

A CRC FREEBOOK



A STRATEGY GUIDE FOR GAME CREATION








V. 2



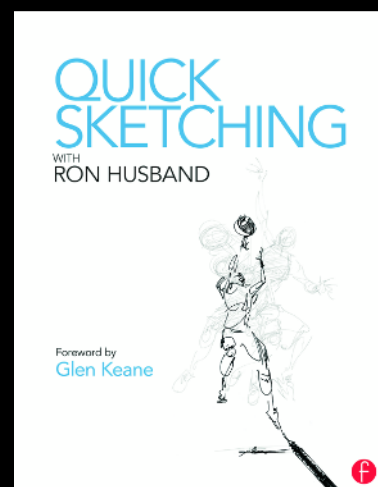
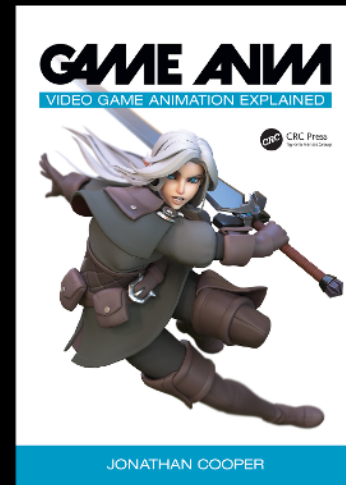
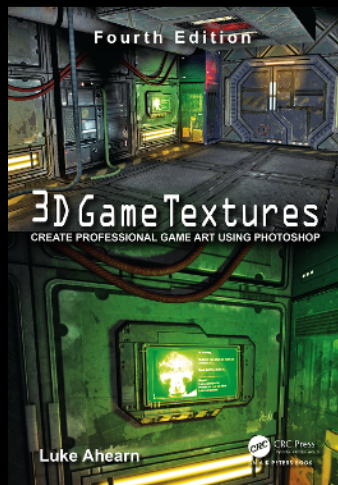
CRC Press
Taylor & Francis Group



TABLE OF CONTENTS

-  Introduction
-  1 • The Role of the Game Designer
-  2 • The Basics of Art
-  3 • The Five Fundamentals of Game Animation
-  4 • Introduction to Game Engine Architecture
-  5 • 3D Concepts
-  6 • The Basics

READ THE LATEST ON GAMING & ANIMATION WITH THESE KEY TITLES



VISIT WWW.CRCPRESS.COM/GAMES-ANIMATION
TO BROWSE OUR FULL RANGE OF TITLES

**SAVE 20% AND FREE STANDARD SHIPPING WITH DISCOUNT CODE
GGCV2**



Introduction

A Strategy Guide for Game Creation V.2 is relevant for those looking to hone their skills in video game development. The new edition provides a brand new set of must know fundamentals and tips in game design.

Game Design Workshop puts you to work prototyping, playtesting, and revising your own games with time-tested methods and tools. These skills will provide the foundation for your career in any facet of the game industry including design, producing, programming, and visual design.

The new edition of **3D Game Textures: Create Professional Game Art Using Photoshop** features the most up-to-date techniques that allow you to create your own unique textures, shaders, and materials.

Taking readers through a complete game production, **Game Anim** provides a clear understanding of expectations of the game animator at every stage, featuring game animation fundamentals and how they fit within an overall project to offer a holistic approach to the field of game animation.

In the new and improved third edition of the highly popular **Game Engine Architecture**, Jason Gregory draws on his nearly two decades of experience at Midway, Electronic Arts and Naughty Dog to present both the theory and practice of game engine software development.

From a steamy jungle to a modern city, or even a sci-fi space station, **3D Game Environments** is the ultimate resource to help you create AAA quality art for a variety of game worlds.

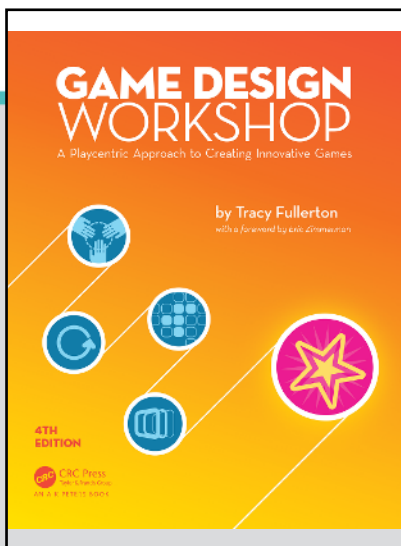
Quick Sketching with Ron Husband offers instruction to quick sketching and all its techniques. From observing positive and negative space and learning to recognize simple shapes in complex forms to action analysis and using line of action, this Disney legend teaches you how to sketch using all these components, and how to do it in a matter of seconds.



CHAPTER

1

THE ROLE OF THE GAME DESIGNER



This chapter is excerpted from
Game Design Workshop
by Tracy Fullerton

© 2018 Taylor & Francis Group. All rights reserved.



[Learn more](#)

Chapter 1

The Role of the Game Designer

The game designer envisions how a game will work during play. She creates the objectives, rules, and procedures; thinks up the dramatic premise and gives it life; and is responsible for planning everything necessary to create a compelling player experience. In the same way that an architect drafts a blueprint for a building or a screenwriter produces the script for a movie, the game designer plans the structural elements of a system that, when set in motion by the players, creates the interactive experience.

As the impact of digital games has increased, there has been an explosion of interest in game design as a career. Now, instead of looking to Hollywood and dreaming of writing the next blockbuster, many creative people are turning to games as a new form of expression.

But what does it take to be a game designer? What kinds of talents and skills do you need? What will be expected of you during the process? And what is the best method of designing for a game? In this chapter, I'll talk about the answers to these questions and outline a method of iterative design that designers can use to judge the success of gameplay against their goals for the player experience throughout the design and development process. This iterative method, which I call the "playcentric" approach, relies on inviting feedback from players early on and is the key to designing games that delight and engage the audience because the game mechanics are developed from the ground up with the player experience at the center of the process.

AN ADVOCATE FOR THE PLAYER

The role of the game designer is, first and foremost, to be an advocate for the player. The game designer must look at the world of games through the player's eyes. This sounds simple, but you'd be surprised how often this concept is ignored. It's far too easy to get caught up in a game's graphics, story line, or new features and forget that what makes a game great is solid gameplay. That's what excites players. Even if they tell you that they love the special effects, art direction, or plot, they won't play for long unless the gameplay hooks them.

As a game designer, a large part of your role is to keep your concentration focused on the player experience and not allow yourself to be distracted by the other concerns of production. Let the art director worry about the imagery, the producer stress over the budget, and the technical director focus on the engine. Your main job is to make sure that when the game is delivered, it provides superior gameplay.

When you first sit down to design a game, everything is fresh and, most likely, you have a vision for

4 Chapter 1: The Role of the Game Designer

what it is that you want to create. At this point in the process, your view of the game and that of the eventual new player are similar. However, as the process unfolds and the game develops, it becomes increasingly difficult to see your creation objectively. After months of testing and tweaking every conceivable aspect, your once-clear view can become muddled. At times like this, it's easy to get too close to your own work and lose perspective.

Playtesters

It is in situations like these when it becomes critical to have playtesters. Playtesters are people who play your game and provide feedback on the experience so that you can move forward with a fresh perspective. By watching other people play the game, you can learn a great deal.

Observe their experience and try to see the game through their eyes. Pay attention to what objects they are focused on, where they touch the screen or move the cursor when they get stuck or frustrated or bored, and write down everything they tell you. They are your guides, and it's your mission to have them lead you inside the game and illuminate any issues lurking below the surface of the design. If you train yourself to do this, you will regain your objectivity and be able to see both the beauty and the flaws in what you've created.

Many game designers don't involve playtesters in their process, or, if they do, it's at the end of production when it's really too late to change the essential elements of the design. Perhaps they are on a tight schedule and feel they don't have time for feedback. Or perhaps they are afraid that feedback will force them to change things they love about their design. Maybe they think that getting a playtest group together will cost too much money. Or they might be under the impression that testing is something only done by large companies or marketing people.

What these designers don't realize is that by divorcing their process from this essential feedback opportunity, they probably cost themselves considerable time, money, and creative heartache. This is because games are not a form of one-way



1.1 Playtest group

communication. Being a superior game designer isn't about controlling every aspect of the game design or dictating exactly how the game should function. It's about building a potential experience, setting all the pieces in place so that everything's ready to unfold when the players begin to participate.

In some ways, designing a game is like being the host of a party. As the host, it's your job to get everything ready—food, drinks, decorations, music to set the mood—and then you open the doors to your guests and see what happens. The results are not always predictable or what you envisioned. A game, like a party, is an interactive experience that is only fully realized after your guests arrive. What type of party will your game be like? Will your players sit like wallflowers in your living room? Will they stumble around trying to find the coatroom closet? Or will they laugh and talk and meet new people, hoping the night will never end?

Inviting players “over to play” and listening to what they say as they experience your game is the best way to understand how your game is working. Gauging reactions, interpreting silent moments, studying feedback, and matching those with specific game elements are the keys to becoming a professional designer. When you learn to listen to your players, you can help your game to grow.

In [Chapter 9](#) on page 277, when I discuss the playtesting process in detail, you'll learn methods and procedures that will help you hold professional-quality

1.2 More playtest groups



playtests and make the most of these tests by asking good questions and listening openly to criticism. For now, though, it's just important to know that playtesting is the heart of the design process explored in this book and that the feedback you receive during these sessions can help you transform your game into a truly enjoyable experience for your players.

Like any living system, games transform throughout their development cycle. No rule is set in stone. No technique is absolute. No particular scheme is the right one. If you understand how fluid the structures are, you can help mold them into the desired shape through repeated testing and careful observation. As a game designer, it's up to you to evolve your game into more than you originally envisioned. That's the art of game design. It's not locking things in place; it's giving birth and parenting. No one, no matter how smart, can conceive and produce a sophisticated game from a blank sheet of paper and perfect it without going through this process. And learning how to work creatively within this process is what this book is all about.

Exercise 1.1: Become a Tester

Take on the role of a tester. Go play a game and observe yourself as you play. Write down what you're doing and feeling. Try to create one page of detailed notes on your behaviors and actions. Then repeat this experience while watching a friend play the same game. Compare the two sets of notes and analyze what you've learned from the process.

Throughout this book, I've included exercises that challenge you to practice the skills that are essential to game design. I've tried to break them down so that you can master them one by one, but by the end of the book, you will have learned a tremendous amount about games, players, and the design process. And you will have designed, prototyped, and playtested at least one original idea of your own. I recommend creating a folder, either digital or analog, of your completed exercises so that you can refer to them as you work your way through the book.

PASSIONS AND SKILLS

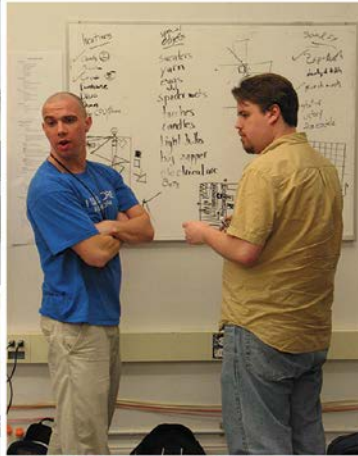
What does it take to become a game designer? There is no one simple answer, no one path to success. There are some basic traits and skills I can suggest, however. First, a great game designer is someone who loves to create playful situations. A passion for games and play is the one thread all great designers have in common. If you don't love what you're doing, you'll never be able to put in the long hours necessary to craft truly innovative games.

To someone on the outside, making games might seem like a trivial task—something that's akin to playing around. But it's not. As any experienced designer can tell you, testing your own game for the ten thousandth time can become work, not play. As the designer, you have to remain dedicated to that ongoing process. You can't just go through the motions. You have to keep that passion alive in yourself, and in the rest of the team, to make sure that the great gameplay you envisioned

in those early days of design is still there in the exhausting, pressure-filled final days before you lock production. To do that, you'll need to develop some other important skills in addition to your love of games and your understanding of the playcentric process.

Communication

The most important skill that you, as a game designer, can develop is the ability to communicate clearly and effectively with all the other people who will be working on your game. You'll have to "sell" your game many times over before it ever hits the store shelves: to your teammates, management, investors, and perhaps even your friends and family. To accomplish this, you'll need good language skills, a crystal-clear vision, and a well-conceived presentation. This is the only way to rally everyone involved to your



1.3 Communicating with team members



cause and secure the support that you'll need to move forward.

But good communication doesn't just mean writing and speaking—it also means becoming a good listener and a great compromiser. Listening to your playtesters and to the other people on your team affords fresh ideas and new directions. Listening also involves your teammates in the creative process, giving them a sense of authorship in the final design that will reinvest them in their own responsibilities on the project. If you don't agree with an idea, you haven't lost anything, and the idea you don't use might spark one that you do.

What happens when you hear something that you don't want to hear? Perhaps one of the hardest things to do in life is compromise. In fact, many game designers think that compromise is a bad word. But compromise is sometimes necessary, and if done well, it can be an important source of creative collaboration.

For example, your vision of the game might include a technical feature that is simply impossible given the available time and resources. What if your programmers come up with an alternative implementation for the feature, but it doesn't capture the essence of the original design? How can you adapt your idea to the practical necessities in such a way as to keep the gameplay intact? You'll have to compromise. As the designer, it's your job to find a way to do it elegantly and successfully so that the game doesn't suffer.

Teamwork

Game production can be one of the most intense collaborative processes you'll ever experience. The interesting and challenging thing about game development teams is the sheer breadth of types of people who work on them. From the hardcore computer scientists, who might be designing the AI or graphic displays, to the talented illustrators and animators who bring the characters to life, to the money-minded executives and business managers who deliver the game to its players, the range of personalities is incredible.



1.4 Team meeting

As the designer, you will interact with almost all of them, and you will find that they all speak different professional languages and have different points of view. Overly technical terms may not translate well to artists or the producer, while the subtle shadings of a character sketch might not be instantly obvious to a programmer. These are generalizations, of course, and many team members may come from multidisciplinary backgrounds, but you can't always count on that. So a big part of your job, and one of the reasons for your documents and specifications, is to serve as a sort of universal translator, making sure that all of these different groups are, in fact, working on the same game.

Throughout this book, I often refer to the game designer as a single team member, but in many cases, the task of game design is a team effort. Whether there is a team of designers on a single game or a collaborative environment where the visual designers, programmers, or producer all have input to the design, the game designer rarely works alone. In [Chapter 12](#) on page 391, I will discuss team structures and how the game designer fits into the complicated puzzle that is a development team.

Process

Being a game designer often requires working under great pressure. You'll have to make critical changes to your game without causing new issues in the process. All too often, a game becomes unbalanced as attempts are made to correct an issue because

8 Chapter 1: The Role of the Game Designer

the designer gets too close to the work and, in the hopes of solving one problem, introduces a host of new problems. But, unable to see this mistake, the designer keeps making changes, while the problems grow worse, until the game becomes such a mess that it loses whatever magic it once had.

Games are fragile systems, and each element is inextricably linked to the others, so a change in one variable can send disruptive ripples throughout. This is particularly catastrophic in the final phases of development, where you run out of time, mistakes are left unfixed, and portions of the game are amputated in hopes of saving what's left. It's gruesome, but it might help you understand why some games with so much potential seem D.O.A.

The one thing that can rescue a game from this terrible fate is instilling in your team the need for good processes from the beginning. Production is a messy business; it is where ideas can get convoluted and objectives can disappear in the chaos of daily crises. But good process, using the playcentric approach of playtesting, and controlled, iterative changes, which I'll discuss throughout this book, can help you stay focused on your goals, prioritize what's truly important, and avoid the pitfalls of an unstructured approach.

Exercise 1.2: D.O.A.

Take one game that you've played that was D.O.A. By D.O.A., I mean "dead on arrival" (i.e., a game that's no fun to play). Write down what you don't like about it. What did the designers miss? How could the game be improved?

Inspiration

A game designer often looks at the world differently from most people. This is in part because of the profession and in part because the art of game design requires someone who is able to see and analyze the underlying relationships and rules of complex systems and to find inspiration for play in common interactions.

When a game designer looks at the world, he often sees things in terms of challenges, structures, and play. Games are everywhere, from how we manage our money to how we form relationships. Everyone has goals in life and must overcome obstacles to achieve those goals. And, of course, there are rules. If you want to win in the financial markets, you have to understand the rules of trading stocks and bonds, profit forecasts, IPOs, and so forth. When you play the markets, the act of investing becomes very similar



1.5 Systems all around us

to a game. The same holds true for winning someone's heart. In courtship, there are social rules that you must follow, and it's in understanding these rules and how you fit into society that helps you to succeed.

If you want to be a game designer, try looking at the world in terms of its underlying systems. Try to analyze how things in your life function. What are the underlying rules? How do the mechanics operate? Are there opportunities for challenge or playfulness? Write down your observations and analyze the relationships. You'll find there is potential for play all around you that can serve as the inspiration for a game. You can use these observations and inspirations as foundations for building new types of gameplay.

Why not look at other games for inspiration? Well, of course, you can and you should. I'll talk about that in just a minute. But if you want to come up with truly original ideas, then don't fall back on existing games for all your ideas. Instead, look at the world around you. Some of the things that have inspired other game designers, and could inspire you, are obvious: personal relationships, buying and selling, competition in the workplace, and on and on. Take ant colonies, for example: They're organized around a sophisticated set of rules, and there's competition both within the colonies and between competing insect groups. Well-known game designer Will Wright made a game about ant colonies in 1991, *SimAnt*. "I was always fascinated by social insects," he says. "Ants are one of the few real examples of intelligence we have that we can study and deconstruct. We're still struggling with the way the human brain works. But if you look at ant colonies, they sometimes exhibit a remarkable degree of intelligence."¹ The game itself was something of a disappointment commercially, but the innate curiosity about how the world works that led Wright to ant colonies has also led him to look at ecological systems such as the Gaia hypothesis as inspiration for *SimEarth* or psychological theories such as Maslow's Hierarchy of Needs as inspiration for artificial intelligence in *The Sims*. Having a strong sense of curiosity and a passion for learning about the world is clearly an important part of Wright's inspiration as a game designer.

What inspires you? Examine things that you are passionate about as systems; break them down in terms of objects, behaviors, relationships, and so forth. Try to understand exactly how each element of the system interacts. This can be the foundation for an interesting game. By practicing the art of extracting and defining the games in all aspects of your life, you will not only hone your skills as a designer, but you'll open up new vistas in what you imagine a game can be.

Exercise 1.3: Your Life as a Game

List five areas of your life that could be games. Then briefly describe a possible underlying game structure for each.

Becoming a Better Player

One way to become an advocate for players is by being a better player yourself. By "better," I don't just mean more skilled or someone who wins all the time—although by studying game systems in depth, you will undoubtedly become a more skilled player. What I mean is using yourself and your experiences with games to develop an unerring sense for good gameplay. The first step to practicing any art form is to develop a deep understanding of what makes that art form work. For example, if you've ever studied a musical instrument, you've probably learned to hear the relationship between the various musical tones. You've developed an ear for music. If you've studied drawing or painting, it's likely that your instructor has urged you to practice looking carefully at light and texture. You've developed an eye for visual composition. If you are a writer, you've learned to read critically. And if you want to be a game designer, you need to learn to play with the same conscious sensitivity to your own experience and critical analysis of the underlying system that these other arts demand.

The following chapters in this section look at the formal, dramatic, and dynamic aspects of games. Together, the concepts in these chapters form a set of tools that you can use to analyze your gameplay experiences and become a better, or more

articulate, player and creative thinker. By practicing these skills, you will develop a game literacy that will make you a better designer. Literacy is the ability to read and write a language, but the concept can also be applied to media or technology. Being game literate means understanding how game systems work, analyzing how they make meaning, and using your understanding to create your own game systems.

I recommend writing your analysis in a game journal. Like a dream journal or a diary, a game journal can help you think through experiences you've had and to remember details of your gameplay long afterwards. As a game designer, these are valuable insights that you might otherwise forget. It is important when writing in your game journal to try to think deeply about your game experience—don't just review the game and talk about its features. Discuss a meaningful moment of gameplay. Try to remember it in detail—why did it strike you? What did you think, feel, do, and so forth? What are the underlying mechanics that made the moment work? The dramatic aspects? Perhaps your insights will form the basis for a future design, perhaps not. But, like sketching or practicing scales on a musical instrument, the act of writing and thinking about design will help you to develop your own way of thinking about games, which is critical to becoming a game designer.

Exercise 1.4: Game Journal

Start a game journal. Don't just try to describe the features of the game, but dig deeply into the choices you made, what you thought and felt about those choices, and the underlying game mechanics that supports those choices. Go into detail; look for the reasons *why* various mechanics of the game exist. Analyze why one moment of gameplay stands out and not another. Commit to writing in your game journal every day.

Creativity

Creativity is hard to quantify, but you'll definitely need to access your creativity to design great

games. Everyone is creative in different ways. Some people come up with lots of ideas without even trying. Others focus on one idea and explore all of its possible facets. Some sit quietly in their rooms thinking to themselves, while others like to bounce ideas around with a group, and they find the interaction to be stimulating. Some seek out stimulation or new experiences to spark their imaginations. Great game designers like Will Wright tend to be people who can tap into their dreams and fantasies and bring those to life as interactive experiences.

Another great game designer, Nintendo's Shigeru Miyamoto, says that he often looks to his childhood and to hobbies that he enjoys for inspiration. "When I was a child, I went hiking and found a lake," he says. "It was quite a surprise for me to stumble upon it. When I traveled around the country without a map, trying to find my way, stumbling on amazing things as I went, I realized how it felt to go on an adventure like this."² Many of Miyamoto's games draw from this sense of exploration and wonder that he remembers from childhood.

Think about your own life experiences. Do you have memories that might spark the idea for a game? One reason that childhood can be such a powerful inspiration for game designers is that when we are children, we are particularly engrossed in playing games. If you watch how kids interact on a playground, it's usually through gameplaying. They make games and learn social order and group dynamics from their play. Games permeate all aspects of kids' lives and are a vital part of their developmental process. So if you go back to your childhood and look at things that you enjoyed, you'll find the raw material for games right there.

Exercise 1.5: Your Childhood

List ten games you played as a child, for example, hide and seek, four square, and tag. Briefly describe what was compelling about each of those games.



1.6 You Don't Know Jack

Creativity might also mean putting two things together that don't seem to be related—like Shakespeare and the Brady Bunch. What can you make of such a strange combination? Well, the designers of *You Don't Know Jack* used silly combinations of high- and low-brow knowledge like this to create a trivia game that challenged players to be equally proficient in both. The result was a hit game with such creative spark that it crossed the usual boundaries of gaming, appealing to players old and young, male and female.

Sometimes creative ideas just come to you, and the trick is to know when to stand by a game idea that seems far-fetched. Keita Takahashi, designer of the quirky and innovative hit game *Katamari Damacy*, was given an assignment while working at Namco to come up with an idea for a racing game. The young artist and sculptor wanted to do something more original than a racing game, however, and says he just “came up with” the idea for the game mechanic of a sticky ball, or *katamari*, that players could roll around, picking up objects that range from paper clips and sushi to palm trees and policemen. Takahashi has said inspiration for the game came from sources as wildly different as the paintings of Pablo Picasso, the novels of John Irving, and Playmobil brand toys, but it is also clear that Takahashi has been influenced by Japanese children's games and sports such as *tamakorogashi* (ballroller) as a designer and is thinking beyond digital games for his future creations. “I would like to



1.7 Beautiful *katamari* and *tamakorogashi*

create a playground for children,” he said. “A normal playground is flat but I want an undulating one, with bumps.”³

I recently designed a game about Henry David Thoreau's time at Walden Pond. I was inspired by his writings and by the thought that underlying his philosophical experiment was an interesting set of rules that he was “playing by” when he set out to “live deliberately.” The game took ten years to make and required a deep commitment to the original idea over those years. When we started making it, the idea of an indie game “about” something like a philosopher's experiment in living was considered somewhat strange and new. Today, personal games, and games about ideas or experiences, are relatively common, especially in the indie space.

Our past experiences, our other interests, our relationships, and our identity all come into play when trying to reach our creativity. Great game designers find a way to tap into their creative souls and bring forth the best parts in their games. However you do it, whether you work alone or in a team, whether you read books or climb mountains, whether you look to other games for inspiration or to life experiences, the bottom line is that there's no single right way to go about it. Everyone has a different style for coming up with ideas and being creative. What matters is not the spark of an idea but what you do with that idea once it emerges, and this is where the playcentric process becomes critical.

A PLAYCENTRIC DESIGN PROCESS

Having a good solid process for developing an idea from the initial concept into a playable and satisfying game experience is another key to thinking like a game designer. The playcentric approach I will illustrate in this book focuses on involving the player in your design process from conception through completion. By that I mean continually keeping the player experience in mind and testing the gameplay with target players through every phase of development.

Setting Player Experience Goals

The sooner you can bring the player into the equation, the better, and the first way to do this is to set “player experience goals.” Player experience goals are just what they sound like: goals that the game designer sets for the type of experience that players will have during the game. These are not features of the game but rather descriptions of the interesting and unique situations in which you hope players will find themselves. For example, “players will have to cooperate to win, but the game will be structured so they can never trust each other,” “players will feel a sense of happiness and playfulness rather than competitiveness,” or “players will have the freedom to pursue the goals of the game in any order they choose.”

Setting player experience goals up front, as a part of your brainstorming process, can also focus your creative process. Notice that these descriptions do not talk about how these experience goals will be implemented in the game. Features will be brainstormed later to meet these goals, and then they will be playtested to see if the player experience goals are being met. At first, though, I advise thinking at a very high level about what is interesting and engaging about your game to players while they are playing and what experiences they will describe to their friends later to communicate the high points of the game.

Learning how to set interesting and engaging player experience goals means getting inside the heads of the players, not focusing on the features of the game as you intend to design it. When you’re just beginning to design games, one of the hardest things

to do is to see beyond features to the actual game experience the players are having. What are they thinking as they make choices in your game? How are they feeling? Are the choices you’ve offered as rich and interesting as they can be?

Prototyping and Playtesting

Another key component to playcentric design is that ideas should be prototyped and playtested early. I encourage designers to construct a playable version of their idea immediately after brainstorming ideas. By this I mean a physical prototype of the core game mechanics. A physical prototype can use paper and pen or index cards or even be acted out. It is meant to be played by the designer and her friends. The goal is to play and perfect this simplistic model before a single programmer, producer, or graphic artist is ever brought onto the project. This way, the game designer receives instant feedback on what players think of the game and can see immediately if they are achieving their player experience goals.

This might sound like common sense, but in the industry today, much of the testing of the core game mechanics comes later in the production cycle, which can lead to disappointing results. Because many games are not thoroughly prototyped or tested early, flaws in the design aren’t identified until late in the process—in some cases, too late to fix. People in the industry are realizing that this lack of player feedback means that many games don’t reach their full potential, and the process of developing games needs to change if that problem is to be solved. The work of professional user research experts like Nicole Lazzaro of XEODesign and Dennis Wixon of Microsoft (see their sidebars on pages 282 and 303) is becoming more and more important to game designers and publishers in their attempts to improve game experiences, especially with the new, sometimes inexperienced, game players that are being attracted to platforms like smartphones or tablets. You don’t need to have access to a professional test lab to use the playcentric approach. In

DESIGNERS YOU SHOULD KNOW

The following is a list of designers who have had a monumental impact on digital games. The list was hard to finalize because so many great individuals have contributed to the craft in so many important ways. The goal was not to be comprehensive but rather to give a taste of some designers who have created foundational works and who it would be good for you, as an aspiring designer yourself, to be familiar with. I'm pleased that many designers on the list contributed interviews and sidebars to this book.

Shigeru Miyamoto

Miyamoto was hired out of industrial design school by Nintendo in 1977. He was the first staff artist at the company. Early in his career, he was assigned to a submarine game called *Radarscope*. This game was like most of the games of the day—simple twitch-game play mechanics, no story, and no characters. He wondered why digital games couldn't be more like the epic stories and fairy tales that he knew and loved from childhood. He wanted to make adventure stories, and he wanted to add emotion to games. Instead of focusing on *Radarscope*, he made up his own beauty-and-the-beast-like story where an ape steals his keeper's girlfriend and runs away. The result was *Donkey Kong*, and the character that you played was Mario (originally named Jumpman). Mario is perhaps the most enduring character in games and one of the most recognized characters in the world. Each time a new console is introduced by Nintendo—starting with the original NES machine—Miyamoto designs a Mario game as its flagship title. He is famous for the wild creativity and imagination in his games. Aside from all the Mario and Luigi games, Miyamoto's list of credits is long. It includes the games *Zelda*, *Starfox*, and *Dikmin*.

Will Wright

Early in his career, in 1987, Wright created a game called *Raid on Bungling Bay*. It was a helicopter game where you attacked islands. He had so much fun programming the little cities on the islands that he decided that making cities was the premise for a fun game. This was the inspiration for *SimCity*. When he first developed *SimCity*, publishers were not interested because they didn't believe anyone would buy it. But Wright persisted, and the game became an instant hit. *SimCity* was a breakout in terms of design in that it was based on creating rather than destroying. Also, it didn't have set goals. These things added some new facets to games. Wright was always interested in simulated reality and has done more than anyone in bringing simulation to the masses. *SimCity* spawned a whole series of titles, including *SimEarth*, *SimAnt*, *SimCopter*, and many others. His game *The Sims* is currently the bestselling game of all time, and *Spore*, his most ambitious project yet, explores new design territory in terms of user-created content. See "A Conversation with Will Wright by Celia Pearce" on page 183.

Sid Meier

Legend has it that Sid Meier bet his buddy, Bill Stealey, that within two weeks he could program a better flying combat game than the one they were playing. Stealey took him up on the offer, and together they founded the company Micro Prose. It took more than two weeks, but the company released the title *Solo Flight* in 1984. Considered by many to be the father of PC gaming, Meier went on to create groundbreaking title after groundbreaking title. His *Civilization* series has had a fundamental influence on the genre of PC strategy games. His game *Sid Meier's Pirates!* was an innovative mix of genres—action, adventure, and role-playing—that also

blended real-time and turn-based gaming. His gameplay ideas have been adopted in countless PC games. Meier's other titles include *Colonization*, *Sid Meier's Gettysburg!*, *Alpha Centauri*, and *Silent Serv*.

Warren Spector

Warren Spector started his career working for board game maker Steve Jackson Games in Austin, Texas. From there, he went on to the paper-based role-playing game company TSR, where he developed board games and wrote RPG supplements and several novels. In 1989, he was ready to add digital games to his portfolio and moved to the developer ORIGIN Systems. There, he worked on the *Ultima* series with Richard Garriott. Spector had an intense interest in integrating characters and stories into games. He pioneered "free-form" gameplay with a series of innovative titles, including *Underworld*, *System Shock*, and *Thief*. His title *Deus Ex* took the concepts of flexible play and drama in games to new heights and is considered one of the finest PC games of all time. See his "Designer Perspective" interview on page 27.

Brenda Romero

Brenda Romero began her career at Sir-tech Software as part of the *Wizardry* role-playing team, where she worked her way up from testing to designer for *Wizardry 8*. While at Sir-tech, she also worked on the *Jagged Alliance* and *Realms of Arkania* series before moving to Atari to work on *Dungeons & Dragons*. Throughout her career, she has been a passionate advocate for diversity in the industry and was awarded the Ambassador Award from the Game Developers Conference as well as a special British Academy for Film and Television Arts award for her contributions to the industry. On page 88, she discusses her groundbreaking analog game series *The Mechanic Is the Message*.

Richard Garfield

In 1990, Richard Garfield was an unknown mathematician and part-time game designer. He had been trying unsuccessfully to sell a board game prototype called *RoboRally* to publishers for seven years. When yet another publisher rejected his concept, he was not surprised. However, this time the publisher, a man named Peter Adkison doing business as *Wizards of the Coast*, asked for a portable card game that was playable in under an hour. Garfield took the challenge and developed a dueling game system where each card in the system could affect the rules in different ways. It was a breakthrough in game design because the system was infinitely expandable. The game was *Magic: The Gathering*, and it singlehandedly spawned the industry of collectible card games. *Magic* has been released in digital format in multiple titles. When Hasbro bought *Wizards of the Coast* in 1995 for \$325 million, Garfield owned a significant portion of the company. See his article "The Design Evolution of *Magic: The Gathering*" on page 219.

Amy Hennig

Amy Hennig began her career in the game industry working as an artist and animator on games for the NES. While she was working at Electronic Arts as an artist on *Michael Jordan: Chaos in the Windy City*, the lead designer left the project and Hennig landed the job. Later, she moved to Crystal Dynamics, where she was director, producer, and writer for *Legacy of Kain: Soul Reaver*. She is well known for her work as a game director and writer on some of the most successful titles in the industry, including the *Uncharted* series for

Naughty Dog and Sony. She has been awarded two Writers Guild of America Video Game Writing Awards in addition to numerous other awards for her work on the Uncharted games. She describes her writing work on this series as being on the “bleeding edge” of the genre of cinematic video games.

Peter Molyneux

The story goes that it all started with an anthill. As a child, Peter Molyneux toyed with one—tearing it down in parts and watching the ants fight to rebuild, dropping food into the world and watching the ants appropriate it, and so on. He was fascinated by the power he had over the tiny, unpredictable creatures. Molyneux went on to become a programmer and game designer and eventually the pioneer of digital “god games.” In his breakout title, *Populous*, you act as a deity lording it over tiny settlers. The game was revolutionary in that it was a strategy game that took place in real time, as opposed to in turns, and you had indirect control over your units. The units had minds of their own. This game and other Molyneux hits had a profound influence on the real-time strategy (RTS) games that were on the horizon. Other titles he has created include *Syndicate*, *Theme Park*, *Dungeon Keeper*, and *Black & White*.

Gary Gygax

In the early 1970s, Gary Gygax was an insurance underwriter in Lake Geneva, Wisconsin. He loved all kinds of games, including tabletop war games. In these games, players controlled large armies of miniatures, acting like generals. Gygax and his friends had fun acting out the personas of different pieces on the battlefield such as commanders, heroes, and so forth. He followed his inclination of what was fun and created a system for battling small parties of miniatures in a game he called *Chainmail*. From there players wanted even more control over and more character information about the individual units. They wanted to play the role of single characters. Gygax, in conjunction with game designer Dave Arneson, developed an elaborate system for role-playing characters that was eventually named *Dungeons & Dragons*. The D&D game system is the direct ancestor of every paper-based and digital RPG since then. The system is directly evident in all of today’s RPGs, including *Diablo*, *Baldur’s Gate*, and *World of Warcraft*.

Richard Garriott

Richard Garriott—a.k.a. “Lord British”—programmed his first game right out of high school in 1979. It was an RPG called *Akalabeth*. He sold it on his own through a local computer store in Austin, Texas. The packaging for this first version was a Ziploc bag. *Akalabeth* later got picked up by a publisher and sold well. Garriott used what he learned to create *Ultima*, one of the most famous game series of all time. The *Ultima* titles evolved over the years—each successive one pushing the envelope in terms of both technology and gameplay—eventually bringing the world of the game online. *Ultima Online*, released in 1997, was a pioneering title in massively multiplayer online worlds. Garriott continues to push the boundaries of online gaming with work on the science fiction MMO *Tabula Rasa*.

Dona Baily

Dona Baily was a young programmer in 1981 who, along with Ed Logg, created the classic arcade video game *Centipede*. At the time, when Baily joined Atari’s coin-op division, she was the only woman employed there. When given a notebook of ideas for possible games to program, all of which involved “lasering or frying

things,” she chose a short description of a bug winding down the screen because, she said, “it didn’t seem bad to shoot a bug.” Centipede went on to become one of the most commercially successful games from the arcade era’s golden age.

Gerald Lawson

Gerald Lawson was an electronic engineer known for his work in the 1970s, designing the Fairchild Channel F video game system and inventing the video game cartridge. The Fairchild Channel F console, while not a commercially successful product, introduced the idea that game software could be stored on swappable cartridges for the first time. Prior to the Channel F, most game systems had the game software programmed into the architecture of the hardware, so games could never be added to or updated. Lawson’s invention was so novel that every cartridge he produced had to be approved by the FCC before distribution as new product. Quickly, his invention became the standard for all future game consoles. Lawson was one of the few African-American engineers working in the industry at that time.

Chapter 9, I describe a number of methods you can use on your own to produce useful improvements to your game design.

I suggest that you do not begin production without a deep understanding of your player experience goals and your core mechanics. This is critical because when the production process commences, it becomes increasingly difficult to alter the software design. Therefore, the further along the design and prototyping are before the production begins, the greater the likelihood of avoiding costly mistakes. You can ensure that your core design concept is sound before production begins by taking a player-centric approach to the design and development process.

Iteration

By “iteration” I simply mean that you design, test, and evaluate the results over and over again throughout the development of your game, each time improving upon the gameplay or features, until the player experience meets your criteria. Iteration is deeply important to the playcentric process. Here is a detailed flow of the iterative process that you should go through when designing a game:

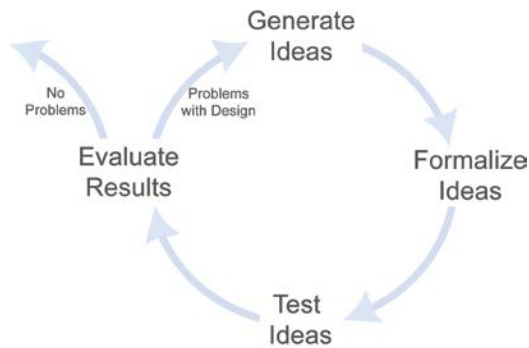
- Player experience goals are set.
- An idea or system is conceived.

- An idea or system is formalized (i.e., written down or prototyped).
- An idea or system is tested against player experience goals (i.e., playtested or exhibited for feedback).
- Results are evaluated and prioritized.
- If results are negative and the idea or system appears to be fundamentally flawed, go back to the first step.
- If results point to improvements, modify and test again.
- If results are positive and the idea or system appears to be successful, the iterative process has been completed.

As you will see, this process is applicable during every aspect of game design, from initial conception through final quality assurance testing.

Step 1: Brainstorming

- Set player experience goals.
- Come up with game concepts or mechanics that you think might achieve your player experience goals.
- Narrow the list down to the top three.
- Write up a short, one-page description for each of these ideas, sometimes called a treatment or concept document.



1.8 Iterative process diagram

- Test your written concepts with potential players (you might also want to create rough visual mock-ups of your ideas at this stage to help communicate the ideas).

Step 2: Physical Prototype

- Create a playable prototype using pen and paper or other craft materials.
- Playtest the physical prototype using the process described in [Chapters 7 and 9](#).
- When the physical prototype demonstrates working gameplay that achieves your player experience goals, write a three- to six-page gameplay treatment describing how the game functions.

Step 3: Presentation (Optional)

- A presentation is often made to secure funds to hire the prototyping team. Even if you do not require funding, going through the exercise of creating a full presentation is a good way to think through your game and introduce it to team members and upper management for feedback.
- Your presentation should include demo artwork and a solid gameplay treatment.
- If you do not secure funding, you can either return to step 1 and start over again on a new concept or solicit feedback from your funding sources and work on modifying the game to fit their needs. Because you have not yet invested in extensive artwork or programming, your costs so far should

be pretty reasonable, and you should have a great deal of flexibility to make any changes.

Step 4: Software Prototype(s)

- When you have your prototyping team in place, you can begin creating rough digital models of the core gameplay. Often, several software prototypes are made, each focusing on different aspects of the system. Digital prototyping is discussed in [Chapter 8](#) beginning on page 241. (If possible, try to do this entirely with temporary graphics that cost very little to make. This will save time and money and speed up the process.)
- Playtest the software prototype(s) using the method process described in [Chapter 9](#).
- When the software prototype(s) demonstrate working gameplay that achieves your player experience goals, move on to develop plans for the full feature set and levels of the game.

Step 5: Design Documentation

- While you have been prototyping and working on your gameplay, you have probably been compiling notes and ideas for the “real” game. Use the knowledge you’ve gained during this prototyping stage to develop a full list of goals for the game, which are documented in a way that is useful and accessible for the team.
- Recently, many designers have moved away from creating large static documents for this purpose, moving instead toward online groupware like wikis and smaller, as-needed form documentation because of the flexible, collaborative nature of modern design processes. The design documentation that comes out of your production process should be thought of as a collaboration tool that changes and grows with production.

Step 6: Production

- Work with all team members to make sure your goals are clear and achievable and that the team is on board with the priorities for these goals.

THE ITERATIVE DESIGN PROCESS

by Eric Zimmerman, game designer and professor, NYU Game Center

Eric Zimmerman is a game designer and a twenty-year veteran of the game industry. Eric cofounded Gamelab, an award-winning New York City-based studio that helped invent casual games with titles like Diner Dash. Other projects range from the pioneering independent online game SiSSYFiGHT 2000 to tabletop games like the strategy board game Quantum and Local No. 12's card game The Metagame. Eric has also created game installations with architect Nathalie Pozzi that have been exhibited in museums and festivals around the world. He is the coauthor with Katie Salen of Rules of Play and is a founding faculty and arts professor at the NYU Game Center. Also see his article with Nathalie Pozzi on playtesting methods on page 293.

The following excerpt is adapted from a longer essay entitled "Play as Research," which appears in the book Design Research, edited by Brenda Laurel (MIT Press, 2004). It appears here with permission from the author. Iterative design is a design methodology based on a cyclic process of prototyping, testing, analyzing, and refining a work in progress. In iterative design, interaction with the designed system is the basis of the design process, informing and evolving a project as successive versions, or iterations, of a design are implemented. This sidebar outlines the iterative process as it occurred in one game with which I was involved—the online multiplayer game SiSSYFiGHT 2000.

What is the process of iterative design? Test, analyze, refine. And repeat. Because the experience of a player cannot ever be completely predicted, in an iterative process design, decisions are based on the experience of the prototype in progress. The prototype is tested, revisions are made, and the project is tested once more. In this way, the project develops through an ongoing dialogue between the designers, the design, and the testing audience.

In the case of games, iterative design means playtesting. Throughout the entire process of design and development, your game is played. You play it. The rest of the development team plays it. Other people in the office play it. People visiting your office play it. You organize groups of testers that match your target audience. You have as many people as possible play the game. In each case, you observe them, ask them questions, then adjust your design and playtest again.

This iterative process of design is radically different from typical retail game development. More often than not, at the start of the design process for a computer or console title, a game designer will think up a finished concept and then write an exhaustive design document that outlines every possible aspect of the game in minute detail. Invariably, the final game never resembles the carefully conceived original. A more iterative design process, on the other hand, will not only streamline development resources, but it will also result in a more robust and successful final product.

Case Study: SiSSYFiGHT 2000

SiSSYFiGHT 2000 is a multiplayer online game in which players create a schoolgirl avatar and then vie with three to six players for dominance of the playground. Each turn, a player selects one of six actions to take, ranging from teasing and tattling to cowering and licking a lolly. The outcome of an action is dependent on other players' decisions, making for highly social gameplay. SiSSYFiGHT 2000 is also a robust online community. You

can play the game at www.sissyfight.com. In the summer of 1999, I was hired by Word.com to help them create their first game. We initially worked to identify the project's play values: the abstract principles that the game design would embody. The list of play values we created included designing for a broad audience of nongamers, a low technology barrier, a game that was easy to learn and play but deep and complex, gameplay that was intrinsically social, and, finally, something that was in line with the smart and ironic Word.com sensibility.

These play values were the parameters for a series of brainstorming sessions interspersed with group play of computer and noncomputer games. Eventually, a game concept emerged: little girls in social conflict on a playground. While every game embodies some kind of conflict, we were drawn toward modeling a conflict that we hadn't seen depicted previously in a game. Technology and production limitations meant that the game would be turn based, although it could involve real-time chat.

When these basic formal and conceptual questions had begun to be mapped out, the shape of the initial prototype became clear. The very first version of SiSSyFiGHt was played with Post-it Notes around a conference table. I designed a handful of basic actions each player could take, and acting as the program, I "processed" the actions each turn and reported the results back to the players, keeping score on a piece of paper.

Designing a first prototype requires strategic thinking about how to most quickly implement a playable version that can begin to address the project's chief uncertainties in a meaningful way. Can you create a paper version of your digital game? Can you design a short version of a game that will last much longer in its final form? Can you test the interaction pattern of a massively multiplayer game with just a handful of players?

In the iterative design process, the most detailed thinking you need at any moment is that which will get you to your next prototype. It is, of course, important to understand the big picture as well: the larger conceptual, technical, and design questions that drive the project as a whole. Just be sure not to let your design get ahead of your iterative research. Keep your eye on the prize, but leave room for play in your design, for the potential to change as you learn from your playtesting, accepting the fact that some of your assumptions will undoubtedly be wrong.

The project team continued to develop the paper prototype, seeking the balance between cooperation and competition that would become the heart of the final gameplay. We refined the base rule set—the actions a player can take each turn and the outcomes that result. These rules were turned into a specification for the first digital prototype: a text-only version on IRC, which we played hot-seat style, taking turns sitting at the same computer. Constructing that early, text-only prototype allowed us to focus on the complexities of the game logic without worrying about implementing interactivity, visual and audio aesthetics, and other aspects of the game.

While we tested gameplay via the text-only iteration, programming for the final version began in Director, and the core game logic we had developed for the IRC prototype was recycled into the Director code with little alteration. Parallel to the game design, the project's visual designers had begun to develop the graphic



SiSSyFiGHt 2000 Interface

language of the game and chart out possible screen layouts. These early drafts of the visuals (revised many times over the course of the entire development) were dropped into the Director version of the game, and the first rough-hewn iteration of SiSSyFiGHT as a multiplayer online game took shape, inspired by Henry Darger's outsider art and retro game graphics.

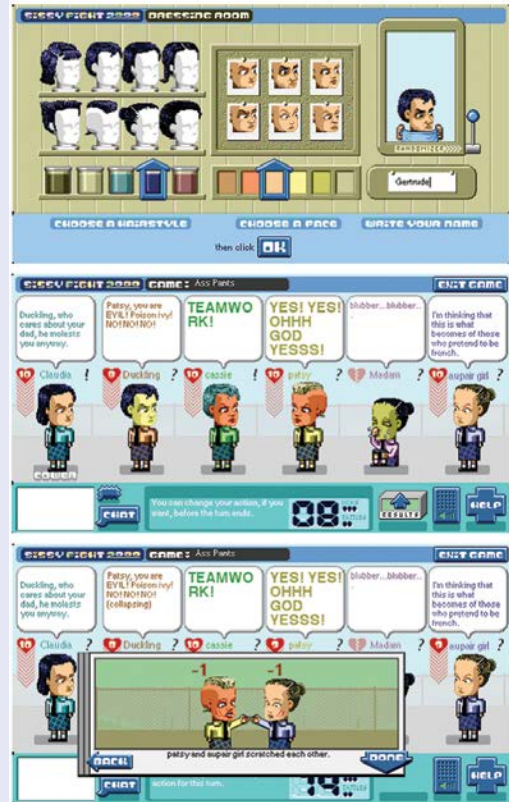
As soon as the web version was playable, the development team played it. And as our ugly duckling grew more refined, the rest of the Word.com staff was roped into testing as well. As the game grew more stable, we descended on our friends' dot-com companies after the workday had ended, sitting them down cold in front of the game and letting them play. All of this testing and feedback helped us refine the game logic, visual aesthetics, and interface. The biggest challenge turned out to be clearly articulating the relationship between player action and game outcome: Because the results of every turn are interdependent on each player's actions, early versions of the game felt frustratingly arbitrary. Only through many design revisions and dialogue with our testers did we manage to structure the results of each turn to unambiguously communicate what had happened that round and why.

When the server infrastructure was completed, we launched the game to an invitation-only beta tester community that slowly grew in the weeks leading up to public release. Certain time slots were scheduled as official testing events, but our beta users could come online anytime and play. We made it very easy for the beta testers to contact us and e-mail in bug reports.

Even with this small sample of a few dozen participants, larger play patterns emerged. For example, as with many multiplayer games, it was highly advantageous to play defensively, leading to standstill matches. In response, we tweaked the game logic to discourage this play style: Any player that "cowered" twice in a row was penalized for acting like a chicken. When the game did launch, our loyal beta testers became the core of the game community, easing new players into the game's social space.

In the case of SiSSyFiGHT 2000, the testing and prototyping cycle of iterative design was successful because at each stage we clarified exactly what we wanted to test and how. We used written and online questionnaires. We debriefed after each testing session. And we strategized about how each version of the game would incorporate the visual, audio, game design, and technical elements of the previous versions, while also laying a foundation for the final form of the experience.

