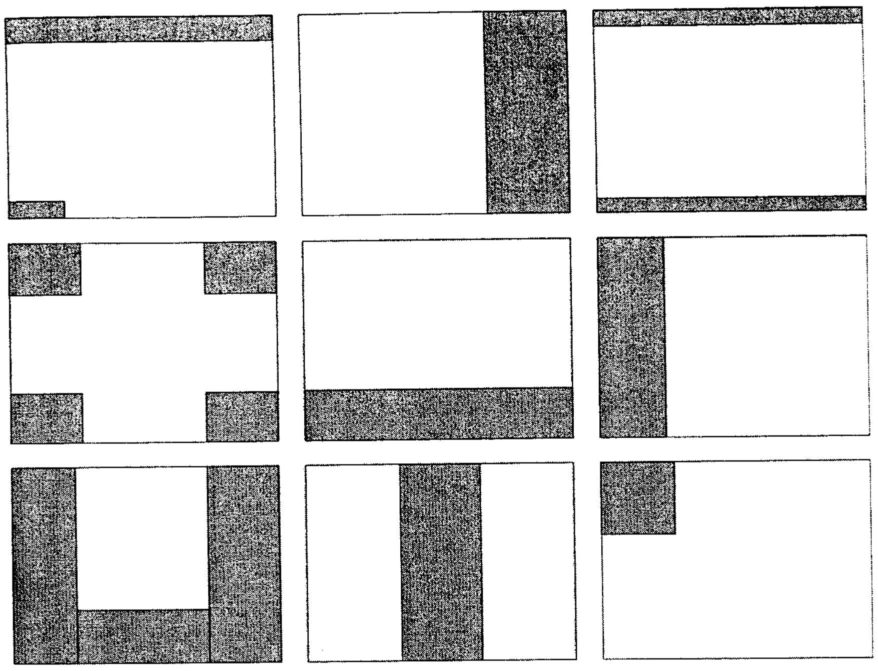


FIGURE 8.1 Common screen layouts.

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

8

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.