To design a game is to construct a set of rules. But the point of game design is not to have players experience rules—it is to have players experience play. Game design is therefore a second-order design problem in which designers craft play, but only indirectly, through the systems of rules that game designers create. Play



SiSSyFiGHT 2000 Game Interfaces

arises out of the rules as they are inhabited and enacted by players, creating emergent patterns of behavior, sensation, social exchange, and meaning. This shows the necessity of the iterative design process. The delicate interaction of rule and play is something too subtle and too complex to script out in advance, requiring the improvisational balancing that only testing and prototyping can provide.

In iterative design, there is a blending of designer and user, of creator and player. It is a process of design through the reinvention of play. Through iterative design, designers create systems and play with them. They become participants, but they do so in order to critique their creations, to bend them, break them, and refashion them into something new. And in these procedures of investigation and experimentation, a special form of discovery takes place. The process of iteration, of design through play, is a way of discovering the answers to questions you didn't even know were there. And that makes it a powerful and important method of design. *SiSSyFiGHT 2000* was developed by Marisa Bowe, Ranjit Bhatnagar, Tomas Clarke, Michelle Golden, Lucas Gonze, Lem Jay Ignacio, Jason Mohr, Daron Murphy, Yoshi Sodeka, Wade Tinney, and Eric Zimmerman.

- Staff up with a full team and plan a set of development “sprints” for each of the goals in your plan. Evaluate your game as a team after each sprint to make sure you are still on target with your player experience goals.
- Don't lose sight of the playcentric process during production—test your artwork, gameplay, characters, and so forth as you move along. As you continue to perform iterative cycles throughout the production phase, the problems you find and the changes you make should get smaller and smaller. This is because you resolved your major issues during the prototyping phases.
- Unfortunately, this is the time when most game designers actually wind up designing their games, and this can lead to numerous problems related to time, money, and frustration.

Step 7: Quality Assurance

- By the time the project is ready for quality assurance testing, you should be very sure that your gameplay is solid. There can still be some issues, so continue playtesting with an eye to usability. Now is the time to make sure your game is accessible to your entire target audience.

As you can see, the playcentric approach involves player feedback throughout the production process, which means you'll be doing lots of prototyping and playtesting at every stage of your game's development. You can't be the advocate for the player if you don't know what the player is thinking, and playtesting is the best mechanism by which you can elicit feedback and gain insight into your game. I cannot emphasize this fact enough, and I encourage any designer to rigorously build into any production schedule the means to continually isolate and playtest all aspects of their game as thoroughly as possible.

Prototypes and Playtesting in the Industry

In the game industry today, designers often skip the creation of a physical prototype altogether and jump straight from the concept stage to writing up the design. The problem with this method is that the software coding has commenced before anyone has a true sense for the game mechanics. The reason this is possible is because many games are simply variations on standard game mechanics, so the designers have a good idea of how the game

will work because they've played it, or a variation of it, as another game.

It's important to remember that the game industry is just that: an industry. Taking risks and spending a lot of time and money creating new gameplay mechanics are difficult to reconcile with a bottom line. However, the game industry is changing and growing rapidly, with new platforms that demand innovative designs. This means designing for different types of players outside the traditional gaming audience. New platforms like VR, AR, smartphones, tablets, gestural and multitouch interfaces, and breakout hits like *Pokémon Go* have proven that there is demand from new audiences if the right new kind of gameplay is offered.

While the industry as a whole is extremely skilled at maintaining steady technological innovation and cultivating core audience demand for those innovations, the same isn't true when it comes to developing original ideas in player experience. To meet the demands of new players using game devices in wildly different contexts than a traditional game audience, we are seeing the need for breakthroughs in player experience just as surely as there has always been a need for breakthroughs in technology to drive the industry forward. But it is difficult to design an original game if you skip the physical prototyping process. What happens is that

you are forced to reference existing games in the design description? This means your game is bound from the outset to be derivative. Breaking away from your references becomes even more difficult as the production takes off. When your team is in place, with programmers coding and artists cranking out graphics, the idea of going back and changing the core gameplay becomes very difficult.

That is why a number of prominent game designers have begun to adopt a playcentric approach. Large companies such as Electronic Arts have created in-house training in preproduction (see sidebar in [Chapter 6](#), page 175) originally run by Chief Visual Officer Glenn Entis. This workshop includes physical prototyping and playtesting as part of the initial development stage. Entis runs development teams through a series of exercises, one of which is coming up with a quick physical prototype. His advice is make it “fast, cheap, public, and physical. If you don't see people on the team arguing,” he says, “you can't know if they are sharing ideas. A physical prototype gets the team talking, interacting.”⁴

Chris Plummer, an executive producer at Electronic Arts Los Angeles, says, “Paper prototypes can be a great tool for low-cost ideation and playtesting of game features or systems that would otherwise cost a lot more to develop in software. It's much easier to justify spending the



1.9 Angry Birds Star Wars and Pokémon Go—unconventional markets and players



1.10 USC Games students at work at weekend game jam

resources to realize a game in software after the game framework is developed and refined through more cost-effective means, such as analog prototypes.”⁵

Smaller companies often engage in “game jams,” events where local independents and students come together for a weekend to generate prototypes for new game projects. The Global Game Jam

is an annual worldwide event that brings together tens of thousands of participants to develop innovative game prototypes. By leveraging their local community of independent game designers, small groups and companies are able to jump-start their new ideas in a collaborative environment.

DESIGNING FOR INNOVATION

As I mentioned earlier, today’s game designers have the challenge—and opportunity—to produce breakthroughs in player experience as part of their basic job description. They will have to do this without taking too many risks in terms of time and money. By innovation, I mean:

- Designing games with unique play mechanics—thinking beyond existing genres of play
- Appealing to new players—people who have different tastes and skills than hard-core gamers
- Designing for new platforms such as smartphones, tablets, and gestural and multitouch interfaces
- Creating games that integrate into daily life, real-world spaces, and the systems around us
- Embracing new business models for games such as free-to-play or subscription
- Trying to solve difficult problems in game design such as:

- ◇ The integration of story and gameplay
- ◇ Deeper empathy for characters in games
- ◇ Creating emotionally rich gameplay
- ◇ Discovering the relationships between games and learning
- Asking difficult questions about what games are, what they can be, and what their impact is on us individually and culturally

The playcentric approach can help foster innovation and give you a solid process within which to explore these provocative, unusual questions about gameplay possibilities, to try ideas that might seem fundamentally unsound but could have within them the seed of a breakthrough idea, and to craft them until they are playable. Real innovation seldom comes from the first spark of an idea; it tends to come from long-term development and experimentation. By interacting with players throughout the design process, experimental ideas have time to develop and mature.

CONCLUSION

My goal in this book is to help you become a game designer. I want to give you the skills and tools you’ll need to take your ideas and craft them into games that aren’t mere extensions of games already on the market. I want to enable you to push the envelope on game design, and the key to doing this is process. The approach you will learn here is about internalizing a playcentric method of design that will make you more

creative and productive, while helping you to avoid many of the pitfalls that plague game designers.

The following chapters in this first section will lay out a vocabulary of design and help you to think critically about the games you play and the games you want to design. Understanding how games work and why players play them is the next step to becoming a game designer.

DESIGNER PERSPECTIVE: CHRISTINA NORMAN

Lead Designer, Riot Games

Christina Norman is an experienced game designer whose credits include Mass Effect (2007), League of Legends (2009), Mass Effect 2 (2011) and Mass Effect 3 (2012).



How did you become a game designer?

I would say I became a game designer at age 9. I was playing Dungeons & Dragons with some kids at school, and our dungeon master moved away. I'd already memorized all the rules, so I was a natural to replace him. This was the starting point of a nine-year-long D&D campaign, and the moment I became a game designer.

The story of how I became employed as a game designer is, of course, entirely different. That story starts with...depression. I had a successful career programming e-commerce web sites, but I felt deeply unfulfilled. I didn't care about what I was doing, so I asked myself—what do you care about? What do you really want to do? The answer was: make games.

I had three things going for me: I was a hardcore gamer, I had created several successful Warcraft 3 mods, and I was a programmer. I applied for a game design job at BioWare and...they rejected me. I applied again as a programmer and they said, okay! After I had been there for a few years I was able to convince the lead designer to give me a shot at game design. Since then it's been all flowers, bunny rabbits, and joy!

On games that have inspired her:

Dungeons & Dragons: This, along with other great pen-and-paper role-playing games, taught me the fundamentals of system design. It was my unquenchable thirst for more Dungeons & Dragons that drove me to CRPGs (what we used to call “computer RPGs”).

Nethack (honorable mention to Diablo 2): Nethack is one of the early “roguelike” games. In this vast procedurally generated world, I endlessly pursued the fabled amulet of Yendor. As I descended through the seemingly endless dungeon levels, I marveled at the intricate and complex systems and their many interactions. Years later, Diablo 2 was the first mainstream game I played that captured much of Nethack's strengths, improving it with AAA production values and addictive multiplayer.

Baldur's Gate 2: This game taught me that games can be an exceptional storytelling medium that really makes you feel. Through my adventures I came to truly care for my party members—I wanted to help them achieve their goals! On top of all this, BG2 remains a mastery of systems design and in my opinion is the best realization of D&D in a video game to date.

Master of Orion 2 (honorable mention to Civilization): This was the first 4X (explore, expand, exploit, exterminate) game that completely captivated me. The idea of starting at a single planet, developing the technology of space flight, and ultimately ruling the entire universe was mind blowing.

Everquest: I didn't just play Everquest, I was transformed by it. I entered the virtual world of Norath a role player. I left it a hardcore raider who would eventually achieve world-first boss kills in World of Warcraft. More importantly, through Everquest I developed an appreciation for how deep, strong, and real online social relationships can be.

What is the most exciting development in the recent game industry?

This is an invigorating time to be a game designer. We're experiencing a renaissance in which small games are dominating the creative landscape. The rise of mobile gaming, self-publishing, and fresh game models has created opportunities for small developers to create innovative games that can also be financially successful. League of Legends started as a small game and benefited from these industry dynamics where scrappy challenges really have a shot!

Disruption rocks!

On her design process:

I don't build games for myself. It's easy to build games that you want to play; it's much harder to truly understand the needs of others. Building games so a diverse audience can enjoy them requires a commitment to understanding how others enjoy games.

The first thing I do when I'm designing a game, or a system, is listen to the people I'm building it for. I try to understand what kind of experience will please them. I then relentlessly pursue delivering that experience without compromise.

Do you use prototypes?

I'm a programmer, so code is my paintbrush. When I want to try an idea out, I code it fast and dirty. From there it's test, iterate, test, iterate, test...and when the design works...build it properly. When I do code-based prototyping, I use whatever tools will let me test ideas the quickest.

I'm also a big fan of building physical prototypes. Sometimes it's just faster to build something as a card game, or board game, than to code it.

On a particularly difficult design problem:

Mass Effect was essentially a hardcore RPG dressed as a shooter. Whether you hit enemies or not was determined by an invisible die roll. This meant that even if you aimed perfectly, you could miss, so guns felt weak and unreliable.

For Mass Effect 2 we wanted guns to feel accurate, powerful, and reliable. We disabled the to-hit rolls, but aiming still felt sub-par. This was my unruly introduction to combat design—I learned that making something work a certain way is different than making it feel great. My team studied the great shooters, learned from them, and then we polished our guns until they felt great.

But it wasn't that simple. Making firing guns feel great required adjusting the pacing of gameplay, which required...reinventing pretty much every system in Mass Effect. By the time we were done, we had an entirely different game than the first one, but the results were worth it—ME2 is currently the fourth highest-rated Xbox 360 game of all time on Metacritic.

What are you most proud of in your career?

Reinventing Mass Effect 2's gameplay required more than design. To achieve that goal, I had to achieve buy-in from the team (not an easy task for a designer on her first design project). In the end, I succeeded because I had a strong vision, I communicated it clearly, and I appealed to the team's collective desire to deliver a great experience to our players.

On advice to designers:

Play many games. Play them hardcore. If you get into the game industry, you'll have less time to play games, and so many insights come from your experience as a player.

Go beyond your own insights. Learn to be a better designer by listening to other players. Just watching someone play a game can teach you a great deal about game design.

Listen to your team. Just because someone's title doesn't include the word "designer" doesn't mean they don't have valuable design insights. Some of the best designers I have worked with have producer, programmer, or QA in their title.

DESIGNER PERSPECTIVE: WARREN SPECTOR

Studio Director, OtherSide Entertainment

Warren Spector is a veteran game designer and producer whose credits include Ultima VI (1990), Wing Commander (1990), Martian Dreams (1991), Underworld (1991), Ultima VII (1993), Wings of Glory (1994), System Shock (1994), Deus Ex (2000), Deus Ex: Invisible War (2003), Thief: Deadly Shadows (2004), Disney Epic Mickey (2010), and Disney Epic Mickey 2 (2012).

On getting into the game industry:

I started out, like most folks, as a gamer, back in the day. Back in 1983, I made my hobby my profession, starting out as an editor at Steve Jackson Games, a small board game company in Austin, Texas. There, I worked on TOON: The Cartoon Roleplaying Game, GURPS, several Car Wars, Ogre, and Illuminati games and learned a ton about game design from people like Steve Jackson, Allen Varney, Scott Haring, and others. In 1987, I was lured away by TSR, makers of Dungeons & Dragons and other fine RPGs and board games. 1989 saw me homesick for Austin, Texas, and feeling like paper gaming was a business/art form that had pretty much plateaued. I was playing a lot of early computer and video games at the time, and when the opportunity to work for Origin came up, I jumped at it. I started out there as an associate producer, working with Richard Garriott and Chris Roberts before moving up to full producer. I spent seven years with Origin, shipping about a dozen titles and moving up from associate producer to producer to executive producer.

On game influences:

There have probably been dozens of games that have influenced me, but here are a few of the biggies:

- **Ultima IV:** This is Richard Garriott's masterpiece. It proved to me (and a lot of other people) that giving players power to make choices enhanced the gameplay experience. And attaching consequences to those choices made the experience even *more* powerful. This was the game that showed me that games could be about more than killing things or solving goofy puzzles. It was also the first game I ever played that made me feel like I was engaged in a dialogue with the game's creator. And that's something I've striven to achieve ever since.
- **Super Mario 64:** I was stunned at how much gameplay Miyamoto and the Mario team managed to squeeze into this game. And it's all done through a control/interface scheme that's so simple that, as a developer, it shames me. Mario can do maybe ten things, I think, and yet the player never feels constrained—you feel empowered and liberated, encouraged to explore, plan, experiment, fail, and try again, without feeling frustrated. You have to be inspired by the combination of simplicity and depth.
- **Star Raiders:** This was the first game that made me believe games were more than just a fad or passing fancy, for me and for, well, humanity at large. "Oh, man," I thought, "we can send people places they'll never be able to go in real life." That's not just kid stuff—that's change-the-world stuff. There's an old saying about not judging someone until you've walked a mile in their shoes, you know? Well,

games are like an experiential shoe store for all mankind. We can allow you to walk in the shoes of anyone we can imagine. How powerful is that?

- Ico: Ico impressed me because it proved to me how powerfully we can affect players on an emotional level. And I'm not just talking about excitement or fear, the stuff we usually traffic in. Ico, through some stellar animation, graphics, sound, and story elements, explores questions of friendship, loyalty, dread, tension, and exhilaration. The power of a virtual touch—of the player holding the hand of a character he's charged to protect, even though she seems weak and moves with almost maddening slowness—the power of that touch blew me away. I have to find a way to get at some of that power in my own work. Interestingly, some recent games, like Last of Us and The Walking Dead, have exploited the human need to make contact with and protect another. Clearly, this is an idea games can exploit exceptionally well—an idea that allows us to move people, emotionally, in ways many nongamers and even some gamers thought impossible.
- Suikoden: This little PlayStation role-playing game showed me new ways of dealing with conversation. I had never before experienced Suikoden's brand of simple, straightforward, binary-choice approach—little things like “Do you fight your father or not? Y/N” or “Do you leave your best friend to almost certain death so you can escape and complete your critically important quest? Y/N” will blow you away! In addition, the game featured two other critical systems: a castle-building mechanic and a related player-controlled ally system. The castle-building bit showed me the power of allowing players to leave a personal mark on the world—the narcissistic aspect of game playing. The ally system, which affected what information you got before embarking on quests, as well as the forces/abilities available to you in mass battles, revealed some of the power of allowing each player to author his or her own unique experience. It is a terrific game that has a lot to teach even the most experienced RPG designers in the business.
- One recent game that inspired me, though perhaps not in the way I expected or the creators of the game intended, was The Walking Dead. Playing that game, I was drawn into a narrative, into an experience, that felt more emotionally compelling than maybe any other game I've played. As an experience, the game was magnificent. As a game? I'm not so sure. I think The Walking Dead worked as well as it did because it was unabashedly cinematic—the creators of the game knew exactly where every player would be at all times, what each player would do, exactly how they would do it...In a sense, that meant The Walking Dead was “just” a movie—but a movie that gives an incredibly convincing illusion of interactivity. As a player, I was charmed by it. As a developer, I was aghast that anyone would make a game where developers would never be surprised by anything players did and where no player would ever do anything the creators didn't intend, plan for, and implement. I'm still working through the contradiction inherent in the idea of a game I loved as a player but felt disappointed in as a developer. Any game that is as enjoyable and, albeit inadvertently, thought provoking is worth including on a list of influences!

On free-form gameplay:

I guess I'm pretty proud of the fact that free-form gameplay, player-authored experiences, and the like are finally becoming not just common but almost expected these days. From the “middle” Ultimas (4–6), to Underworld, to System Shock, to Thief, to Deus Ex, there's been this small cadre of us arguing, through our work, in favor of less linear, designer-centric games, and, thanks to the efforts of folks at Origin, Looking Glass Studios, Ion Storm, Rockstar/DMA, Bioware, Lionhead, Bethesda, and others, people are finally beginning to take notice. And it isn't just the hardcore gamers—the mass market is waking up, too. That's pretty cool.

I'm hugely proud of having had the privilege of working alongside some amazingly talented people. It's standard practice in all media to give one person credit for the creation of a product, but that's nonsense. Nowhere is it more nonsensical than in games. Game development is the most intensely collaborative endeavor I can imagine. It's been an honor to work with Richard Garriott, Paul Neurath, Doug Church, Harvey Smith, Paul Weaver, and many others (who will now be offended that I didn't single them out here!). I know I've learned a lot from all of them and hope I've taught a little bit in return.

Advice to designers:

Learn to program. You don't have to be an ace, but you should know the basics. In addition to a solid technical foundation, get as broad-based an education as you can. As a designer, you never know what you're going to need to know—behavioral psychology will help you immensely, as will architecture, economics, and history. Get some art/graphics experience, if you can, so you can speak intelligently with artists even if you lack the skills to become one yourself. Do whatever it takes to become an effective communicator in written and verbal modes. And most importantly, make games. Get hold of one of the many free game engines out there and build things. Get yourself on a mods team and build some maps, some missions, anything you can. Heck, make something amazing in Minecraft! You can do all of this on your own or at one of the many institutions of higher learning now (finally!) offering courses, even degrees, in game development and game studies. It doesn't really matter how you get your training and gain some experience—of life as much as game development—just make sure you get it. Oh, and make sure you really, really, really want to make games for a living. It's gruelingly hard work, with long hours and wrecked relationships to prove it. There are a lot of people who want the same job you do. Don't go into it unless you're absolutely certain it's the career for you. There's no room here for dilettantes!

FURTHER READING

Kelley, Tom. *The Art of Innovation: Lessons in Creativity from IDEO, America's Leading Design Firm*. New York: Random House, 2001.

Laramée, François Dominic, ed. *Game Design Perspectives*. Hingham: Charles River Media, 2002.

Moggridge, Bill. *Designing Interactions*. Cambridge: The MIT Press, 2007.

The Imagineers. *The Imagineering Way*. New York: Disney Editions, 2003.

Tinsman, Brian. *The Game Inventor's Guidebook*. Iola: KP Books, 2003.

END NOTES

1. Phipps, Keith, "Will Wright Interview by Keith Phipps" A.V. Club. February 2, 2005. <https://www.avclub.com/will-wright-1798208435>
2. Sheff, David. *Game Over: How Nintendo Conquered the World*. New York: Vintage Books, 1994, p. 51.
3. Hermida, Alfred. "Katamari Creator Dreams of Playground." BBC News.com November 2005. <http://news.bbc.co.uk/2/hi/technology/4392964.stm>
4. Entis, Glenn. "Pre-Production Workshop." EA@USC Lecture Series. March 23, 2005.
5. Plummer, Chris. E-mail interview, May 2007.



Taylor & Francis

Taylor & Francis Group

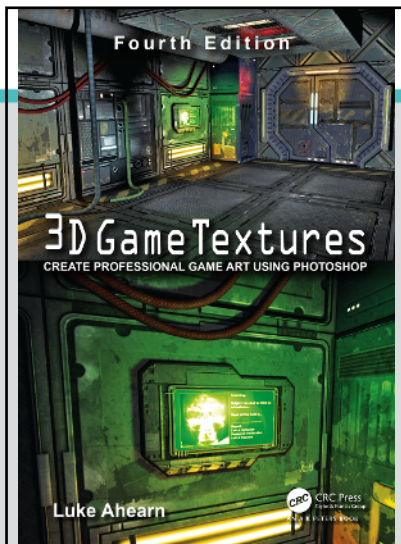
<http://taylorandfrancis.com>



CHAPTER

2

THE BASICS OF ART

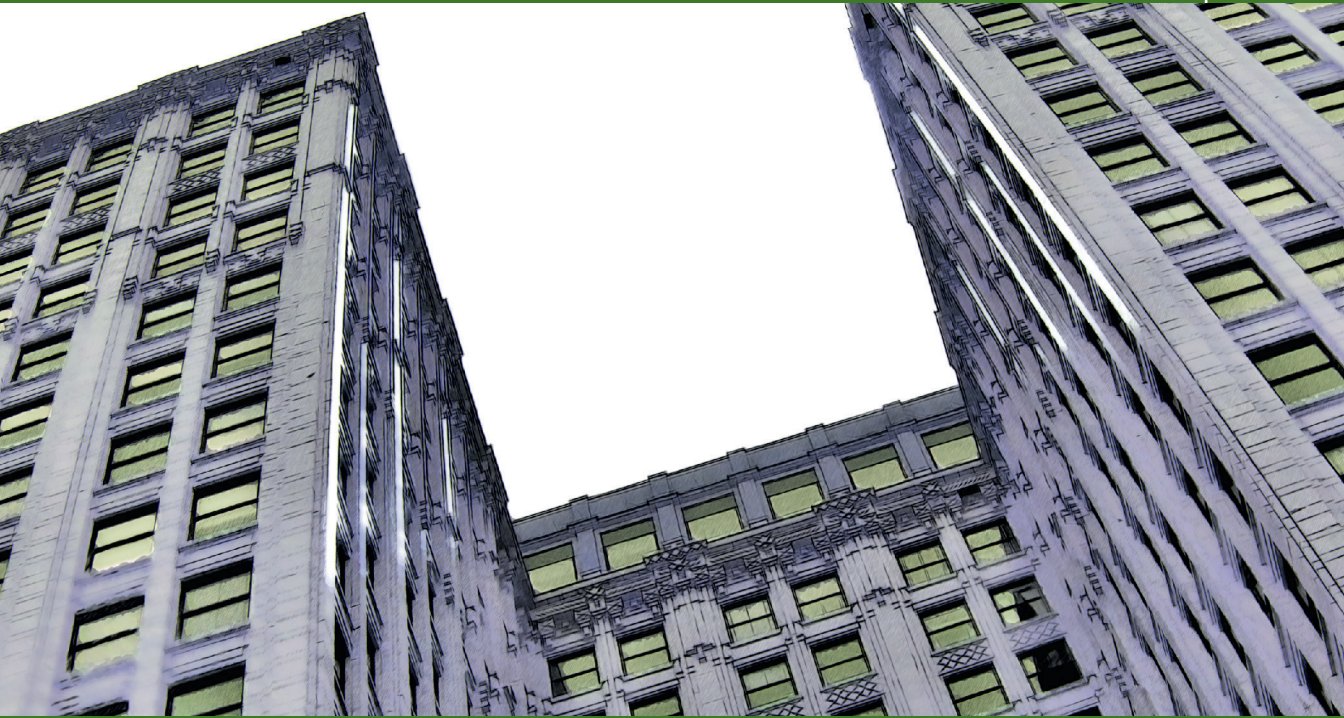


This chapter is excerpted from
3D Game Textures
by Luke Ahearn

© 2016 Taylor & Francis Group. All rights reserved.



[Learn more](#)



The Basics of Art

Art is born of the observation and investigation of nature.

—Cicero
Roman author, orator, and politician, 106 BC–43 BC

Introduction

Creating art for computer games requires both artistic and technical skills. We will look at both but first we will look at the very basics of art. The goal of this chapter isn't to turn you into an artist, but it will help you create better textures to understand these fundamentals. If you have any art training, this may be material you will want to skim over. If you don't, hopefully it will help point you in the right direction to learn what you need to in order to become a great game artist.

There are a few basic aspects of visual art that are simple to understand but can take years to master. And even though teaching you traditional fine art skills is beyond the scope of this book, it is critical that you have an understanding of these basic aspects of visual art so that you can create game textures. Fortunately, these basic aspects of art are easy to present in

book form and you can learn the vocabulary of an artist without difficulty. By studying these basics of art, you will learn to see the world as an artist does and to understand what you see and then be better able to create a texture set for a game world.

The basic aspects of visual art that we will focus on are as follows:

- Shape and form
- Light and shadow
- Texture
- Color
- Perspective

Learning to observe the world around you, understand what you are seeing, and then explain it verbally is what an artist does. Communicating with other artists and team members is critical in the development process. Just being able to make a decent texture isn't enough; you need to be able to create the texture that is needed and that need will be communicated to you through the nomenclature of the artist.

Technology is, of course, critical to the larger picture of game textures, but the actual basics of art are where great textures begin. It is way too common for a book or class to start with a tutorial on tiling in Photoshop or even game engine technology and that is skipping what the game artist really needs, a fundamental understanding of the visual world. It is very common for people to know this stuff, but it is equally as common for a person to have absolutely no understanding of any of it.

The skills that are the most important for the creation of game textures are the ability to understand what you are seeing in the real world and the ability to recreate it in the computer. Often a texture artist is required to break a scene down to its core materials and build a texture set based on those materials, so learning this skill is essential. Although you don't need to have an advanced degree in art to create great textures, let's face it: Almost anyone can learn what buttons to click in Photoshop, but the person who understands and skillfully applies the basics of art can make a texture that stands out above the rest.

There are many types of art and aspects of visual art that you should further explore to develop as a game artist. The following are some of the things you can study and/or practice:

- Figure drawing
- Still life
- Calligraphy
- Photography
- Painting (oil, water color, etc.)
- Lighting (for film, still photography, the stage, or computer graphics)
- Color theory and application
- Sculpture

- Drafting and architectural rendering
- Anatomy, which usually starts with stick figures and adds the skeleton, muscles, and then skin
- Set design
- Technical illustration

It is even worth the time to study other areas of interest beyond art, such as the sciences, particularly the behavior of the physical world. Almost all commercial game engines process light in real time and don't rely on painting it into the texture. There are still many situations where that skill is needed so we will look at working with textures for both methods. The more you understand and are able to reproduce effects such as reflection, refraction, blowing smoke, and so on, the more success you will find as a game artist. We now have technologies that reproduce the real world to a much greater extent than ever before, but it still takes an artist to create the input and adjust the output for these effects to look their best.

The areas of study that will help you deal with real-world behaviors are endless. You can start by simply observing the world. Watch how water drips or flows, the variations of light and shadow on different surfaces at different times of the day, how a tree grows from the ground—straight like a young pine or flared at the base like an old oak. Study a crack up close and you will begin to see many interesting details. Look at any object, photograph it if possible, and study the surface. An old dumpster was new once. Try to determine what made it look the way it does today. What may have dented the various parts, a car hitting it or the mechanism that lifts it to empty it? Where is it rusted and why? Was it repainted, vandalized, damaged, or repaired? Is there dirt splattered and clumped on, running down the sides? Where is it, behind a restaurant, a chemical factory, or a construction site?

An excellent book to inspire this type of activity is *Digital Texturing and Painting*, by Owen Demers (New Riders, 2001).

You can also take tours of museums, go on architectural tours or nature walks, join a photography club or a figure-drawing class—there is no end to the classes, clubs, disciplines, and other situations that will open up your mind to new inspirations and teach you new tools and techniques for texture creation. And, of course, playing games, watching movies, and reading graphic novels are basic food to the game artist.

There are many elements of traditional art, but we will narrow our focus to those elements that are most pertinent to texture creation. Let's start with shape and form.

Shape (2D) and Form (3D)

A *shape* (height and width) is simply a two-dimensional (flat) outline of a form. Circles, squares, rectangles, and triangles are all examples of shape. Shape is what we first use to draw a picture before we understand such

concepts as light, shadow, and depth. As children we draw what we see in a crude 2D way. Look at the drawings of very young children and you will see that they are almost always composed of pure basic shapes: triangular roof, square door, circular sun. Even as adults, when we understand shadows and perspective, we have trouble drawing what we see before us and instead rely on a whole series of mental notes and assumptions as to what we think we are seeing. There are exercises to help develop the ability to draw what we actually see. Most notably, the book *The New Drawing on the Right Side of the Brain*, by Betty Edwards (Tarcher, 1999), offers many such exercises. One of the most famous of these involves the drawing of a human face from a photo. After you have performed this exercise, you turn the photo upside down and draw it again. The upside-down results are often far better than the right-side-up ones, even on first try. This is because once you turn the image upside down, your brain is no longer able to make any mental assumptions about what you think you are seeing; you can see only what's really there. Your brain hasn't yet developed a set of rules and assumptions about the uncommon sight of an upside-down human face.

One of the first skills that you can practice as an artist is trying to see the shapes that make up the objects that surround you. Figure 1.1 has some examples of this shape training, ranging from the simple to the complex. This is a very important skill to acquire. As a texture artist, you will often need to see an object's fundamental shape amid all the clutter and confusion in a scene so that you can create the 2D art that goes over the 3D objects of the world.

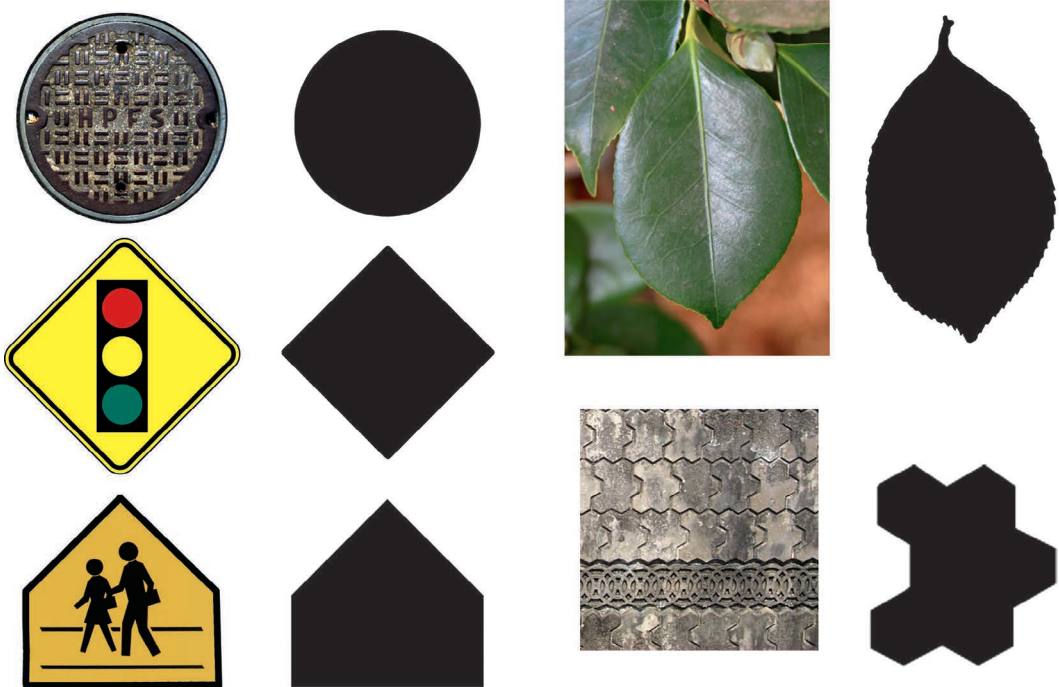


FIG 1.1 Here are some examples of shapes that compose everyday objects. These shapes range from simple to complex.

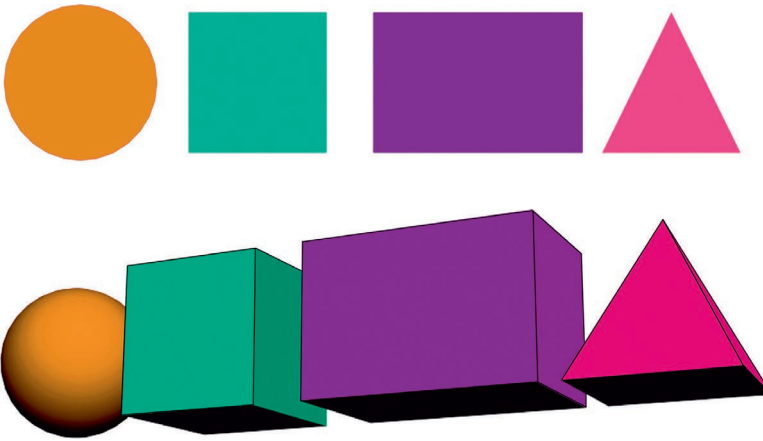


FIG 1.2 Here are examples of shapes and forms. Notice how it is shadow that turns a circle into a sphere.

Form is three-dimensional (height, width, and depth) and includes simple objects like spheres, cubes, and pyramids. See Figure 1.2 for examples and visual comparisons. You will see later that as a texture artist, you are creating art on flat shapes (essentially squares and rectangles) that are later placed on the surfaces of forms. An example is shown in Figure 1.3, where a cube is turned into a crate (a common prop in many computer games). When a shape is cut into a base material in Photoshop and some highlights and shadows are added, the illusion of form is created. A texture can be created rather quickly using this method. See Figure 1.4 for a very simple example of a space door created using an image of rust, some basic shapes, and some standard Photoshop Layer Effects.

Of course, mapping those textures to more complex shapes like weapons, vehicles, and characters gets more complex, and the textures themselves

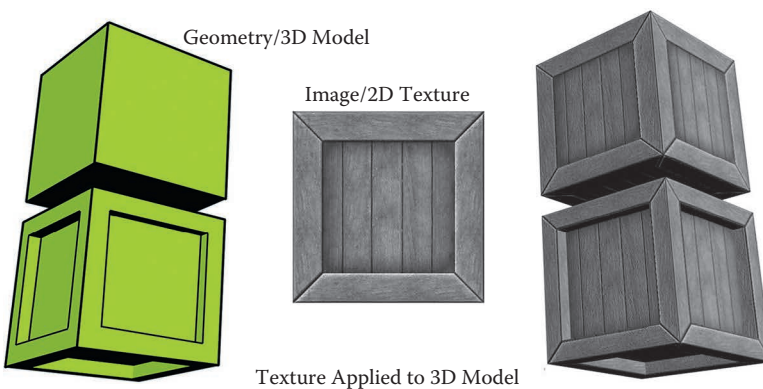


FIG 1.3 A game texture is basically a 2D image applied, or *mapped*, to a 3D shape to add visual detail. In this example, a cube is turned into a crate using texture. A more complex 3D shape makes a more interesting crate using the same 2D image.

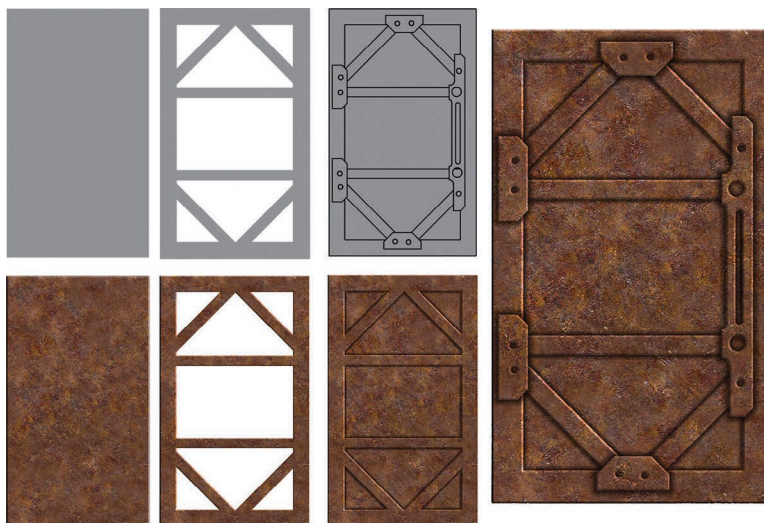


FIG 1.4 Here is an example of how shapes can be cut into an image and with some simple layer effects can be turned into a texture in Photoshop.

reflect this complexity. Paradoxically, as the speed, quality, and complexity of game technology increase, artists are actually producing more simplified textures in many cases. The complexity comes in the understanding and implementation of the technology. This book will gradually introduce this complexity so you won't be overwhelmed by it.

As with shapes, you can practice looking for the forms that make up the objects around you. In Figure 1.5 you can see some examples of this concept.



FIG 1.5 Here are some examples of the forms that make up the objects around you.

Light and Shadow

Of all the topics in traditional art, light and shadow are arguably the most important due to their difficulty to master and importance to the final work. Light and shadow give depth to and—as a result—define what we see. At their simplest, light and shadow are easy to see and understand. Most of us are familiar with shadow—our own shadow cast by the sun, making animal silhouettes with our hands on a wall, or a single light source shining on a sphere and the round shadow that it casts. That’s where this book begins. Light and shadow quickly get more complicated, so the examples in this book will get more complex as well. The book starts by discussing the ability to see and analyze light and shadow in this chapter; moves up to creating and tweaking light and shadow in Photoshop using Layer Styles, for the most part; and finally looks at some basic hand tweaking of light and shadow. If you want to master the ability to hand-paint light and shadow on complex and organic surfaces, you are advised to take traditional art classes in illustration, sketching, and painting.

We all know that the absence of light is darkness, and in total darkness we can see nothing at all, but the presence of too much light will also make it difficult to see. Too much light blows away shadow, removes depth, and desaturates color. In the previous section, we looked at how shape and form differ. We see that difference primarily as light and shadow, as in the example of the circle and a sphere. But even if the sphere were lit evenly with no shadows and looked just like the circle, the difference would become apparent when the object was rotated. The sphere would always look round if rotated, whereas once you began to rotate the circle it would begin to look like an oval until it eventually disappeared when completely sideways. In Figure 1.4, in which a shape was cut into an image of rusted metal and made to look like a metal space door using Photoshop Layer Effects, the highlights and shadows were faked using the various tools and their settings. In Figure 1.6 you can see the



FIG 1.6 Here is the same door texture from the previous section. Notice the complete lack of depth as we look at it from angles other than straight on. The illusion of depth is shattered.

same door texture rotated from front to side. Notice the complete lack of depth in the image on the far right. The illusion is shattered.

Understanding light and shadow is very important in the process of creating quality textures. We will go into more depth on this topic as we work through this book. One of the main reasons for dwelling on the topic is the importance of light and shadow visually, but in addition, you will see that many necessary decisions are based on whether light and shadow should be represented using texture, geometry, or technology. To make this decision intelligently in serious game production involves the input and expertise of many people. Although what looks best is ideally the first priority, what runs best on the target computer is usually what the decision boils down to. So keep in mind that in game development you don't want to make any assumptions about light and shadow—instead, ask questions.

This book covers different scenarios of how light and shadow may be handled in a game. It can be challenging to make shadows look good in any one of the situations. Too little shadow and you lack depth; too much and the texture starts to look flat. Making shadows too long or intense is an easy mistake. Unless the game level specifically calls for that on some rare occasion, don't do it. Technology sometimes handles the highlights and shadows. This feature is challenging because it is a new way of thinking that baffles many people who are unfamiliar with computer graphics. It can also be a bit overwhelming because you go from creating one texture for a surface to creating three or more textures that all work together on one surface. Naming and storing those textures can get confusing if you let the process get away from you.

Overall, you want your textures to be as versatile as possible, and to a great extent, that includes the ability to use those textures under various lighting conditions. See Figure 1.7 for an example of a texture in which the shadows and highlights have been improperly implemented and another one that has been correctly created. For this reason we will purposely use highlights and shadows to a minimal amount. You will find that if your texture needs more depth than a modest amount of highlight and/or shadow give you, you most likely need to create geometry or use a shader—or consider removing the source of the shadow! If there is no need for a large electrical box on a wall, for example, don't paint it in if it draws attention to itself and looks flat. If there is a need for an object and you are creating deep and harsh shadows because of it, you might need to create the geometry for the protruding element.

You will find that as game development technology accelerates, things like pipes, doorknobs, and ledges will be easily created with the larger polygon budgets or by using the advanced shaders at our disposal. Many texture surface properties are no longer painted on. Reflections, specular highlights, bump mapping, and other aspects of highlight and shadow are now processed in real time.



FIG 1.7 The crate on the left has conflicting light sources. The shadow from the edge of the crate is coming up from the bottom, is too dark, is too long, and even has a gap in it. The highlights on the edges are in conflict with the shadow cast on the inner panel of the crate, and they are too *hot*, or bright. The crate on the right has a more subtle, low-contrast, and diffuse highlight-and-shadow scheme and will work better in more diverse situations.

The rest of this book takes various approaches to light and shadow using Photoshop's Layer Effects to automate this process and other tools to hand-paint highlights and shadows. One of the main benefits of creating your own highlights and shadows in your textures is that you can control them and make them more interesting, as well as consistent. Nothing is worse than a texture with shadows from conflicting light sources: harsh, short shadows on some elements of the texture and longer, more diffuse shadows on others. See Figure 1.8 for an example. The human eye can detect these types of errors even if the human seeing the image can't quite understand why it looks wrong. One of the artist's greatest abilities is not only being able to create art but also being able to consciously know and verbalize what he is seeing. In Figure 1.9, you can see the various types of shadows created as



FIG 1.8 Here is a really bad texture created from two sources. Notice the difference in the shadows and highlights. The human eye can detect these errors, even if the human seeing it can't understand why the image looks wrong.

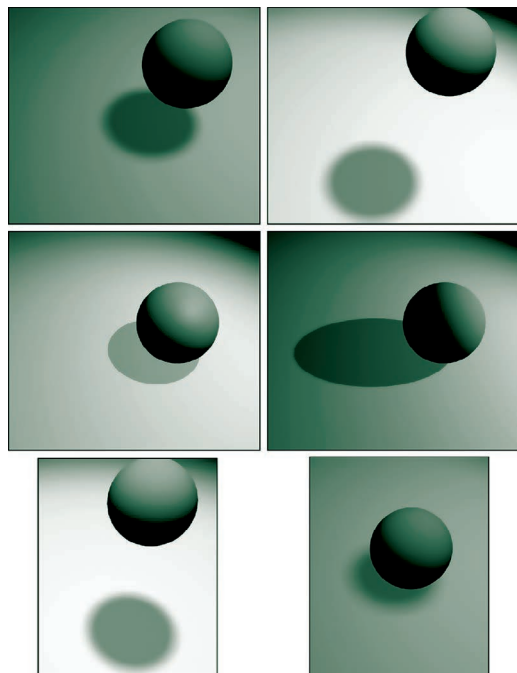


FIG 1.9 With one light source and a simple object, you can see the range of shadows we can create. Each shadow tells us information about the object and the light source, such as location, intensity, and so on.

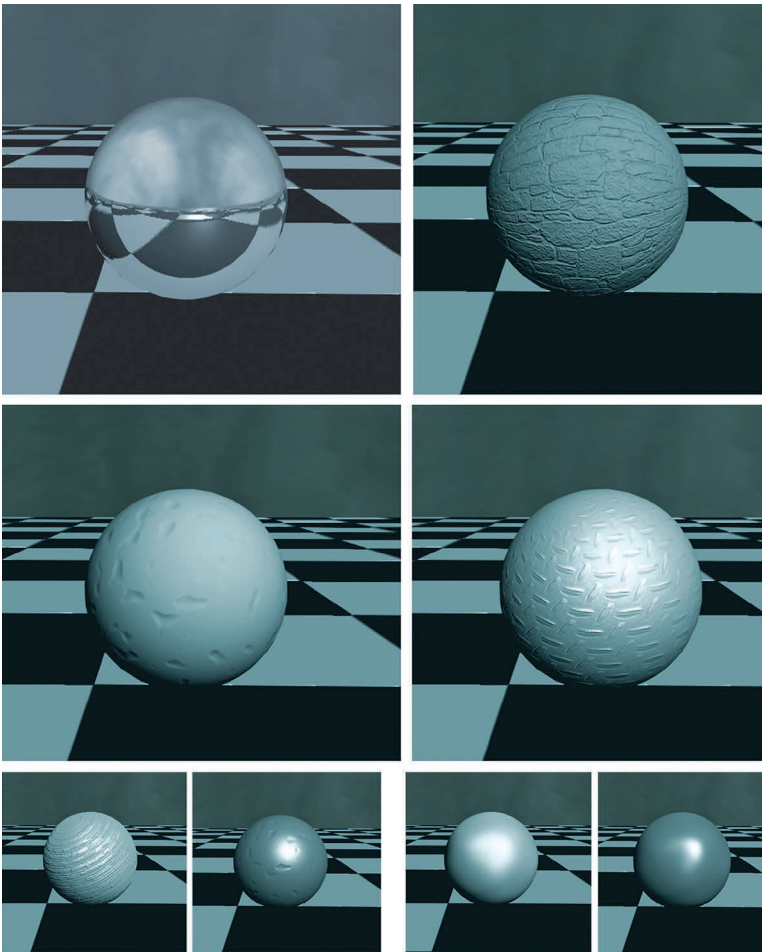


FIG 1.10 With one light source and a simple object with various highlights on it, you can see that the object appears to be created of various materials. Keep in mind that what you are seeing is only highlight and shadow. How much does only this aspect of an image tell you about the material?

the light source changes. This is a simple demonstration. If you ever have the opportunity to light a 3D scene or movie set, you will discover that the range of variables for light and shadow can be quite large.

Highlights also tell us a good bit about the light source as well as the object itself. In Figure 1.10, you can see another simple illustration of how different materials will have different highlight patterns and intensities. These materials lack any texture or color and simply show the highlights and shadows created on the surface by one consistent light source.

For a more advanced and in-depth discussion on the subject of light and shadow for 3D scenes, I recommend *Essential CG Lighting Techniques with 3ds Max*, by Darren Brooker (Focal, 2006).

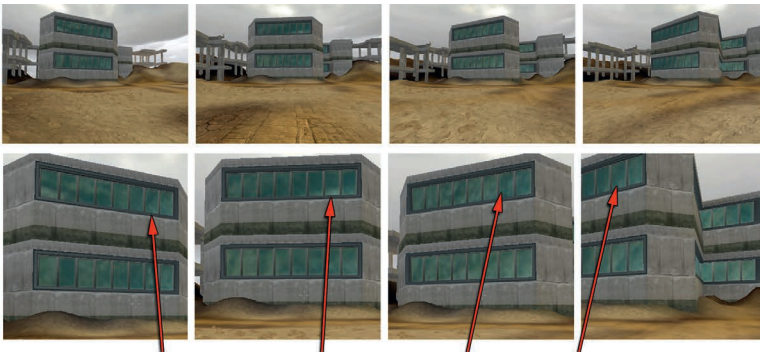
Texture

In the bulk of this book, as in the game industry, we use the term *texture* to refer to a 2D static image. What we refer to as textures in this book are also sometimes called *materials* or even *tile sets* (from older games), but we will stick to the term *texture*. The one exception is that in this section we talk about the word *texture* as it is used in traditional art: painting, sculpture, and so on. A side note on vocabulary: Keep in mind that vocabulary is very important and can be a confusing aspect of working in the game industry. There is much room for miscommunication. Different words can often mean the same thing, and the same words can often mean many different things. Acronyms can be especially confusing: RAM, POV, MMO, and RPG all mean different things in different industries. POV means *point of view* in the game industry but *personally owned vehicle* in the government sector, and it also stands for *persistence of vision*. So to clarify, the term *texture*—usually meaning a 2D image applied to a polygon (the face of a 3D object)—in this section of this chapter refers to an aspect of an image and not the image itself. We draw this distinction for the following conversation on traditional art.

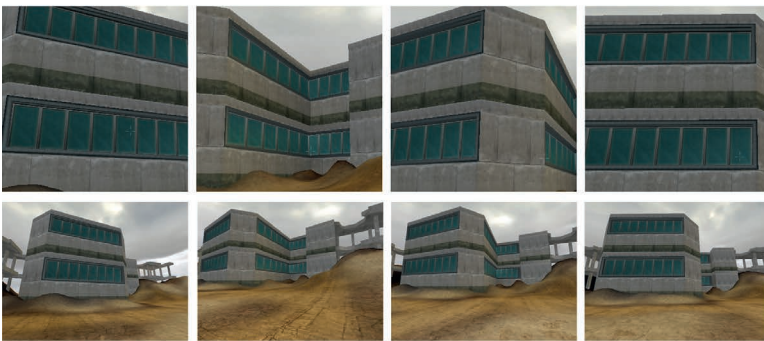
In traditional art, there are two types of texture: tactile and visual. *Tactile texture* is a term used when you are able to actually touch the physical texture of the art or object. Smooth and cold (marble, polished metal, glass) is as much a texture as coarse and rough. In art this term applies to sculptures and the like, but many paintings have thick and very pronounced brush or palette knife strokes. Vincent van Gogh was famous for this technique. Some painters even add materials such as sand to their paint to bring more physical or tactile texture to their work.

Visual texture is the illusion of what a surface's texture might feel like if we could touch it. Visual texture is composed of fine highlights and shadows. As computer game texture artists, we deal solely with this aspect of texture. So, for example, an image on your monitor might look like rough stone, smooth metal, or even a beautiful woman ... and if you try to kiss that beautiful woman, she is still just a monitor. Not that I ever tried that, mind you.

There are many ways to convey texture in a two-dimensional piece of art. In computer games we are combining 2D and 3D elements and must often decide which to use. With 2D we are almost always forced to use strictly 2D imagery for fine visual texture. And though the faster processors, larger quantities of RAM, and the latest crop of 3D graphic cards allow us to use larger and more detailed textures and more geometry, a great deal of visual texture is still static—noticeably so, to a trained artist. This limitation is starting to melt away as complex shader systems are coming into the mainstream of real-time games. The real-time processing of bump maps, specular highlights, and a long list of other, more complex effects add to our game worlds a depth of realism not even dreamed of in the recent past. This book will teach you both the current method of building texture sets and the increasingly common method of building material sets that use textures and



Windows have a reflection of the sky and that reflection moves as the player does



Windows with painted textures stay the same no matter where the player goes

FIG 1.11 Visual texture is composed of fine highlights and shadows. A shader allows for the real-time processing of visual texture, among other effects, and adds much more realism to a scene as the surface reacts with the world around it. In this example I used a specular map. These effects are best seen in 3D, but you can see here that the windows in the building on the top row have a reflection of the sky in them and that reflection moves as the player does. The windows in the building on the lower row are painted textures and stay the same no matter where the player walks. The bottom two rows are close-ups to help you see the effect. If you pick one window in the close-up images and look closely, you will see that the cloud reflections are in different places in each frame.

shader effects together. I will discuss this more at length later in the book, but for now you can see some visual examples of these effects. In Figure 1.11 you can see how in the 2D strip the object rotates but the effects stay static on the surface, whereas on the 3D strip the object rotates and the effect moves realistically across the surface.

The game artist's job is often to consider what tools and techniques we have at our disposal and determine which one will best accomplish the job. We must often make a trade-off between what looks good and what runs well. As you begin to paint textures, you will find that some of the techniques of traditional art don't work in the context of game texturing. As traditional artists, we usually do a painting that represents one static viewpoint, and we can paint into it strong light sources and a great deal of depth, but that

amount of depth representation goes beyond tactile texture and becomes faked geometry and looks flat in a dynamic, real-time 3D world. As mentioned earlier in this chapter, this approach will not work in a 3D game in which a player can move about and examine the texture. Once again, we must choose what to represent using a static 2D image, what can be processed in real time using a shader, and what must be represented using actual geometry. There are many solutions for this problem; among them are restricting the players' ability to move around the texture, removing the element of overt depth representation, or adding actual geometry for the parts of the texture represented by the overt depth representation (see Figure 1.12).

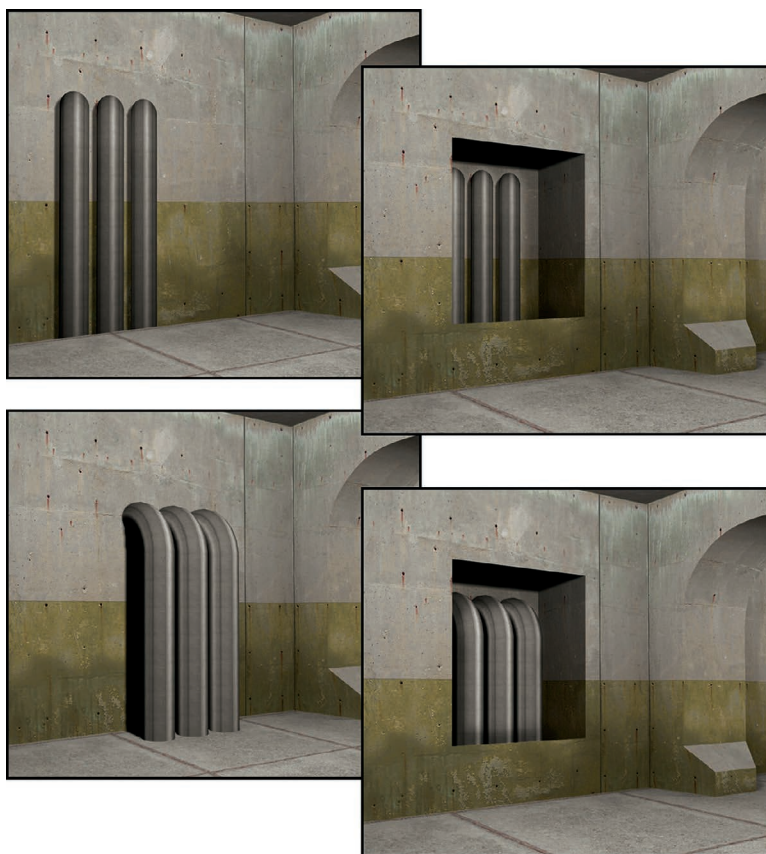


FIG 1.12 There are several possibilities in dealing with overt depth representation. Upper left: The pipes are painted into the texture and totally lack any depth; notice how they dead-end into the floor. Upper right: Restricting the players' ability to move around the texture can alleviate some of the problem. Lower left: Adding actual geometry, if possible, for the parts of the texture that cause the overt depth is the best solution (this method uses less texture memory but more polygons). Finally, lower right: Adding the actual geometry into the recess is an option that looks pretty interesting and actually allows for a reduction of geometry. The removal of polygons from the backsides of the pipes more than offsets the added faces of the recess.

Color

We all know what color is in an everyday fashion: “Get me those pliers. No, the ones with black handles.” “I said to paint the house green. I didn’t mean neon green!” That’s all fine for civilian discussions of color, but when you begin to speak with artists about color, you need to learn to speak of color intelligently, and that takes a little more education and some practice. You will also learn to choose and combine colors, too. In games, as in movies, interior design, and other visual disciplines, color is very important. Color tells us much about the world and situation we are in.

At one game company we developed a massive multiplayer game that started in a small town—saturated green grass, blue water, butterflies—you get the picture; this was a nice, safe place. As you moved away from town, the colors darkened and lost saturation. The grass went from a bright green to a less-saturated brownish-green. There were other visual clues to the change as well. Most people can look at grass and tell whether it is healthy, dying, kept up, or growing wild. Away from town, the grass was long and clumpy, dying, and growing over the path. But even before we changed any other aspect of the game—still using the same grass texture from town that was well trimmed—we simply lowered the saturation of the colors on the fly and you could feel the life drain from the world as you walked away from town. As you create textures, you will probably have some form of direction on color choice—or maybe not. You might need to know what colors to choose to convey what is presented in the design document and what colors will work well together.

This section lays out a simple introduction to the vocabulary of color, color mixing (on the computer), and color choices and their commonly accepted meanings. I decided to skip the complex science of color and stick to the practical and immediately useful aspects of color. Color can get very complex and esoteric, but you would benefit from taking your education further and learning how color works on a scientific basis. Although this chapter will be a strong starting point, you will eventually move on from working with only the colors contained in the texture that you are creating to determining how those colors interact with other elements in the world, such as lighting. To start with, however, a game texture artist needs the ability to communicate, create, and choose colors.

First, let’s discuss the way in which we discuss color. There are many color *models*, or ways of looking at and communicating color verbally. There are models that concern printing, physics, pigment, and light. They each have their own vocabulary, concepts, and tools for breaking out color. As digital artists, we use the models that describe light, because we are working with colored pixels that emit light. A little later we will take a closer look at those color systems from the standpoint of color mixing, but for now we will look at the vocabulary of color. In game development you will almost always use the red-green-blue (RGB) color model to mix color and the hue, saturation, and

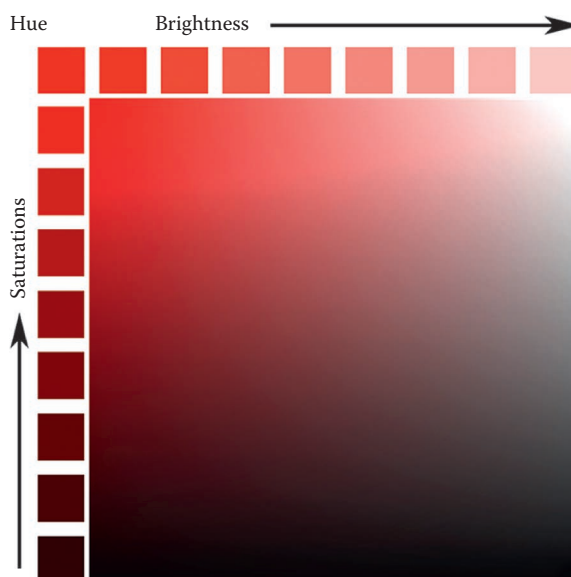


FIG 1.13 In this image you can see a representation of HSB.

brightness (HSB) color model (both explained next) to discuss color. You will see that Photoshop allows for the numeric input and visual selection of color in various ways. When you discuss color choices and changes and then go to enact them, you are often translating between two or more models. Don't worry; this is not difficult, and most people don't even realize that they are doing it.

First, we will look at the HSB model, since this is the most common way for digital artists to communicate concerning color. In Figure 1.13 you can see examples of these aspects of color. These three properties of color are the main aspects of color that we need to be concerned with when discussing color:

- *Hue* is the name of the color (red, yellow, green).
- *Saturation* (or *chroma*) is the strength or purity of the color.
- *Brightness* is the lightness or darkness of the color.

Hue

Most people use the word *color* when they really mean *hue*. Although there are many, many colors, there are far fewer hues. Variations of saturation and brightness create the nearly unlimited colors we see in the world. For example, scarlet, maroon, pink, and crimson are all *colors*, but the base *hue* is red.

Learning about color and its various properties is best done through visual examples. The most often used method is the *color wheel*, developed by Johannes Itten. We will look at the color wheel a little later. In Photoshop

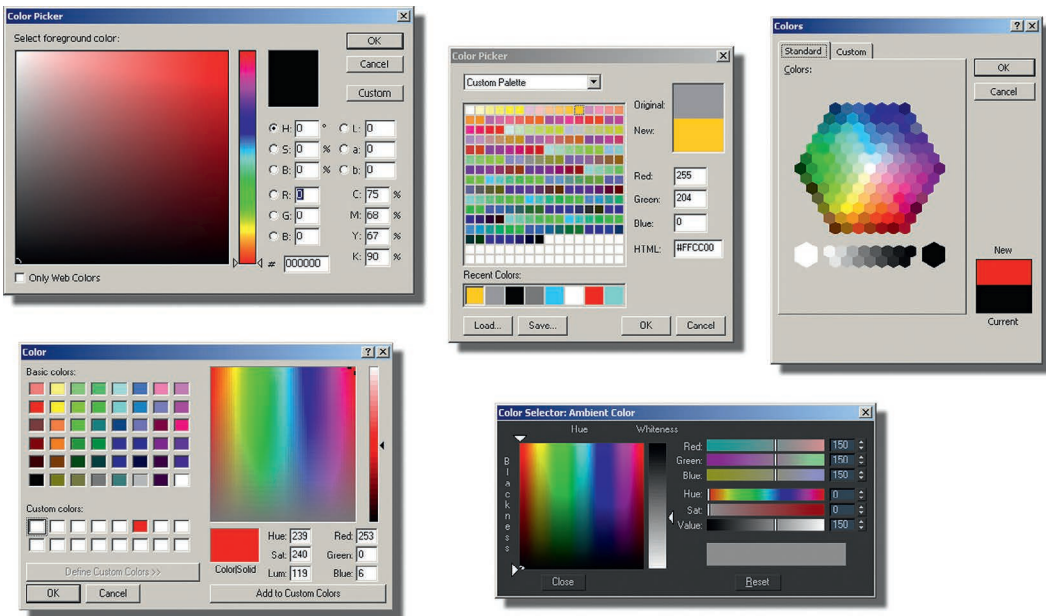


FIG 1.14 Here are Color Pickers from various applications.

you will recognize the Color Picker, which offers various methods for choosing and controlling color, both numerically and visually. The Color Picker has various ways to choose color, but the most commonly used is RGB (Figure 1.14).

Saturation

Saturation quite simply is the amount of white in a color. In Figure 1.15 you can see the saturation of a color being decreased as white is added. If you have access to a software package like Photoshop and open the Color Picker, you can slide the picker from the pure hue to a less saturated hue and watch the saturation numbers in the HSB slots go down as the color gets less saturated. Notice how the brightness doesn't change unless you start dragging down and adding black to the color. Also, you might want to look down at the RGB numbers and notice how the red in RGB doesn't change, but the green and blue do.



FIG 1.15 The saturation of the color red at 100%, decreasing to 0% by adding white.



FIG 1.16 The brightness of the color red at 100%, decreasing to 0% by adding black.

Brightness

Brightness is the amount of black in a color. In Figure 1.16 you can see the brightness of a color being decreased. As in the previous discussion of saturation, you can open the Color Picker in Photoshop and this time, instead of decreasing the saturation, you can decrease the brightness by dragging down. You can look at the HSB and the RGB slots and see the brightness numbers decreasing. Also notice that this time in the RGB slots the red numbers decrease, but the blue and green are already at zero and stay there.

Like most other aspects of color, brightness is affected by other factors. What colors are next to each other? What are the properties of the lights in the world? Another job of the texture artist is making sure that the textures in the world are consistent. That involves balancing the hue, saturation, and brightness of the color, in most cases. Figure 1.17 depicts an example of a texture that might have looked okay in Photoshop but that needed to

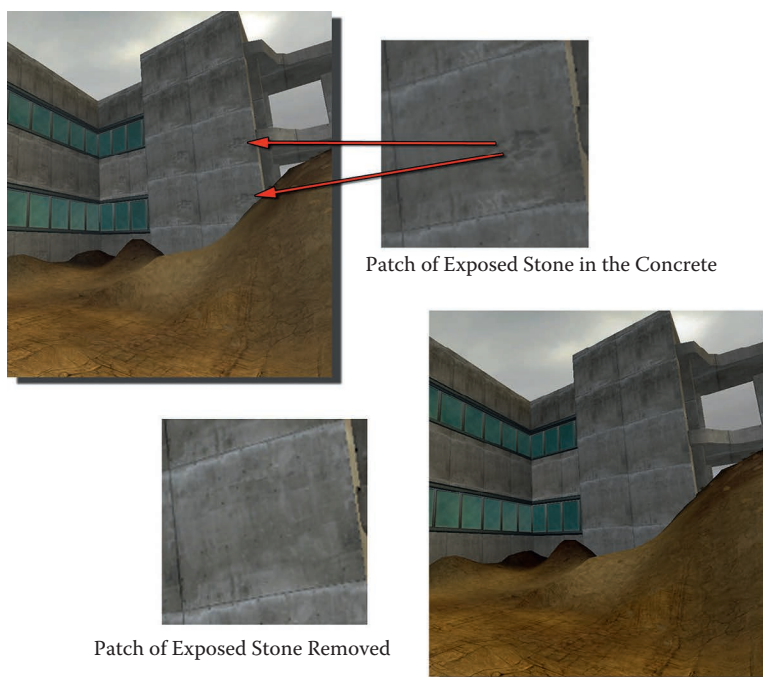


FIG 1.17 Here is an example of a texture that might have looked okay in Photoshop but that needed to be corrected to fit in the scene correctly. This is a subtle example. Notice the patch of exposed stone in the concrete on the building that repeats?

be corrected to fit the scene. You can see that a great deal of contrast and intensity of color makes tiling the image a greater challenge.

Color Systems: Additive and Subtractive

There are two types of color systems: additive and subtractive. *Subtractive color* is the physical mixing of paints, or *pigments*, to create a color. It is called *subtractive* because light waves are absorbed (or subtracted from the spectrum) by the paint and only the reflected waves are seen. A red pigment, therefore, is reflecting only red light and absorbing all the others. In the subtractive system, you get black by mixing all the colors together—*theoretically*. It is a challenge to mix pigments that result in a true black or a vibrant color. That is one of the reasons art supply stores have so many choices when it comes to paint. One of our advantages of working in the additive system is that we can get consistent and vibrant results with light. We won't dwell on the subtractive system since we won't be using it.

In the *additive system*, light is added together (as it is on a computer screen) to create a color, so naturally we deal with the additive system as computer artists because we are working with light. In Figure 1.18 you can see how the additive system works. I simply went into 3ds Max and created three spotlights that were pure red, green, and blue and created my own *additive color wheel*, or a visual representation of how the colors interact. Black is the absence of light (the area outside the spotlights), and white is all light (the center area where all three lights overlap each other): The combination of red, green, and blue is the additive system. If you look at the Color Picker in Photoshop (Figure 1.19), you will see a vertical rectangle of color graduating from red through the colors and back to red. This allows you to select a hue and use the Color Picker Palette to change the value and intensity.

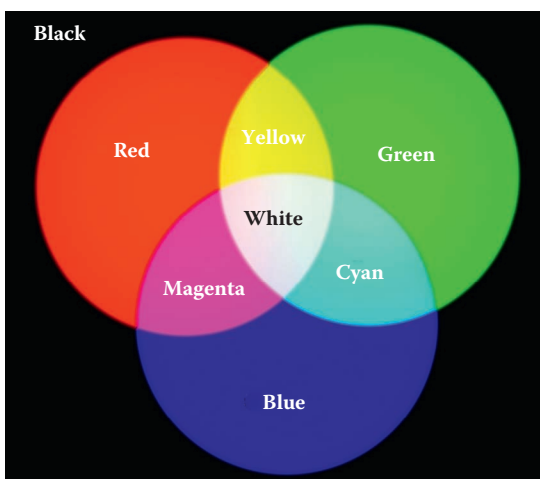


FIG 1.18 The additive system works by adding lights. Black is the absence of light (the area outside the spotlights), and white is all light (the center area where all three lights overlap each other): The combination of red, green, and blue is the additive system.

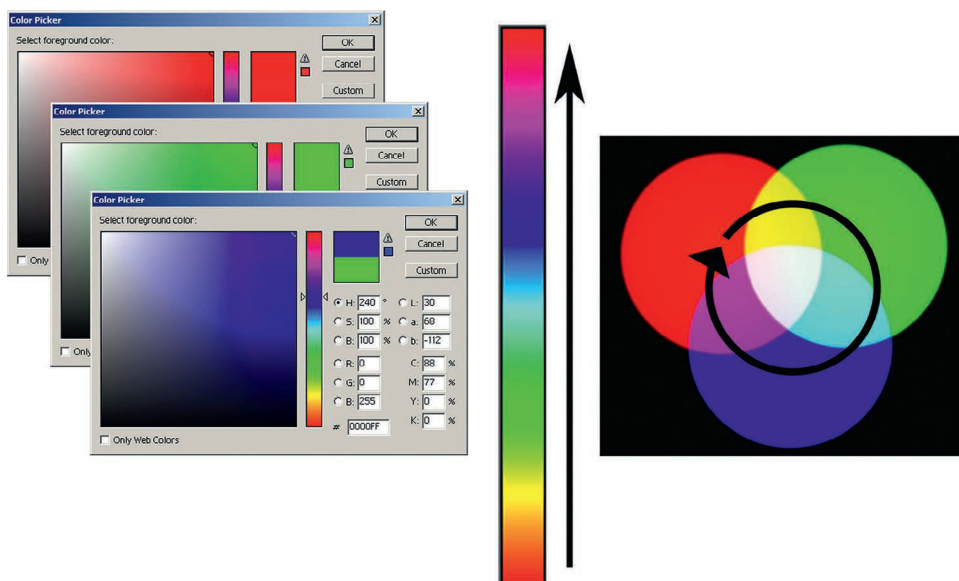


FIG 1.19 The Color Picker in Photoshop has a vertical rectangle of color graduating from red through the colors and back to red. This allows you to select a hue and use the Color Picker Palette to change the value and intensity.

Primary Colors

The three *primary colors* in the additive color system are red, green, and blue (RGB). They are referred to as primary colors because you can mix them and make all the other colors, but you can't create the primary colors by mixing any other color. Many projection televisions use an additive system where you can see the red, green, and blue lens that project these three colors to create the image you see.

Secondary Colors

The *secondary colors* are yellow, magenta, and cyan. When you mix equal amounts of two primary colors together, you get a secondary color. You can see that these colors are located between the primary colors on the color wheel and on the Photoshop Color Picker vertical strip.

Color Emphasis

Color is often used for emphasis. Look at Figure 1.20. All things being equal, the larger shapes dominate, but the small shapes demand your attention once color is added. Of course, there are many other forms of emphasis that you can use in creating art, but color can be the most powerful—and overused. Ever come across a webpage that has a busy background and every font, color, and mode of emphasis devised by man splashed across it? There is almost no actual emphasis, since all the elements cancel each other out. Let this be a cautionary tale to you: Often, less is more.

In another example using a photograph, Figure 1.21, you can see that in the first black-and-white photo, your eye would most likely be drawn to the dark opening of the doghouse and you would most likely assume that the subject

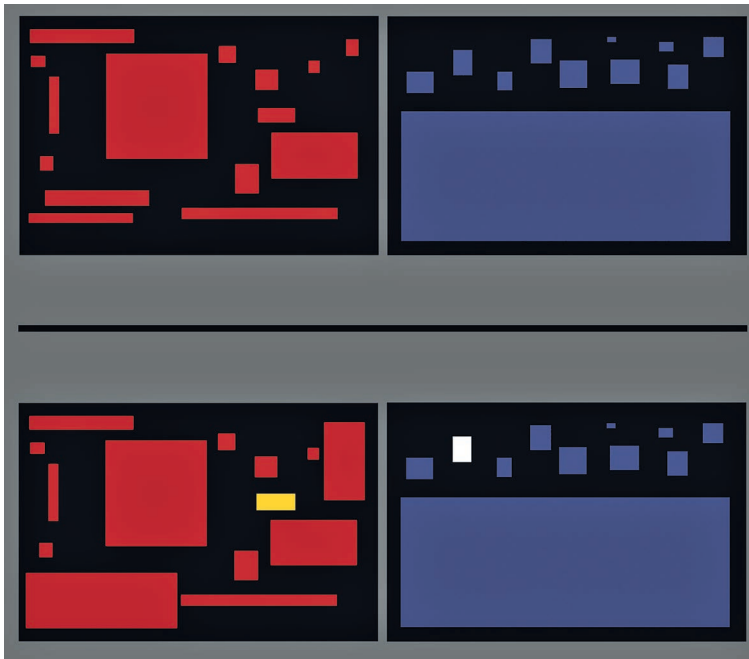


FIG 1.20 All things being equal, the larger shapes dominate, but the small shapes demand your attention once color is added.

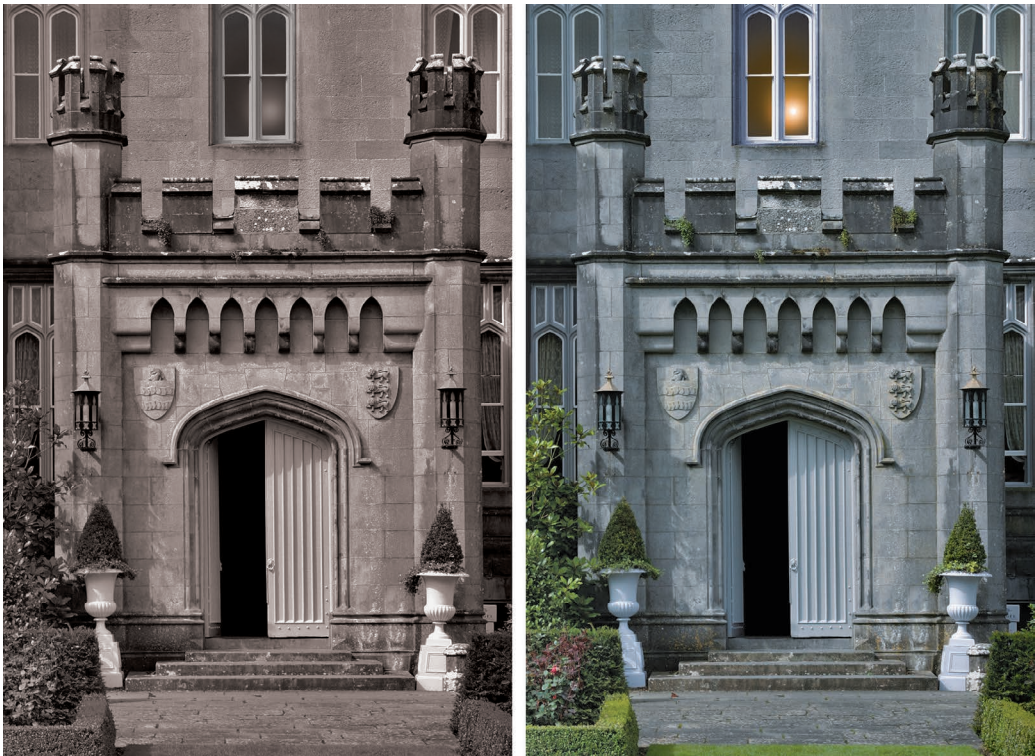


FIG 1.21 Your eye is most likely drawn to the open door in the black-and-white photo, but add color and the window above draws the primary interest.

of this picture is the doghouse. In the second version, the colorful flower draws the primary interest; it still competes with the doghouse door for attention, but you would probably make the assumption that the focus of this picture was the flower.

In a game scene, you can see the use of color drawing the attention of a player to an important item. Look at Figure 1.22. In the first version of the scene, you are drawn to the fire and then to all the items in the shadows. In the second version, the red crate draws your attention and clearly means something. Depending on the world logic of the game you are playing, that could simply mean that you can interact with the object, or it could mean that the item is dangerous. That decision brings us to our next topic: color expression.



FIG 1.22 In a room full of normal objects, the players' eyes will be drawn to the fire and then equally to the objects. In a room full of normal objects, a red crate draws attention, especially given the fact that there are other normal crates around it.

Color Expression or Warm and Cool Colors

When you start painting textures and choosing colors, you will want to know how they interact in terms of contrast, harmony, and even message. There is a lot of information on this topic, and once again, Johannes Itten (the guy who did the color wheel) enters the picture. Itten provided artists with a great deal of information on how color works and how colors work together. Itten was among the first people to look at color not just from a scientific point of view but from an artistic and emotional point of view. He was very interested in how colors made people feel. From his research we get the vocabulary of warm and cool colors.

We all are familiar with this convention, since it is mostly based on the natural world. When asked to draw a flame, we reach for the red or orange crayon. Ice is blue; the sun, yellow. Each warm and cool color has commonly associated feelings, both positive and negative. The brighter or more pure the color, the more positive the association. Darker and duller colors tend to have negative connotations associated with them.

The warm colors are red and yellow, and the cool colors are blue and green. Children will color the sun yellow and color ice blue and use the black crayon to scratch out things they don't like. Traffic lights are hot when you should stop or be cautious (red and yellow) and cool when it is okay to go (green). Red and orange are hot and usually associated with fire, lava, and coals. How many red and black shirts do you see at the mall? Red and black together generally symbolize demonic obsession. Red by itself can mean royalty and strength as well as demonism. Deep red can be erotic. Yellow is a hot color like the sun—a light giver. Yellow is rich like gold as a pure color. A deep yellow (amber) window in the dark of a cold night can mean fire and warmth. But washed out or pale yellow can mean envy or betrayal. Calling a person yellow is an insult, meaning that he is a coward. Judas is portrayed as wearing yellow garments in many paintings. During the Inquisition, people who were considered guilty of heresy were made to wear yellow. For green, we think of lush jungles teeming with life. As green washes out, we get a sense of dread and decay (zombie and orc skin). Vibrant green in a certain context can be toxic waste and radioactive slime. Blue in its saturated state is cold like ice, but fresh like water and the sky. Darker blues indicate misery. Purple is mysterious and royal.

Keep in mind that color is context-sensitive. Water is generally blue (would you drink dark-green water?). But not just any blue will do. In the real world, if we come across water that is a saturated blue that we can't see through, we get suspicious. Was this water dyed? Are there weird chemicals in there? If anything lives in that, what could it be?! Blood is generally red, but what if an enemy bled green? What if the game you are playing is about an alien race taking over earth and one of your companions bleeds green from an injury during combat? In a fantasy game, you might come across coins. Which coin do you take: the one made of bright yellowish metal or the one made of gray-green metal? With no previous information on the color of coins in this world, most people would pick the brighter yellow. Look at Figure 1.23. What are some of the assumptions you might make about these three scenes?

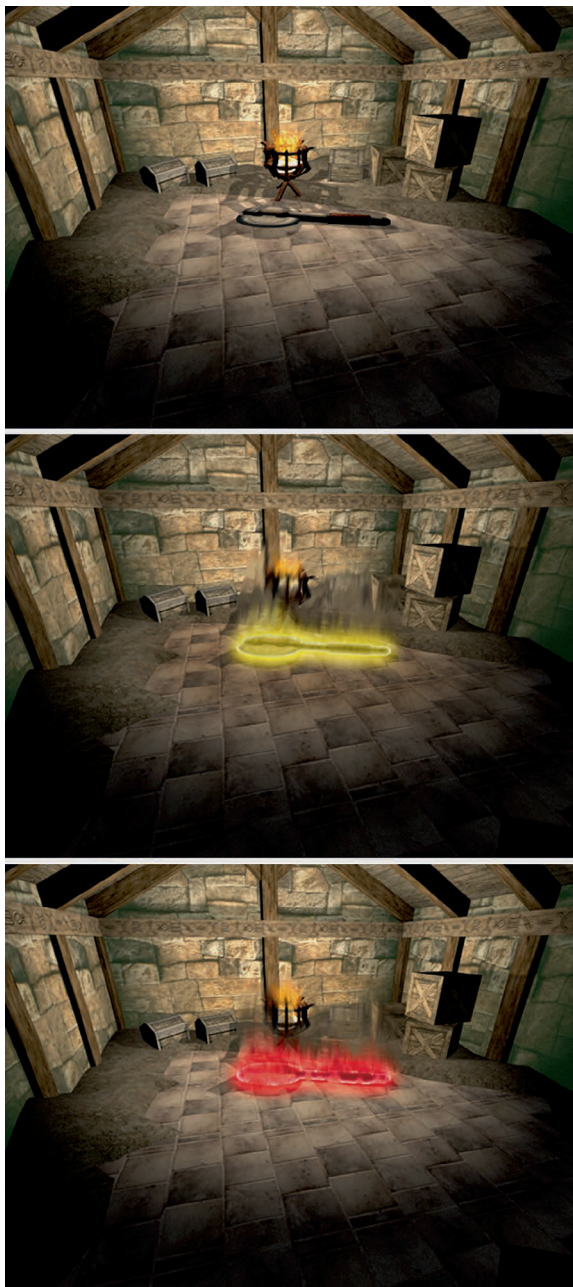


FIG 1.23 These three scenes are the same except for the ax. What questions and/or assumptions run through your mind looking at each version?

Looking at color in this way might make it seem a bit mechanical, but it still takes a talented artist to make the right color choices. You can memorize all the information in the world, but it usually comes down to having a good eye and being able to convey that vision in your work and to your coworkers.

Perspective

We discussed earlier in this chapter that dramatic perspective (Figure 1.24) is usually not used in the creation of a game texture, although sometimes perspective is present and needs to be understood. In addition, understanding perspective is not only a valuable artistic tool to have available but will also help you when you are taking digital reference images and when you are cleaning and straightening those images. We will look at the artistic aspects of perspective now; later, in the chapter on cleaning and storing your assets, we will talk about fixing those images.

Quite simply, *perspective* is the illusion that something far away from us is smaller than closer objects. This effect can be naturally occurring, as in a photo, or a mechanically created illusion in a painting. You can see samples of this illusion in Figure 1.25. In 2D artwork, perspective is a technique used to recreate that illusion and give the artwork a 3D depth. Perspective uses overlapping objects, horizon lines, and vanishing points to create a feeling of depth. You can see in Figure 1.26 an image and the same image with the

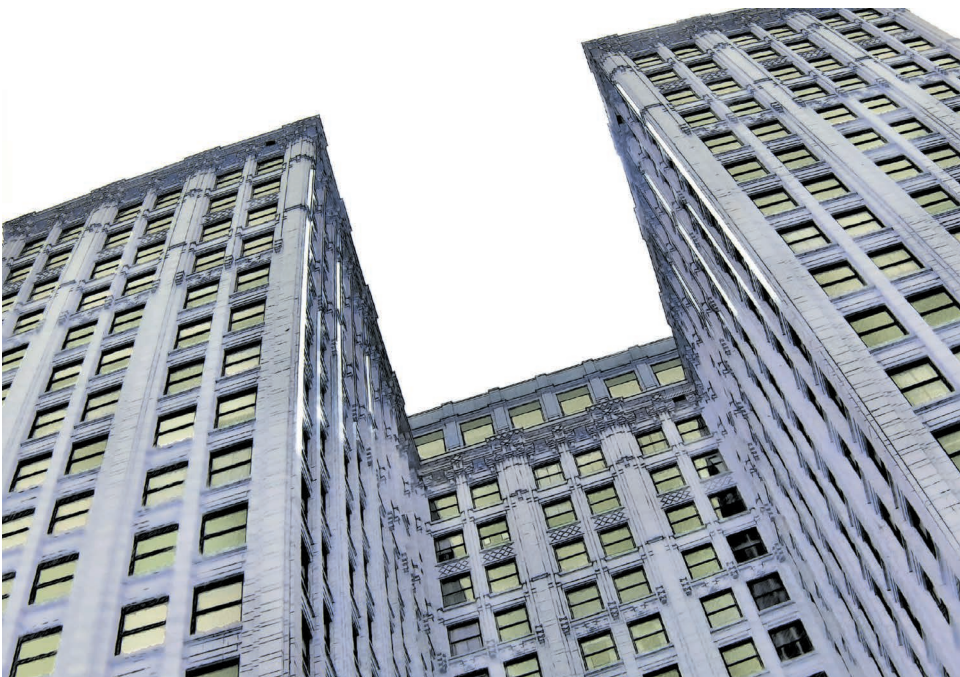


FIG 1.24 Although dramatic perspective is used in traditional art, it is not used in a game texture. But there is some notion of perspective, so it is best to understand the concept.

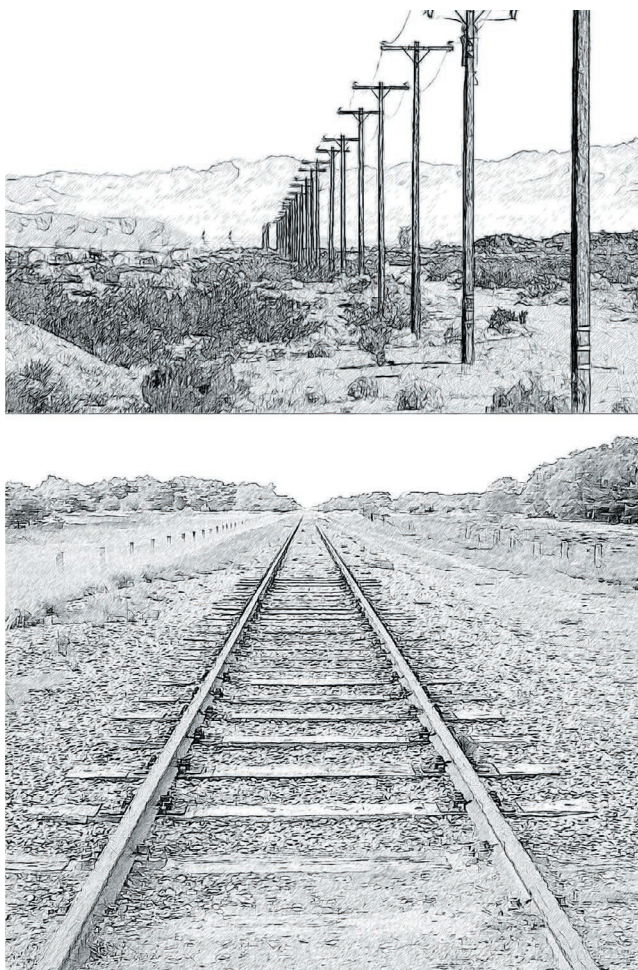


FIG 1.25 Perspective is the illusion that something far away from us is smaller. Are the telephone poles actually getting smaller in this image? Are the train tracks really getting closer together?

major lines of perspective as they converge on one point, called the *vanishing point*. Several types of perspective are used to achieve different effects.

One-Point Perspective

One-point perspective is when all the major lines of an image converge on one point. This effect is best illustrated when you're looking down a set of straight railroad tracks or a long road. The lines of the road and track, although we know they are the same distance apart, seem to meet and join together at some point in the far distance—the vanishing point. In one-point perspective, all the lines move away from you (the z-axis) and converge at the vanishing point.

Vertical and horizontal or up and down and right and left lines (the x- and y-axes) remain straight, as seen in Figure 1.27.

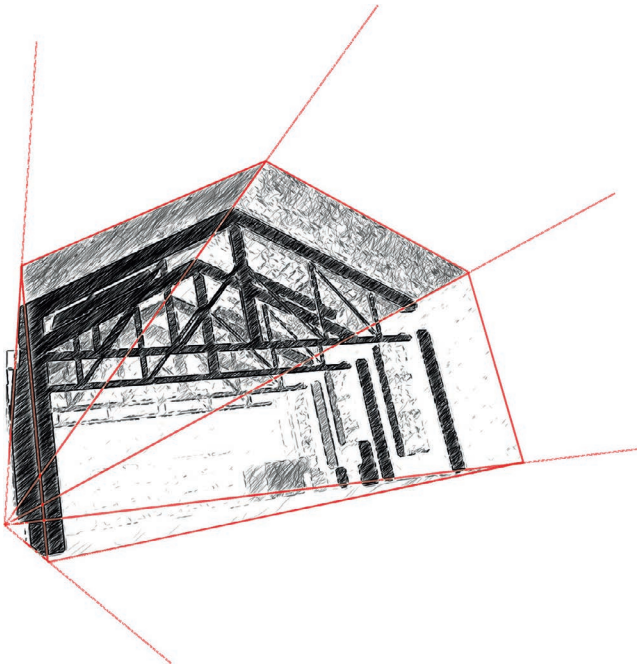


FIG 1.26 In 2D artwork, perspective is a technique used to recreate that illusion and give the artwork a 3D depth.

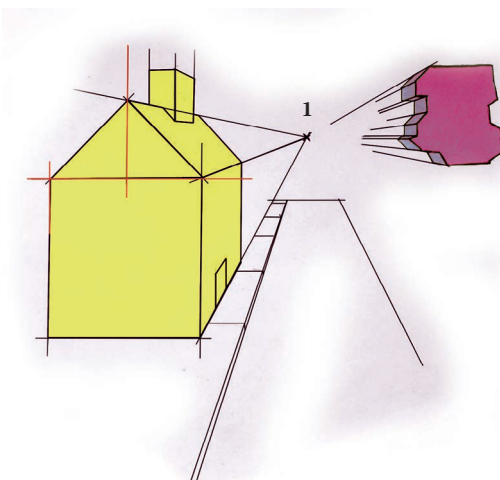


FIG 1.27 In one-point perspective, all the lines that move away from the viewer seem to meet at a far point on the horizon. This point is called the *vanishing point*.

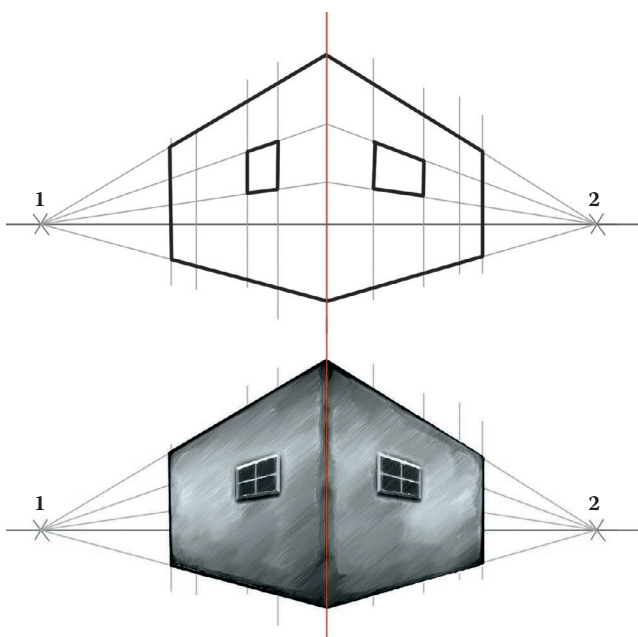


FIG 1.28 Two-point perspective has two vanishing points on the horizon line. All lines, except the vertical ones, will converge onto one of the two vanishing points.

Two-Point Perspective

One-point perspective works fine if you happen to be looking directly at the front of something or standing in the middle of some railroad tracks, but what if the scene is viewed from the side? Then you shift into *two-point perspective*. Two-point perspective has two vanishing points on the horizon line. All lines, except the vertical, will converge onto one of the two vanishing points. See Figure 1.28.

Three-Point Perspective

Three-point perspective is probably the most challenging of all. In three-point perspective every line will eventually converge on one of three points. Three-point perspective is the most dramatic and can often be seen in comic books when the hero is flying over buildings or kicking butt in the alley below as the buildings tower above. Figure 1.29 shows three-point perspective.

Perspective, from the texture artist's point of view while photographing surfaces for game art, can be the enemy. We will look at that issue in a coming chapter when we talk about collecting and cleaning your images. From the art education point of view, knowing what perspective is and what it looks like is enough for now.

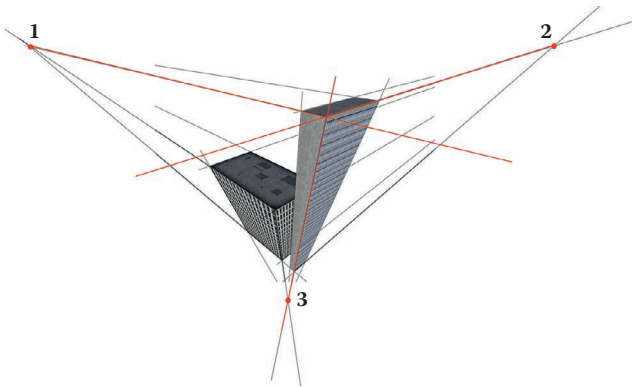


FIG 1.29 In three-point perspective, every line will eventually converge on one of three vanishing points.

Quick Studies of the World Around You

The following pages display some quick studies that I did of random objects. I tried to work through each of them as a game artist might to give you some quick and general examples of how a game artist might break them down. We will do this type of exercise in more depth throughout the book, but in the tutorial portions of the book, those breakouts will be more specific and focused on the goal at hand. This is a general look and introduction on the thought process of recreating surfaces and materials in a digital environment. I covered all that was introduced in this chapter: shape and form, light and shadow, texture, and color, as well as considering other aspects of the object or material. I didn't touch on perspective in these exercises because I wanted to limit the exercise to recreating 2D surfaces (textures), and perspective is not as critical as the other concepts in this chapter. In the following pages, Figures 1.30 to 1.35 each have a caption that discusses the particulars of each study.

Conclusion

This chapter was an overview of the most basic yet critical aspects of traditional art. Understanding the concepts in this chapter and further exploring them on your own will make you a much better texture artist. We are now ready to get more technical and look at the mechanical issues of creating game textures.

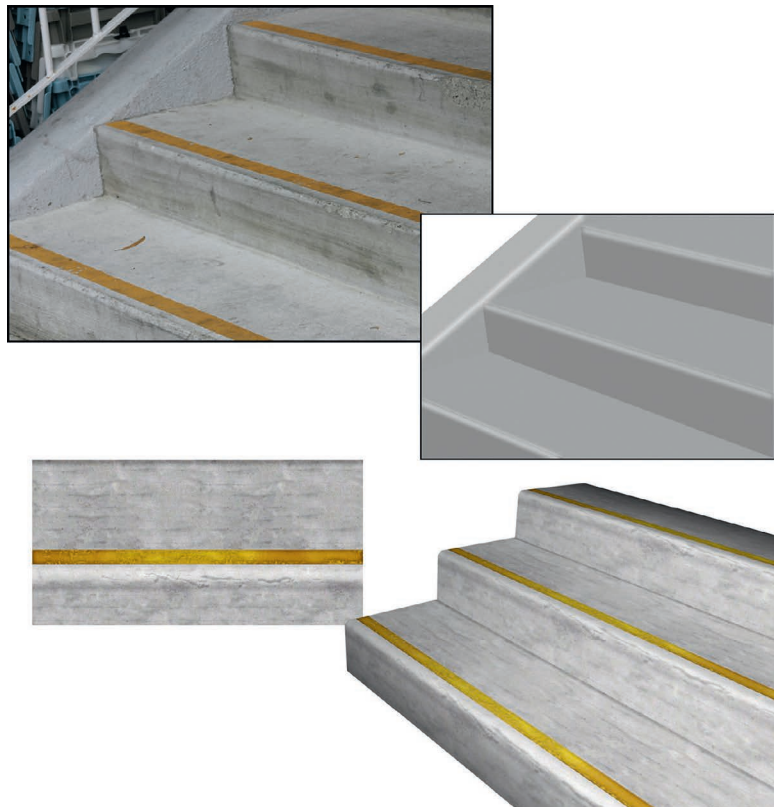


FIG 1.30 The upper-left image is a digital photo of some simple concrete stairs. You might have an art lead e-mail you an image like this and tell you that she wants a texture based on these stairs. Fortunately, this is a rather simple form; it doesn't have a lot of color or detail to distract us. Look at the simple recreation of the stairs to the right showing the basic light and shadow patterns on the stairs. The lower-left image shows the 2D texture created in Photoshop to be applied to a 3D model of the stairs. If you look at the yellow stripe on the stairs and compare it to the stripe on the texture, you can see the highlights painted in the texture where the edge of the step is and the shadow under the lip of the edge. If you were able to closely examine the original digital image of the stairs, you would see an almost infinite amount of detail. Part of the texture artist's job is to know when to draw the line. Here I didn't include every scuff and mark from the original stair image because that approach wouldn't work. You will learn in coming chapters that such details usually stand out and draw attention to the repeating pattern of a texture, or, in the case of fabrics and fine meshes, can create noise or static in the texture. I created this texture pretty quickly; given more time, I would experiment with the chips and wear on the edge of the steps to add more character.

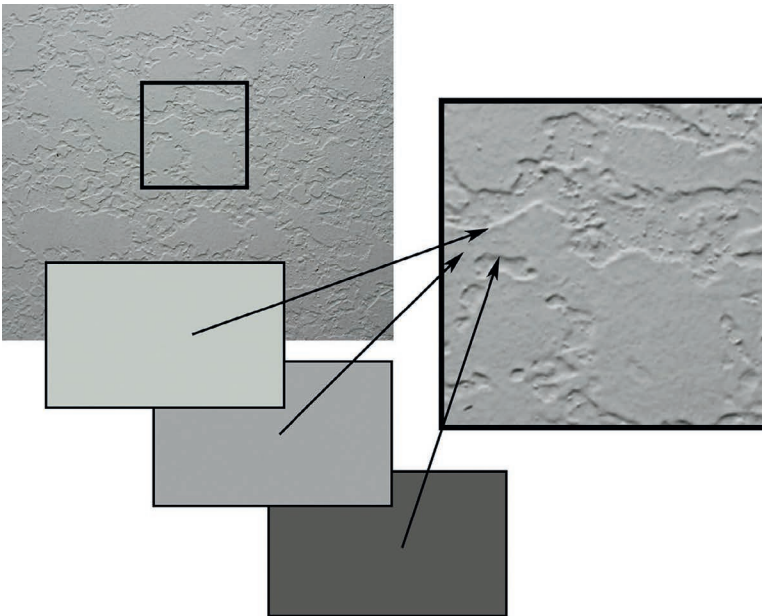


FIG 1.31 This is a straight-on photo of an interior plaster wall. I included this obviously unexciting image to demonstrate that even in such a simple surface, there can be complex highlight and shadow going on. Look at the color swatches of the highlight, shadow, and midtone. Notice that the colors are not simple black, white, and gray. The highlight is not pure white or light gray, but a very pale green. Look at the close-up of the image. You can clearly see the consistent behavior of light as it highlights the upper ridges of the plaster and shadow falls from the lower edges. Once you start studying such seemingly commonplace things, like a wall that you walk by a hundred times a day, you will start to notice, understand, and remember how various lights, materials, and other factors affect a surface. Do you convey that simple raised pattern in the texture using geometry or a shader? Of course, that depends on many factors, and hopefully by the end of this book you will know what questions to ask to determine the answers to such questions.



FIG 1.32 This image simply shows the world that I need to wash my car. Seriously, look at the various parts of complex objects and you will see a variety of surface behaviors. Notice how the paint is highly reflective and mirrors the world around the car. The metal is not flat like a mirror, so notice the distortion of the reflected image. The windows, while reflecting the surrounding world as well, are translucent, so you can see what's behind the window and on the other side of the car. The window also has a patina of dirt and spots on it. If you needed to recreate this object as realistically as possible, you would have to take all those aspects into consideration and determine the best way to achieve the effect. Look at the close-up of the rim. You can see that the highlights are not mirror-like in their accuracy but rather are a diffuse notion of highlight. This looks simple to paint, but wheels rotate and will instantly look bad if not painted properly. Using a real-time process for highlights eliminates this problem. Though the tires are flat black and reveal only a faint notion of highlight, depending on the detail level, you may be dealing with complex mapping and shader effects here, too. All of this seems obvious, but taking the time to examine the object you are recreating and to understand what you are seeing and how to verbalize it helps when turning the object into game art. If you were to make materials or textures for this vehicle, you would need to know many things about the technology and how the car will be used in the game. Can we have real-time environmental reflections? Can we fake them using a shader? Do we have to carefully paint in a vague notion of metallic highlights that work in all situations the car may be in? And the windows: Can we do a translucent/reflective surface with an alpha channel for dirt? If the car is used in a driving game in which the vehicle is the focus of the game and the player gets to interact up close and personal with the car, I am sure a lot of attention will be given to these questions. But if this car is a static prop sitting on a street that the player blazes past, then over-the-top effects may only be a waste of development time and computer resources.



FIG 1.33 This sewer intrigued me: a simple shape of a common item that many might overlook as not worthy of serious attention. Some may have the attitude that it is only a sewer grate, so make it and move on. But a shiny new sewer grate with clean edges would stand out in a grungy urban setting. Look at this sewer grate. It is made of iron and looks solid and heavy. It was probably laid down decades ago and has had thousands of cars drive over it, people walk over it, millions of gallons of rainwater pour through it. On the image at the upper left you can look at the iron and see how it is rusted, but it's so well worn that the rust is polished off in most places. Dirt has built up in the cracks between the grate, the rim, and the concrete. Even little plants have managed to grow. Look at the close-up at the upper right and you can see just how beat-up this iron is and how discolored it has become. At the lower left, I desaturated and cleaned up a portion of the image to see just how the light and shadow are hitting it and to get a feel for the quality of the surface. In this image, you can more clearly see the roughness of the cement and the metal, and although the circular grate looks round from a distance, up close there are no straight edges and smooth curves. All this detail can't be depicted 100% in a game texture, but knowing it's there and understanding what you are seeing will allow you to convey a richer version of the grate as you learn to focus on those details that add realism and character. On the lower right is a texture I did, and you can see that I was able to quickly achieve a mottled and grungy look for the metal and the edges. There are a few places at the top where I started the process of eating away at the concrete and the metal a bit.

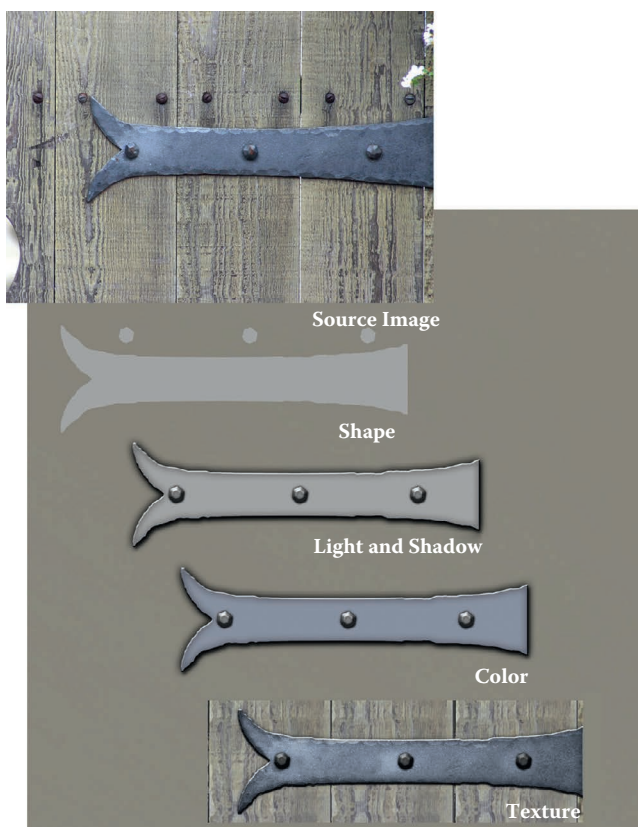


FIG 1.34 This image is similar to the sewer in approach. Here I wanted to point out how a simple shape can be turned into an ornate hinge with little effort. The top image is the original digital photo of the hinge. I drew the shape of the hinge in Photoshop. You may notice that I drew the screws separately. This is because you need the shapes separately to work with them in Photoshop—you will see why later in the book. In Photoshop I applied and adjusted the Layer Effects and then colored the hinge close to the overall color of the original. After that it was a matter of applying the right filters and doing some hand work to get the edges looking right. We will be doing this type of work throughout the book. I will remind you from time to time that although the best approach may be to use a photo source or any one of the other methods available, the focus of this book is to help you develop a set of Photoshop skills that will allow you to not depend on any one method. These skills will improve your abilities when you're working in any of the other methods.

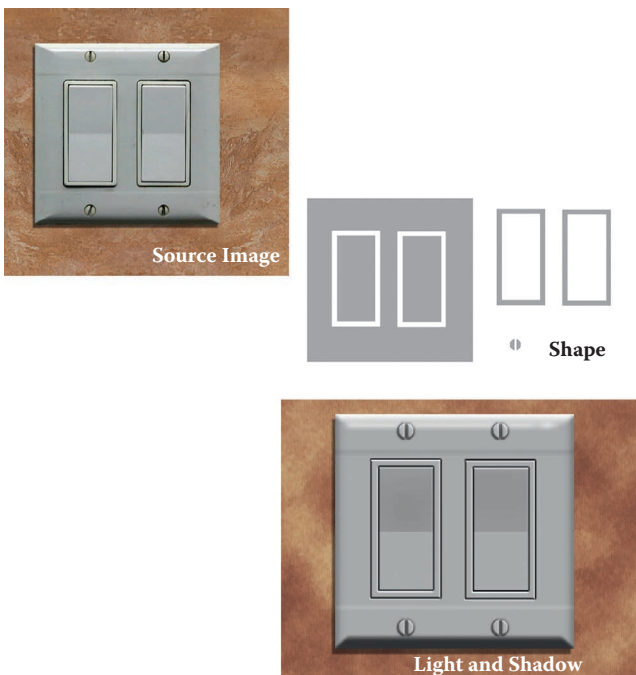


FIG 1.35 This light switch is a common object that you might need to create. Instead of taking the time to clean up and manipulate a photo, you can just make one more quickly from scratch. The switch is composed of simple shapes with the Layer Effects applied. The wall behind the switch was a quick series of filters to add a base for this exercise.

Chapter Exercises

1. Examine and sketch your own series of basic shapes from world objects. Start with a ball, a cereal box, and other simple objects and work up to more complex items (what basic shapes is a car made from?). Pick three different objects and sketch each using only basic shapes. Compare the accuracy of your drawing by holding it at arm's length next to the object in the background. If you are having trouble with this exercise, it may help to use a picture so that you can work side by side and compare the two images—yours and the original.
2. Train your eye to see light and shadow. Start with a flashlight or one light source in a dark room and shine it at an object on a bare tabletop. The shadow should be pretty easy to identify. Move the light back and at different angles and watch the shadow move. Now turn on another light source. Is it brighter than the light you are holding? Is it a different color? See how the dual shadows interact with each other. What happens to the shadow when the object is moved instead of the light? Also study how light and shadow affect color. Notice the various shades of color across a surface that may at first seem to be one solid color. Better yet, take an

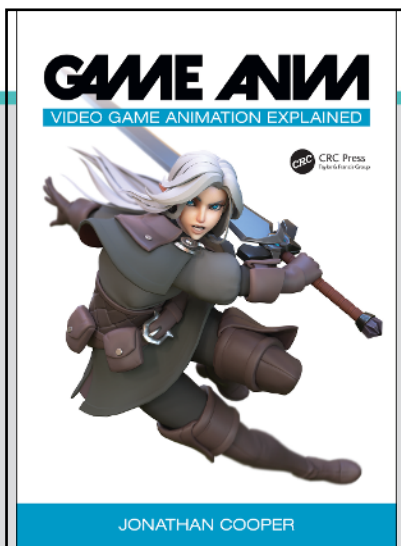
- image into Photoshop and sample the various parts of the surface from darkest to lightest and take note of the range of shades the surface holds.
3. Focus on visual texture. Pick any object and look at it closely. A common mistake new artists make is to create a base texture and stop. A perfectly colored, shiny, and cleaner-than-clean plastic-looking object is going to look fake. In real life, even a new object has some texture—if not an actual texture, like a leather grain, then a notion of the imperfection of the real world. Sometimes a subtle noise or cloud overlay is enough to create the almost indistinguishable texture that creates a more believable surface. Any object in the real world will have scratches from wear and tear, and other signs of use will develop pretty quickly. Study the texture of common objects and see what has created this object's texture. Is a wooden porch step smooth from use or rough due to weather and decay?
 4. Look closely at what color you are “really” seeing. As noted in this chapter, we mentally fill in a lot of blanks when it comes to trying to determine what we are seeing. Color is no exception. Look at images from the Internet, or better yet, take digital images with your own camera and load them into Photoshop or a similar image program. Sample the colors of various spots and see that almost no color is going to be pure, perfectly even, or at all where you might think it would be on the color selector. A red mug is not going to be a pure red mug; it will most likely be a shade darker or lighter or contain other colors.
 5. Color expression: Can you make a puppy look evil? Try painting its eyes red. Take an image of a garden and desaturate it. What other images can you take and simply change the colors (all of them across the image or only one color) and create various moods and messages? A house with rich amber windows at night might seem inviting. How does a window with a red tint or a green cast make you feel? Try combining colors, too. An amber window with a predominantly black surrounding may feel like a lonely sight in the void, as opposed to the fanciful and richer blue surrounding.
 6. Do your own quick studies of the world. Take pictures or just examine a surface or object and determine every bit of information you can based on what you learned in this chapter. Define the basic shape, note how light and shadow play on the object and how the object affects the light and shadow of the world around it, and sample and determine the colors of the object and the amount of variation of the color along the surface of the object. How would you begin to create the surface of this object in Photoshop? Don't try yet, but think about it. Later you will learn the tools necessary to create surfaces, but for now you need to be able to dissect a surface and determine what colors, shapes, and other qualities the surface contains.



CHAPTER

3

THE FIVE FUNDAMENTALS OF GAME ANIMATION



This chapter is excerpted from
Game Anim
by Jonathan Cooper

© 2019 Taylor & Francis Group. All rights reserved.



[Learn more](#)

The Five Fundamentals of Game Animation

The 12 animation principles are a great foundation for any animator to understand, and failure to do so will result in missing some of the underlying fundamentals of animation—visible in many a junior’s work. Ultimately, however, they were written with the concept of linear entertainment like TV and film in mind, and the move to 3D kept all of these elements intact due to the purely aesthetic change in the medium. Three-dimensional animated cartoons and visual effects are still part of a linear medium, so they will translate only to certain elements of video game animation—often only if the game is cartoony in style.

As such, it’s time to propose an additional set of principles unique to game animation that don’t replace but instead complement the originals. These are what I have come to know as the core tenets of our new nonlinear entertainment medium, which, when taken into consideration, form the basis of video game characters that not only look good, but feel good under player control—something the original 12 didn’t have to consider. Many elements are essential in order to create great game animation, and they group under five fundamental areas:

1. Feel
2. Fluidity
3. Readability
4. Context
5. Elegance

Feel

The single biggest element that separates video game animation from traditional linear animation is interactivity. The very act of the player controlling and modifying avatars, making second-to-second choices, ensures that the animator must relinquish complete authorship of the experience. As such, any uninterrupted animation that plays start to finish is a period of time the player is essentially locked out of the decision-making process, rendered impotent while waiting for the animation to complete (or reach the desired result, such as landing a punch).

The time taken between a player's input and the desired reaction can make the difference between creating the illusion that the player is embodying the avatar or becoming just a passive viewer on the sidelines. That is why cutscenes are the only element in video games that for years have consistently featured a "skip" option—because they most reflect traditional noninteractive media, which is antithetical to the medium.

Response

Game animation must always consider the response time between player input and response as an intrinsic part of how the character or interaction will "feel" to the player. While generally the desire is to have the response be as quick as possible (fewer frames), that is dependent on the context of the action. For example, heavy/stronger actions are expected to be slower, and enemy attacks must be slow enough to be seen by the player to give enough time to respond.

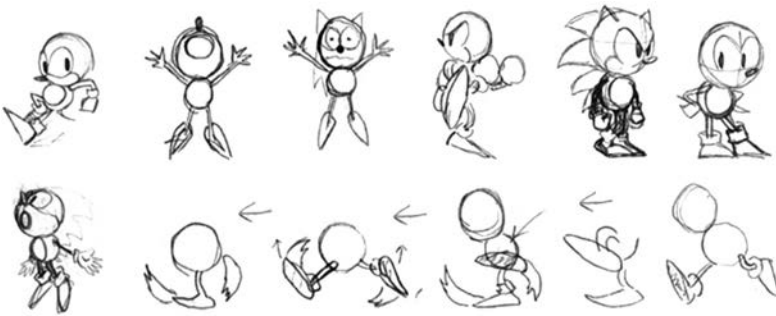
It will be the game animator's challenge, often working in concert with a designer and/or programmer, to offer the correct level of response to provide the best "feel," while also retaining a level of visual fidelity that satisfies all the intentions of the action and the character. It is important not to sacrifice the weight of the character or the force of an action for the desire to make everything as responsive as possible, so a careful balancing act and as many tricks as available must be employed.

Ultimately, though, the best mantra is that "gameplay wins." The most fluid and beautiful animation will always be cut or scaled back if it interferes too much with gameplay, so it is important for the game animator to have a player's eye when creating response-critical animations, and, most importantly, play the game!

Inertia & Momentum

Inertia is a great way to not only provide a sense of feel to player characters, but also to make things fun. While some characters will be required to turn on a dime and immediately hit a run at full speed, driving a car around a track that could do the same would not only feel unrealistic but mean there would be no joy to be had in approaching a corner at the correct speed for the minimum lap time. The little moments when you are nudging an avatar because you understand their controls are where mastery of a game is to be found, and much of this is provided via inertia.

Judging death-defying jumps in a platform game is most fun when the character must be controlled in an analogue manner, whereby they take some time to reach full speed and continue slightly after the input is released. This is as much a design/programming challenge as it is animation, but the animator often controls the initial inertia boost and slowdown in stop/start animations.



Original sketches from Sonic the Hedgehog (circa 1990). The animation frames are already heavily displaying inertia. (Courtesy of SEGA of America.)

Momentum is often conveyed by how long it takes a character to change from current to newly desired directions and headings. The general principle is that the faster a character is moving, the longer it takes to change direction via larger turn-circles at higher speeds or longer plant-and-turn animations in the case of turning 180°.

Larger turn-circles can be made to feel better by immediately showing the intent of the avatar, such as having the character lean into the turn and/or look with his or her head, but ultimately we are again balancing within a very small window of time lest we render our characters unresponsive.

A classic example is the difference between the early Mario and Sonic the Hedgehog series. Both classic Mario and Sonic's run animations rely heavily on inertia and have similar long ramp-ups to full speed. While Mario immediately starts cartoonishly running at full speed as his legs spin on the ground to gain traction, Sonic slowly transitions from a walk to a run to a sprint. While Mario subjectively feels better, this is by design, as Sonic's gameplay centers on high speeds and "flow," so stopping or slowing down is punitive for not maintaining momentum.

Visual Feedback

A key component of the "feel" of any action the player and avatar perform is the visual representation of that action. A simple punch can be made to feel stronger with a variety of techniques related to animation, beginning with the follow-through following the action. A long, lingering held pose will do wonders for telling the player he or she just performed a powerful action. The damage animation on the attacked enemy is a key factor in informing the player just how much damage has been suffered, with exaggeration being a key component here.

In addition, employing extra tricks such as camera-shake will help further sell the impact of landing the punch or gunshot, not to mention visual effects of blood or flashes to further register the impact in the player's mind. Many fighting games employ a technique named "hit-stop" that freezes the

characters for a single frame whenever a hit is registered. This further breaks the flow of clean arcs in the animations and reinforces the frame on which the impact took place.

As many moves are performed quickly so as to be responsive, they might get lost on the player, especially during hectic actions. Attacking actions can be reinforced by additional effects that draw the arc of the punch, kick, or sword-swipe on top of the character in a similar fashion to the “smears” and “multiples” of old. When a sword swipe takes only 2 frames to create its arc, the player benefits mostly from the arcing effect it leaves behind.

Slower actions can be made to feel responsive simply by showing the player that at least part of their character is responding to their commands. A rider avatar on a horse can be seen to immediately turn the horse’s head with the reins even if the horse itself takes some time to respond and traces a wide circle as it turns. This visual feedback will feel entirely more responsive than a slowly turning horse alone would following the exact same wide turn.

Much of the delay in visual feedback comes not from the animation alone, but the way different game engines handle inputs from the joypad in the player’s hands. Games like the Call of Duty series place an onus on having their characters and weapons instantly respond to the player’s inputs with minimal lag and high frame rates, whereas other game engines focused more on graphics postprocessing will have noticeably longer delays (measured in milliseconds) between a jump button-press and the character even beginning the jump animation, for example. This issue is further exacerbated by modern HDTVs that have lag built in and so often feature “Game Mode” settings to minimize the effect. All this said, it is still primarily an animator’s goal to make characters as responsive as possible within reason.

Fluidity

Rather than long flowing animations, games are instead made of lots of shorter animations playing in sequence. As such, they are often stopping, starting, overlapping, and moving between them. It is a video game animator’s charge to be involved in how these animations flow together so as to maintain the same fluidity put into the individual animations themselves, and there are a variety of techniques to achieve this, with the ultimate goal being to reduce any unsightly movement that can take a player out of the experience by highlighting where one animation starts and another ends.

Blending and Transitions

In classic 2D game sprites, an animation either played or it didn’t. This binary approach carried into 3D animation until developers realized that, due to characters essentially being animated by poses recorded as numerical values, they could manipulate those values in a variety of ways. The first such

improvement that arrived was the ability to blend across (essentially cross-fading animations during a transitory stage) every frame, taking an increasing percentage of the next animation's value and a decreasing percentage of the current as one animation ended and another began. While more calculation intensive, this opened up opportunities for increasing the fluidity between individual animations and removing unsightly pops between them.



Prince of Persia: Sands of Time was the first game to really focus on small transitions for fluidity. (Copyright 2003 Ubisoft Entertainment. Based on Prince of Persia®, created by Jordan Mechner. Prince of Persia is a trademark of Waterwheel Licensing LLC in the US and/or other countries used under license.)

A basic example of this would be an idle and a run. Having the idle immediately cancel and the run immediately play on initial player input will cause the character to break into a run at full speed, but the character will unsightly pop as he or she starts and stops due to the potential repeated nature of the player's input. This action can be made more visually appealing by blending between the idle and run over several frames, causing the character to more gradually move between the different poses. Animators should have some degree of control over the length of blends between any two animations to make them as visually appealing as possible, though always with an eye on the gameplay response of the action.

The situation above can be improved further (albeit with more work) by creating brief bespoke animations between idle and run (starting) and back again (stopping), with blends between all of them. What if the player started running in the opposite direction he or she is facing? An animator could create a transition for each direction that turned the character as he or she began running in order to completely control the character's weight-shift as he or she leans into the desired direction and pushes off with his or her feet.

What if the character isn't running but only walking? Again, the animator could also create multiple directional transitions for that speed. As you can see, the number of animations can quickly spiral in number, so a balance must be found among budget, team size, and the desired level of fluidity.

Seamless Cycles

Even within a single animation, it is essential to maintain fluidity of motion, and that includes when a cycling animation stops and restarts. A large percentage of game animations repeat back on themselves, so it is important to again ensure the player cannot detect when this transition occurs. As such, care must be taken to maintain momentum through actions so the end of the animation perfectly matches the start.

It is not simply enough to ensure the last frame of a cycle identically matches the first; the game animator must also preserve momentum on each body part to make the join invisible. This can be achieved by modifying the curves before and after the last frame to ensure they create clean arcs and continue in the same direction. For motion-capture, where curves are mostly unworkable, there are techniques that can automatically provide a preservation of momentum as a cycle restarts that are described later in this book.

Care should also be taken to maintain momentum when creating an animation that transitions into a cycle, such as how the stopping animation should seamlessly match the idle. For maximum fluidity, the best approach in this case is to copy the approved idle animation and stopping transition into the same scene to manually match the curves leading into the idle, exporting only the stopping transition from that scene.

Settling

This kind of approach should generally be employed whenever a pose must be assumed at the end of an animation, time willing. It is rather unsightly to have a large movement like an attack animation end abruptly in the combat idle pose, especially with all of the character's body parts arriving simultaneously. Offsetting individual elements such as the arms and root are key to a more visually pleasing settle.

Notably, however, games often suffer from too quickly resuming the idle pose at the end of an animation in order to return control to the player to promote response, but this can be avoided by animating a long tail on the end of an animation and, importantly, allowing the player to exit out at a predetermined frame before the end if new input is provided. This ability to interrupt an animation before finishing allows the animator to use the desired number of frames required for a smooth and fluid settle into the following animation.

Settling is generally achieved by first copying the desired end pose to the end of an animation but ensuring some elements like limbs (even divided



Uncharted 4: A Thief's End makes heavy use of "abort frames" to exit gameplay animations and cinematics before completion for fluidity. (Courtesy of Sony Interactive Entertainment.)

into shoulder and forearms) arrive at their final position at different times, with earlier elements hitting, then overshooting, their goal, creating overlapping animation. Settling the character's root (perhaps the single most important element, as it moves everything not planted) is best achieved by having it arrive at the final pose with different axes at different times. Perhaps it achieves its desired height (Y-axis) first as it is still moving left to right (X-axis), causing the root to hit, then bounce past the final height and back again. Offsetting in the order of character root, head, and limbs lessens the harshness of a character fully assuming the end pose on a single frame—though care must be taken to not overdo overlap such that it results in limbs appearing weak and floppy.

Readability

After interactivity, the next biggest differentiator between game and traditional animation, in 3D games at least, is that game animations will more often than not be viewed from all angles. This bears similarity to the traditional principle "staging," but animators cannot cheat or animate to the camera, nor can they control the composition of a scene, so actions must be created to be appealing from all angles. What this means is when working on an animation, it is not enough to simply get it right from a front or side view. Game animators must take care to always be rotating and approving their motion from all angles, much like a sculptor walking around a work.

Posing for Game Cameras

To aid the appeal and readability of any given action, it is best to avoid keeping a movement all in one axis. For example, a combo of three punches should not only move the whole character forward as he or she attacks, but also slightly to the left and right, twisting as they do so.

Similarly, the poses the character ends in after every punch should avoid body parts aligning with any axes, such as arms and legs that appear to bend only when viewed from the side. Each pose must be dynamic, with lines of action drawn through the character that are not in line with any axes.

Lines of action are simplified lines that can be drawn through any single pose to clearly illustrate the overall motion for the viewer. Strong poses can be illustrated in this way with a single arcing or straight line, whereas weaker and badly thought-out poses will generally have less-discernible lines that meander and are not instantly readable to the viewer. Lines that contrast greatly between one pose and the next (contrasting actions) promote a more readable motion for the viewer than multiple similar or weak poses.

For the motions themselves, swiping actions always read better than stabbing motions, as they cover an arc that will be seen by the player regardless of camera angle. Even without the aid of a trail effect, a swipe passes through multiple axes (and therefore camera angles), so even if players are viewing from a less-than-ideal angle, they should still have an idea of what happened, especially if the character dramatically changes the line of action during poses throughout the action.

All this said, work to the game being made. If the camera is fixed to the side, such as in a one-on-one fighting game, then actions should be created to be most readable from that angle. Similarly, if you are creating a run animation for a game mostly viewed from the rear, then ensure the cycle looks best from that angle before polishing for others.



*League of Legends pushes animation incredibly far due to the far overhead cameras and frenetic onscreen action.
(Courtesy of Riot Games.)*

Silhouettes

At the character design/concept stage, the animator should get involved in helping guide how a character might look, not just to avoid issues such as hard, armorlike clothing at key versatile joints such as shoulders or waists. The animator should also help guide the design so as to help provide the best silhouettes when posed. A character with an appealing silhouette makes the job of animating far easier when attempting to create appeal than one composed entirely of unimaginative blobs or shapeless tubes for limbs.



Team Fortress 2 uses distinct character silhouettes for gameplay, making the animator's job of bringing appeal much easier. (Used with permission from Valve Corp.)

It is advisable to request “proxy” versions of characters at early stages of development so they can be roughly animated and viewed in the context of the gameplay camera, which, due to wide fields of view (for spatial awareness gameplay purposes), often warps the extremities of character as they reach the screen’s edge. Generally, the most appealing characters look chunkier and thicker than they might in real life, due to them being warped and stretched once viewed from the wide-angle game camera.

Collision & Center of Mass/Balance

As with all animation, consideration must be given to the center of mass (COM; or center of balance) of a character at any given frame, especially as multiple animations transition between one another so as to avoid

unnatural movements when blending. The COM is generally found over the leg that is currently taking the full weight of the character's root when in motion or between both feet if they are planted on the ground when static. Understanding this basic concept of balance will not only greatly aid posing but also avoid many instances of motions looking wrong to players without them knowing the exact issue.

This is especially true when considering the character's collision (location) in the game world. This is the single point where a character will pivot when rotated (while moving) and, more importantly, where the character will be considered to exist in the game at any given time. The game animator will always animate the character's position in the world when animating away from the 3D scene origin, though not so if cycles are exported in place. Importantly, animations are always considered to be exported relative to this prescribed location, so characters should end in poses that match others (such as idles) relative to this position. This will be covered in full in the following chapter.

Context

Whereas in linear animation, the context of any given action is defined by the scene in which it plays and what has happened in the story up to that point and afterward, the same is impossible in game animation. Oftentimes, the animator has no idea which action the player performed beforehand or the setting in which the character is currently performing the action. More often than not, the animation is to be used repeatedly throughout the game in a variety of settings, and even on a variety of different characters.

Distinction vs Homogeneity

Due to the unknown setting of most game animations, the animator must look for opportunities to give character to the player and nonplayer characters whenever possible, and must also consider when he or she should avoid it.

If, for example, the animator knows that a particular run cycle is only to be performed on that character being animated, then he or she can imbue it with as much personality as matches the character description. It's even better if the animator can create a variety of run cycles for that character in different situations. Is the character strong and confident initially, but later suffers loss or failure and becomes despondent? Is the character chasing after someone or perhaps running away from a rolling boulder about to crush him or her? The level of distinction the animator should put into the animation depends on how much control he or she has over the context in which it will be seen.



The player character generally moves at a much higher fidelity and with more distinction than NPCs. (Copyright 2007–2017 Ubisoft Entertainment. All Rights Reserved. Assassin's Creed, Ubisoft, and the Ubisoft logo are trademarks of Ubisoft Entertainment in the US and/or other countries.)

If an animation is not designed for the player character but instead to be used on multiple nonplayer characters, then the level of distinction and notability should generally be dialed down so as to not stand out. Walks and runs must instead be created to look much more generic, unless the animation is shared by a group of NPCs only (all soldiers might run differently from all civilians). Almost always, the player character is unique among a game world's inhabitants, so this should be reflected in his or her animations.

Repetition

Similarly, within a cycling animation, if the action is expected to be repeated endlessly, such as an idle or run cycle, then care must be taken to avoid any individual step or arm swing standing out against the rest, lest it render the rhythm of repetition too apparent to the player—such as every fourth step having a noticeably larger bounce for example.

Stand-out personality can instead be added to on-off actions or within cycles via “cycle breakers” such as the character shifting his or her footing after standing still too long, performing a slight stumble to break up a tired run, or even by modifying the underlying animation with additive actions—covered in more detail in the following chapter.



Uncharted: Drake's Fortune utilized additive poses to avoid repetition when in cover.

Onscreen Placement

A key factor in setting the exaggeration of movement is the relative size on the screen of the character as defined by the camera distance and field of view. While cameras have gotten closer and closer as the fidelity of characters has risen, players still need to see a lot of the environment on screen for awareness purposes, so many games may show characters that are quite small. Far cameras require actions to be much larger than life so as to be read by the player.

The same is true of enemy actions that are far off in the distance, such as damage animations to tell the player he or she landed a shot. Conversely, only really close cameras such as those employed in cutscenes afford subtleties like facial expressions—here, overly theatrical gestures will generally look out of place. It is important as a game animator to be aware of the camera for any particular action you are animating and to animate accordingly within the style of the project. The wide field of view of the gameplay camera will even distort the character enough to affect the look of your animation, so, as ever, the best way to evaluate the final look of your animation is in the game.

Elegance

Game animations rarely just play alone, instead requiring underlying systems within which they are triggered, allowing them to flow in and out of one another at the player's input—often blending seamlessly, overlapping one another, and combining multiple actions at once to ensure the player is unaware of the individual animations affording their avatar motion.

If not designing them outright, it is the game animator's duty to work with others to bring these systems and characters to life, and the efficiency of

any system can have a dramatic impact on the production and the team's ability to make changes further down the line toward the end of a project. Just as a well-animated character displays efficiency of movement, a good, clean, and efficient system to play them can work wonders for the end result.

Simplicity of Design

Industrial designer Dieter Rams, as the last of his 10 principles of good design, stated that good design involves "as little design as possible," concentrating only on the essential aspects. A good game animation system should similarly involve no more design than required, as bloated systems can quickly become unworkable as the project scales to the oft-required hundreds or thousands of animations.

Every unique aspect of character-based gameplay will require a system to play back animations, from the navigation around the world to combat to jumping and climbing to conversation and dialogue and many more. Here, the game animator must aid in creating systems to play back all the varied animation required to bring each element of character control to life, and often the desire to create many animations will come into conflict with the realities of production such as project length and budget.



DOOM opted for full-body damage animations over body parts alone for visual control. (DOOM® Copyright 2016 id Software LLC, a ZeniMax Media company. All Rights Reserved.)

Thankfully, there are many tricks that a team can employ to maximize their animation potential, such as reuse and sharing, layering and combining animations to create multiple combinations, or ingenious blending solutions to increase the fluidity without having to account for absolutely every possible transition outcome. While the simplest solution is to do nothing

more than play animations in sequence, this will rarely produce the best and most fluid visuals, so the smartest approach is to manipulate animations at runtime in the game engine to get the most out of the animations the team has the time to create. Again, we'll cover some of the potential systemic solutions in the following chapter.

Bang for the Buck

Just as we look to share animations, being smart about choices at the design stage should create a workable method of combining animations throughout production. This will in turn prevent unique solutions being required for every new system. For example, a well-thought-out system for opening doors in a game could be expanded to interacting with and opening crates if made efficiently. When building any one system, anticipating uses beyond the current requirements should always be considered.

A good approach to system design will produce the maximum quality of motion for the minimum amount of overhead (work). It must be stressed that every new animation required not only involves the initial creation but later modification over multiple iterations, as well as debugging toward the end of the project. Every stage of development is multiplied by every asset created, so avoiding adding 20 new animations for each object type is not only cost effective but allows more objects to be added to the game. (All that said, sometimes the solution to a system is just to brute-force create lots of animations if your budget allows it.)

Sharing & Standardization

As mentioned earlier, it is important to know when to keep animations generic and when to make unique ones for each example. If the game requires the player character interact with many objects in a game, then it would be wise to standardize the objects' sizes so one animation accommodates all objects of a particular size.

The same goes for world dimensions, where if a character can vault over objects throughout the game, then it makes sense to standardize the height of vaultable objects in the environment so the same animation will work anywhere—not least so the player can better read the level layout and know where the character can and cannot vault.



Gears of War featured high and low cover heights supported by different sets of animations. (Copyright Microsoft. All rights reserved. Used with permission from Microsoft Corporation.)

That said, if your gameplay is primarily about picking up objects or vaulting over things, then it may be worth creating more unique animations to really highlight that area and spend less effort elsewhere. This, again, feeds back into the idea of bang for the buck and knowing what is important to your particular game.

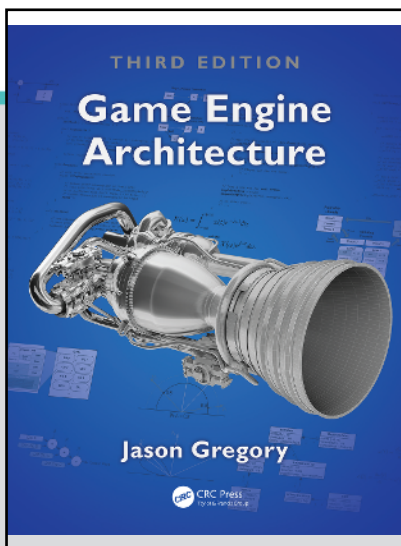
All these decisions must come into play when designing systems for your game, as very few teams can afford unique and bespoke animations for each and every situation. Nevertheless, beautiful game animation can come from even single-person teams that focus on one thing and do it very very well. This is the crux of what good design is, and every aspect of game development benefits from clever and elegant design, regardless of game type.



CHAPTER

4

INTRODUCTION TO GAME ENGINE ARCHITECTURE



This chapter is excerpted from
Game Engine Architecture
by Jason Gregory

© 2018 Taylor & Francis Group. All rights reserved.



[Learn more](#)

Introduction

When I got my first game console in 1979—a way-cool Intellivision system by Mattel—the term “game engine” did not exist. Back then, video and arcade games were considered by most adults to be nothing more than toys, and the software that made them tick was highly specialized to both the game in question and the hardware on which it ran. Today, games are a multi-billion-dollar mainstream industry rivaling Hollywood in size and popularity. And the software that drives these now-ubiquitous three-dimensional worlds—*game engines* like Epic Games’ Unreal Engine 4, Valve’s Source engine and, Crytek’s CRYENGINE® 3, Electronic Arts DICE’s Frostbite™ engine, and the Unity game engine—have become fully featured reusable software development kits that can be licensed and used to build almost any game imaginable.

While game engines vary widely in the details of their architecture and implementation, recognizable coarse-grained patterns have emerged across both publicly licensed game engines and their proprietary in-house counterparts. Virtually all game engines contain a familiar set of core components, including the rendering engine, the collision and physics engine, the animation system, the audio system, the game world object model, the artificial intelligence system and so on. Within each of these components, a relatively small number of semi-standard design alternatives are also beginning to emerge.

There are a great many books that cover individual game engine subsystems, such as three-dimensional graphics, in exhaustive detail. Other books cobble together valuable tips and tricks across a wide variety of game technology areas. However, I have been unable to find a book that provides its reader with a reasonably complete picture of the entire gamut of components that make up a modern game engine. The goal of this book, then, is to take the reader on a guided hands-on tour of the vast and complex landscape of game engine architecture.

In this book you will learn:

- how real industrial-strength production game engines are architected;
- how game development teams are organized and work in the real world;
- which major subsystems and design patterns appear again and again in virtually every game engine;
- the typical requirements for each major subsystem;
- which subsystems are genre- or game-agnostic, and which ones are typically designed explicitly for a specific genre or game; and
- where the engine normally ends and the game begins.

We'll also get a first-hand glimpse into the inner workings of some popular game engines, such as Quake, Unreal and Unity, and some well-known middleware packages, such as the Havok Physics library, the OGRE rendering engine and Rad Game Tools' Granny 3D animation and geometry management toolkit. And we'll explore a number of proprietary game engines that I've had the pleasure to work with, including the engine Naughty Dog developed for its *Uncharted* and *The Last of Us* game series.

Before we get started, we'll review some techniques and tools for large-scale software engineering in a game engine context, including:

- the difference between logical and physical software architecture;
- configuration management, revision control and build systems; and
- some tips and tricks for dealing with one of the common development environments for C and C++, Microsoft Visual Studio.

In this book I assume that you have a solid understanding of C++ (the language of choice among most modern game developers) and that you understand basic software engineering principles. I also assume you have some

exposure to linear algebra, three-dimensional vector and matrix math and trigonometry (although we'll review the core concepts in Chapter 5). Ideally, you should have some prior exposure to the basic concepts of real time and event-driven programming. But never fear—I will review these topics briefly, and I'll also point you in the right direction if you feel you need to hone your skills further before we embark.

1.1 Structure of a Typical Game Team

Before we delve into the structure of a typical game engine, let's first take a brief look at the structure of a typical game development team. Game studios are usually composed of five basic disciplines: engineers, artists, game designers, producers and other management and support staff (marketing, legal, information technology/technical support, administrative, etc.). Each discipline can be divided into various subdisciplines. We'll take a brief look at each below.

1.1.1 Engineers

The engineers design and implement the software that makes the game, and the tools, work. Engineers are often categorized into two basic groups: *runtime* programmers (who work on the engine and the game itself) and *tools* programmers (who work on the offline tools that allow the rest of the development team to work effectively). On both sides of the runtime/tools line, engineers have various specialties. Some engineers focus their careers on a single engine system, such as rendering, artificial intelligence, audio or collision and physics. Some focus on gameplay programming and scripting, while others prefer to work at the systems level and not get too involved in how the game actually plays. Some engineers are generalists—jacks of all trades who can jump around and tackle whatever problems might arise during development.

Senior engineers are sometimes asked to take on a technical leadership role. Lead engineers usually still design and write code, but they also help to manage the team's schedule, make decisions regarding the overall technical direction of the project, and sometimes also directly manage people from a human resources perspective.

Some companies also have one or more technical directors (TD), whose job it is to oversee one or more projects from a high level, ensuring that the teams are aware of potential technical challenges, upcoming industry developments, new technologies and so on. The highest engineering-related position at a game studio is the chief technical officer (CTO), if the studio has one. The

CTO's job is to serve as a sort of technical director for the entire studio, as well as serving a key executive role in the company.

1.1.2 Artists

As we say in the game industry, "Content is king." The artists produce all of the visual and audio content in the game, and the quality of their work can literally make or break a game. Artists come in all sorts of flavors:

- *Concept artists* produce sketches and paintings that provide the team with a vision of what the final game will look like. They start their work early in the concept phase of development, but usually continue to provide visual direction throughout a project's life cycle. It is common for screenshots taken from a shipping game to bear an uncanny resemblance to the concept art.
- *3D modelers* produce the three-dimensional geometry for everything in the virtual game world. This discipline is typically divided into two sub-disciplines: foreground modelers and background modelers. The former create objects, characters, vehicles, weapons and the other objects that populate the game world, while the latter build the world's static background geometry (terrain, buildings, bridges, etc.).
- *Texture artists* create the two-dimensional images known as textures, which are applied to the surfaces of 3D models in order to provide detail and realism.
- *Lighting artists* lay out all of the light sources in the game world, both static and dynamic, and work with color, intensity and light direction to maximize the artfulness and emotional impact of each scene.
- *Animators* imbue the characters and objects in the game with motion. The animators serve quite literally as actors in a game production, just as they do in a CG film production. However, a game animator must have a unique set of skills in order to produce animations that mesh seamlessly with the technological underpinnings of the game engine.
- *Motion capture actors* are often used to provide a rough set of motion data, which are then cleaned up and tweaked by the animators before being integrated into the game.
- *Sound designers* work closely with the engineers in order to produce and mix the sound effects and music in the game.

- *Voice actors* provide the voices of the characters in many games.
- Many games have one or more *composers*, who compose an original score for the game.

As with engineers, senior artists are often called upon to be team leaders. Some game teams have one or more *art directors*—very senior artists who manage the look of the entire game and ensure consistency across the work of all team members.

1.1.3 Game Designers

The game designers' job is to design the interactive portion of the player's experience, typically known as *gameplay*. Different kinds of designers work at different levels of detail. Some (usually senior) game designers work at the macro level, determining the story arc, the overall sequence of chapters or levels, and the high-level goals and objectives of the player. Other designers work on individual levels or geographical areas within the virtual game world, laying out the static background geometry, determining where and when enemies will emerge, placing supplies like weapons and health packs, designing puzzle elements and so on. Still other designers operate at a highly technical level, working closely with gameplay engineers and/or writing code (often in a high-level scripting language). Some game designers are ex-engineers, who decided they wanted to play a more active role in determining how the game will play.

Some game teams employ one or more *writers*. A game writer's job can range from collaborating with the senior game designers to construct the story arc of the entire game, to writing individual lines of dialogue.

As with other disciplines, some senior designers play management roles. Many game teams have a game director, whose job it is to oversee all aspects of a game's design, help manage schedules, and ensure that the work of individual designers is consistent across the entire product. Senior designers also sometimes evolve into producers.

1.1.4 Producers

The role of producer is defined differently by different studios. In some game companies, the producer's job is to manage the schedule and serve as a human resources manager. In other companies, producers serve in a senior game design capacity. Still other studios ask their producers to serve as liaisons between the development team and the business unit of the company (finance, legal, marketing, etc.). Some smaller studios don't have producers at all. For

example, at Naughty Dog, literally everyone in the company, including the two co-presidents, plays a direct role in constructing the game; team management and business duties are shared between the senior members of the studio.

1.1.5 Other Staff

The team of people who directly construct the game is typically supported by a crucial team of support staff. This includes the studio's executive management team, the marketing department (or a team that liaises with an external marketing group), administrative staff and the IT department, whose job is to purchase, install and configure hardware and software for the team and to provide technical support.

1.1.6 Publishers and Studios

The marketing, manufacture and distribution of a game title are usually handled by a *publisher*, not by the game studio itself. A publisher is typically a large corporation, like Electronic Arts, THQ, Vivendi, Sony, Nintendo, etc. Many game studios are not affiliated with a particular publisher. They sell each game that they produce to whichever publisher strikes the best deal with them. Other studios work exclusively with a single publisher, either via a long-term publishing contract or as a fully owned subsidiary of the publishing company. For example, THQ's game studios are independently managed, but they are owned and ultimately controlled by THQ. Electronic Arts takes this relationship one step further, by directly managing its studios. *First-party developers* are game studios owned directly by the console manufacturers (Sony, Nintendo and Microsoft). For example, Naughty Dog is a first-party Sony developer. These studios produce games exclusively for the gaming hardware manufactured by their parent company.

1.2 What Is a Game?

We probably all have a pretty good intuitive notion of what a game is. The general term "game" encompasses board games like chess and *Monopoly*, card games like poker and blackjack, casino games like roulette and slot machines, military war games, computer games, various kinds of play among children, and the list goes on. In academia we sometimes speak of *game theory*, in which multiple agents select strategies and tactics in order to maximize their gains within the framework of a well-defined set of game rules. When used in the context of console or computer-based entertainment, the word "game"

usually conjures images of a three-dimensional virtual world featuring a humanoid, animal or vehicle as the main character under player control. (Or for the old geezers among us, perhaps it brings to mind images of two-dimensional classics like *Pong*, *Pac-Man*, or *Donkey Kong*.) In his excellent book, *A Theory of Fun for Game Design*, Raph Koster defines a game to be an interactive experience that provides the player with an increasingly challenging sequence of patterns which he or she learns and eventually masters [30]. Koster's assertion is that the activities of learning and mastering are at the heart of what we call "fun," just as a joke becomes funny at the moment we "get it" by recognizing the pattern.

For the purposes of this book, we'll focus on the subset of games that comprise two- and three-dimensional virtual worlds with a small number of players (between one and 16 or thereabouts). Much of what we'll learn can also be applied to HTML5/JavaScript games on the Internet, pure puzzle games like *Tetris*, or massively multiplayer online games (MMOG). But our primary focus will be on game engines capable of producing first-person shooters, third-person action/platform games, racing games, fighting games and the like.

1.2.1 Video Games as Soft Real-Time Simulations

Most two- and three-dimensional video games are examples of what computer scientists would call *soft real-time interactive agent-based computer simulations*. Let's break this phrase down in order to better understand what it means.

In most video games, some subset of the real world—or an imaginary world—is *modeled* mathematically so that it can be manipulated by a computer. The model is an approximation to and a simplification of reality (even if it's an *imaginary* reality), because it is clearly impractical to include every detail down to the level of atoms or quarks. Hence, the mathematical model is a *simulation* of the real or imagined game world. Approximation and simplification are two of the game developer's most powerful tools. When used skillfully, even a greatly simplified model can sometimes be almost indistinguishable from reality—and a lot more fun.

An *agent-based* simulation is one in which a number of distinct entities known as "agents" interact. This fits the description of most three-dimensional computer games very well, where the agents are vehicles, characters, fireballs, power dots and so on. Given the agent-based nature of most games, it should come as no surprise that most games nowadays are implemented in an object-oriented, or at least loosely object-based, programming language.

All interactive video games are *temporal simulations*, meaning that the virtual game world model is *dynamic*—the state of the game world changes over time as the game’s events and story unfold. A video game must also respond to unpredictable inputs from its human player(s)—thus *interactive temporal simulations*. Finally, most video games present their stories and respond to player input in real time, making them *interactive real-time simulations*. One notable exception is in the category of turn-based games like computerized chess or turn-based strategy games. But even these types of games usually provide the user with some form of real-time graphical user interface. So for the purposes of this book, we’ll assume that all video games have at least *some* real-time constraints.

At the core of every real-time system is the concept of a *deadline*. An obvious example in video games is the requirement that the screen be updated at least 24 times per second in order to provide the illusion of motion. (Most games render the screen at 30 or 60 frames per second because these are multiples of an NTSC monitor’s refresh rate.) Of course, there are many other kinds of deadlines in video games as well. A physics simulation may need to be updated 120 times per second in order to remain stable. A character’s artificial intelligence system may need to “think” at least once every second to prevent the appearance of stupidity. The audio library may need to be called at least once every 1/60 second in order to keep the audio buffers filled and prevent audible glitches.

A “soft” real-time system is one in which missed deadlines are not catastrophic. Hence, all video games are *soft real-time systems*—if the frame rate dies, the human player generally doesn’t! Contrast this with a *hard real-time system*, in which a missed deadline could mean severe injury to or even the death of a human operator. The avionics system in a helicopter or the control-rod system in a nuclear power plant are examples of hard real-time systems.

Mathematical models can be *analytic* or *numerical*. For example, the analytic (closed-form) mathematical model of a rigid body falling under the influence of constant acceleration due to gravity is typically written as follows:

$$y(t) = \frac{1}{2}gt^2 + v_0t + y_0. \quad (1.1)$$

An analytic model can be evaluated for any value of its independent variables, such as the time t in the above equation, given only the initial conditions v_0 and y_0 and the constant g . Such models are very convenient when they can be found. However, many problems in mathematics have no closed-form solution. And in video games, where the user’s input is unpredictable, we cannot hope to model the entire game analytically.

A numerical model of the same rigid body under gravity can be expressed

as follows:

$$y(t + \Delta t) = F(y(t), \dot{y}(t), \ddot{y}(t), \dots). \quad (1.2)$$

That is, the height of the rigid body at some future time $(t + \Delta t)$ can be found as a function of the height and its first, second, and possibly higher-order time derivatives at the current time t . Numerical simulations are typically implemented by running calculations repeatedly, in order to determine the state of the system at each discrete time step. Games work in the same way. A main “game loop” runs repeatedly, and during each iteration of the loop, various game systems such as artificial intelligence, game logic, physics simulations and so on are given a chance to calculate or update their state for the next discrete time step. The results are then “rendered” by displaying graphics, emitting sound and possibly producing other outputs such as force-feedback on the joystick.

1.3 What Is a Game Engine?

The term “game engine” arose in the mid-1990s in reference to first-person shooter (FPS) games like the insanely popular *Doom* by id Software. *Doom* was architected with a reasonably well-defined separation between its core software components (such as the three-dimensional graphics rendering system, the collision detection system or the audio system) and the art assets, game worlds and rules of play that comprised the player’s gaming experience. The value of this separation became evident as developers began licensing games and retooling them into new products by creating new art, world layouts, weapons, characters, vehicles and game rules with only minimal changes to the “engine” software. This marked the birth of the “mod community”—a group of individual gamers and small independent studios that built new games by modifying existing games, using free toolkits provided by the original developers.

Towards the end of the 1990s, some games like *Quake III Arena* and *Unreal* were designed with reuse and “modding” in mind. Engines were made highly customizable via scripting languages like id’s *Quake C*, and engine licensing began to be a viable secondary revenue stream for the developers who created them. Today, game developers can license a game engine and reuse significant portions of its key software components in order to build games. While this practice still involves considerable investment in custom software engineering, it can be much more economical than developing all of the core engine components in-house.

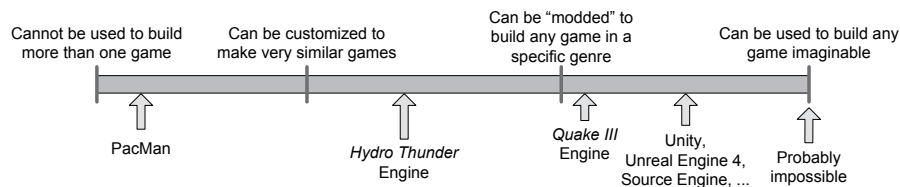


Figure 1.1. Game engine reusability gamut.

The line between a game and its engine is often blurry. Some engines make a reasonably clear distinction, while others make almost no attempt to separate the two. In one game, the rendering code might “know” specifically how to draw an orc. In another game, the rendering engine might provide general-purpose material and shading facilities, and “orc-ness” might be defined entirely in data. No studio makes a perfectly clear separation between the game and the engine, which is understandable considering that the definitions of these two components often shift as the game’s design solidifies.

Arguably a *data-driven architecture* is what differentiates a game engine from a piece of software that is a game but not an engine. When a game contains hard-coded logic or game rules, or employs special-case code to render specific types of game objects, it becomes difficult or impossible to reuse that software to make a different game. We should probably reserve the term “game engine” for software that is extensible and can be used as the foundation for many different games without major modification.

Clearly this is not a black-and-white distinction. We can think of a gamut of reusability onto which every engine falls. Figure 1.1 takes a stab at the locations of some well-known games/engines along this gamut.

One would think that a game engine could be something akin to Apple QuickTime or Microsoft Windows Media Player—a general-purpose piece of software capable of playing virtually *any* game content imaginable. However, this ideal has not yet been achieved (and may never be). Most game engines are carefully crafted and fine-tuned to run a particular game on a particular hardware platform. And even the most general-purpose multiplatform engines are really only suitable for building games in one particular genre, such as first-person shooters or racing games. It’s safe to say that the more general-purpose a game engine or middleware component is, the less optimal it is for running a particular game on a particular platform.

This phenomenon occurs because designing any efficient piece of software invariably entails making trade-offs, and those trade-offs are based on assumptions about how the software will be used and/or about the target

hardware on which it will run. For example, a rendering engine that was designed to handle intimate indoor environments probably won't be very good at rendering vast outdoor environments. The indoor engine might use a binary space partitioning (BSP) tree or portal system to ensure that no geometry is drawn that is being occluded by walls or objects that are closer to the camera. The outdoor engine, on the other hand, might use a less-exact occlusion mechanism, or none at all, but it probably makes aggressive use of level-of-detail (LOD) techniques to ensure that distant objects are rendered with a minimum number of triangles, while using high-resolution triangle meshes for geometry that is close to the camera.

The advent of ever-faster computer hardware and specialized graphics cards, along with ever-more-efficient rendering algorithms and data structures, is beginning to soften the differences between the graphics engines of different genres. It is now possible to use a first-person shooter engine to build a strategy game, for example. However, the trade-off between generality and optimality still exists. A game can always be made more impressive by fine-tuning the engine to the specific requirements and constraints of a particular game and/or hardware platform.

1.4 Engine Differences across Genres

Game engines are typically somewhat genre specific. An engine designed for a two-person fighting game in a boxing ring will be very different from a massively multiplayer online game (MMOG) engine or a first-person shooter (FPS) engine or a real-time strategy (RTS) engine. However, there is also a great deal of overlap—all 3D games, regardless of genre, require some form of low-level user input from the joystick, keyboard and/or mouse, some form of 3D mesh rendering, some form of heads-up display (HUD) including text rendering in a variety of fonts, a powerful audio system, and the list goes on. So while the Unreal Engine, for example, was designed for first-person shooter games, it has been used successfully to construct games in a number of other genres as well, including the wildly popular third-person shooter franchise *Gears of War* by Epic Games, the hit action-adventure games in the *Batman: Arkham* series by Rocksteady Studios, the well-known fighting game *Tekken 7* by Bandai Namco Studios, and the first three role-playing third-person shooter games in the *Mass Effect* series by BioWare.

Let's take a look at some of the most common game genres and explore some examples of the technology requirements particular to each.



Figure 1.2. *Overwatch* by Blizzard Entertainment (Xbox One, PlayStation 4, Windows). (See Color Plate I.)

1.4.1 First-Person Shooters (FPS)

The first-person shooter (FPS) genre is typified by games like *Quake*, *Unreal Tournament*, *Half-Life*, *Battlefield*, *Destiny*, *Titanfall* and *Overwatch* (see Figure 1.2). These games have historically involved relatively slow on-foot roaming of a potentially large but primarily corridor-based world. However, modern first-person shooters can take place in a wide variety of virtual environments including vast open outdoor areas and confined indoor areas. Modern FPS traversal mechanics can include on-foot locomotion, rail-confined or free-roaming ground vehicles, hovercraft, boats and aircraft. For an overview of this genre, see http://en.wikipedia.org/wiki/First-person_shooter.

First-person games are typically some of the most technologically challenging to build, probably rivaled in complexity only by third-person shooters, action-platformer games, and massively multiplayer games. This is because first-person shooters aim to provide their players with the illusion of being immersed in a detailed, hyperrealistic world. It is not surprising that many of the game industry's big technological innovations arose out of the games in this genre.

First-person shooters typically focus on technologies such as:

- efficient rendering of large 3D virtual worlds;

- a responsive camera control/aiming mechanic;
- high-fidelity animations of the player’s virtual arms and weapons;
- a wide range of powerful handheld weaponry;
- a forgiving player character motion and collision model, which often gives these games a “floaty” feel;
- high-fidelity animations and artificial intelligence for the non-player characters (NPCs)—the player’s enemies and allies; and
- small-scale online multiplayer capabilities (typically supporting between 10 and 100 simultaneous players), and the ubiquitous “death match” gameplay mode.

The rendering technology employed by first-person shooters is almost always highly optimized and carefully tuned to the particular type of environment being rendered. For example, indoor “dungeon crawl” games often employ binary space partitioning trees or portal-based rendering systems. Outdoor FPS games use other kinds of rendering optimizations such as occlusion culling, or an offline sectorization of the game world with manual or automated specification of which target sectors are visible from each source sector.

Of course, immersing a player in a hyperrealistic game world requires much more than just optimized high-quality graphics technology. The character animations, audio and music, rigid body physics, in-game cinematics and myriad other technologies must all be cutting-edge in a first-person shooter. So this genre has some of the most stringent and broad technology requirements in the industry.

1.4.2 Platformers and Other Third-Person Games

“Platformer” is the term applied to third-person character-based action games where jumping from platform to platform is the primary gameplay mechanic. Typical games from the 2D era include *Space Panic*, *Donkey Kong*, *Pitfall!* and *Super Mario Brothers*. The 3D era includes platformers like *Super Mario 64*, *Crash Bandicoot*, *Rayman 2*, *Sonic the Hedgehog*, the *Jak and Daxter* series (Figure 1.3), the *Ratchet & Clank* series and *Super Mario Galaxy*. See <http://en.wikipedia.org/wiki/Platformer> for an in-depth discussion of this genre.

In terms of their technological requirements, platformers can usually be lumped together with third-person shooters and third-person action/adventure games like *Just Cause 2*, *Gears of War 4* (Figure 1.4), the *Uncharted* series, the *Resident Evil* series, the *The Last of Us* series, *Red Dead Redemption 2*, and the list goes on.

Third-person character-based games have a lot in common with first-person shooters, but a great deal more emphasis is placed on the main character's abilities and locomotion modes. In addition, high-fidelity full-body character animations are required for the player's avatar, as opposed to the somewhat less-taxing animation requirements of the "floating arms" in a typical FPS game. It's important to note here that almost all first-person shooters have an online multiplayer component, so a full-body player avatar must be rendered in addition to the first-person arms. However, the fidelity of these FPS player avatars is usually not comparable to the fidelity of the non-player characters in these same games; nor can it be compared to the fidelity of the player avatar in a third-person game.

In a platformer, the main character is often cartoon-like and not particularly realistic or high-resolution. However, third-person shooters often feature a highly realistic humanoid player character. In both cases, the player character typically has a very rich set of actions and animations.

Some of the technologies specifically focused on by games in this genre include:



Figure I.3. *Jak II* by Naughty Dog (Jak, Dexter, Jak and Dexter, and Jak II © 2003, 2013™ SIE. Created and developed by Naughty Dog, PlayStation 2.) (See Color Plate II.)



Figure 1.4. *Gears of War 4* by The Coalition (Xbox One). (See Color Plate III.)

- moving platforms, ladders, ropes, trellises and other interesting locomotion modes;
- puzzle-like environmental elements;
- a third-person “follow camera” which stays focused on the player character and whose rotation is typically controlled by the human player via the right joystick (on a console) or the mouse (on a PC—note that while there are a number of popular third-person shooters on a PC, the platformer genre exists almost exclusively on consoles); and
- a complex camera collision system for ensuring that the view point never “clips” through background geometry or dynamic foreground objects.

1.4.3 Fighting Games

Fighting games are typically two-player games involving humanoid characters pummeling each other in a ring of some sort. The genre is typified by games like *Soul Calibur* and *Tekken 3* (see Figure 1.5). The Wikipedia page http://en.wikipedia.org/wiki/Fighting_game provides an overview of this genre.

Traditionally games in the fighting genre have focused their technology efforts on:

- a rich set of fighting animations;
- accurate hit detection;
- a user input system capable of detecting complex button and joystick combinations; and
- crowds, but otherwise relatively static backgrounds.

Since the 3D world in these games is small and the camera is centered on the action at all times, historically these games have had little or no need for world subdivision or occlusion culling. They would likewise not be expected to employ advanced three-dimensional audio propagation models, for example.

Modern fighting games like EA's *Fight Night Round 4* and NetherRealm Studios' *Injustice 2* (Figure 1.6) have upped the technological ante with features like:

- high-definition character graphics;
- realistic skin shaders with subsurface scattering and sweat effects;
- photo-realistic lighting and particle effects;
- high-fidelity character animations; and



Figure 1.5. *Tekken 3* by Namco (PlayStation). (See Color Plate IV.)

- physics-based cloth and hair simulations for the characters.

It's important to note that some fighting games like Ninja Theory's *Heavenly Sword* and *For Honor* by Ubisoft Montreal take place in a large-scale virtual world, not a confined arena. In fact, many people consider this to be a separate genre, sometimes called a *brawler*. This kind of fighting game can have technical requirements more akin to those of a third-person shooter or a strategy game.

1.4.4 Racing Games

The racing genre encompasses all games whose primary task is driving a car or other vehicle on some kind of track. The genre has many subcategories. Simulation-focused racing games ("sims") aim to provide a driving experience that is as realistic as possible (e.g., *Gran Turismo*). Arcade racers favor over-the-top fun over realism (e.g., *San Francisco Rush*, *Cruis'n USA*, *Hydro Thunder*). One subgenre explores the subculture of street racing with tricked out consumer vehicles (e.g., *Need for Speed*, *Juiced*). Kart racing is a subcategory in which popular characters from platformer games or cartoon characters from TV are re-cast as the drivers of whacky vehicles (e.g., *Mario Kart*, *Jak X*, *Freaky Flyers*). Racing games need not always involve time-based competition. Some kart racing games, for example, offer



Figure 1.6. *Injustice 2* by NetherRealm Studios (PlayStation 4, Xbox One, Android, iOS, Microsoft Windows). (See Color Plate V.)



Figure 1.7. *Gran Turismo Sport* by Polyphony Digital (PlayStation 4). (See Color Plate VI.)

modes in which players shoot at one another, collect loot or engage in a variety of other timed and untimed tasks. For a discussion of this genre, see http://en.wikipedia.org/wiki/Racing_game.

A racing game is often very linear, much like older FPS games. However, travel speed is generally much faster than in an FPS. Therefore, more focus is placed on very long corridor-based tracks, or looped tracks, sometimes with various alternate routes and secret short-cuts. Racing games usually focus all their graphic detail on the vehicles, track and immediate surroundings. As an example of this, Figure 1.7 shows a screenshot from the latest installment in the well-known *Gran Turismo* racing game series, *Gran Turismo Sport*, developed by Polyphony Digital and published by Sony Interactive Entertainment. However, kart racers also devote significant rendering and animation bandwidth to the characters driving the vehicles.

Some of the technological properties of a typical racing game include the following techniques:

- Various “tricks” are used when rendering distant background elements, such as employing two-dimensional cards for trees, hills and mountains.
- The track is often broken down into relatively simple two-dimensional regions called “sectors.” These data structures are used to optimize rendering and visibility determination, to aid in artificial intelligence and path finding for non-human-controlled vehicles, and to solve many other technical problems.
- The camera typically follows behind the vehicle for a third-person per-



Figure 1.8. *Age of Empires* by Ensemble Studios (Windows). (See Color Plate VII.)

spective, or is sometimes situated inside the cockpit first-person style.

- When the track involves tunnels and other “tight” spaces, a good deal of effort is often put into ensuring that the camera does not collide with background geometry.

1.4.5 Strategy Games

The modern strategy game genre was arguably defined by *Dune II: The Building of a Dynasty* (1992). Other games in this genre include *Warcraft*, *Command & Conquer*, *Age of Empires* and *Starcraft*. In this genre, the player deploys the battle units in his or her arsenal strategically across a large playing field in an attempt to overwhelm his or her opponent. The game world is typically displayed at an oblique top-down viewing angle. A distinction is often made between turn-based strategy games and real-time strategy (RTS). For a discussion of this genre, see https://en.wikipedia.org/wiki/Strategy_video_game.

The strategy game player is usually prevented from significantly changing the viewing angle in order to see across large distances. This restriction permits developers to employ various optimizations in the rendering engine of a strategy game.



Figure 1.9. *Total War: Warhammer 2* by Creative Assembly (Windows). (See Color Plate VIII.)

Older games in the genre employed a grid-based (cell-based) world construction, and an orthographic projection was used to greatly simplify the renderer. For example, Figure 1.8 shows a screenshot from the classic strategy game *Age of Empires*.

Modern strategy games sometimes use perspective projection and a true 3D world, but they may still employ a grid layout system to ensure that units and background elements, such as buildings, align with one another properly. A popular example, *Total War: Warhammer 2*, is shown in Figure 1.9.

Some other common practices in strategy games include the following techniques:

- Each unit is relatively low-res, so that the game can support large numbers of them on-screen at once.
- Height-field terrain is usually the canvas upon which the game is designed and played.
- The player is often allowed to build new structures on the terrain in addition to deploying his or her forces.
- User interaction is typically via single-click and area-based selection of units, plus menus or toolbars containing commands, equipment, unit types, building types, etc.



Figure 1.10. *World of Warcraft* by Blizzard Entertainment (Windows, MacOS). (See Color Plate IX.)

1.4.6 Massively Multiplayer Online Games (MMOG)

The massively multiplayer online game (MMOG or just MMO) genre is typified by games like *Guild Wars 2* (AreaNet/NCsoft), *EverQuest* (989 Studios/ SOE), *World of Warcraft* (Blizzard) and *Star Wars Galaxies* (SOE/Lucas Arts), to name a few. An MMO is defined as any game that supports huge numbers of simultaneous players (from thousands to hundreds of thousands), usually all playing in one very large, *persistent* virtual world (i.e., a world whose internal state persists for very long periods of time, far beyond that of any one player's gameplay session). Otherwise, the gameplay experience of an MMO is often similar to that of their small-scale multiplayer counterparts. Subcategories of this genre include MMO role-playing games (MMORPG), MMO real-time strategy games (MMORTS) and MMO first-person shooters (MMOFPS). For a discussion of this genre, see <http://en.wikipedia.org/wiki/MMOG>. Figure 1.10 shows a screenshot from the hugely popular MMORPG *World of Warcraft*.

At the heart of all MMOGs is a very powerful battery of servers. These servers maintain the authoritative state of the game world, manage users signing in and out of the game, provide inter-user chat or voice-over-IP (VoIP) services and more. Almost all MMOGs require users to pay some kind of regular

subscription fee in order to play, and they may offer micro-transactions within the game world or out-of-game as well. Hence, perhaps the most important role of the central server is to handle the billing and micro-transactions which serve as the game developer's primary source of revenue.

Graphics fidelity in an MMO is almost always lower than its non-massively multiplayer counterparts, as a result of the huge world sizes and extremely large numbers of users supported by these kinds of games.

Figure 1.11 shows a screen from Bungie's latest FPS game, *Destiny 2*. This game has been called an MMOFPS because it incorporates some aspects of the MMO genre. However, Bungie prefers to call it a "shared world" game because unlike a traditional MMO, in which a player can see and interact with literally any other player on a particular server, *Destiny* provides "on-the-fly match-making." This permits the player to interact only with the other players with whom they have been matched by the server; this matchmaking system has been significantly improved for *Destiny 2*. Also unlike a traditional MMO, the graphics fidelity in *Destiny 2* is on par with first- and third-person shooters.

We should note here that the game *Player Unknown's Battlegrounds* (PUBG) has recently popularized a subgenre known as *battle royale*. This type of game blurs the line between regular multiplayer shooters and massively multiplayer online games, because they typically pit on the order of 100 players against each other in an online world, employing a survival-based "last man standing" gameplay style.



Figure 1.11. *Destiny 2* by Bungie. © 2018 Bungie Inc. (Xbox One, PlayStation 4, PC) (See Color Plate X.)

1.4.7 Player-Authored Content

As social media takes off, games are becoming more and more collaborative in nature. A recent trend in game design is toward *player-authored content*. For example, Media Molecule's *LittleBigPlanet*,TM *LittleBigPlanet*TM 2 (Figure 1.12) and *LittleBigPlanet*TM 3: *The Journey Home* are technically *puzzle platformers*, but their most notable and unique feature is that they encourage players to create, publish and share their own game worlds. Media Molecule's latest installment in this engaging genre is *Dreams* for the PlayStation 4 (Figure 1.13).

Perhaps the most popular game today in the player-created content genre is *Minecraft* (Figure 1.14). The brilliance of this game lies in its simplicity: *Minecraft* game worlds are constructed from simple cubic voxel-like elements mapped with low-resolution textures to mimic various materials. Blocks can be solid, or they can contain items such as torches, anvils, signs, fences and panes of glass. The game world is populated with one or more player characters, animals such as chickens and pigs, and various “mobs”—good guys like villagers and bad guys like zombies and the ubiquitous *creepers* who sneak up on unsuspecting players and explode (only scant moments after warning the player with the “hiss” of a burning fuse).

Players can create a randomized world in *Minecraft* and then dig into the generated terrain to create tunnels and caverns. They can also construct their own structures, ranging from simple terrain and foliage to vast and complex



Figure 1.12. *LittleBigPlanet*TM 2 by Media Molecule. © 2014 Sony Interactive Entertainment (PlayStation 3). (See Color Plate XI.)



Figure I.13. *Dreams* by Media Molecule, © 2017 Sony Computer Computer Europe (PlayStation 4). (See Color Plate XII.)

buildings and machinery. Perhaps the biggest stroke of genius in *Minecraft* is *redstone*. This material serves as “wiring,” allowing players to lay down circuitry that controls pistons, hoppers, mine carts and other dynamic elements in the game. As a result, players can create virtually anything they can imagine, and then share their worlds with their friends by hosting a server and inviting them to play online.



Figure I.14. *Minecraft* by Markus “Notch” Persson / Mojang AB (Windows, MacOS, Xbox 360, PlayStation 3, PlayStation Vita, iOS). (See Color Plate XIII.)

1.4.8 Virtual, Augmented and Mixed Reality

Virtual, augmented and mixed reality are exciting new technologies that aim to immerse the viewer in a 3D world that is either entirely generated by a computer, or is augmented by computer-generated imagery. These technologies have many applications outside the game industry, but they have also become viable platforms for a wide range of gaming content.

1.4.8.1 Virtual Reality

Virtual reality (VR) can be defined as an immersive multimedia or computer-simulated reality that simulates the user's presence in an environment that is either a place in the real world or in an imaginary world. Computer-generated VR (CG VR) is a subset of this technology in which the virtual world is exclusively generated via computer graphics. The user views this virtual environment by donning a headset such as HTC Vive, Oculus Rift, Sony PlayStation VR, Samsung Gear VR or Google Daydream View. The headset displays the content directly in front of the user's eyes; the system also tracks the movement of the headset in the real world, so that the virtual camera's movements can be perfectly matched to those of the person wearing the headset. The user typically holds devices in his or her hands which allow the system to track the movements of each hand. This allows the user to interact in the virtual world: Objects can be pushed, picked up or thrown, for example.

1.4.8.2 Augmented and Mixed Reality

The terms *augmented reality* (AR) and *mixed reality* (MR) are often confused or used interchangeably. Both technologies present the user with a view of the real world, but with computer graphics used to enhance the experience. In both technologies, a viewing device like a smart phone, tablet or tech-enhanced pair of glasses displays a real-time or static view of a real-world scene, and computer graphics are overlaid on top of this image. In real-time AR and MR systems, accelerometers in the viewing device permit the virtual camera's movements to track the movements of the device, producing the illusion that the device is simply a window through which we are viewing the actual world, and hence giving the overlaid computer graphics a strong sense of realism.

Some people make a distinction between these two technologies by using the term "augmented reality" to describe technologies in which computer graphics are overlaid on a live, direct or indirect view of the real world, but are not anchored to it. The term "mixed reality," on the other hand, is more often

applied to the use of computer graphics to render imaginary objects which are anchored to the real world and appear to exist within it. However, this distinction is by no means universally accepted.

Here are a few examples of AR technology in action:

- The U.S. Army provides its soldiers with improved tactical awareness using a system dubbed “tactical augmented reality” (TAR)—it overlays a video-game-like heads-up display (HUD) complete with a mini-map and object markers onto the soldier’s view of the real world (<https://youtu.be/x8p19j8C6VI>).
- In 2015, Disney demonstrated some cool AR technology that renders a 3D cartoon character on top of a sheet of real-world paper on which a 2D version of the character is colored with a crayon (<https://youtu.be/SWzurBQ81CM>).
- PepsiCo also pranked commuting Londoners with an AR-enabled bus stop. People sitting in the bus stop enclosure were treated to AR images of a prowling tiger, a meteor crashing, and an alien tentacle grabbing unwitting passers by off the street (<https://youtu.be/Go9rf9GmYpM>).

And here are a few examples of MR:

- Starting with Android 8.1, the camera app on the Pixel 1 and Pixel 2 supports *AR Stickers*, a fun feature that allows users to place animated 3D objects and characters into videos and photos.
- Microsoft’s HoloLens is another example of mixed reality. It overlays world-anchored graphics onto a live video image, and can be used for a wide range of applications including education and training, engineering, health care, and entertainment.

1.4.8.3 VR/AR/MR Games

The game industry is currently experimenting with VR and AR/MR technologies, and is trying to find its footing within these new media. Some traditional 3D games have been “ported” to VR, yielding very interesting, if not particularly innovative, experiences. But perhaps more exciting, entirely new game genres are starting to emerge, offering gameplay experiences that could not be achieved without VR or AR/MR.

For example, *Job Simulator* by Owlchemy Labs plunges the user into a virtual job museum run by robots, and asks them to perform tongue-in-cheek approximations of various real-world jobs, making use of game mechanics

that simply wouldn't work on a non-VR platform. Owlchemy's next installment, *Vacation Simulator*, applies the same whimsical sense of humour and art style to a world in which the robots of Job Simulator invite the player to relax and perform various tasks. Figure 1.15 shows a screenshot from another innovative (and somewhat disturbing!) game for HTC Vive called *Accounting*, from the creators of "Rick & Morty" and *The Stanley Parable*.

1.4.8.4 VR Game Engines

VR game engines are technologically similar in many respects to first-person shooter engines, and in fact many FPS-capable engines such as Unity and Unreal Engine support VR "out of the box." However, VR games differ from FPS games in a number of significant ways:

- *Stereoscopic rendering.* A VR game needs to render the scene twice, once for each eye. This doubles the number of graphics primitives that must be rendered, although other aspects of the graphics pipeline such as visibility culling can be performed only once per frame, since the eyes are reasonably close together. As such, a VR game isn't quite as expensive to render as the same game would be to render in split-screen multiplayer mode, but the principle of rendering each frame twice from two (slightly) different virtual cameras is the same.
- *Very high frame rate.* Studies have shown that VR running at below 90



Figure 1.15. *Accounting* by Squanchtendo and Crows Crows Crows (HTC Vive). (See Color Plate XIV.)

frames per second is likely to induce disorientation, nausea, and other negative user effects. This means that not only do VR systems need to render the scene twice per frame, they need to do so at 90+ FPS. This is why VR games and applications are generally required to run on high-powered CPU and GPU hardware.

- *Navigation issues.* In an FPS game, the player can simply walk around the game world with the joypad or the WASD keys. In a VR game, a small amount of movement can be realized by the user physically walking around in the real world, but the safe physical play area is typically quite small (the size of a small bathroom or closet). Travelling by “flying” tends to induce nausea as well, so most games opt for a point-and-click teleportation mechanism to move the virtual player/camera across larger distances. Various real-world devices have also been conceived that allow a VR user to “walk” in place with their feet in order to move around in a VR world.

Of course, VR makes up for these limitations somewhat by enabling new user interaction paradigms that aren’t possible in traditional video games. For example,

- users can reach in the real world to touch, pick up and throw objects in the virtual world;
- a player can dodge an attack in the virtual world by dodging physically in the real world;
- new user interface opportunities are possible, such as having floating menus attached to one’s virtual hands, or seeing a game’s credits written on a whiteboard in the virtual world;
- a player can even pick up a pair of virtual VR goggles and place them onto his or her head, thereby transporting them into a “nested” VR world—an effect that might best be called “VR-ception.”

1.4.8.5 Location-Based Entertainment

Games like *Pokémon Go* neither overlay graphics onto an image of the real world, nor do they generate a completely immersive virtual world. However, the user’s view of the computer-generated world of *Pokémon Go* does react to movements of the user’s phone or tablet, much like a 360-degree video. And the game is aware of your actual location in the real world, prompting you to go searching for Pokémon in nearby parks, malls and restaurants. This kind of game can’t really be called AR/MR, but neither does it fall into the VR category. Such a game might be better described as a form of *location-based*

entertainment, although some people do use the AR moniker for these kinds of games.

1.4.9 Other Genres

There are of course many other game genres which we won't cover in depth here. Some examples include:

- sports, with subgenres for each major sport (football, baseball, soccer, golf, etc.);
- role-playing games (RPG);
- God games, like *Populous* and *Black & White*;
- environmental/social simulation games, like *SimCity* or *The Sims*;
- puzzle games like *Tetris*;
- conversions of non-electronic games, like chess, card games, go, etc.;
- web-based games, such as those offered at Electronic Arts' Pogo site;

and the list goes on.

We have seen that each game genre has its own particular technological requirements. This explains why game engines have traditionally differed quite a bit from genre to genre. However, there is also a great deal of technological overlap between genres, especially within the context of a single hardware platform. With the advent of more and more powerful hardware, differences between genres that arose because of optimization concerns are beginning to evaporate. It is therefore becoming increasingly possible to reuse the same engine technology across disparate genres, and even across disparate hardware platforms.

1.5 Game Engine Survey

1.5.1 The Quake Family of Engines

The first 3D first-person shooter (FPS) game is generally accepted to be *Castle Wolfenstein 3D* (1992). Written by id Software of Texas for the PC platform, this game led the game industry in a new and exciting direction. id Software went on to create *Doom*, *Quake*, *Quake II* and *Quake III*. All of these engines are very similar in architecture, and I will refer to them as the Quake family of engines. Quake technology has been used to create many other games and even other engines. For example, the lineage of *Medal of Honor* for the PC platform goes something like this:

- *Quake III* (id Software);
- *Sin* (Ritual);
- *F.A.K.K. 2* (Ritual);
- *Medal of Honor: Allied Assault* (2015 & Dreamworks Interactive); and
- *Medal of Honor: Pacific Assault* (Electronic Arts, Los Angeles).

Many other games based on Quake technology follow equally circuitous paths through many different games and studios. In fact, Valve's Source engine (used to create the *Half-Life* games) also has distant roots in Quake technology.

The *Quake* and *Quake II* source code is freely available, and the original Quake engines are reasonably well architected and "clean" (although they are of course a bit outdated and written entirely in C). These code bases serve as great examples of how industrial-strength game engines are built. The full source code to *Quake* and *Quake II* is available at <https://github.com/id-Software/Quake-2>.

If you own the Quake and/or Quake II games, you can actually build the code using Microsoft Visual Studio and run the game under the debugger using the real game assets from the disk. This can be incredibly instructive. You can set breakpoints, run the game and then analyze how the engine actually works by stepping through the code. I highly recommend downloading one or both of these engines and analyzing the source code in this manner.

1.5.2 Unreal Engine

Epic Games, Inc. burst onto the FPS scene in 1998 with its legendary game *Unreal*. Since then, the Unreal Engine has become a major competitor to Quake technology in the FPS space. Unreal Engine 2 (UE2) is the basis for *Unreal Tournament 2004* (UT2004) and has been used for countless "mods," university projects and commercial games. Unreal Engine 4 (UE4) is the latest evolutionary step, boasting some of the best tools and richest engine feature sets in the industry, including a convenient and powerful graphical user interface for creating shaders and a graphical user interface for game logic programming called *Blueprints* (previously known as Kismet).

The Unreal Engine has become known for its extensive feature set and cohesive, easy-to-use tools. The Unreal Engine is not perfect, and most developers modify it in various ways to run their game optimally on a particular hardware platform. However, Unreal is an incredibly powerful prototyping tool and commercial game development platform, and it can be used to build virtually any 3D first-person or third-person game (not to mention games in

other genres as well). Many exciting games in all sorts of genres have been developed with UE4, including *Rime* by Tequila Works, *Genesis: Alpha One* by Radiation Blue, *A Way Out* by Hazelight Studios, and *Crackdown 3* by Microsoft Studios.

The Unreal Developer Network (UDN) provides a rich set of documentation and other information about all released versions of the Unreal Engine (see <http://udn.epicgames.com/Main/WebHome.html>). Some documentation is freely available. However, access to the full documentation for the latest version of the Unreal Engine is generally restricted to licensees of the engine. There are plenty of other useful websites and wikis that cover the Unreal Engine. One popular one is <http://www.beyondunreal.com>.

Thankfully, Epic now offers full access to Unreal Engine 4, source code and all, for a low monthly subscription fee plus a cut of your game's profits if it ships. This makes UE4 a viable choice for small independent game studios.

1.5.3 The Half-Life Source Engine

Source is the game engine that drives the well-known *Half-Life 2* and its sequels *HL2: Episode One* and *HL2: Episode Two*, *Team Fortress 2* and *Portal* (shipped together under the title *The Orange Box*). Source is a high-quality engine, rivaling Unreal Engine 4 in terms of graphics capabilities and tool set.

1.5.4 DICE's Frostbite

The Frostbite engine grew out of DICE's efforts to create a game engine for *Battlefield Bad Company* in 2006. Since then, the Frostbite engine has become the most widely adopted engine within Electronic Arts (EA); it is used by many of EA's key franchises including *Mass Effect*, *Battlefield*, *Need for Speed*, *Dragon Age*, and *Star Wars Battlefront II*. Frostbite boasts a powerful unified asset creation tool called FrostEd, a powerful tools pipeline known as Backend Services, and a powerful runtime game engine. It is a proprietary engine, so it's unfortunately unavailable for use by developers outside EA.

1.5.5 Rockstar Advanced Game Engine (RAGE)

RAGE is the engine that drives the insanely popular *Grand Theft Auto V*. Developed by RAGE Technology Group, a division of Rockstar Games' Rockstar San Diego studio, RAGE has been used by Rockstar Games' internal studios to develop games for PlayStation 4, Xbox One, PlayStation 3, Xbox 360, Wii, Windows, and MacOS. Other games developed on this proprietary engine include *Grand Theft Auto IV*, *Red Dead Redemption* and *Max Payne 3*.

1.5.6 CRYENGINE

Crytek originally developed their powerful game engine known as CRYENGINE as a tech demo for NVIDIA. When the potential of the technology was recognized, Crytek turned the demo into a complete game and *Far Cry* was born. Since then, many games have been made with CRYENGINE including *Crysis*, *Codename Kingdoms*, *Ryse: Son of Rome*, and *Everyone's Gone to the Rapture*. Over the years the engine has evolved into what is now Crytek's latest offering, CRYENGINE V. This powerful game development platform offers a powerful suite of asset-creation tools and a feature-rich runtime engine featuring high-quality real-time graphics. CRYENGINE can be used to make games targeting a wide range of platforms including Xbox One, Xbox 360, PlayStation 4, PlayStation 3, Wii U, Linux, iOS and Android.

1.5.7 Sony's PhyreEngine

In an effort to make developing games for Sony's PlayStation 3 platform more accessible, Sony introduced PhyreEngine at the Game Developer's Conference (GDC) in 2008. As of 2013, PhyreEngine has evolved into a powerful and full-featured game engine, supporting an impressive array of features including advanced lighting and deferred rendering. It has been used by many studios to build over 90 published titles, including thatgamecompany's hits *fIOW*, *Flower* and *Journey*, and Coldwood Interactive's *Unravel*. PhyreEngine now supports Sony's PlayStation 4, PlayStation 3, PlayStation 2, PlayStation Vita and PSP platforms. PhyreEngine gives developers access to the power of the highly parallel Cell architecture on PS3 and the advanced compute capabilities of the PS4, along with a streamlined new world editor and other powerful game development tools. It is available free of charge to any licensed Sony developer as part of the PlayStation SDK.

1.5.8 Microsoft's XNA Game Studio

Microsoft's XNA Game Studio is an easy-to-use and highly accessible game development platform based on the C# language and the Common Language Runtime (CLR), and aimed at encouraging players to create their own games and share them with the online gaming community, much as YouTube encourages the creation and sharing of home-made videos.

For better or worse, Microsoft officially retired XNA in 2014. However, developers can port their XNA games to iOS, Android, Mac OS X, Linux and Windows 8 Metro via an open-source implementation of XNA called MonoGame. For more details, see <https://www.windowscentral.com/xna-dead-long-live-xna>.

1.5.9 Unity

Unity is a powerful cross-platform game development environment and runtime engine supporting a wide range of platforms. Using Unity, developers can deploy their games on mobile platforms (e.g., Apple iOS, Google Android), consoles (Microsoft Xbox 360 and Xbox One, Sony PlayStation 3 and PlayStation 4, and Nintendo Wii, Wii U), handheld gaming platforms (e.g., Playstation Vita, Nintendo Switch), desktop computers (Microsoft Windows, Apple Macintosh and Linux), TV boxes (e.g., Android TV and tvOS) and virtual reality (VR) systems (e.g., Oculus Rift, Steam VR, Gear VR).

Unity's primary design goals are ease of development and cross-platform game deployment. As such, Unity provides an easy-to-use integrated editor environment, in which you can create and manipulate the assets and entities that make up your game world and quickly preview your game in action right there in the editor, or directly on your target hardware. Unity also provides a powerful suite of tools for analyzing and optimizing your game on each target platform, a comprehensive asset conditioning pipeline, and the ability to manage the performance-quality trade-off uniquely on each deployment platform. Unity supports scripting in JavaScript, C# or Boo; a powerful animation system supporting animation retargeting (the ability to play an animation authored for one character on a totally different character); and support for networked multiplayer games.

Unity has been used to create a wide variety of published games, including *Deus Ex: The Fall* by N-Fusion/Eidos Montreal, *Hollow Knight* by Team Cherry, and the subversive retro-style *Cuphead* by StudioMDHR. The Webby Award winning short film *Adam* was rendered in real time using Unity.

1.5.10 Other Commercial Game Engines

There are lots of other commercial game engines out there. Although indie developers may not have the budget to purchase an engine, many of these products have great online documentation and/or wikis that can serve as a great source of information about game engines and game programming in general. For example, check out the Tombstone engine (<http://tombstoneengine.com/>) by Terathon Software, the LeadWerks engine (<https://www.leadwerks.com/>), and HeroEngine by Idea Fabrik, PLC (<http://www.heroengine.com/>).

1.5.11 Proprietary In-House Engines

Many companies build and maintain proprietary in-house game engines. Electronic Arts built many of its RTS games on a proprietary engine called Sage, developed at Westwood Studios. Naughty Dog's *Crash Bandicoot* and

Jak and Daxter franchises were built on a proprietary engine custom tailored to the PlayStation and PlayStation 2. For the *Uncharted* series, Naughty Dog developed a brand new engine custom tailored to the PlayStation 3 hardware. This engine evolved and was ultimately used to create Naughty Dog's *The Last of Us* series on the PlayStation 3 and PlayStation 4, as well as its most recent releases, *Uncharted 4: A Thief's End* and *Uncharted: The Lost Legacy*. And of course, most commercially licensed game engines like Quake, Source, Unreal Engine 4 and CRYENGINE all started out as proprietary in-house engines.

1.5.12 Open Source Engines

Open source 3D game engines are engines built by amateur and professional game developers and provided online for free. The term "open source" typically implies that source code is freely available and that a somewhat open development model is employed, meaning almost anyone can contribute code. Licensing, if it exists at all, is often provided under the Gnu Public License (GPL) or Lesser Gnu Public License (LGPL). The former permits code to be freely used by anyone, as long as their code is also freely available; the latter allows the code to be used even in proprietary for-profit applications. Lots of other free and semi-free licensing schemes are also available for open source projects.

There are a staggering number of open source engines available on the web. Some are quite good, some are mediocre and some are just plain awful! The list of game engines provided online at http://en.wikipedia.org/wiki/List_of_game_engines will give you a feel for the sheer number of engines that are out there. (The list at http://www.worldofleveldesign.com/categories/level_design_tutorials/recommended-game-engines.php is a bit more digestible.) Both of these lists include both open-source and commercial game engines.

OGRE is a well-architected, easy-to-learn and easy-to-use 3D rendering engine. It boasts a fully featured 3D renderer including advanced lighting and shadows, a good skeletal character animation system, a two-dimensional overlay system for heads-up displays and graphical user interfaces, and a post-processing system for full-screen effects like bloom. OGRE is, by its authors' own admission, not a full game engine, but it does provide many of the foundational components required by pretty much any game engine.

Some other well-known open source engines are listed here:

- Panda3D is a script-based engine. The engine's primary interface is the Python custom scripting language. It is designed to make prototyping

3D games and virtual worlds convenient and fast.

- Yake is a game engine built on top of OGRE.
- Crystal Space is a game engine with an extensible modular architecture.
- Torque and Irrlicht are also well-known open-source game engines.
- While not technically open-source, the Lumberyard engine does provide source code to its developers. It is a free cross-platform engine developed by Amazon, and based on the CRYENGINE architecture.

1.5.13 2D Game Engines for Non-programmers

Two-dimensional games have become incredibly popular with the recent explosion of casual web gaming and mobile gaming on platforms like Apple iPhone/iPad and Google Android. A number of popular game/multimedia authoring toolkits have become available, enabling small game studios and independent developers to create 2D games for these platforms. These toolkits emphasize ease of use and allow users to employ a graphical user interface to create a game rather than requiring the use of a programming language. Check out this YouTube video to get a feel for the kinds of games you can create with these toolkits: <https://www.youtube.com/watch?v=3Zq1yo0lxOU>

- *Multimedia Fusion 2* (<http://www.clickteam.com/website/world>) is a 2D game/multimedia authoring toolkit developed by Clickteam. Fusion is used by industry professionals to create games, screen savers and other multimedia applications. Fusion and its simpler counterpart, The Games Factory 2, are also used by educational camps like PlanetBravo (<http://www.planetbravo.com>) to teach kids about game development and programming/logic concepts. Fusion supports the iOS, Android, Flash, and Java platforms.
- *Game Salad Creator* (<http://gamesalad.com/creator>) is another graphical game/multimedia authoring toolkit aimed at non-programmers, similar in many respects to Fusion.
- *Scratch* (<http://scratch.mit.edu>) is an authoring toolkit and graphical programming language that can be used to create interactive demos and simple games. It is a great way for young people to learn about programming concepts such as conditionals, loops and event-driven programming. Scratch was developed in 2003 by the Lifelong Kindergarten group, led by Mitchel Resnick at the MIT Media Lab.

1.6 Runtime Engine Architecture

A game engine generally consists of a tool suite and a runtime component. We'll explore the architecture of the runtime piece first and then get into tool architecture in the following section.

Figure 1.16 shows all of the major runtime components that make up a typical 3D game engine. Yeah, it's *big!* And this diagram doesn't even account for all the tools. Game engines are definitely large software systems.

Like all software systems, game engines are built in *layers*. Normally upper layers depend on lower layers, but not vice versa. When a lower layer depends upon a higher layer, we call this a *circular dependency*. Dependency cycles are to be avoided in any software system, because they lead to undesirable coupling between systems, make the software untestable and inhibit code reuse. This is especially true for a large-scale system like a game engine.

What follows is a brief overview of the components shown in the diagram in Figure 1.16. The rest of this book will be spent investigating each of these components in a great deal more depth and learning how these components are usually integrated into a functional whole.

1.6.1 Target Hardware

The target hardware layer represents the computer system or console on which the game will run. Typical platforms include Microsoft Windows, Linux and MacOS-based PCs; mobile platforms like the Apple iPhone and iPad, Android smart phones and tablets, Sony's PlayStation Vita and Amazon's Kindle Fire (among others); and game consoles like Microsoft's Xbox, Xbox 360 and Xbox One, Sony's PlayStation, PlayStation 2, PlayStation 3 and PlayStation 4, and Nintendo's DS, GameCube, Wii, Wii U and Switch. Most of the topics in this book are platform-agnostic, but we'll also touch on some of the design considerations peculiar to PC or console development, where the distinctions are relevant.

1.6.2 Device Drivers

Device drivers are low-level software components provided by the operating system or hardware vendor. Drivers manage hardware resources and shield the operating system and upper engine layers from the details of communicating with the myriad variants of hardware devices available.

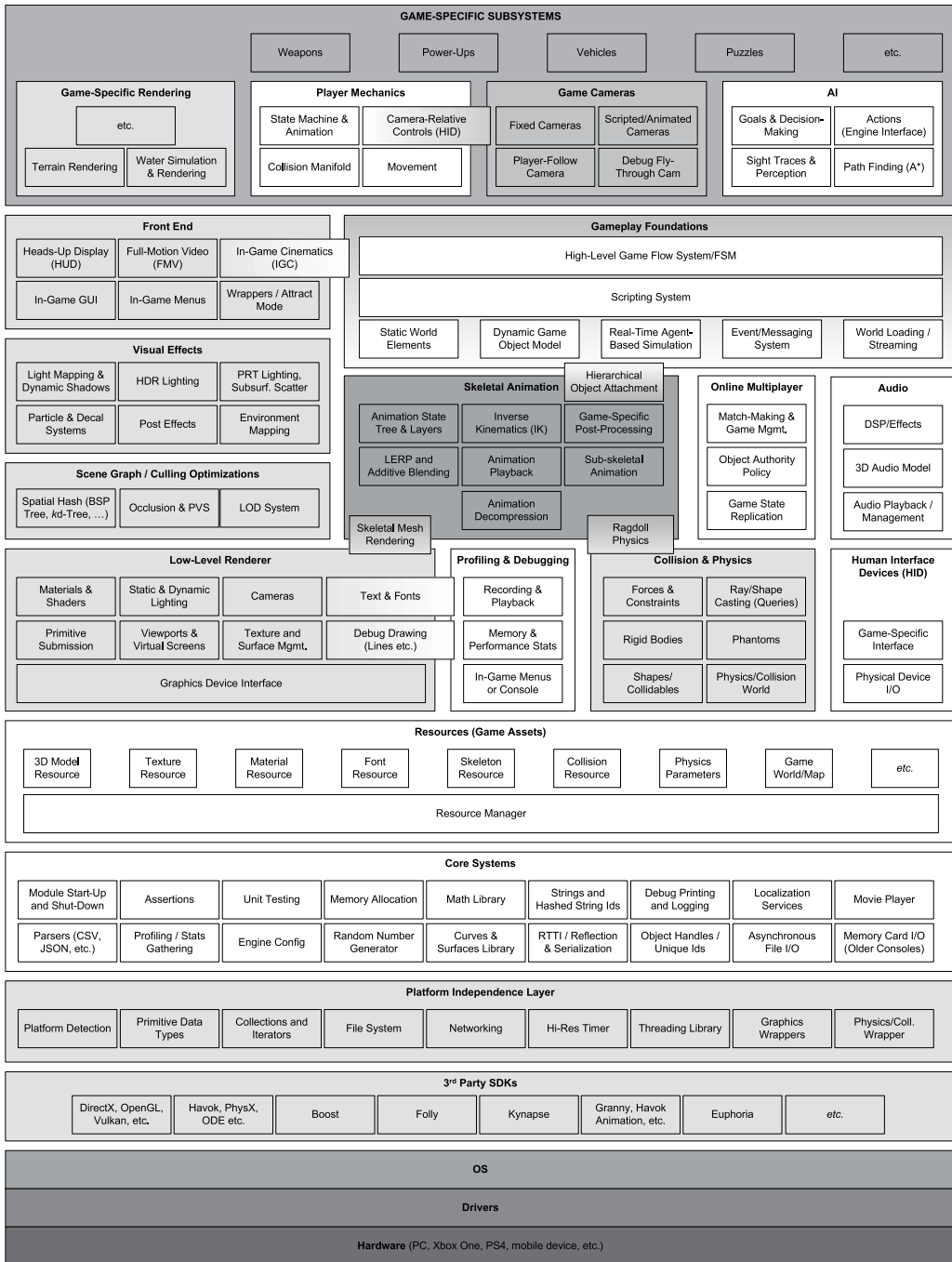


Figure 1.16. Runtime game engine architecture.

1.6.3 Operating System

On a PC, the operating system (OS) is running all the time. It orchestrates the execution of multiple programs on a single computer, one of which is your game. Operating systems like Microsoft Windows employ a time-sliced approach to sharing the hardware with multiple running programs, known as preemptive multitasking. This means that a PC game can never assume it has full control of the hardware—it must “play nice” with other programs in the system.

On early consoles, the operating system, if one existed at all, was just a thin library layer that was compiled directly into your game executable. On those early systems, the game “owned” the entire machine while it was running. However, on modern consoles this is no longer the case. The operating system on the Xbox 360, PlayStation 3, Xbox One and PlayStation 4 can interrupt the execution of your game, or take over certain system resources, in order to display online messages, or to allow the player to pause the game and bring up the PS4’s “XMB” user interface or the Xbox One’s dashboard, for example. On the PS4 and Xbox One, the OS is continually running background tasks, such as recording video of your playthrough in case you decide to share it via the PS4’s Share button, or downloading games, patches and DLC, so you can have fun playing a game while you wait. So the gap between console and PC development is gradually closing (for better or for worse).

1.6.4 Third-Party SDKs and Middleware

Most game engines leverage a number of third-party software development kits (SDKs) and middleware, as shown in Figure 1.17. The functional or class-based interface provided by an SDK is often called an application programming interface (API). We will look at a few examples.



Figure 1.17. Third-party SDK layer.

1.6.4.1 Data Structures and Algorithms

Like any software system, games depend heavily on *container* data structures and algorithms to manipulate them. Here are a few examples of third-party libraries that provide these kinds of services:

- *Boost*. Boost is a powerful data structures and algorithms library, designed in the style of the standard C++ library and its predecessor, the standard template library (STL). (The online documentation for Boost is also a great place to learn about computer science in general!)
- *Folly*. Folly is a library used at Facebook whose goal is to extend the standard C++ library and Boost with all sorts of useful facilities, with an emphasis on maximizing code performance.
- *Loki*. Loki is a powerful generic programming template library which is exceedingly good at making your brain hurt!

The C++ Standard Library and STL

The C++ standard library also provides many of the same kinds of facilities found in third-party libraries like Boost. The subset of the standard library that implements generic container classes such as `std::vector` and `std::list` is often referred to as the *standard template library* (STL), although this is technically a bit of a misnomer: The standard template library was written by Alexander Stepanov and David Musser in the days before the C++ language was standardized. Much of this library's functionality was absorbed into what is now the C++ standard library. When we use the term STL in this book, it's usually in the context of the subset of the C++ standard library that provides generic container classes, not the original STL.

1.6.4.2 Graphics

Most game rendering engines are built on top of a hardware interface library, such as the following:

- *Glide* is the 3D graphics SDK for the old Voodoo graphics cards. This SDK was popular prior to the era of hardware transform and lighting (hardware T&L) which began with DirectX 7.
- *OpenGL* is a widely used portable 3D graphics SDK.
- *DirectX* is Microsoft's 3D graphics SDK and primary rival to OpenGL.
- *libgcm* is a low-level direct interface to the PlayStation 3's RSX graphics hardware, which was provided by Sony as a more efficient alternative to OpenGL.
- *Edge* is a powerful and highly efficient rendering and animation engine produced by Naughty Dog and Sony for the PlayStation 3 and used by a number of first- and third-party game studios.

- *Vulkan* is a low-level library created by the Khronos™ Group which enables game programmers to submit rendering batches and GPGPU compute jobs directly to the GPU as command lists, and provides them with fine-grained control over memory and other resources that are shared between the CPU and GPU. (See Section 4.11 for more on GPGPU programming.)

1.6.4.3 Collision and Physics

Collision detection and rigid body dynamics (known simply as “physics” in the game development community) are provided by the following well-known SDKs:

- *Havok* is a popular industrial-strength physics and collision engine.
- *PhysX* is another popular industrial-strength physics and collision engine, available for free download from NVIDIA.
- *Open Dynamics Engine (ODE)* is a well-known open source physics/collision package.

1.6.4.4 Character Animation

A number of commercial animation packages exist, including but certainly not limited to the following:

- *Granny*. Rad Game Tools’ popular Granny toolkit includes robust 3D model and animation exporters for all the major 3D modeling and animation packages like Maya, 3D Studio MAX, etc., a runtime library for reading and manipulating the exported model and animation data, and a powerful runtime animation system. In my opinion, the Granny SDK has the best-designed and most logical animation API of any I’ve seen, commercial or proprietary, especially its excellent handling of time.
- *Havok Animation*. The line between physics and animation is becoming increasingly blurred as characters become more and more realistic. The company that makes the popular Havok physics SDK decided to create a complimentary animation SDK, which makes bridging the physics-animation gap much easier than it ever has been.
- *OrbisAnim*. The OrbisAnim library produced for the PS4 by SN Systems in conjunction with the ICE and game teams at Naughty Dog, the Tools and Technology group of Sony Interactive Entertainment, and Sony’s Advanced Technology Group in Europe includes a powerful and efficient animation engine and an efficient geometry-processing engine for rendering.

1.6.4.5 Biomechanical Character Models

- *Endorphin and Euphoria*. These are animation packages that produce character motion using advanced biomechanical models of realistic human movement.

As we mentioned previously, the line between character animation and physics is beginning to blur. Packages like Havok Animation try to marry physics and animation in a traditional manner, with a human animator providing the majority of the motion through a tool like Maya and with physics augmenting that motion at runtime. But a firm called Natural Motion Ltd. has produced a product that attempts to redefine how character motion is handled in games and other forms of digital media.

Its first product, *Endorphin*, is a Maya plug-in that permits animators to run full biomechanical simulations on characters and export the resulting animations as if they had been hand animated. The biomechanical model accounts for center of gravity, the character's weight distribution, and detailed knowledge of how a real human balances and moves under the influence of gravity and other forces.

Its second product, *Euphoria*, is a real-time version of *Endorphin* intended to produce physically and biomechanically accurate character motion at runtime under the influence of unpredictable forces.

1.6.5 Platform Independence Layer

Most game engines are required to be capable of running on more than one hardware platform. Companies like Electronic Arts and ActivisionBlizzard Inc., for example, always target their games at a wide variety of platforms because it exposes their games to the largest possible market. Typically, the only game studios that do not target at least two different platforms per game are first-party studios, like Sony's Naughty Dog and Insomniac studios. Therefore, most game engines are architected with a platform independence layer, like the one shown in Figure 1.18. This layer sits atop the hardware, drivers, operating system and other third-party software and shields the rest of the engine from the majority of knowledge of the underlying platform by "wrapping" certain interface functions in custom functions over which you, the game developer, will have control on every target platform.

There are two primary reasons to "wrap" functions as part of your game engine's platform independence layer like this: First, some application programming interfaces (APIs), like those provided by the operating system, or even some functions in older "standard" libraries like the C standard library,

differ significantly from platform to platform; wrapping these functions provides the rest of your engine with a consistent API across all of your targeted platforms. Second, even when using a fully cross-platform library such as Havok, you might want to insulate yourself from future changes, such as transitioning your engine to a different collision/physics library in the future.

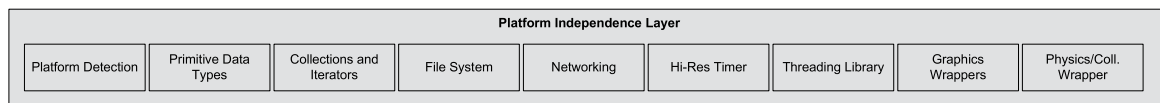


Figure I.18. Platform independence layer.

1.6.6 Core Systems

Every game engine, and really every large, complex C++ software application, requires a grab bag of useful software utilities. We'll categorize these under the label "core systems." A typical core systems layer is shown in Figure 1.19. Here are a few examples of the facilities the core layer usually provides:

- *Assertions* are lines of error-checking code that are inserted to catch logical mistakes and violations of the programmer's original assumptions. Assertion checks are usually stripped out of the final production build of the game. (Assertions are covered in Section 3.2.3.3.)
- *Memory management*. Virtually every game engine implements its own custom memory allocation system(s) to ensure high-speed allocations and deallocations and to limit the negative effects of memory fragmentation (see Section 6.2.1).
- *Math library*. Games are by their nature highly mathematics-intensive. As such, every game engine has at least one, if not many, math libraries. These libraries provide facilities for vector and matrix math, quaternion rotations, trigonometry, geometric operations with lines, rays, spheres, frusta, etc., spline manipulation, numerical integration, solving systems of equations and whatever other facilities the game programmers require.
- *Custom data structures and algorithms*. Unless an engine's designers decided to rely entirely on third-party packages such as Boost and Folly, a suite of tools for managing fundamental data structures (linked lists, dynamic arrays, binary trees, hash maps, etc.) and algorithms (search, sort, etc.) is usually required. These are often hand coded to minimize or eliminate dynamic memory allocation and to ensure optimal runtime performance on the target platform(s).

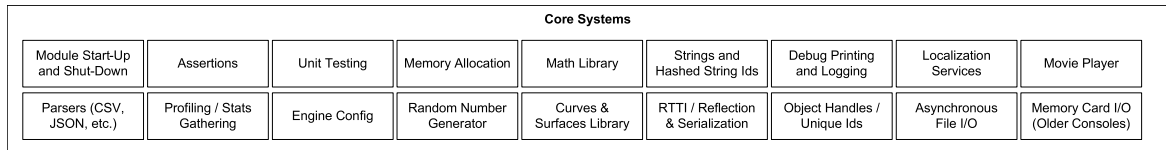


Figure 1.19. Core engine systems.

A detailed discussion of the most common core engine systems can be found in Part II.

1.6.7 Resource Manager

Present in every game engine in some form, the resource manager provides a unified interface (or suite of interfaces) for accessing any and all types of game assets and other engine input data. Some engines do this in a highly centralized and consistent manner (e.g., Unreal's packages, OGRE's Resource-Manager class). Other engines take an ad hoc approach, often leaving it up to the game programmer to directly access raw files on disk or within compressed archives such as Quake's PAK files. A typical resource manager layer is depicted in Figure 1.20.

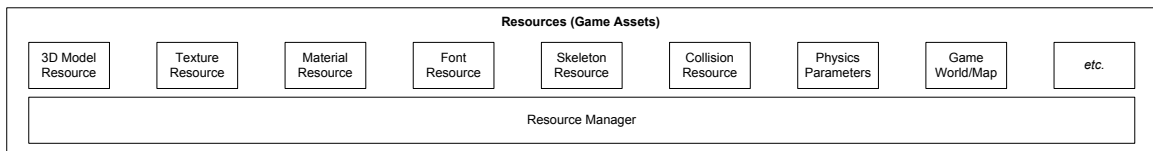


Figure 1.20. Resource manager.

1.6.8 Rendering Engine

The rendering engine is one of the largest and most complex components of any game engine. Renderers can be architected in many different ways. There is no one accepted way to do it, although as we'll see, most modern rendering engines share some fundamental design philosophies, driven in large part by the design of the 3D graphics hardware upon which they depend.

One common and effective approach to rendering engine design is to employ a layered architecture as follows.

1.6.8.1 Low-Level Renderer

The *low-level renderer*, shown in Figure 1.21, encompasses all of the raw rendering facilities of the engine. At this level, the design is focused on rendering a collection of geometric primitives as quickly and richly as possible, without much regard for which portions of a scene may be visible. This component is broken into various subcomponents, which are discussed below.

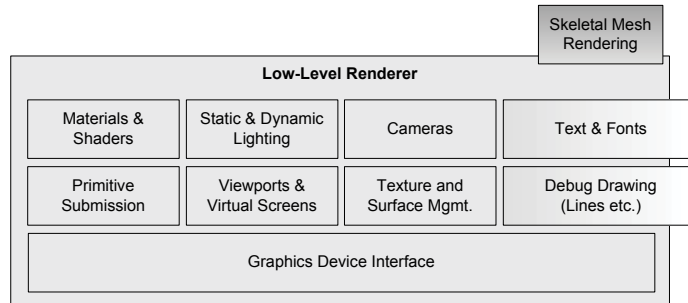


Figure 1.21. Low-level rendering engine.

Graphics Device Interface

Graphics SDKs, such as DirectX, OpenGL or Vulkan, require a reasonable amount of code to be written just to enumerate the available graphics devices, initialize them, set up render surfaces (back-buffer, stencil buffer, etc.) and so on. This is typically handled by a component that I’ll call the *graphics device interface* (although every engine uses its own terminology).

For a PC game engine, you also need code to integrate your renderer with the Windows message loop. You typically write a “message pump” that services Windows messages when they are pending and otherwise runs your render loop over and over as fast as it can. This ties the game’s keyboard polling loop to the renderer’s screen update loop. This coupling is undesirable, but with some effort it is possible to minimize the dependencies. We’ll explore this topic in more depth later.

Other Renderer Components

The other components in the low-level renderer cooperate in order to collect submissions of *geometric primitives* (sometimes called *render packets*), such as meshes, line lists, point lists, particles, terrain patches, text strings and whatever else you want to draw, and render them as quickly as possible.

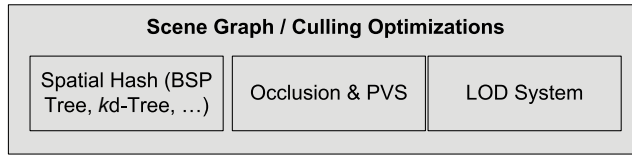


Figure 1.22. A typical scene graph/spatial subdivision layer, for culling optimization.

The low-level renderer usually provides a viewport abstraction with an associated camera-to-world matrix and 3D projection parameters, such as field of view and the location of the near and far clip planes. The low-level renderer also manages the state of the graphics hardware and the game's shaders via its *material system* and its *dynamic lighting system*. Each submitted primitive is associated with a material and is affected by n dynamic lights. The material describes the texture(s) used by the primitive, what device state settings need to be in force, and which vertex and pixel shader to use when rendering the primitive. The lights determine how dynamic lighting calculations will be applied to the primitive. Lighting and shading is a complex topic. We'll discuss the fundamentals in Chapter 11, but these topics are covered in depth in many excellent books on computer graphics, including [16], [49] and [2].

1.6.8.2 Scene Graph/Culling Optimizations

The low-level renderer draws all of the geometry submitted to it, without much regard for whether or not that geometry is actually visible (other than back-face culling and clipping triangles to the camera frustum). A higher-level component is usually needed in order to limit the number of primitives submitted for rendering, based on some form of visibility determination. This layer is shown in Figure 1.22.

For very small game worlds, a simple *frustum cull* (i.e., removing objects that the camera cannot "see") is probably all that is required. For larger game worlds, a more advanced *spatial subdivision* data structure might be used to improve rendering efficiency by allowing the potentially visible set (PVS) of objects to be determined very quickly. Spatial subdivisions can take many forms, including a binary space partitioning tree, a quadtree, an octree, a *kd-tree* or a sphere hierarchy. A spatial subdivision is sometimes called a scene graph, although technically the latter is a particular kind of data structure and does not subsume the former. Portals or occlusion culling methods might also be applied in this layer of the rendering engine.

Ideally, the low-level renderer should be completely agnostic to the type of spatial subdivision or scene graph being used. This permits different game

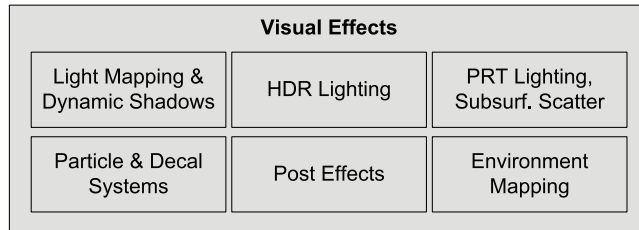


Figure 1.23. Visual effects.

teams to reuse the primitive submission code but to craft a PVS determination system that is specific to the needs of each team's game. The design of the OGRE open source rendering engine (<http://www.ogre3d.org>) is a great example of this principle in action. OGRE provides a plug-and-play scene graph architecture. Game developers can either select from a number of preimplemented scene graph designs, or they can provide a custom scene graph implementation.

1.6.8.3 Visual Effects

Modern game engines support a wide range of visual effects, as shown in Figure 1.23, including:

- particle systems (for smoke, fire, water splashes, etc.);
- decal systems (for bullet holes, foot prints, etc.);
- light mapping and environment mapping;
- dynamic shadows; and
- full-screen post effects, applied after the 3D scene has been rendered to an off-screen buffer.

Some examples of full-screen post effects include:

- high dynamic range (HDR) tone mapping and bloom;
- full-screen anti-aliasing (FSAA); and
- color correction and color-shift effects, including bleach bypass, saturation and desaturation effects, etc.

It is common for a game engine to have an *effects system* component that manages the specialized rendering needs of particles, decals and other visual effects. The particle and decal systems are usually distinct components of the rendering engine and act as inputs to the low-level renderer. On the other

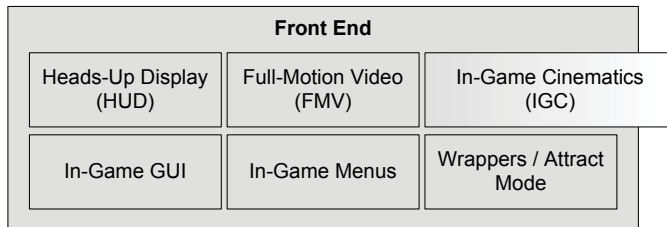


Figure 1.24. Front end graphics.

hand, light mapping, environment mapping and shadows are usually handled internally within the rendering engine proper. Full-screen post effects are either implemented as an integral part of the renderer or as a separate component that operates on the renderer's output buffers.

1.6.8.4 Front End

Most games employ some kind of 2D graphics overlaid on the 3D scene for various purposes. These include:

- the game's *heads-up display* (HUD);
- in-game menus, a console and /or other *development tools*, which may or may not be shipped with the final product; and
- possibly an in-game *graphical user interface* (GUI), allowing the player to manipulate his or her character's inventory, configure units for battle or perform other complex in-game tasks.

This layer is shown in Figure 1.24. Two-dimensional graphics like these are usually implemented by drawing textured quads (pairs of triangles) with an orthographic projection. Or they may be rendered in full 3D, with the quads bill-boarded so they always face the camera.

We've also included the *full-motion video* (FMV) system in this layer. This system is responsible for playing full-screen movies that have been recorded earlier (either rendered with the game's rendering engine or using another rendering package).

A related system is the *in-game cinematics* (IGC) system. This component typically allows cinematic sequences to be choreographed within the game itself, in full 3D. For example, as the player walks through a city, a conversation between two key characters might be implemented as an in-game cinematic. IGCs may or may not include the player character(s). They may be done as a deliberate cut-away during which the player has no control, or they may be subtly integrated into the game without the human player even realizing that

an IGC is taking place. Some games, such as Naughty Dog's *Uncharted 4: A Thief's End*, have moved away from pre-rendered movies entirely, and display *all* cinematic moments in the game as real-time IGCs.

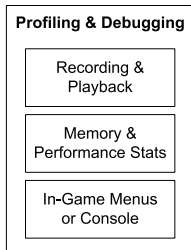


Figure 1.25. Profiling and debugging tools.

1.6.9 Profiling and Debugging Tools

Games are real-time systems and, as such, game engineers often need to profile the performance of their games in order to optimize performance. In addition, memory resources are usually scarce, so developers make heavy use of memory analysis tools as well. The profiling and debugging layer, shown in Figure 1.25, encompasses these tools and also includes in-game debugging facilities, such as debug drawing, an in-game menu system or console and the ability to record and play back gameplay for testing and debugging purposes.

There are plenty of good general-purpose software profiling tools available, including:

- Intel's *VTune*,
- IBM's *Quantify* and *Purify* (part of the *PurifyPlus* tool suite),
- *Insure++* by Parasoft, and
- *Valgrind* by Julian Seward and the Valgrind development team.

However, most game engines also incorporate a suite of custom profiling and debugging tools. For example, they might include one or more of the following:

- a mechanism for manually instrumenting the code, so that specific sections of code can be timed;
- a facility for displaying the profiling statistics on-screen while the game is running;
- a facility for dumping performance stats to a text file or to an Excel spreadsheet;
- a facility for determining how much memory is being used by the engine, and by each subsystem, including various on-screen displays;
- the ability to dump memory usage, high water mark and leakage stats when the game terminates and/or during gameplay;
- tools that allow debug print statements to be peppered throughout the code, along with an ability to turn on or off different categories of debug output and control the level of verbosity of the output; and

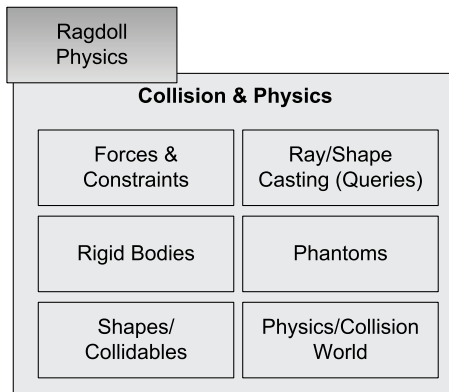


Figure 1.26. Collision and physics subsystem.

- the ability to record game events and then play them back. This is tough to get right, but when done properly it can be a very valuable tool for tracking down bugs.

The PlayStation 4 provides a powerful core dump facility to aid programmers in debugging crashes. The PlayStation 4 is always recording the last 15 seconds of gameplay video, to allow players to share their experiences via the Share button on the controller. Because of this, the PS4's core dump facility automatically provides programmers not only with a complete call stack of what the program was doing when it crashed, but also with a screenshot of the moment of the crash and 15 seconds of video footage showing what was happening just prior to the crash. Core dumps can be automatically uploaded to the game developer's servers whenever the game crashes, even after the game has shipped. These facilities revolutionize the tasks of crash analysis and repair.

1.6.10 Collision and Physics

Collision detection is important for every game. Without it, objects would interpenetrate, and it would be impossible to interact with the virtual world in any reasonable way. Some games also include a realistic or semi-realistic dynamics simulation. We call this the "physics system" in the game industry, although the term *rigid body dynamics* is really more appropriate, because we are usually only concerned with the motion (kinematics) of rigid bodies and the forces and torques (dynamics) that cause this motion to occur. This layer is depicted in Figure 1.26.

Collision and physics are usually quite tightly coupled. This is because when collisions are detected, they are almost always resolved as part of the physics integration and constraint satisfaction logic. Nowadays, very few game companies write their own collision/physics engine. Instead, a third-party SDK is typically integrated into the engine.

- *Havok* is the gold standard in the industry today. It is feature-rich and performs well across the boards.
- *PhysX* by NVIDIA is another excellent collision and dynamics engine. It was integrated into Unreal Engine 4 and is also available for free as a stand-alone product for PC game development. PhysX was originally designed as the interface to Ageia's physics accelerator chip. The SDK is now owned and distributed by NVIDIA, and the company has adapted PhysX to run on its latest GPUs.

Open source physics and collision engines are also available. Perhaps the best-known of these is the Open Dynamics Engine (ODE). For more information, see <http://www.ode.org>. I-Collide, V-Collide and RAPID are other popular non-commercial collision detection engines. All three were developed at the University of North Carolina (UNC). For more information, see http://www.cs.unc.edu/~geom/I_COLLIDE/index.html and http://www.cs.unc.edu/~geom/V_COLLIDE/index.html.

1.6.11 Animation

Any game that has organic or semi-organic characters (humans, animals, cartoon characters or even robots) needs an animation system. There are five basic types of animation used in games:

- sprite/texture animation,
- rigid body hierarchy animation,
- skeletal animation,
- vertex animation, and
- morph targets.

Skeletal animation permits a detailed 3D character mesh to be posed by an animator using a relatively simple system of bones. As the bones move, the vertices of the 3D mesh move with them. Although morph targets and vertex animation are used in some engines, skeletal animation is the most prevalent animation method in games today; as such, it will be our primary focus in this book. A typical skeletal animation system is shown in Figure 1.27.

You'll notice in Figure 1.16 that the skeletal mesh rendering component bridges the gap between the renderer and the animation system. There is a tight cooperation happening here, but the interface is very well defined. The animation system produces a pose for every bone in the skeleton, and then these poses are passed to the rendering engine as a palette of matrices. The renderer transforms each vertex by the matrix or matrices in the palette, in order to generate a final blended vertex position. This process is known as *skinning*.

There is also a tight coupling between the animation and physics systems when *rag dolls* are employed. A rag doll is a limp (often dead) animated character, whose bodily motion is simulated by the physics system. The physics system determines the positions and orientations of the various parts of the body by treating them as a constrained system of rigid bodies. The animation system calculates the palette of matrices required by the rendering engine in order to draw the character on-screen.

1.6.12 Human Interface Devices (HID)

Every game needs to process input from the player, obtained from various *human interface devices* (HIDs) including:

- the keyboard and mouse,
- a joystick, or
- other specialized game controllers, like steering wheels, fishing rods, dance pads, the Wiimote, etc.

We sometimes call this component the *player I/O* component, because

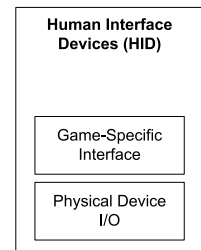


Figure 1.28. The player input/output system, also known as the human interface device (HID) layer.

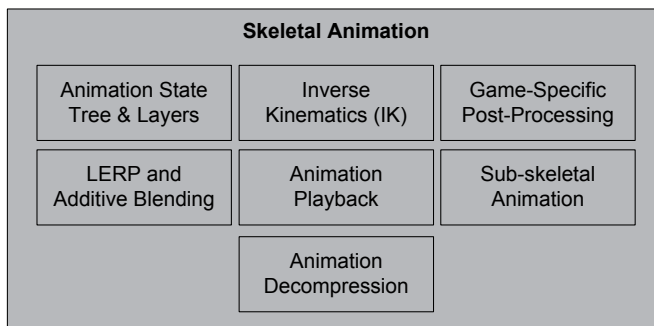


Figure 1.27. Skeletal animation subsystem.

we may also provide *output* to the player through the HID, such as force-feedback/ rumble on a joypad or the audio produced by the Wiimote. A typical HID layer is shown in Figure 1.28.

The HID engine component is sometimes architected to divorce the low-level details of the game controller(s) on a particular hardware platform from the high-level game controls. It massages the raw data coming from the hardware, introducing a dead zone around the center point of each joypad stick, debouncing button-press inputs, detecting button-down and button-up events, interpreting and smoothing accelerometer inputs (e.g., from the PlayStation Dualshock controller) and more. It often provides a mechanism allowing the player to customize the mapping between physical controls and logical game functions. It sometimes also includes a system for detecting chords (multiple buttons pressed together), sequences (buttons pressed in sequence within a certain time limit) and gestures (sequences of inputs from the buttons, sticks, accelerometers, etc.).

1.6.13 Audio

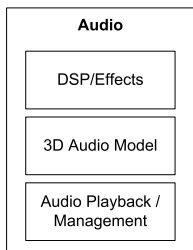


Figure 1.29. Audio subsystem.

Audio is just as important as graphics in any game engine. Unfortunately, audio often gets less attention than rendering, physics, animation, AI and gameplay. Case in point: Programmers often develop their code with their speakers turned off! (In fact, I've known quite a few game programmers who didn't even *have* speakers or headphones.) Nonetheless, no great game is complete without a stunning audio engine. The audio layer is depicted in Figure 1.29.

Audio engines vary greatly in sophistication. Quake's audio engine is pretty basic, and game teams usually augment it with custom functionality or replace it with an in-house solution. Unreal Engine 4 provides a reasonably robust 3D audio rendering engine (discussed in detail in [45]), although its feature set is limited and many game teams will probably want to augment and customize it to provide advanced game-specific features. For DirectX platforms (PC, Xbox 360, Xbox One), Microsoft provides an excellent runtime audio engine called XAudio2. Electronic Arts has developed an advanced, high-powered audio engine internally called SoundR!OT. In conjunction with first-party studios like Naughty Dog, Sony Interactive Entertainment (SIE) provides a powerful 3D audio engine called Scream, which has been used on a number of PS3 and PS4 titles including Naughty Dog's *Uncharted 4: A Thief's End* and *The Last of Us: Remastered*. However, even if a game team uses a preexisting audio engine, every game requires a great deal of custom software development, integration work, fine-tuning and attention to detail in order to produce high-quality audio in the final product.

1.6.14 Online Multiplayer/Networking

Many games permit multiple human players to play within a single virtual world. Multiplayer games come in at least four basic flavors:

- *Single-screen multiplayer.* Two or more human interface devices (joypads, keyboards, mice, etc.) are connected to a single arcade machine, PC or console. Multiple player characters inhabit a single virtual world, and a single camera keeps all player characters in frame simultaneously. Examples of this style of multiplayer gaming include *Smash Brothers*, *Lego Star Wars* and *Gauntlet*.
- *Split-screen multiplayer.* Multiple player characters inhabit a single virtual world, with multiple HIDs attached to a single game machine, but each with its own camera, and the screen is divided into sections so that each player can view his or her character.
- *Networked multiplayer.* Multiple computers or consoles are networked together, with each machine hosting one of the players.
- *Massively multiplayer online games (MMOG).* Literally hundreds of thousands of users can be playing simultaneously within a giant, persistent, online virtual world hosted by a powerful battery of central servers.

The multiplayer networking layer is shown in Figure 1.30.

Multiplayer games are quite similar in many ways to their single-player counterparts. However, support for multiple players can have a profound impact on the design of certain game engine components. The game world object model, renderer, human input device system, player control system and animation systems are all affected. Retrofitting multiplayer features into a pre-existing single-player engine is certainly not impossible, although it can be a daunting task. Still, many game teams have done it successfully. That said, it is usually better to design multiplayer features from day one, if you have that luxury.

It is interesting to note that going the other way—converting a multiplayer game into a single-player game—is typically trivial. In fact, many game engines treat single-player mode as a special case of a multiplayer game, in which there happens to be only one player. The Quake engine is well known for its *client-on-top-of-server* mode, in which a single executable, running on a single PC, acts both as the client and the server in single-player campaigns.

1.6.15 Gameplay Foundation Systems

The term *gameplay* refers to the action that takes place in the game, the rules that govern the virtual world in which the game takes place, the abilities of the

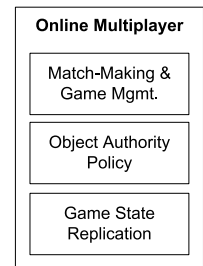


Figure 1.30. Online multiplayer networking subsystem.

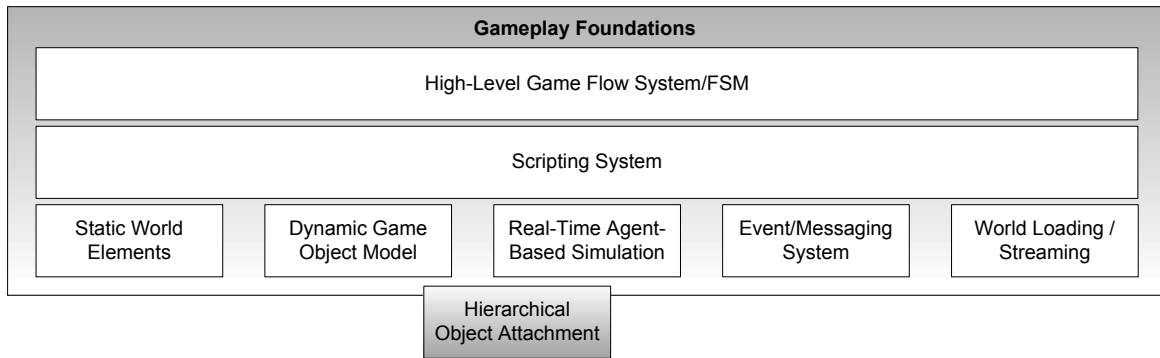


Figure 1.31. Gameplay Foundation systems.

player character(s) (known as *player mechanics*) and of the other characters and objects in the world, and the goals and objectives of the player(s). Gameplay is typically implemented either in the native language in which the rest of the engine is written or in a high-level scripting language—or sometimes both. To bridge the gap between the gameplay code and the low-level engine systems that we’ve discussed thus far, most game engines introduce a layer that I’ll call the *gameplay foundations* layer (for lack of a standardized name). Shown in Figure 1.31, this layer provides a suite of core facilities, upon which game-specific logic can be implemented conveniently.

1.6.15.1 Game Worlds and Object Models

The gameplay foundations layer introduces the notion of a game world, containing both static and dynamic elements. The contents of the world are usually modeled in an object-oriented manner (often, but not always, using an object-oriented programming language). In this book, the collection of object types that make up a game is called the *game object model*. The game object model provides a real-time simulation of a heterogeneous collection of objects in the virtual game world.

Typical types of game objects include:

- static background geometry, like buildings, roads, terrain (often a special case), etc.;
- dynamic rigid bodies, such as rocks, soda cans, chairs, etc.;
- player characters (PC);
- non-player characters (NPC);

- weapons;
- projectiles;
- vehicles;
- lights (which may be present in the dynamic scene at runtime, or only used for static lighting offline);
- cameras;

and the list goes on.

The game world model is intimately tied to a *software object model*, and this model can end up pervading the entire engine. The term software object model refers to the set of language features, policies and conventions used to implement a piece of object-oriented software. In the context of game engines, the software object model answers questions, such as:

- Is your game engine designed in an object-oriented manner?
- What language will you use? C? C++? Java? OCaml?
- How will the static class hierarchy be organized? One giant monolithic hierarchy? Lots of loosely coupled components?
- Will you use templates and policy-based design, or traditional polymorphism?
- How are objects referenced? Straight old pointers? Smart pointers? Handles?
- How will objects be uniquely identified? By address in memory only? By name? By a global unique identifier (GUID)?
- How are the lifetimes of game objects managed?
- How are the states of the game objects simulated over time?

We'll explore software object models and game object models in considerable depth in Section 16.2.

1.6.15.2 Event System

Game objects invariably need to communicate with one another. This can be accomplished in all sorts of ways. For example, the object sending the message might simply call a member function of the receiver object. An event-driven architecture, much like what one would find in a typical graphical user interface, is also a common approach to inter-object communication. In an event-driven system, the sender creates a little data structure called an *event* or *message*, containing the message's type and any argument data that are to be sent. The event is passed to the receiver object by calling its *event handler function*. Events can also be stored in a queue for handling at some future time.

1.6.15.3 Scripting System

Many game engines employ a scripting language in order to make development of game-specific gameplay rules and content easier and more rapid. Without a scripting language, you must recompile and relink your game executable every time a change is made to the logic or data structures used in the engine. But when a scripting language is integrated into your engine, changes to game logic and data can be made by modifying and reloading the script code. Some engines allow script to be reloaded while the game continues to run. Other engines require the game to be shut down prior to script recompilation. But either way, the turnaround time is still much faster than it would be if you had to recompile and relink the game's executable.

1.6.15.4 Artificial Intelligence Foundations

Traditionally, artificial intelligence has fallen squarely into the realm of game-specific software—it was usually not considered part of the game engine per se. More recently, however, game companies have recognized patterns that arise in almost every AI system, and these foundations are slowly starting to fall under the purview of the engine proper.

For example, a company called Kynogon developed a middleware SDK named Kynapse, which provides much of the low-level technology required to build commercially viable game AI. This technology was purchased by Autodesk and has been superseded by a totally redesigned AI middleware package called Gameware Navigation, designed by the same engineering team that invented Kynapse. This SDK provides low-level AI building blocks such as nav mesh generation, path finding, static and dynamic object avoidance, identification of vulnerabilities within a play space (e.g., an open window from which an ambush could come) and a well-defined interface between AI and animation.

1.6.16 Game-Specific Subsystems

On top of the gameplay foundation layer and the other low-level engine components, gameplay programmers and designers cooperate to implement the features of the game itself. Gameplay systems are usually numerous, highly varied and specific to the game being developed. As shown in Figure 1.32, these systems include, but are certainly not limited to the mechanics of the player character, various in-game camera systems, artificial intelligence for the control of non-player characters, weapon systems, vehicles and the list goes on. If a clear line could be drawn between the engine and the game,

it would lie between the game-specific subsystems and the gameplay foundations layer. Practically speaking, this line is never perfectly distinct. At least some game-specific knowledge invariably seeps down through the gameplay foundations layer and sometimes even extends into the core of the engine itself.

1.7 Tools and the Asset Pipeline

Any game engine must be fed a great deal of data, in the form of game assets, configuration files, scripts and so on. Figure 1.33 depicts some of the types of game assets typically found in modern game engines. The thicker dark-grey arrows show how data flows from the tools used to create the original source assets all the way through to the game engine itself. The thinner light-grey arrows show how the various types of assets refer to or use other assets.

1.7.1 Digital Content Creation Tools

Games are multimedia applications by nature. A game engine's input data comes in a wide variety of forms, from 3D mesh data to texture bitmaps to animation data to audio files. All of this source data must be created and manipulated by artists. The tools that the artists use are called *digital content creation* (DCC) applications.

A DCC application is usually targeted at the creation of one particular type of data—although some tools can produce multiple data types. For example, Autodesk's Maya and 3ds Max and Pixologic's ZBrush are prevalent in the creation of both 3D meshes and animation data. Adobe's Photoshop and its ilk are aimed at creating and editing bitmaps (textures). SoundForge is a popular tool for creating audio clips. Some types of game data cannot be created using an off-the-shelf DCC app. For example, most game engines provide a custom editor for laying out game worlds. Still, some engines do make use of

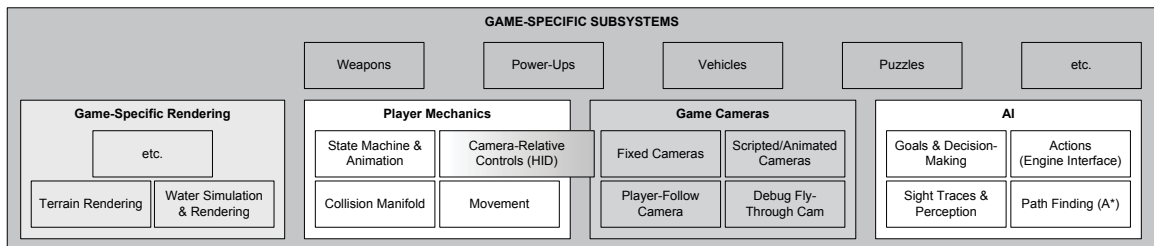


Figure 1.32. Game-specific subsystems.

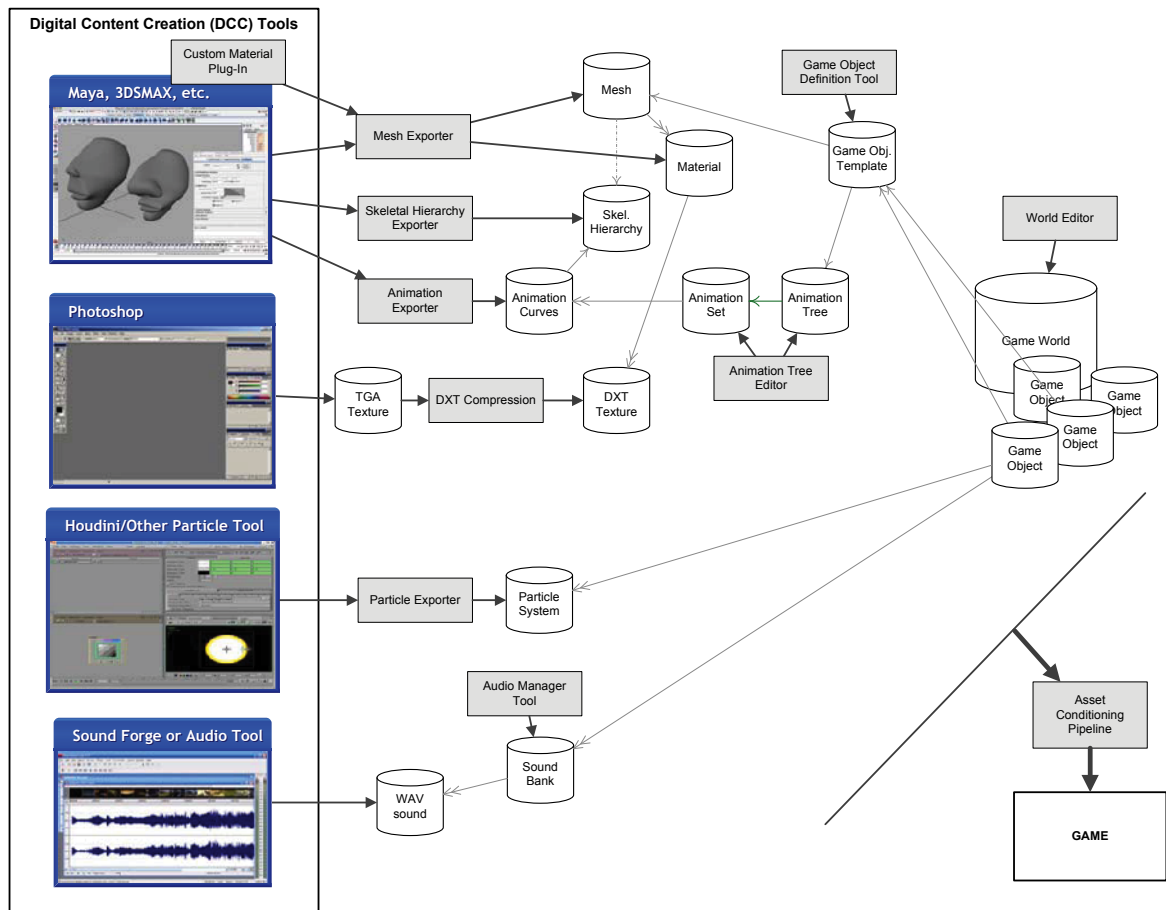


Figure 1.33. Tools and the asset pipeline.

preexisting tools for game world layout. I've seen game teams use 3ds Max or Maya as a world layout tool, with or without custom plug-ins to aid the user. Ask most game developers, and they'll tell you they can remember a time when they laid out terrain height fields using a simple bitmap editor, or typed world layouts directly into a text file by hand. Tools don't have to be pretty—game teams will use whatever tools are available and get the job done. That said, tools must be relatively *easy to use*, and they absolutely must be *reliable*, if a game team is going to be able to develop a highly polished product in a timely manner.

1.7.2 The Asset Conditioning Pipeline

The data formats used by digital content creation (DCC) applications are rarely suitable for direct use in-game. There are two primary reasons for this.

1. The DCC app's in-memory model of the data is usually much more complex than what the game engine requires. For example, Maya stores a directed acyclic graph (DAG) of scene nodes, with a complex web of interconnections. It stores a history of all the edits that have been performed on the file. It represents the position, orientation and scale of every object in the scene as a full hierarchy of 3D transformations, decomposed into translation, rotation, scale and shear components. A game engine typically only needs a tiny fraction of this information in order to render the model in-game.
2. The DCC application's file format is often too slow to read at runtime, and in some cases it is a closed proprietary format.

Therefore, the data produced by a DCC app is usually exported to a more accessible standardized format, or a custom file format, for use in-game.

Once data has been exported from the DCC app, it often must be further processed before being sent to the game engine. And if a game studio is shipping its game on more than one platform, the intermediate files might be processed differently for each target platform. For example, 3D mesh data might be exported to an intermediate format, such as XML, JSON or a simple binary format. Then it might be processed to combine meshes that use the same material, or split up meshes that are too large for the engine to digest. The mesh data might then be organized and packed into a memory image suitable for loading on a specific hardware platform.

The pipeline from DCC app to game engine is sometimes called the *asset conditioning pipeline* (ACP). Every game engine has this in some form.

1.7.2.1 3D Model/Mesh Data

The visible geometry you see in a game is typically constructed from triangle meshes. Some older games also make use of volumetric geometry known as *brushes*. We'll discuss each type of geometric data briefly below. For an in-depth discussion of the techniques used to describe and render 3D geometry, see Chapter 11.

3D Models (Meshes)

A mesh is a complex shape composed of triangles and vertices. Renderable geometry can also be constructed from quads or higher-order subdivision

surfaces. But on today's graphics hardware, which is almost exclusively geared toward rendering rasterized triangles, all shapes must eventually be translated into triangles prior to rendering.

A mesh typically has one or more *materials* applied to it in order to define visual surface properties (color, reflectivity, bumpiness, diffuse texture, etc.). In this book, I will use the term "mesh" to refer to a single renderable shape, and "model" to refer to a composite object that may contain multiple meshes, plus animation data and other metadata for use by the game.

Meshes are typically created in a 3D modeling package such as 3ds Max, Maya or SoftImage. A powerful and popular tool by Pixologic called ZBrush allows ultra high-resolution meshes to be built in a very intuitive way and then down-converted into a lower-resolution model with normal maps to approximate the high-frequency detail.

Exporters must be written to extract the data from the digital content creation (DCC) tool (Maya, Max, etc.) and store it on disk in a form that is digestible by the engine. The DCC apps provide a host of standard or semi-standard export formats, although none are perfectly suited for game development (with the possible exception of COLLADA). Therefore, game teams often create custom file formats and custom exporters to go with them.

Brush Geometry

Brush geometry is defined as a collection of convex hulls, each of which is defined by multiple planes. Brushes are typically created and edited directly in the game world editor. This is essentially an "old school" approach to creating renderable geometry, but it is still used in some engines.

Pros:

- fast and easy to create;
- accessible to game designers—often used to "block out" a game level for prototyping purposes;
- can serve both as collision volumes and as renderable geometry.

Cons:

- low-resolution;
- difficult to create complex shapes;
- cannot support articulated objects or animated characters.

1.7.2.2 Skeletal Animation Data

A *skeletal mesh* is a special kind of mesh that is bound to a skeletal hierarchy for the purposes of articulated animation. Such a mesh is sometimes called a

skin because it forms the skin that surrounds the invisible underlying skeleton. Each vertex of a skeletal mesh contains a list of indices indicating to which joint(s) in the skeleton it is bound. A vertex usually also includes a set of joint weights, specifying the amount of influence each joint has on the vertex.

In order to render a skeletal mesh, the game engine requires three distinct kinds of data:

1. the mesh itself,
2. the skeletal hierarchy (joint names, parent-child relationships and the base pose the skeleton was in when it was originally bound to the mesh), and
3. one or more animation clips, which specify how the joints should move over time.

The mesh and skeleton are often exported from the DCC application as a single data file. However, if multiple meshes are bound to a single skeleton, then it is better to export the skeleton as a distinct file. The animations are usually exported individually, allowing only those animations which are in use to be loaded into memory at any given time. However, some game engines allow a bank of animations to be exported as a single file, and some even lump the mesh, skeleton and animations into one monolithic file.

An unoptimized skeletal animation is defined by a stream of 4×3 matrix samples, taken at a frequency of at least 30 frames per second, for each of the joints in a skeleton (of which there can be 500 or more for a realistic humanoid character). Thus, animation data is inherently memory-intensive. For this reason, animation data is almost always stored in a highly compressed format. Compression schemes vary from engine to engine, and some are proprietary. There is no one standardized format for game-ready animation data.

1.7.2.3 Audio Data

Audio clips are usually exported from Sound Forge or some other audio production tool in a variety of formats and at a number of different data sampling rates. Audio files may be in mono, stereo, 5.1, 7.1 or other multi-channel configurations. Wave files (.wav) are common, but other file formats such as PlayStation ADPCM files (.vag) are also commonplace. Audio clips are often organized into banks for the purposes of organization, easy loading into the engine, and streaming.

1.7.2.4 Particle Systems Data

Modern games make use of complex particle effects. These are authored by artists who specialize in the creation of visual effects. Third-party tools, such as Houdini, permit film-quality effects to be authored; however, most game engines are not capable of rendering the full gamut of effects that can be created with Houdini. For this reason, many game companies create a custom particle effect editing tool, which exposes only the effects that the engine actually supports. A custom tool might also let the artist see the effect exactly as it will appear in-game.

1.7.3 The World Editor

The game world is where everything in a game engine comes together. To my knowledge, there are no commercially available game world editors (i.e., the game world equivalent of Maya or Max). However, a number of commercially available game engines provide good world editors:

- Some variant of the *Radiant* game editor is used by most game engines based on Quake technology.
- The *Half-Life 2* Source engine provides a world editor called *Hammer*.
- *UnrealEd* is the Unreal Engine's world editor. This powerful tool also serves as the asset manager for all data types that the engine can consume.
- *Sandbox* is the world editor in CRYENGINE.

Writing a good world editor is difficult, but it is an extremely important part of any good game engine.

1.7.4 The Resource Database

Game engines deal with a wide range of asset types, from renderable geometry to materials and textures to animation data to audio. These assets are defined in part by the raw data produced by the artists when they use a tool like Maya, Photoshop or SoundForge. However, every asset also carries with it a great deal of *metadata*. For example, when an animator authors an animation clip in Maya, the metadata provides the asset conditioning pipeline, and ultimately the game engine, with the following information:

- A unique id that identifies the animation clip at runtime.
- The name and directory path of the source Maya (.ma or .mb) file.
- The *frame range*—on which frame the animation begins and ends.
- Whether or not the animation is intended to loop.

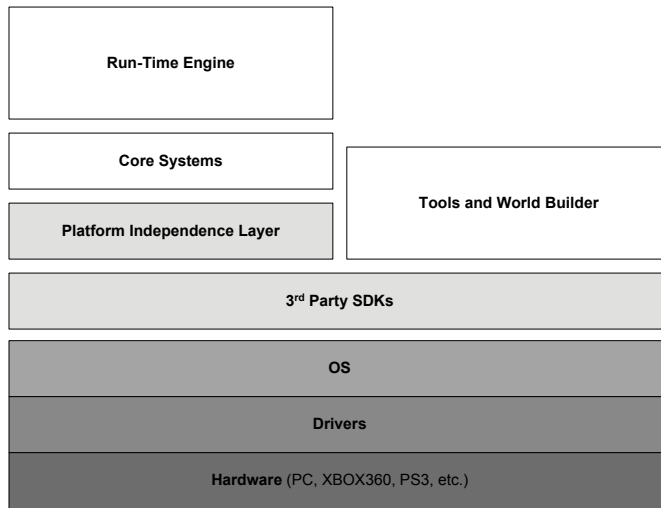


Figure 1.34. Stand-alone tools architecture.

- The animator’s choice of compression technique and level. (Some assets can be highly compressed without noticeably degrading their quality, while others require less or no compression in order to look right in-game.)

Every game engine requires some kind of database to manage all of the metadata associated with the game’s assets. This database might be implemented using an honest-to-goodness relational database such as MySQL or Oracle, or it might be implemented as a collection of text files, managed by a revision control system such as Subversion, Perforce or Git. We’ll call this metadata the *resource database* in this book.

No matter in what format the resource database is stored and managed, some kind of user interface must be provided to allow users to author and edit the data. At Naughty Dog, we wrote a custom GUI in C# called Builder for this purpose. For more information on Builder and a few other resource database user interfaces, see Section 7.2.1.3.

1.7.5 Some Approaches to Tool Architecture

A game engine’s tool suite may be architected in any number of ways. Some tools might be stand-alone pieces of software, as shown in Figure 1.34. Some tools may be built on top of some of the lower layers used by the runtime engine, as Figure 1.35 illustrates. Some tools might be built into the game itself.

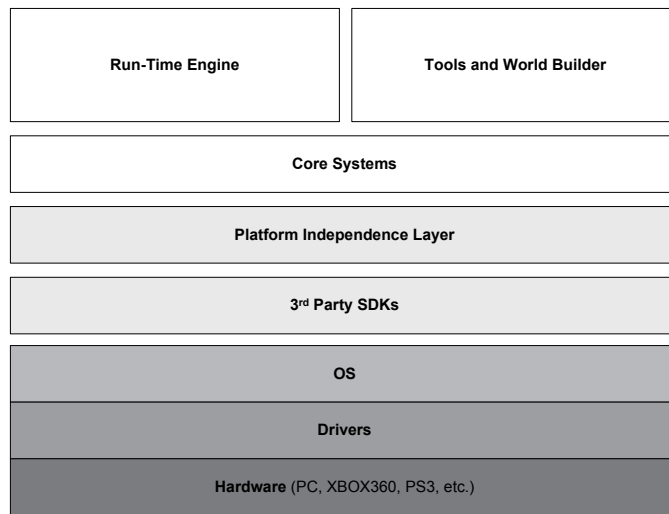


Figure 1.35. Tools built on a framework shared with the game.

For example, Quake- and Unreal-based games both boast an in-game console that permits developers and “modders” to type debugging and configuration commands while running the game. Finally, web-based user interfaces are becoming more and more popular for certain kinds of tools.

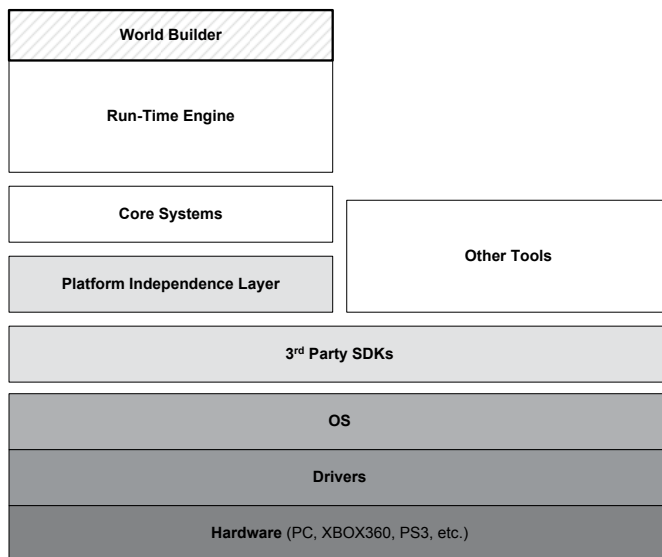


Figure 1.36. Unreal Engine's tool architecture.

As an interesting and unique example, Unreal’s world editor and asset manager, UnrealEd, is built right into the runtime game engine. To run the editor, you run your game with a command-line argument of “editor.” This unique architectural style is depicted in Figure 1.36. It permits the tools to have total access to the full range of data structures used by the engine and avoids a common problem of having to have two representations of every data structure—one for the runtime engine and one for the tools. It also means that running the game from within the editor is very fast (because the game is actually already running). Live in-game editing, a feature that is normally very tricky to implement, can be developed relatively easily when the editor is a part of the game. However, an in-engine editor design like this does have its share of problems. For example, when the engine is crashing, the tools become unusable as well. Hence a tight coupling between engine and asset creation tools can tend to slow down production.

1.7.5.1 Web-Based User Interfaces

Web-based user interfaces are quickly becoming the norm for certain kinds of game development tools. At Naughty Dog, we use a number of web-based UIs. Naughty Dog’s localization tool serves as the front-end portal into our localization database. *Tasker* is the web-based interface used by all Naughty Dog employees to create, manage, schedule, communicate and collaborate on game development tasks during production. A web-based interface known as *Connector* also serves as our window into the various streams of debugging information that are emitted by the game engine at runtime. The game spits out its debug text into various named channels, each associated with a different engine system (animation, rendering, AI, sound, etc.) These data streams are collected by a lightweight Redis database. The browser-based Connector interface allows users to view and filter this information in a convenient way.

Web-based UIs offer a number of advantages over stand-alone GUI applications. For one thing, web apps are typically easier and faster to develop and maintain than a stand-alone app written in a language like Java, C# or C++. Web apps require no special installation—all the user needs is a compatible web browser. Updates to a web-based interface can be pushed out to the users without the need for an installation step—they need only refresh or restart their browser to receive the update. Web interfaces also force us to design our tools using a client-server architecture. This opens up the possibility of distributing our tools to a wider audience. For example, Naughty Dog’s localization tool is available directly to outsourcing partners around the globe

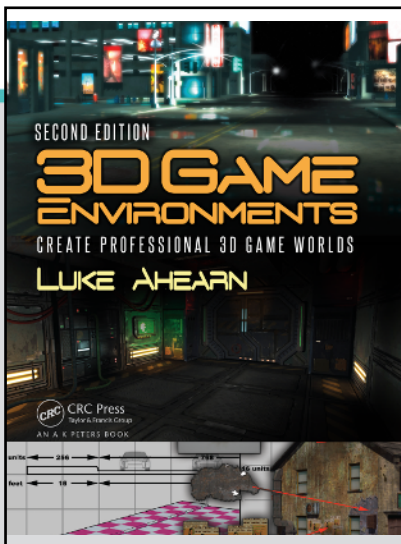
who provide language translation services to us. Stand-alone tools still have their place of course, especially when specialized GUIs such as 3D visualization are required. But if your tool only needs to present the user with editable forms and tabular data, a web-based tool may be your best bet.



CHAPTER

5

3D CONCEPTS



This chapter is excerpted from
3D Game Environments
by Luke Ahearn

© 2017 Taylor & Francis Group. All rights reserved.

 [Learn more](#)

Three-dimensional concepts

Introduction

This chapter is only an introduction to the concepts of three-dimensional (3D) modeling you will most likely be working with. Once you understand these concepts, you can more easily use the tools at your disposal to create the art in this book. For the details on how to do any of the specific functions for any given 3D application, you need to consult the documentation for that application. The good news is that, as game artists, we work in both two dimensional (2D) and 3D, but at a pretty basic level of functionality, so that you can easily achieve these results in virtually any 3D package.

A note on texture and polygon budgets: in environmental (and all game) art, we still need to control our asset budgets. Even though we are able to use much larger assets (polygon counts and textures), we don't want to use a high-polygon model if it is for a simple background world prop. By definition, environmental art is still easier and more basic than modeling characters, vehicles, or weapons. This is true for a few reasons. Environmental art needs to tile and be efficient. It cannot overshadow the characters and important features of the world. It's like formatting text. If every word in this book was bold and underlined, it wouldn't mean anything; it would only be annoying to read. Bold and italics are reserved for *special* words, very, very special words. So, too, must assets be reserved for special places and events in the game. The front of a castle may require more textures, polygons, and just plain artistic attention than the empty hut down the road. Likewise, a racing game is going to focus its assets on the cars and not the buildings blurring past in the background. This is not to say the buildings are not created to the same

standards as the cars, but simply that the car will require more resources to achieve the required level of detail.

Texture mapping

Although creating the texture is a 2D process, texture mapping—the process of applying 2D art to the 3D objects in the world—is part of the 3D process. Textures add a huge amount of detail and richness to a 3D model. A texture map is mapped (applied) to the surface of the shape in various ways, like wrapping a picture around a box to make it look like a specific object—a crate, for example (Figure 2.1).

In environmental art, we create a lot of 2D and 3D assets. Here, we will look at the most often-used 3D concepts you need to be familiar with. We will create the 2D art as well, but, for a deeper education on 2D art creation, please refer to my book *3D Game Textures* (Focal Press 2006).

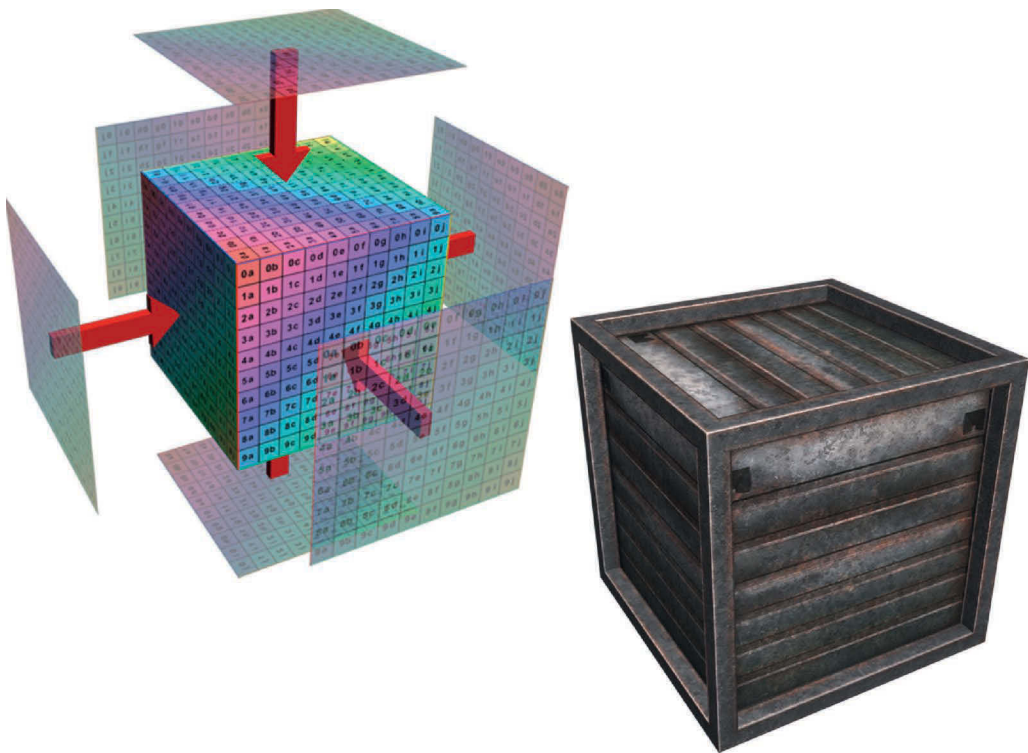


FIGURE 2.1 A texture map is an image that is applied to the surface of a shape, like wrapping a picture around a box to make it look like a crate in this figure.

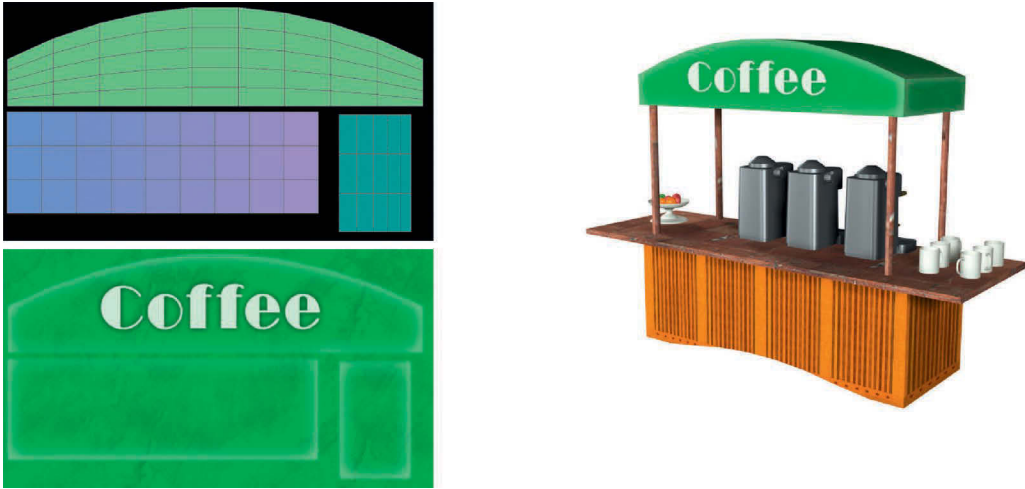


FIGURE 2.2 The process of UV texture layout and mapping.

We will address texture creation on a simpler level in this book, and we will turn our focus to the application of textures to the mesh. In *3D Game Textures*, the emphasis was strictly on 2D art and texture creation, but, here, we learn how to map the textures to the 3D assets we create and to use shaders. The creation of textures for the models in this book will be addressed on a case-by-case basis as we create them for each project. This includes the creation of the UV template and the application of the texture to the model (Figure 2.2).

Note that there is a distinction between a texture and a skin. A skin is the art that goes on a more complex model such as a character, a monster, or a weapon. Skins are generally not tileable and are created for a specific mesh. A texture is generally the art that covers the game world surfaces: grass, floor tiles, walls, and the like, and UV-ing these surfaces is much simpler than skinning an organic model.

Mapping types

When applying textures to a mesh in a 3D program, there are some tools that will apply the textures rapidly in a set fashion with the push of a button. While these tools are useful and quick, they have drawbacks that make them useless in some applications or, at the very least, produce results that must be cleaned up. Such push-button tools assume that the mapping type is applied to the entire mesh. However, mapping types become more useful when used on a face-by-face basis. We will get to

that level of mapping in the projects when we actually start laying out UVs. Right now, we will look at the push-button mapping types, since they are the basis for more complex mapping. They are as follows.

Planar

Planar mapping works like a projector. The texture is projected onto the 3D surface from one direction. This can be used on walls and other flat planar surfaces but is limited and can't be used on complex objects since the process of projecting the texture in one direction also creates smearing on the sides of the 3D model that don't face the planar projection directly (Figure 2.3).

Box

Box mapping projects the texture onto the model from six sides. This works great on boxes! Used on a more organic mesh, there will be seams and smearing on portions of the model (Figure 2.4).

Spherical

Spherical mapping surrounds the object and projects the map from all sides in a spherical pattern. The drawback to this mapping type is that you can see an edge where the textures meet unless you have created a texture that tiles correctly. Also, the texture gets gathered up, or pinched, at the top and bottom of the sphere and needs to be dealt with in the texture. Spherical mapping is obviously great for planets and other spherical objects (Figure 2.5).

Cylindrical

Cylindrical mapping projects the map by wrapping it around in a cylindrical shape. Cylindrical mapping can be used on tree trunks, columns, and other cylindrical objects (Figure 2.6).

UV editing

These mapping types are all limited in their uses. Later on, we will start the process of editing the UV coordinates for a model. This is the process you will use most of the time when mapping a texture to an object. It is a face-by-face process that can be tedious but is extremely important for the efficiency and quality of the assets.

Multitexturing

Multitexturing is the process of laying multiple textures on one mesh. This is powerful because you can mix and match many smaller, simpler textures over a surface to get a very wide variety of looks on your meshes. (See the "Multitexturing or Multiple UV Channels" section in Chapter 1 for a detailed explanation of multitexturing.) Figure 2.7 shows a simple example of multitexturing.

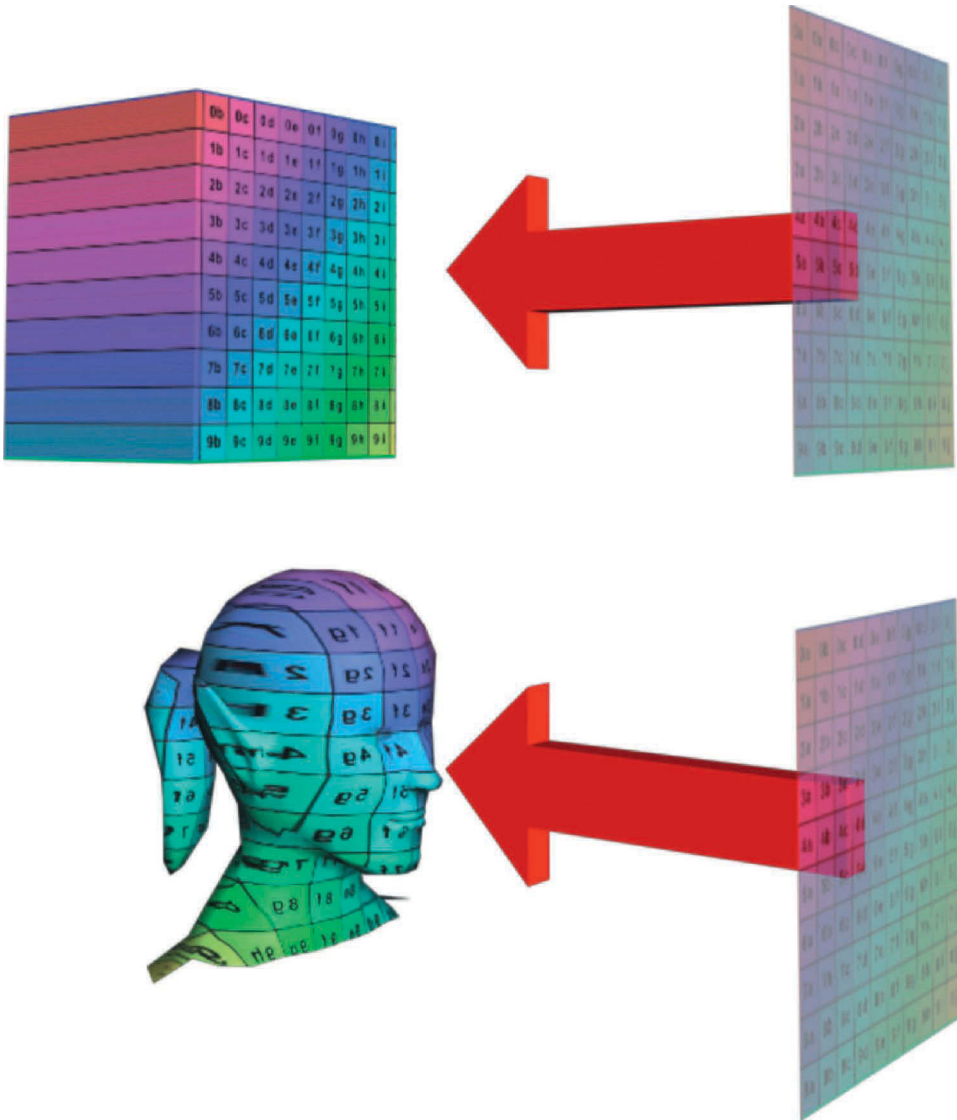


FIGURE 2.3 Planar mapping projects the texture onto the 3D mesh from one direction.

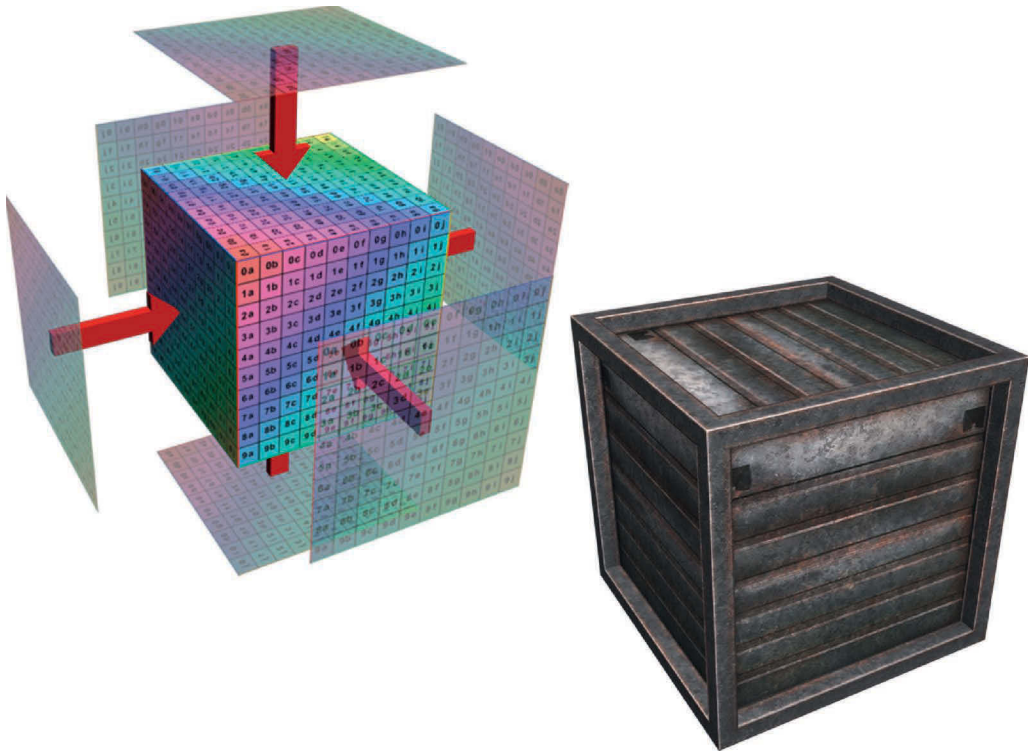


FIGURE 2.4 Box mapping.

3D

The very basics of 3D start with the vertex. The vertex is represented by a dot on screen, but, in reality, it is a mathematical location in 3D space defined by three numbers, or the xyz location. Three or more vertices connected to each other is a polygon. Many polygons together create a mesh (Figure 2.8).

You can edit 3D objects at many levels, from an individual vertex to an element. The basic parts of a mesh are as follows (see Figure 2.9):

- Vertex
- Edge
- Face
- Element

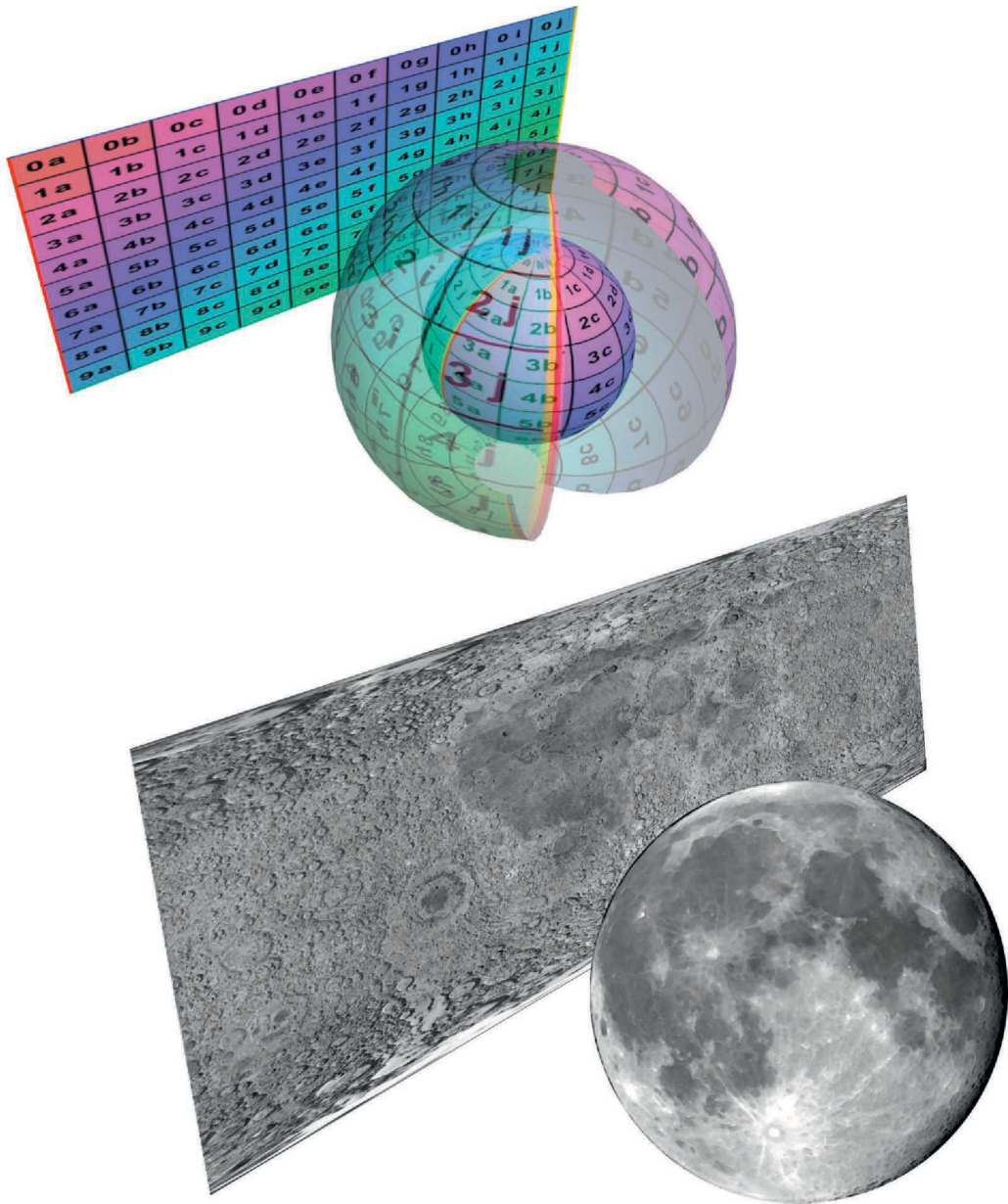


FIGURE 2.5 Spherical mapping.

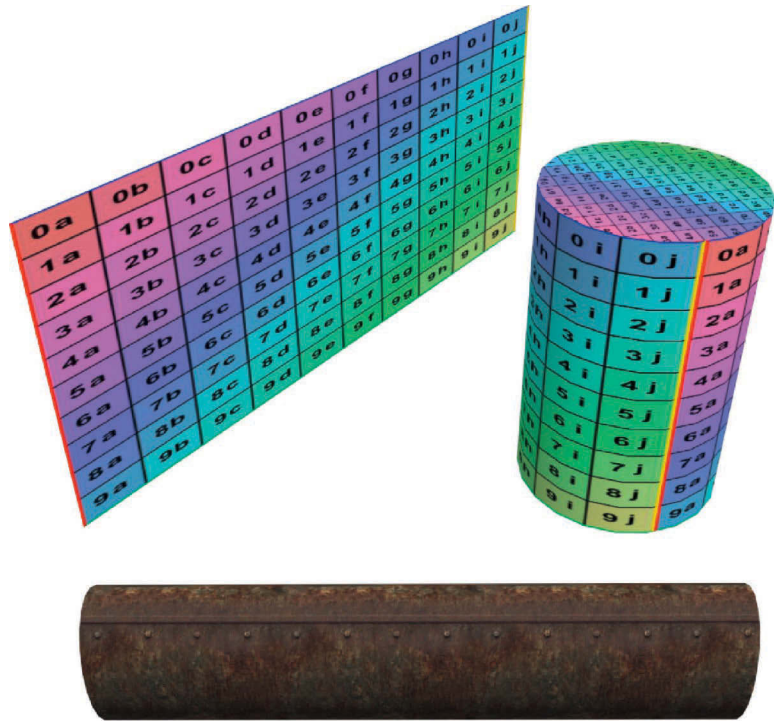


FIGURE 2.6 Cylindrical mapping projects the map by wrapping it around in a cylindrical shape.

3D space

When working in 3D on a computer, we are actually looking at 2D images that update fast enough that we feel as if we are actually looking at a 3D object when we move about it. That being the case, we need to utilize many tools and functions to help us overcome these limits. First, we will look at the little window into our 3D world, which is usually called a *viewport*.

Viewports

Viewports are like the portholes of a ship: tiny, restricted openings looking into to a much larger world. To overcome this restriction, we need to use any trick or tool we can. First off, buy as many big monitors as you can afford and your system can support. With the drop in the cost of flat screen monitors and video cards, it can be feasible to make this upgrade. You can put all your menus on one screen and work on art on another. I usually have my 3D application on one screen (say, on my left side) with



FIGURE 2.7 Multitexturing.

all the menus on the other (on the right) and Photoshop in the reverse order so I can quickly switch between the two applications.

Here are some other tips for working efficiently:

- Get used to working in one viewport at a time rather than four at once, if possible. This gives you a larger view of the world.
- Use hotkeys and shortcuts so you are able to gradually remove menu bars, your goal being to work in *expert mode* as often as possible where there are no menus on screen.
- Create custom menu bars if you need to use menu bars. Applications often come with bloated menus for many functions you may never use; these take up a large amount of screen space.
- Get used to switching viewports using hotkeys, so, if you need to line something up using a specific viewport, you can do so quickly and in full screen.

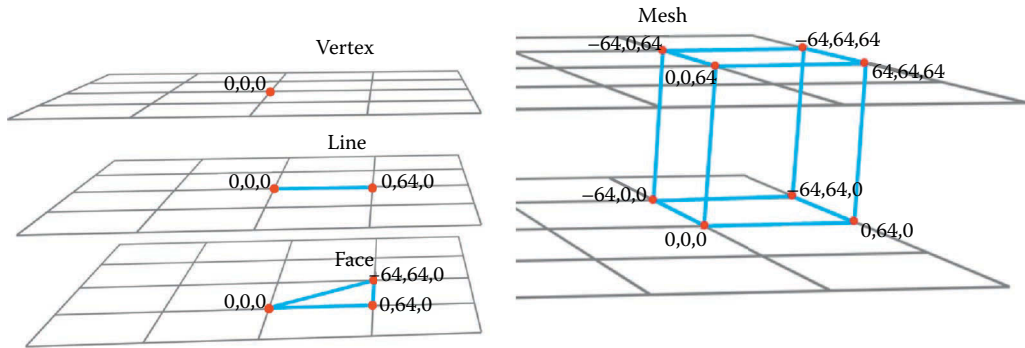


FIGURE 2.8 The creation of a 3D object.

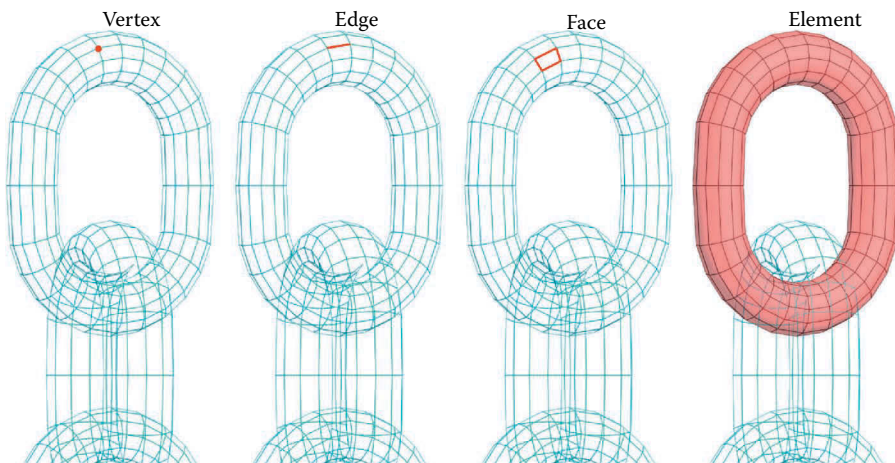


FIGURE 2.9 The basic parts of a mesh.

- Learn and use the zoom functions. This will allow you to quickly zero in on an object or back completely out for a bird's-eye view of the world.
- Learn viewport navigation, that is, the modes of moving around the viewports in the fastest, most efficient way. Some 3D programs have accelerator hotkeys that allow for a larger movement across the world when held down. For example, holding Ctrl when moving your view across the world may move you \times times (\times being any multiplier such as $4\times$ or $10\times$) farther per mouse movement.
- Don't forget the right-mouse click, or pop-up, menus. These can usually be customized as well.

Player perspective

Most 3D applications offer a walk-through mode, or, at the very least, the ability to set up a camera to look at the world from the position and at the focal angle the player will see it in the game. After flying around the world you are modeling, you may be surprised at how much you can or can't see at the player's level. The world will often appear much larger when you are taken out of the sky and put on the ground at the player's eye level. The focal length of the camera alone will have a dramatic effect on how large the world appears. It is common practice to give the game camera a slightly fish-eyed view to compensate for the limits of the monitor.

Shortcuts and hotkeys

Learn to use shortcuts and hotkeys! We all know what they are by now. Pressing a key or using a set of keystrokes is far faster than navigating the menus. Do anything you can to speed up your work. Learn to create macros, actions, and custom menus, and learn a new shortcut or hotkey combo every so often. An added benefit to this efficiency is reducing stress on your fingers and wrists. If you work at a computer all day, the number of clicks and mouse movements adds up. The cumulative wear and tear is tremendous when you start doing the math.

Units of measurement

When setting up to model items for a world space, you need to know the units of measurements in that world. What unit is used to communicate the size of an object: feet, units, or meters? And how do these units translate in the game? Later, we look at world scale and measurement in more detail and examine what we need to know in order to determine what our units of measurement are and what they mean.

It is common in most games to use generic units as a measurement, and those units tend to be in powers of two (16, 32, 64, 128). The units of measurement are important for consistency, accuracy, and technical efficiency. If you create a model and a texture for a game world using all the same units of measurement, things will come together much more smoothly. But what does a generic unit equal? It could be a foot or a mile—that will be determined as you develop a game.

Grids and snaps

The grid and the snap settings are related to the unit of measurement. Grids are just that—a grid of lines spaced evenly apart. You can set the spacing of the grid. This is handy when used in conjunction with snaps. A snap is a setting that controls how strongly your cursor will snap to a specific point. A strong snap setting will grab your cursor when you are close to a certain object and snap it precisely in place; a weaker snap allows you to get closer to the snap position before grabbing the cursor from you. That place can be defined by you, and, usually, you want your cursor snapping to the intersections of the grid or the nearest vertex. This is very helpful when creating world art, since you can snap a line or

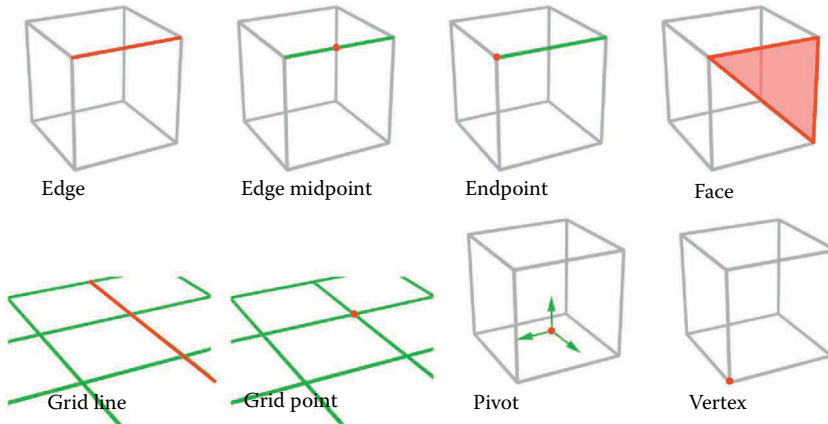


FIGURE 2.10 Various ways to precisely snap objects together.

a shape to a precise size on the grid, and that precisely sized object will easily fit a precisely created texture and then into a game world based on the same settings. This also speeds things up. Imagine trying to drag a shape out to a precise size or hand entering the sizes for every primitive you create. Some of the common snap types (Figure 2.10) are as follows:

- *Edge*—Snaps along an edge
- *Edge midpoint*—Snaps to the middle of an edge
- *Endpoint*—Snaps to the end point of edges on a mesh
- *Face*—Snaps to the surface of a triangular face
- *Grid line*—Snaps to any point on a grid line
- *Grid point*—Snaps to the intersections of a grid
- *Pivot*—Snaps to the pivot point of an object
- *Vertex*—Snaps to the vertices of a mesh

Note: In 3D applications, snaps can operate in different ways. Some modes offer a two-dimensional snap that only snaps to a specific grid or plane, or a three-dimensional snap where the cursor snaps to anything that is set for snapping on any plane. 3D snapping lets you create and move objects in all three dimensions.

Snap is also available to transforms as well. You can set the rotation or scaling to snap at certain angles or percentages.

Hide/unhide

You can also hide objects if you wish, and that makes working much easier. Not only is the scene clearer, but there will also be a smaller load

on the computer. If you are experiencing poor performance, you can hide objects to speed things up. There are usually multiple ways to hide and unhide objects: (a) using groups, (b) selections, or (c) entity types (lights, objects, and so on).

Freeze

Freeze allows you to freeze an object so you can't interact with it, but it is there for your reference. This can make life easier, as you won't be selecting unwanted objects as you work.

Drawing modes

You usually have options as to how the objects are drawn on the screen, meaning you can look at your world in wire frame, flat shaded, textured, textured and lit, and lighting only, among others. This is useful for many reasons; for instance, it allows you to examine the soundness of your geometry, see how the textured model may look in the game, or see what face you may have selected at the time (Figure 2.11).

Grouping

You can group objects together, which is very useful for scene management in large scenes. I strongly suggest that you name your groups well. As a scene becomes more complex, you can hide, select, and deal with a grouping of objects much faster than you can with a bunch of individual items. For example, you may have a large factory with control panels, several different piles of crates, gun racks, and other groupings of items in a large area.

Selecting

When you work in 3D, especially in a game world where multiple meshes or entities can occupy the same exact location as several others (in the case of collision and detail meshes), you need to learn to find what you are looking for quickly, isolate it, and leave everything else so others can find what they need when they work in the same space. For this, there are many tools that can help you. For almost any 3D application or world editor, you should have the tools to hide or unhide items quickly; freeze items (you can see them but not alter them, or they can't be unhidden but must be specifically unfrozen); and select items by name, mesh color, material, alphabetically, or by many other attributes. When you are working in a game world, as opposed to working on a 3D scene that contains only the art, you will have many other items potentially in view. These can include (but are not limited to) AI paths, event trigger markers, and many other game-related events such as power-ups, switches, particle effects, and player spawn points. When looking from any given view at your world, you will see hundreds or thousands of crisscrossing colored lines, and that is nothing but confusing. When working on your own, it is a good idea to get used to naming your meshes and groups at the very least; and, when working on a development team, it is very



FIGURE 2.11 Drawing modes let you look at your 3D world in various ways.

important to learn the naming conventions and other organizational conventions set forth by the developers.

3D creation

Now, we get to the concepts for the actual creation of 3D assets. There are numerous paths to the completion of a 3D model. As you learn these

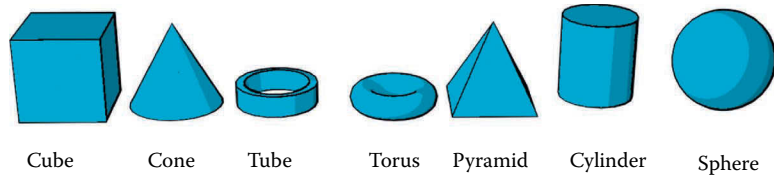


FIGURE 2.12 Basic 3D shapes.

tools, you will get better at knowing what path to take. Some methods may take longer or create messy geometry while being the perfect solution in another case. We start with basic shapes, called *primitives*, in 3D modeling. These shapes are so basic and common that there are dedicated methods to their creation. These shapes are usually the following (Figure 2.12):

- Cube or box
- Cone
- Tube
- Torus
- Pyramid
- Cylinder
- Sphere

Mesh editing

Some of the most common functions you will use in game modeling involve working with vertices, edges, and faces. For vertices, we can weld them together, break them apart, align them, and use a function called *soft select*. Soft select has a gradually lessening effect on the vertices surrounding the selected vertex. This is based on the parameters you set and is very useful for forming or tweaking terrain. Edge functions include chamfer and bridge, among others. And face functions (we will use them a lot) include extrude, bevel, inset, outline, and hinge from edge. Figure 2.13 illustrates some examples of various mesh-editing functions.

Axis

The axis shows the direction the coordinates are running in the 3D window. What this means is easier to understand if you know what the Cartesian coordinate system is. The Cartesian coordinate system is a method that determines where a point is in 3D space using three bits of information—the x, y, and z coordinates. The standard locations for these are as follows:

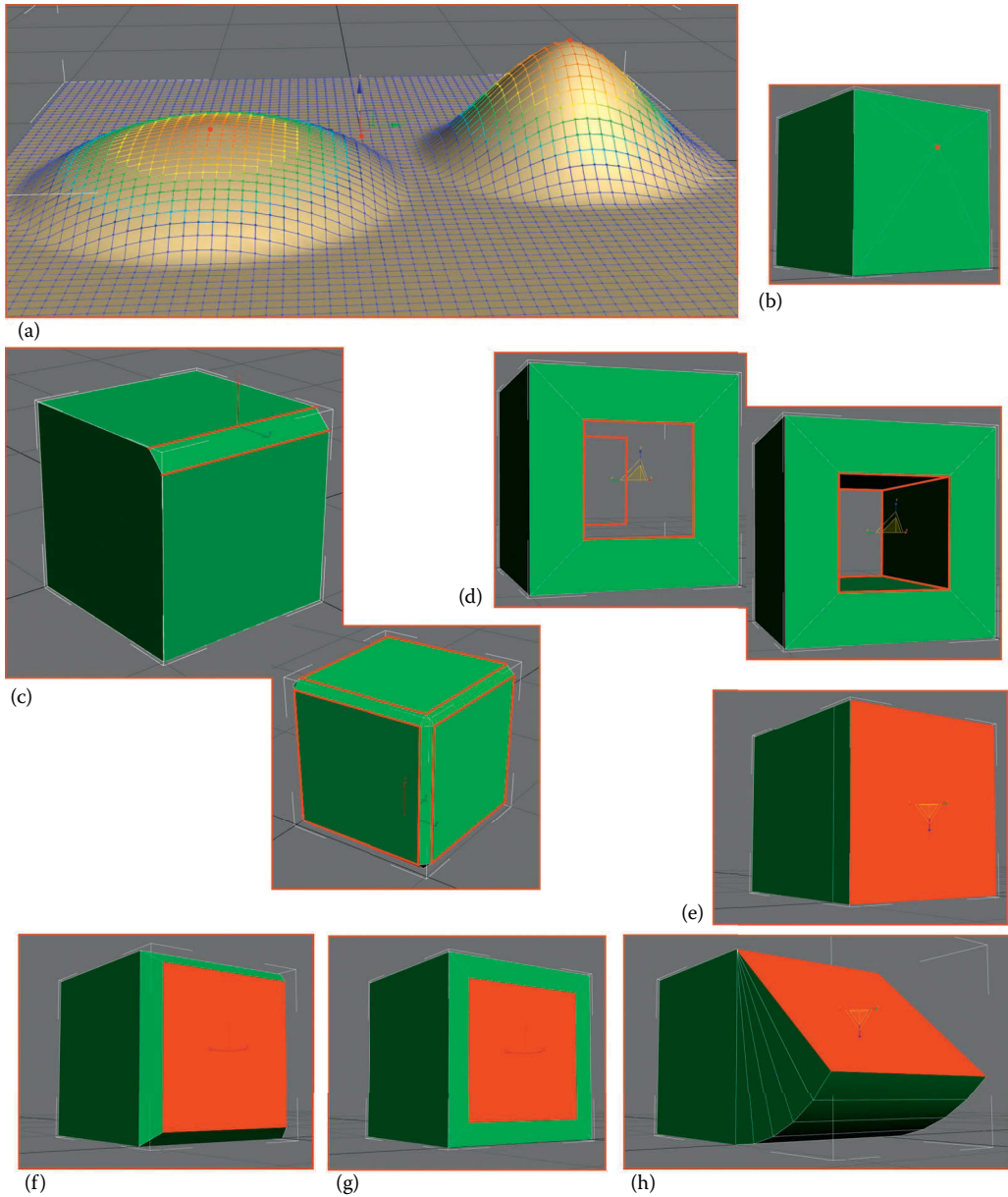


FIGURE 2.13 Mesh-editing functions: (a) vertex soft select, (b) add vertex, (c) chamfer edge, (d) bridge, (e) extrude face, (f) bevel face, (g) inset face, and (h) hinge from edge.

- x = left to right
- y = up and down
- z = front to back

Using the grid makes it easier to navigate 3D space. The starting (origin) point is $0,0,0$. Any movement away from this point will result in a negative or positive value in one of the x , y , or z values (Figure 2.14).

There are many ways to view a coordinate system (Figure 2.15). A few of the most common are as follows:

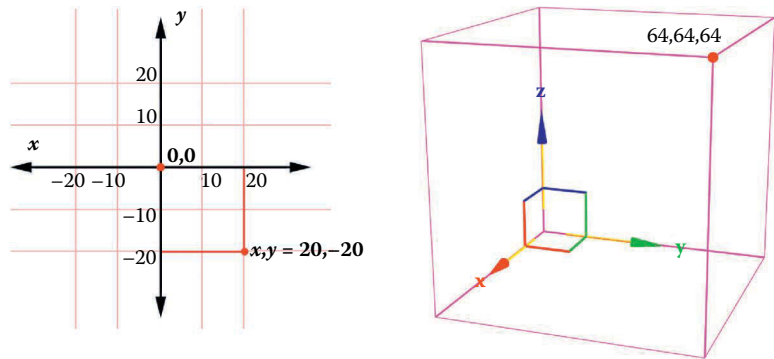


FIGURE 2.14 Coordinate system.

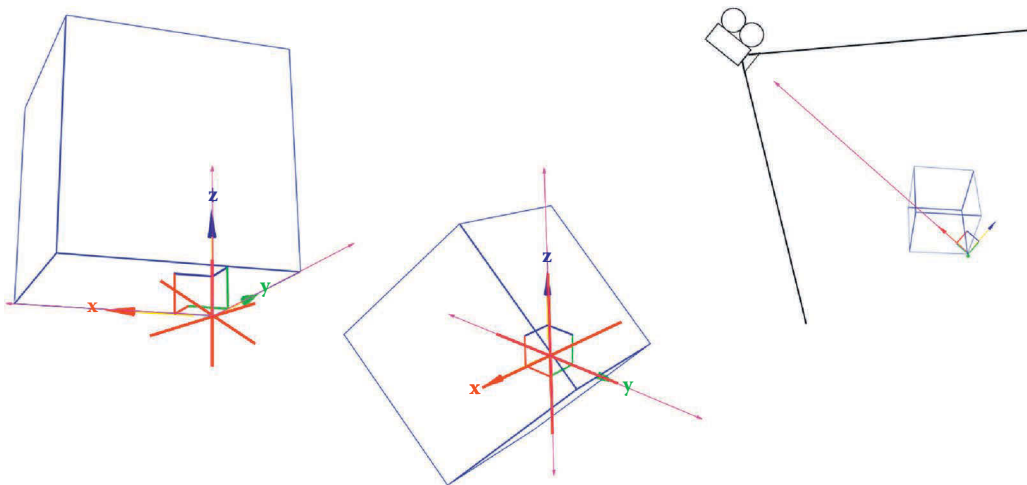


FIGURE 2.15 Local space, world space, and view space.

Object space or local This uses the xyz coordinate system of the selected object. An object's coordinate system is held by its pivot point. You can actually edit the local coordinate system, moving and rotating how the axis points are orientated.

World space This system is fixed and centered in the world despite your view or the objects involved. All vertex data are based on the coordinate system of the world, which originates at 0,0,0.

View space The coordinate system dynamically moves as the view or the camera moves.

Pivot points

An object's pivot point is the location at which the object rotates. In Figure 2.16, you can see some examples of how moving the pivot point affects the movement of an object. The pivot point can also influence how modifiers and transforms affect a mesh (Figure 2.17). It is easier to see the effects in the 3D application where you can see the changes in real time.

2D shapes

You can draw a line in a 3D space and create a 3D object from it. This is a great way to start odd shapes. There are 2D primitives as well, just like 3D primitives. You can start with a predefined shape or a shape created from a line. Most splines are controlled at their vertices in four ways (Figure 2.18).

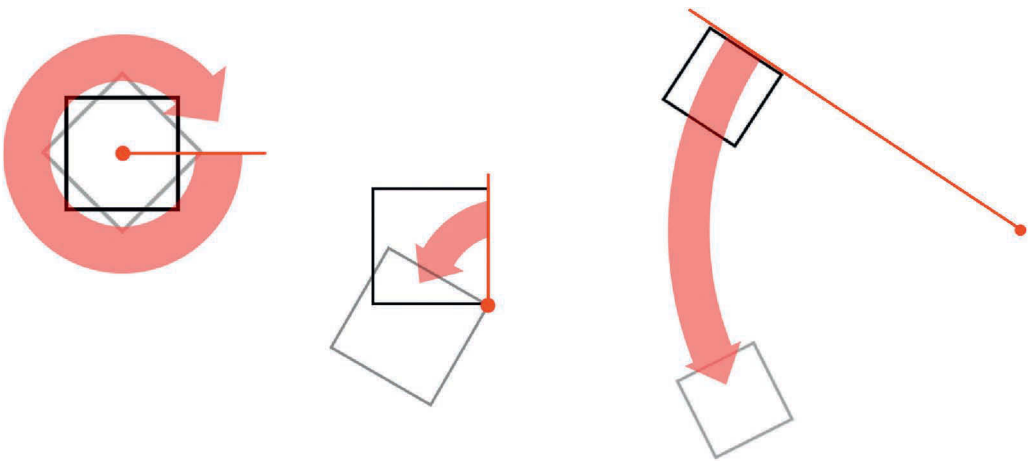


FIGURE 2.16 An object's pivot point is the location at which the object rotates; you can see how moving the pivot point affects the movement of an object.

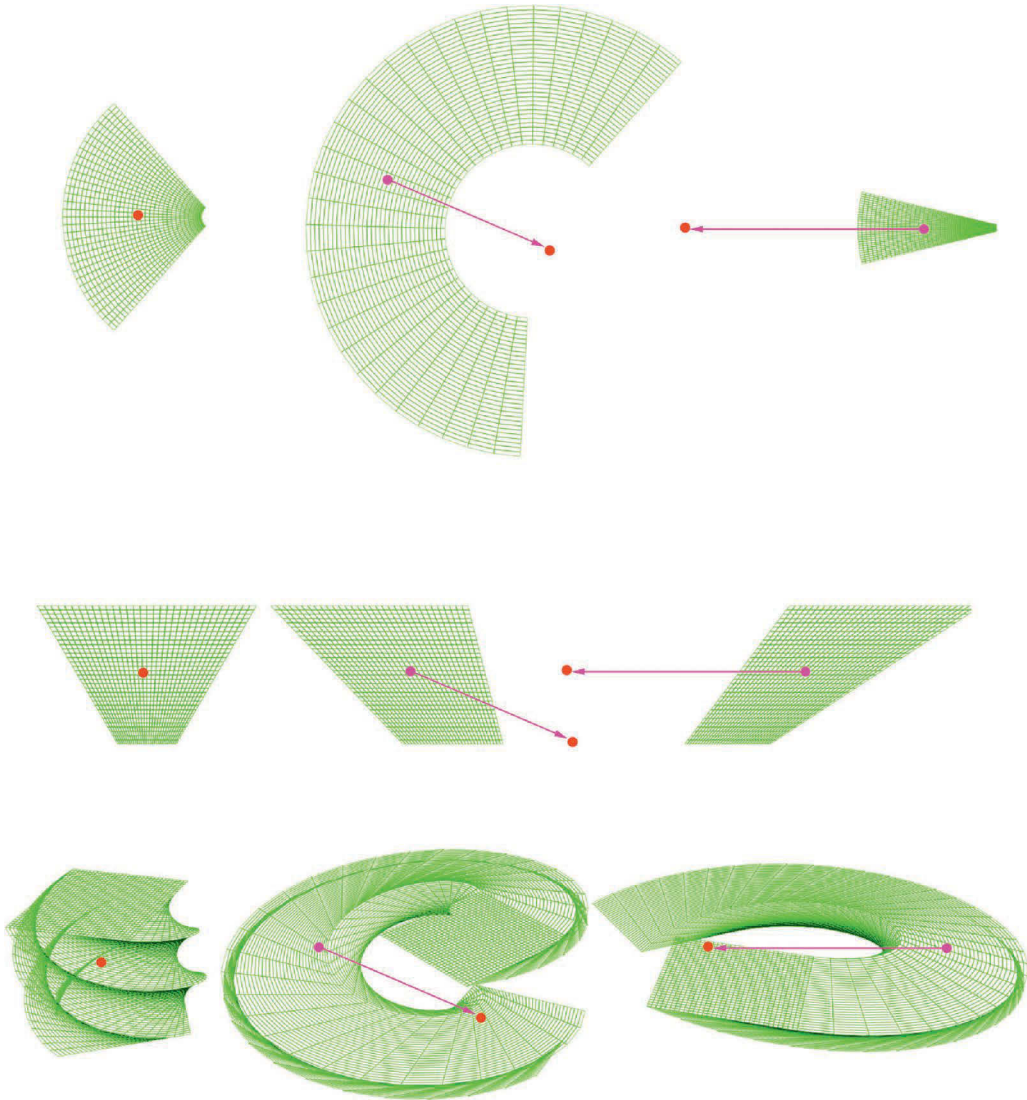


FIGURE 2.17 The pivot point influences how modifiers and transforms affect a mesh.

- *Bezier corner*—Each handle operates independently, resulting in a peak between curves.
- *Bezier curve*—Handles will operate together to create a smooth curve.
- *Linear*—No handles; this is a straight, linear corner.
- *Smooth*—No handles; this is a smooth curve of predetermined angle.

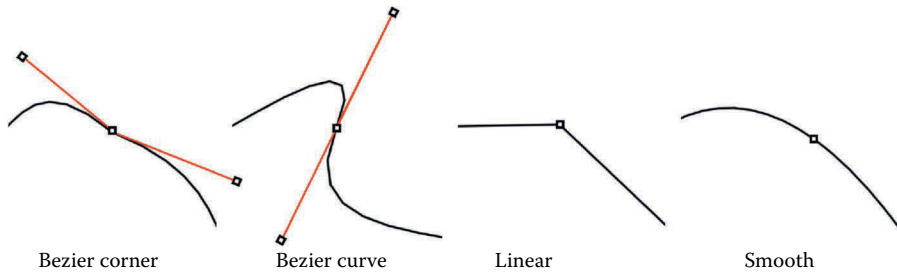


FIGURE 2.18 Spline control.

The great thing about splines is their editability. You can convert between the corner types, add and delete points, and even perform Boolean operations on them.

Creating 3D objects from 2D shapes

After you have created your spline path, you can perform several procedures on them to create a 3D shape. These include extrude, lathe, and loft.

Extrude

Extrude adds depth to the 2D shape (Figure 2.19).

Lathe

The concept of lathing is from woodworking. A lathe is a tool that rotates a block of wood very fast while a sharp tool is placed against the wood.

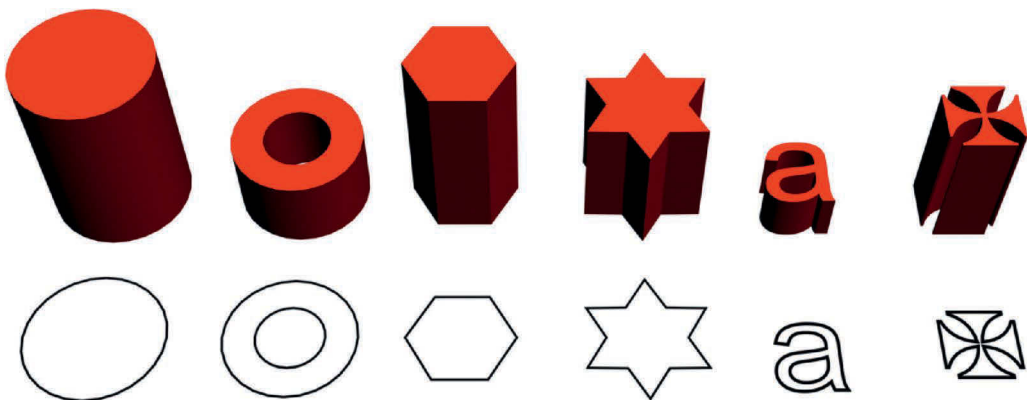


FIGURE 2.19 Extruding 2D shapes.

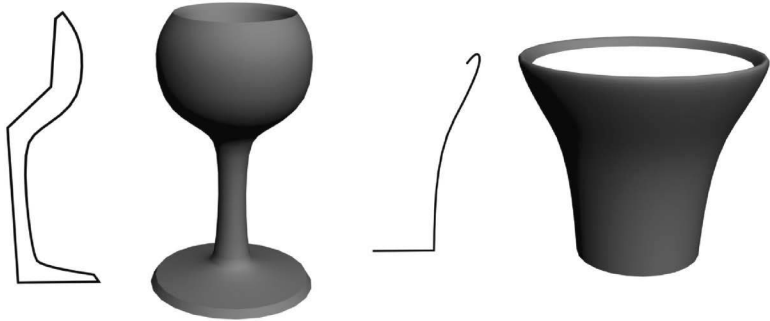


FIGURE 2.20 Lathing.

The carving action creates radial cuts in the wood. You can use a lathe in 3D to create goblets, vases, and other symmetrical geometry. We will use the lathe to create the body of a fire hydrant in the coming exercises (Figure 2.20).

Loft

Lofting is like extruding a shape, except that it follows a spline path. In games, this is great for pipes and hoses (Figure 2.21).

Transforms

As you would expect, transforms transform an object. You can alter the size and position of an object by moving, scaling, or rotating it. See Figure 2.22 for some visual examples of transforms.

Finally, there are numerous ways to copy and align objects. In addition to the basic transforms, you will probably use mirror and align

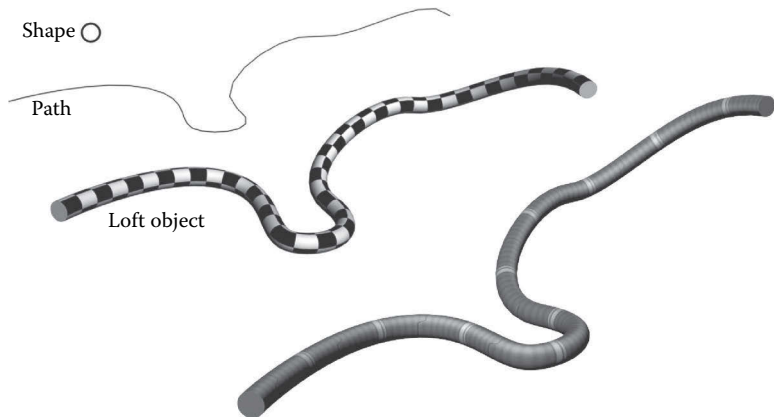


FIGURE 2.21 Lofting.

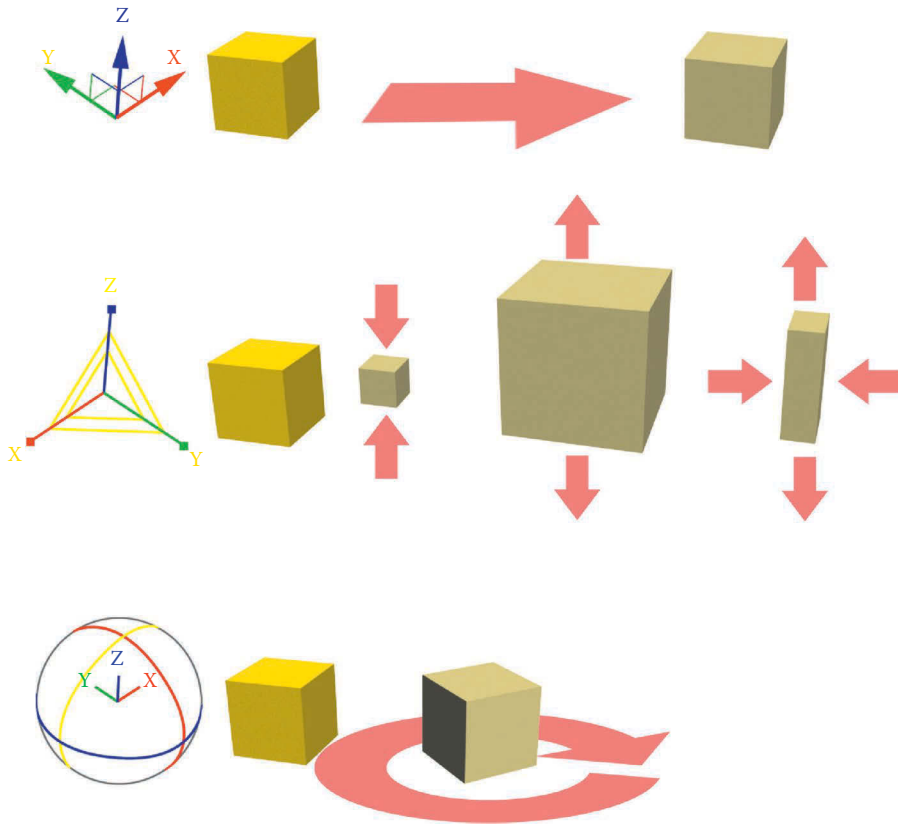


FIGURE 2.22 Transforms.

functions a good bit. Mirror flips an object on a set axis. If you model a car, you don't model the entire model; you model half of it and mirror it and attach the two halves together (Figure 2.23). Alignments combined with copy functions can create a perfectly spaced row of columns or a rubble of stones across a ground surface.

Deforms

Deforms alter or deform a mesh. You can bend, twist, taper, ripple, and even free-form deform a mesh. There are many deforms for meshes. See Figure 2.24 for examples of deforms and their effects on a mesh.

If you understand these basic concepts, you can quickly learn how to accomplish them in your 3D application of choice. Once you know that, you will be able to create most simple game art and will have a solid base

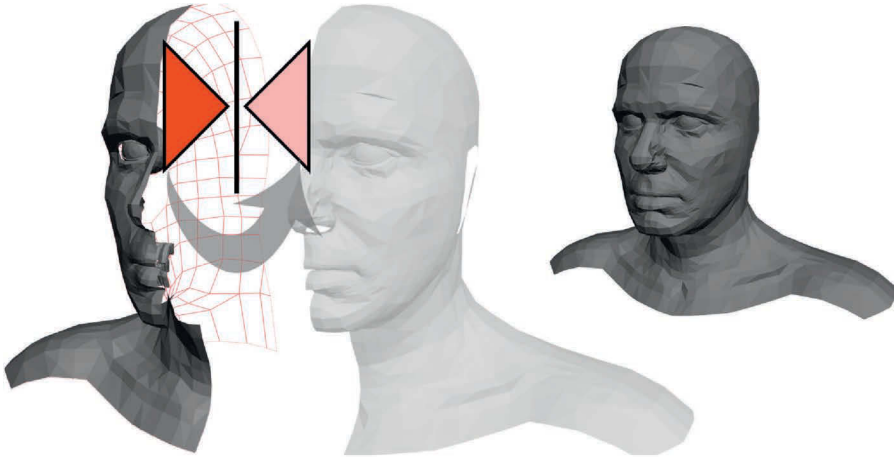


FIGURE 2.23 Mirroring.

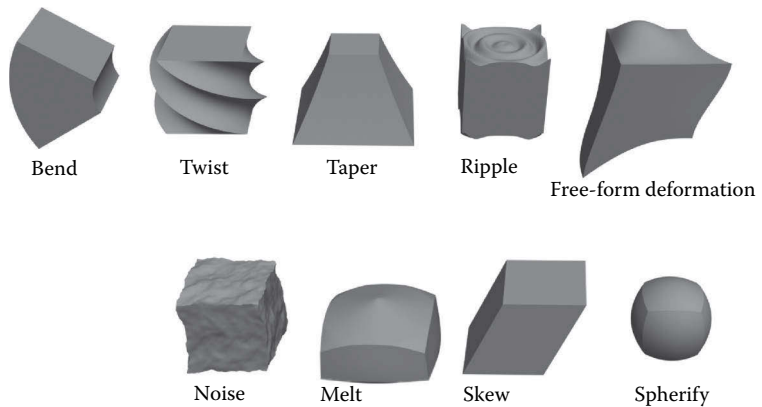


FIGURE 2.24 Deforms.

from which to further learn on your own. In the next chapter, we will look at shaders. Shaders are an exciting topic, as they add so much visual depth and immersion to a scene. Although shaders can get complex, the basics are easy to learn and implement and have a huge impact on the visuals of a game world.



Taylor & Francis

Taylor & Francis Group

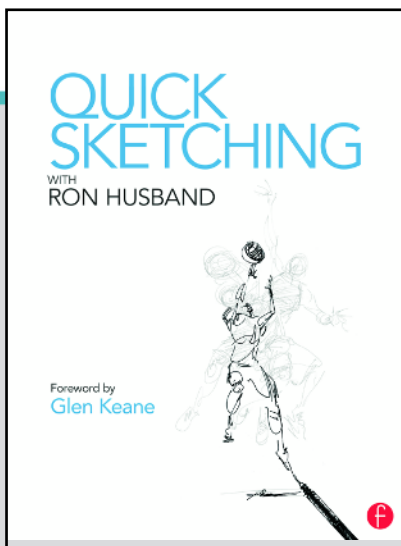
<http://taylorandfrancis.com>



CHAPTER

6

THE BASICS



This chapter is excerpted from
Quick Sketching with Ron Husband
by Ron Husband

© 2013 Taylor & Francis Group. All rights reserved.

 [Learn more](#)

1 The Basics

My definition of a “true” quick sketch is simply “a sketch done quickly” (i.e., an image captured from a live object on paper in about twenty to thirty seconds). My quick sketches differ from typical classroom quick sketch/gesture drawings that are usually done in one- to two-minute poses by a still or posed model. Typically, “built in” to the models’ poses are balance, perspective, proportion, silhouette, etc. for the duration of the pose. Most models work hard to give the artist the best poses possible and the results are “frozen in time” for the artist to capture everything on paper.

It is my belief that in order to capture a true quick sketch, one must also capture the action seen with the eye. The eye then relays what was observed to the brain and is coupled with real life observations, action analyses, photographs, books read, and any other tips of wisdom from past and present art instruction, and flows out the end of one’s pen or pencil. This goes beyond drawing a “posed pose” to capturing on paper what was there, never to be repeated again in the same way.

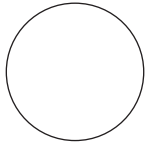
Therefore quick sketches should depict plainly and without question what is going on, the *who*, *what*, *when*, *where*, *why*, and *how*. It may not necessarily answer the when or where, but one look should tell you *who* (male, female, young, or old), *what* the captured subject is doing, and *how* it’s being done. This practice has led to my lifelong love affair with quick sketching. My sketchbook is so much a part of me I feel I’m not fully dressed unless I have my sketchbook in hand.

What Motivates You?

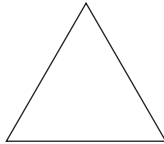
Find something you love to draw: men, women, older men, older women, children, jockeys, horses, you name the living subject and you can take a lifetime observing them. Because they are living and breathing, they will all walk, run, crawl, jump, or drag along in a variety of ways depending on location, motivation, and any other factors that happen at a particular moment in time. Houses, man-made objects, bowls of fruit, etc. can be drawn or painted in marvelous ways and they will always be there in the same lifeless way until some outside force moves them. Living humans and animal life are constantly animating “a life story,” waiting to be stamped on paper by your pen or pencil.

Simple Shapes

Think of the subject in simple shapes. Observe the objects around you: tables, chairs, televisions, books, etc. all consist of shapes—circles, ovals, squares, rectangles, or some combination of these basic shapes.



Circle



Triangle

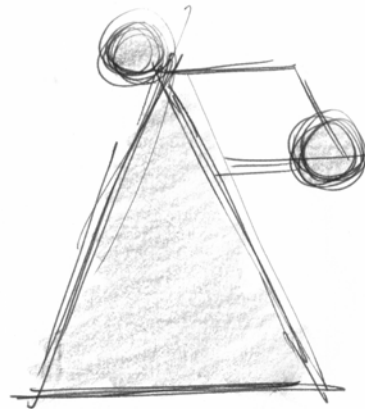


Square



Rectangle

Can you see the shape in your subjects?

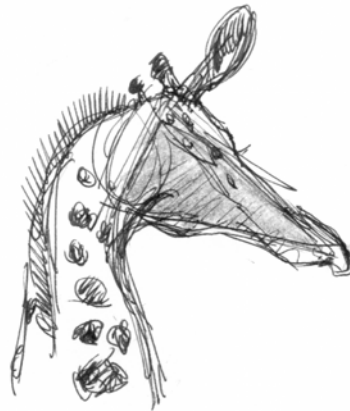
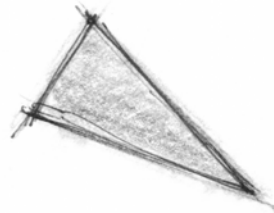
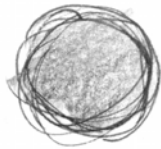
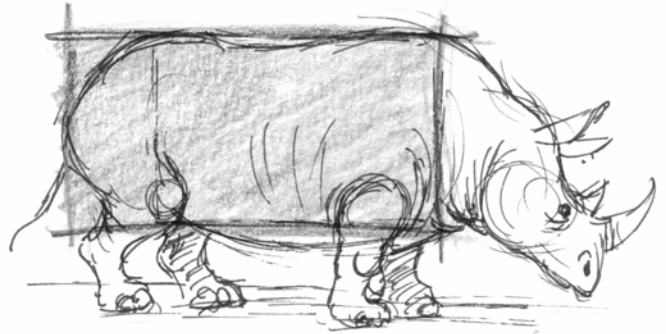
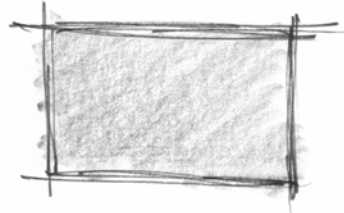


TIP 1

I use an assortment of different sketchpads. My everyday sketchpad is 8½ by 11 inches and ranges from inexpensive paper to a more expensive acid-free, white stock pad. When I am going to a formal gathering a regular pad of paper would look out of place (such as a formal wedding or retirement party) so I use an 8½ by 6 inch fine leather-bound book.

I also like to use a black ink pen. I've tried other pens but the Pilot V5 gives me a nice thin line without smearing or running.

Once I have my paper and pen ready, I usually sit down and do some practice drawings. (I think about the basic shape: circle, triangle, square or rectangle.)



People and animals are made up of similar shapes, with smooth edges; so again, look for these basic shapes in the objects you draw.

Observing these shapes in your subjects helps develop “an eye” for making identifiable shapes quickly. Remember, we are laying the foundation for quick sketching. Observation is key to analyzing any action or activity. Making mental notes of how someone does a specific action pays dividends when it is time to put pencil to paper. Artists must become people watchers; recognizing the differences in those who make up the human race. As bird watchers note the peculiarities of how one species of fowl behaves juxtaposed to another, we’re drawn to the realization that there are as many ways of doing things (i.e., walking, standing, gesturing, etc.) as there are people on the face of the earth.

Body Types

There are as many different body types in humans (and animals) as there are individuals on this planet. The one you choose to commit to paper must convey enough information to tell the viewer who it is, their age, what they are doing, and take into account any interesting body characteristics.

The ancient Egyptian artists were greatly aware of giving the most information possible to the viewer. By looking at most two-dimensional Egyptian artwork, it is not uncommon to see a straight-on eye in a profile head. This gives more information about the eye than a side view can and the profile head gives more information than a straight-on view of the head. For example, the straight-on head cannot tell you the shape of the nose or how far the brow hangs over the eyes or the shape of the chin. By using a straight-on view of the upper torso, combined with a profile of both legs, the ancient artist has attempted to provide us with the greatest amount of information about the figure.

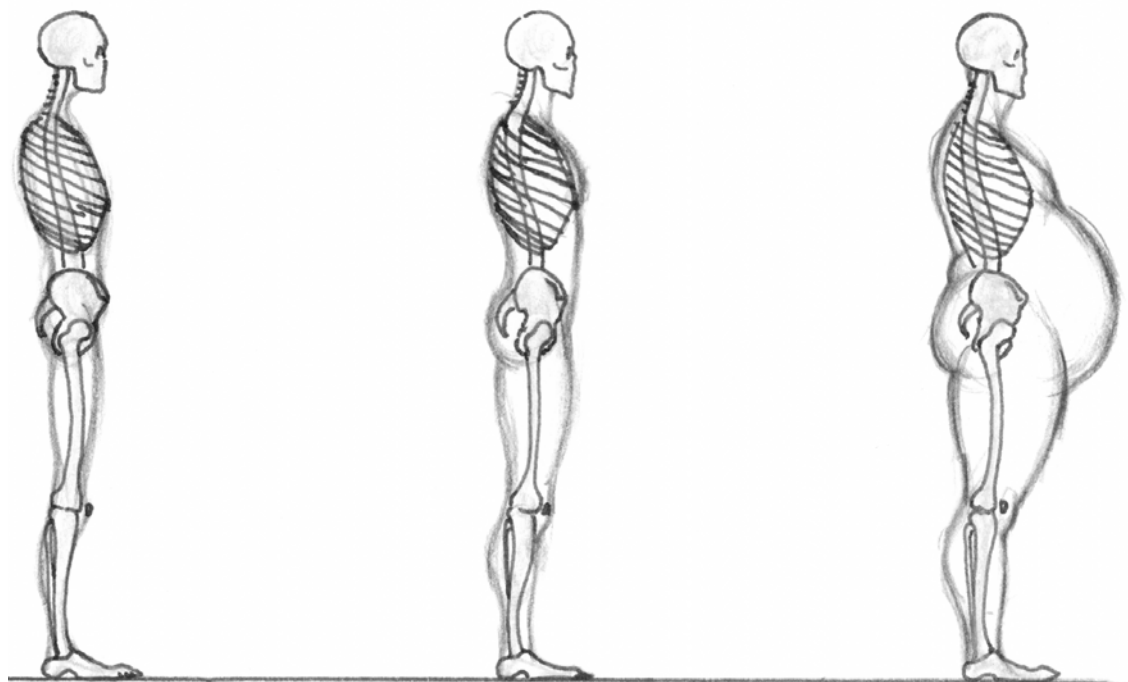


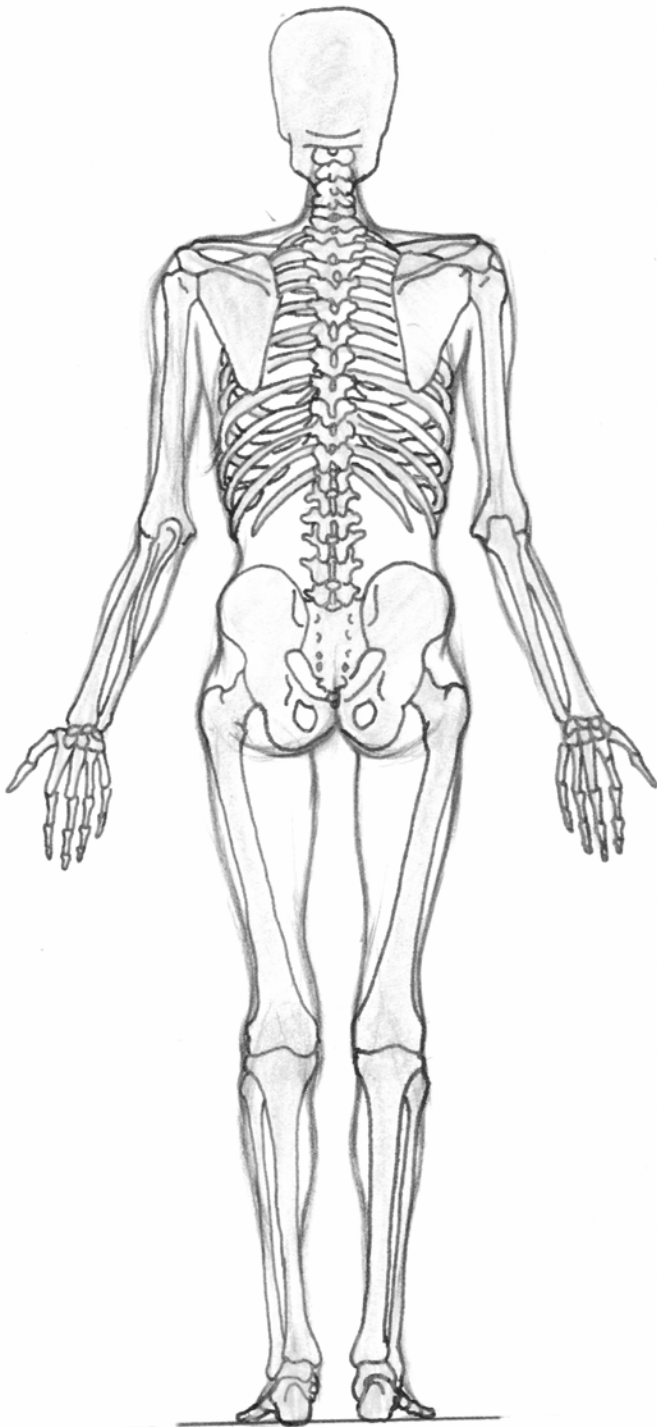
Quick sketching is an attempt to get this kind of information on paper in as short a time as possible and have a drawing that conveys enough information to be easily recognizable; a sizeable task, but one that can be mastered through the discipline of constantly drawing, drawing, and drawing some more.

A good drawing is one that utilizes perspective, proportion, weight, and balance to convey to the viewer its own unique story. In order to accomplish this, it is important to have a working knowledge of the skeleton and muscles of the human body.

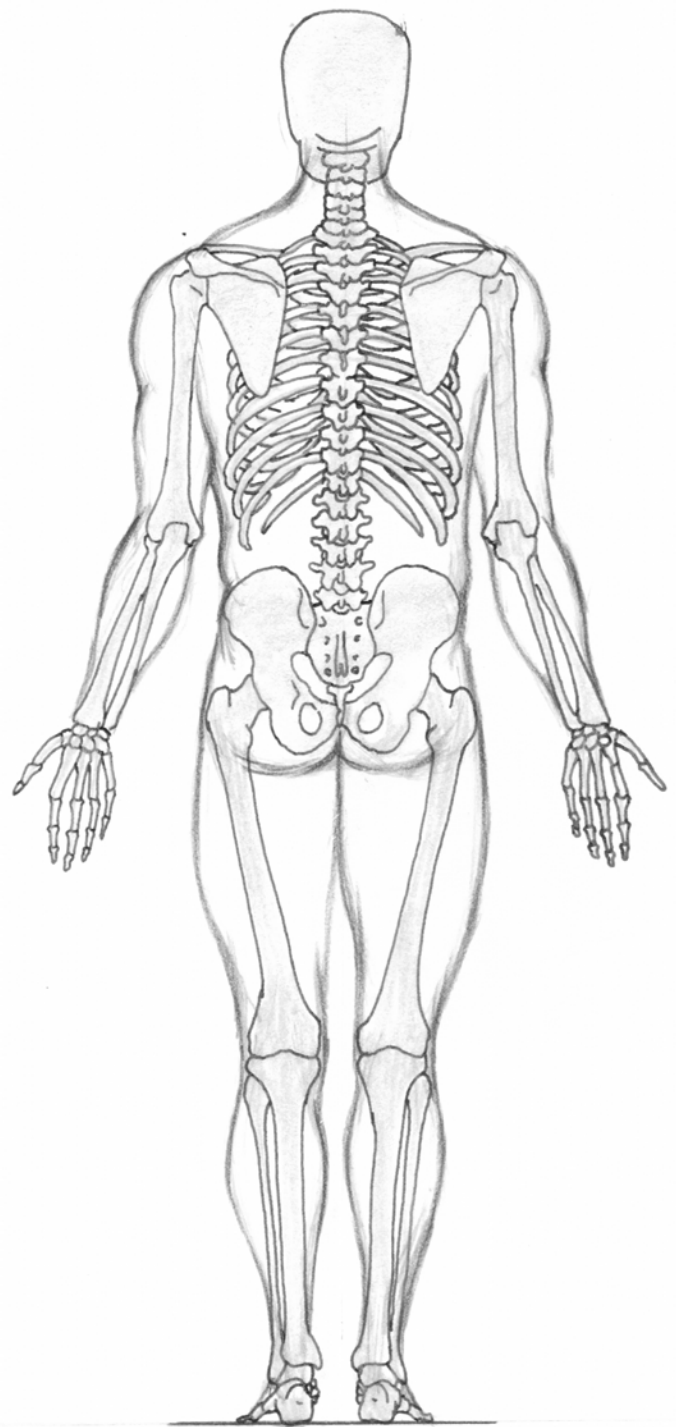
Skeleton/Muscle Tone/Fat

Think about the many body types you see daily. Not the TV/movie, photoshopped, magazine bodies, but the ones you see in everyday life. Keep in mind the one you choose to commit to paper is "one of a kind" even though he/she has the same body parts as everybody else.

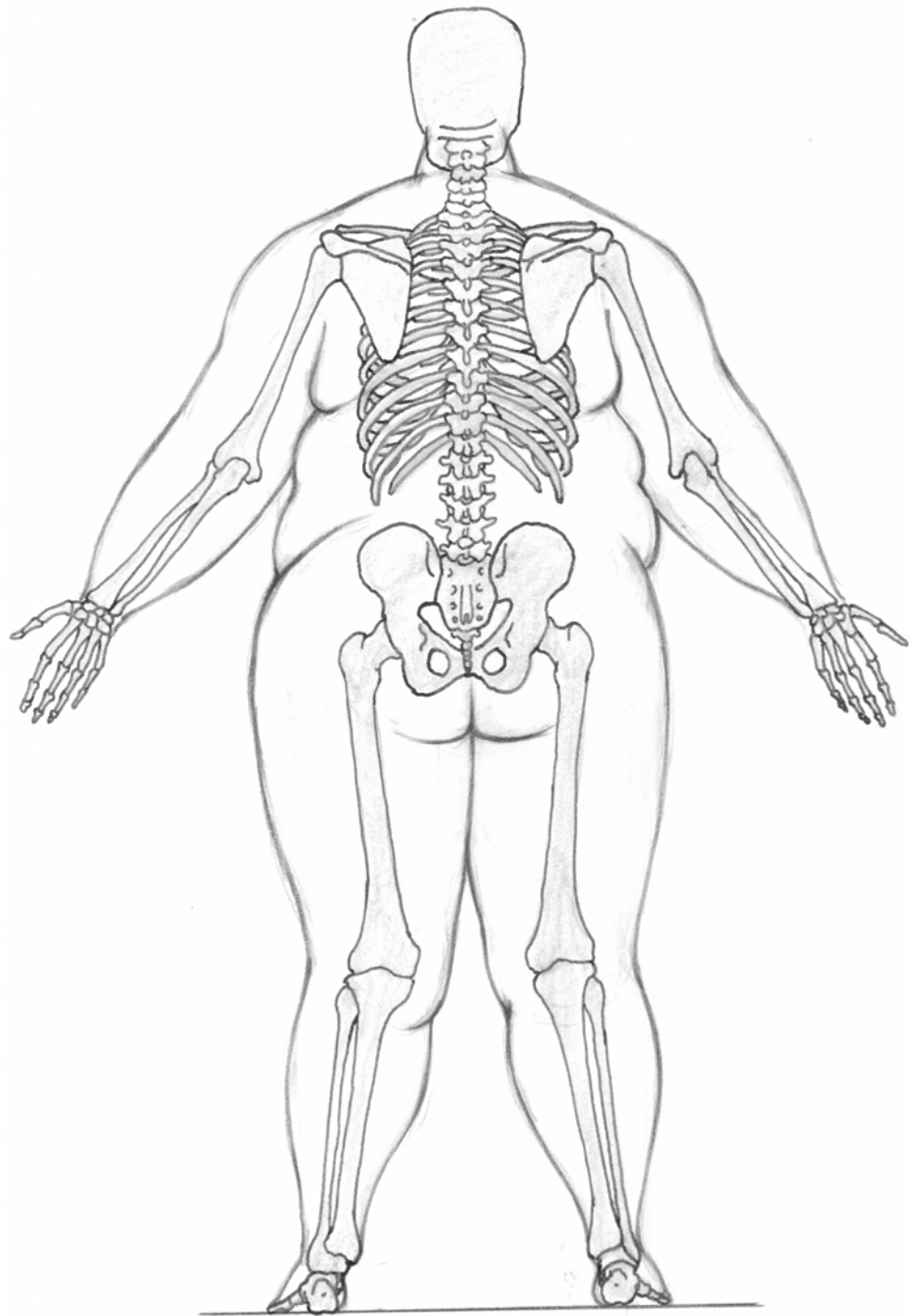




Thin body type



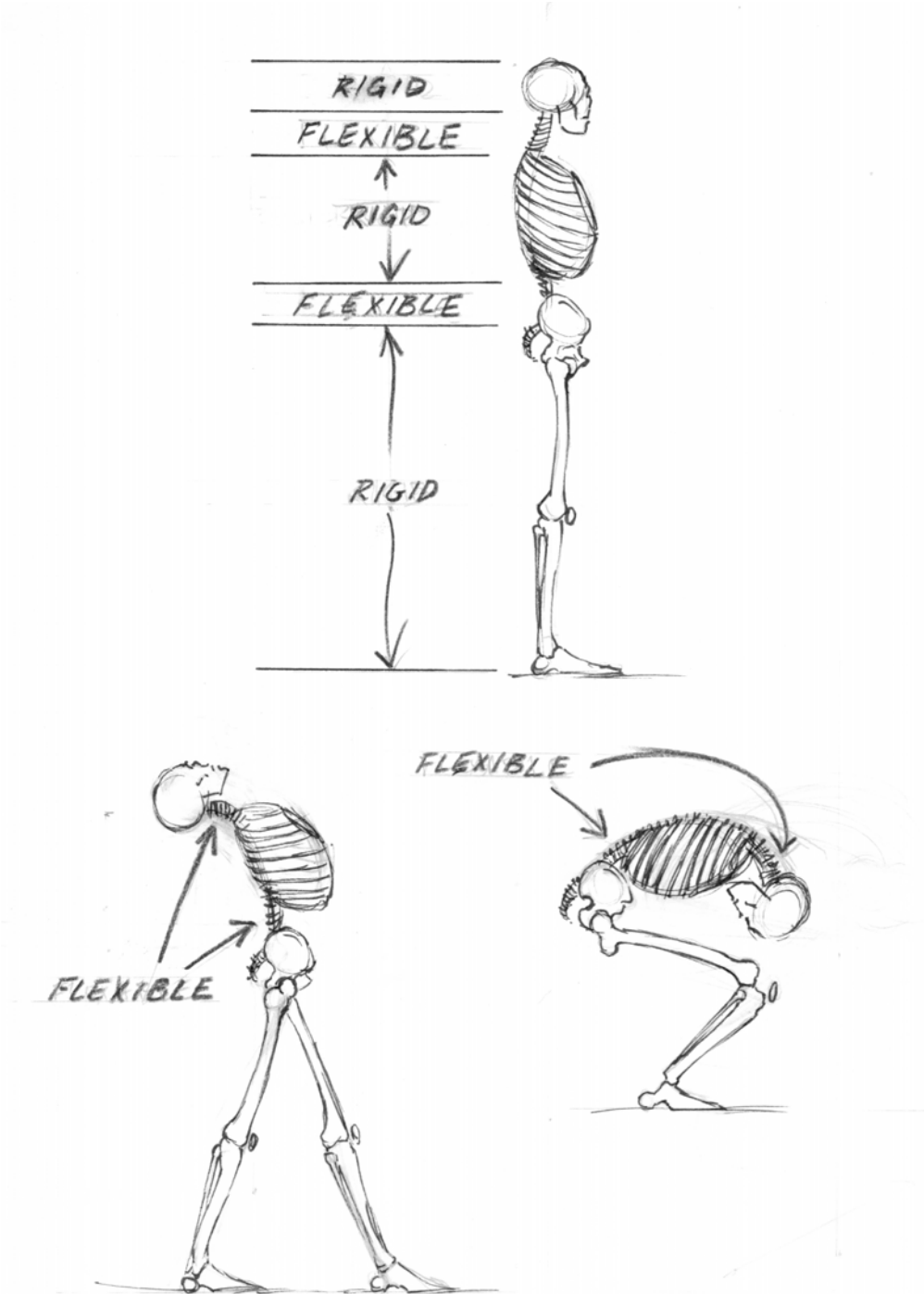
Muscular body type (skeleton remains unchanged)



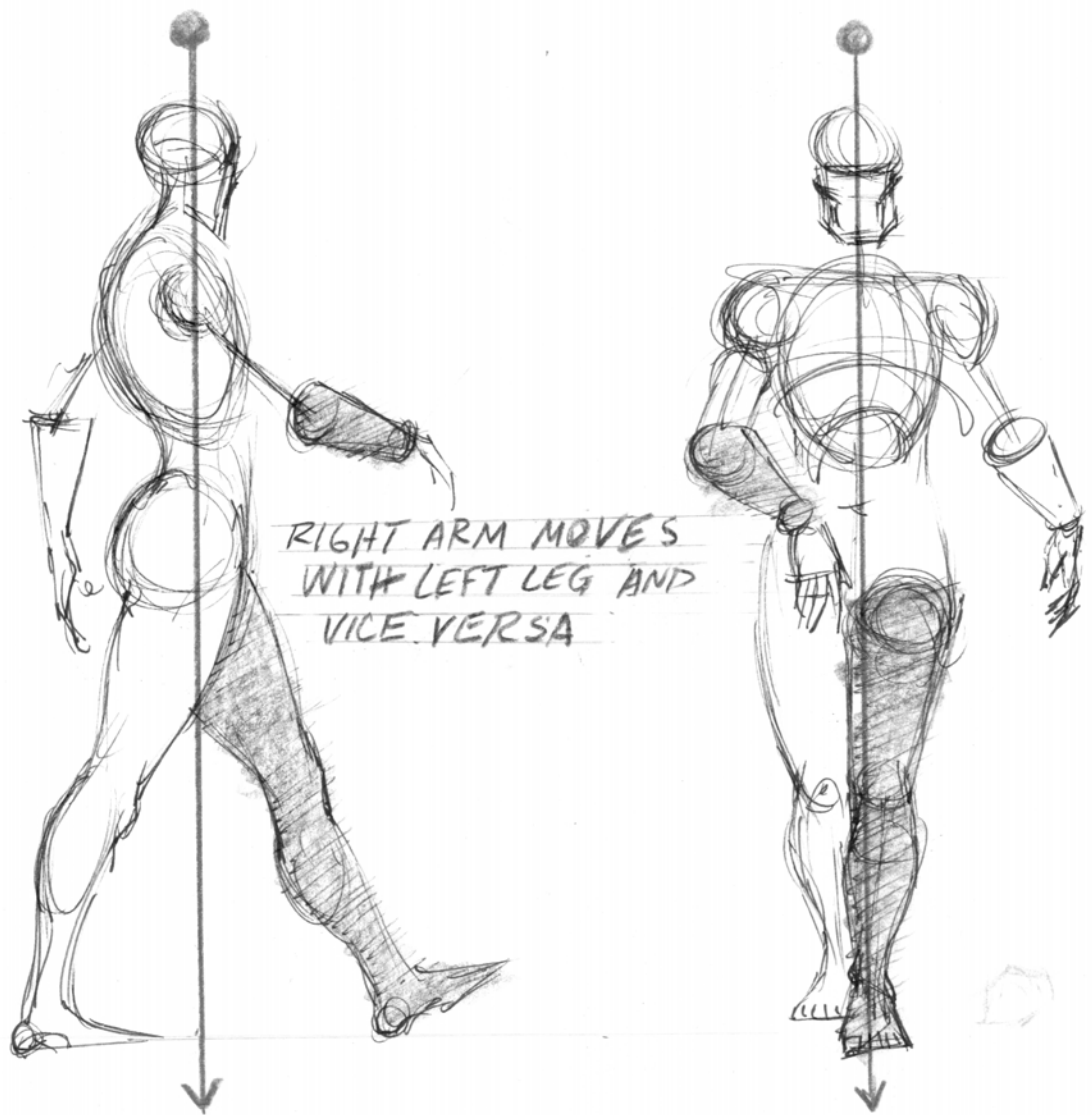
Heavy body type (skeleton remains unchanged)

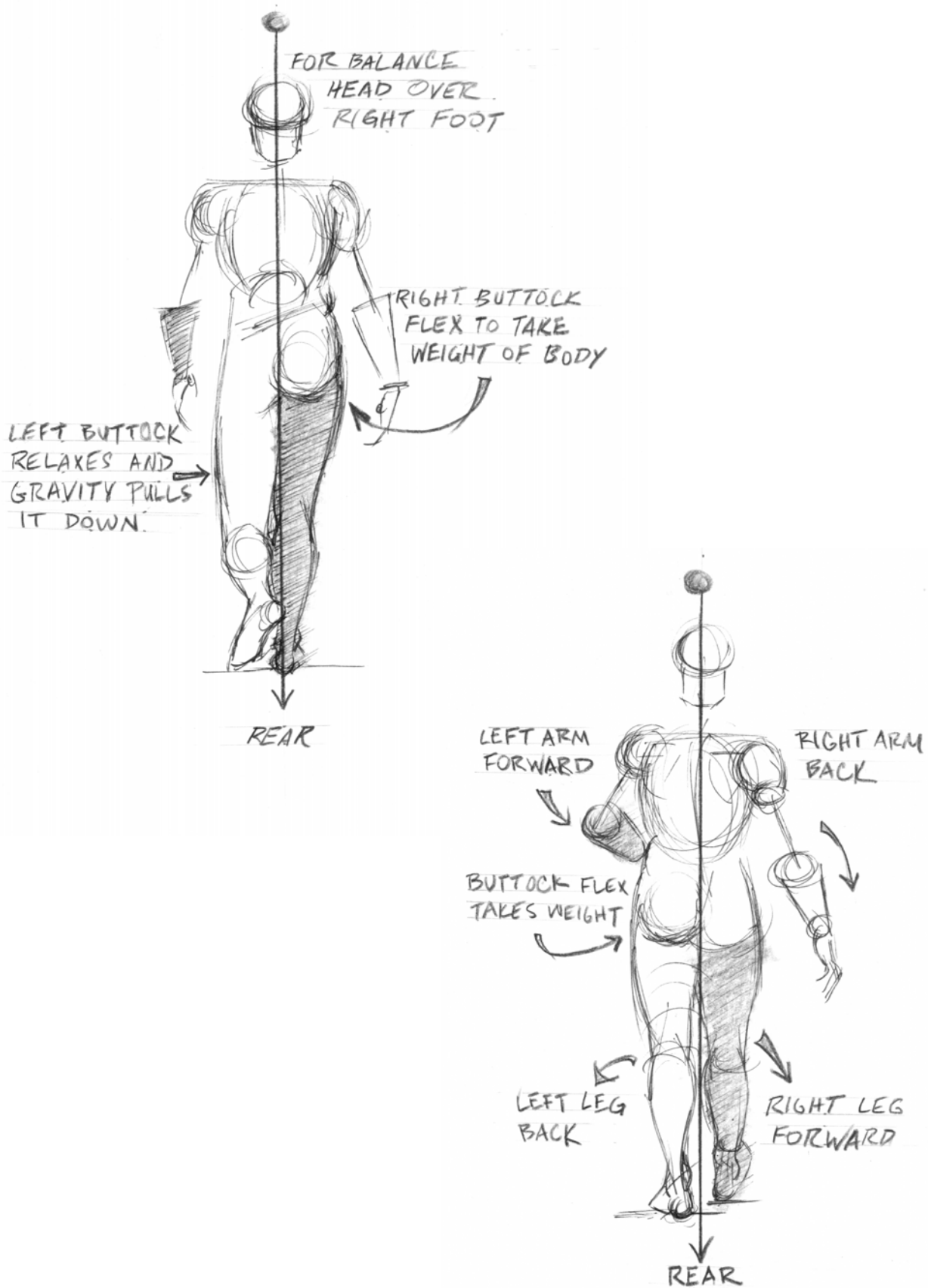
Movement and Balance

Keep flexible parts flexible and rigid parts rigid



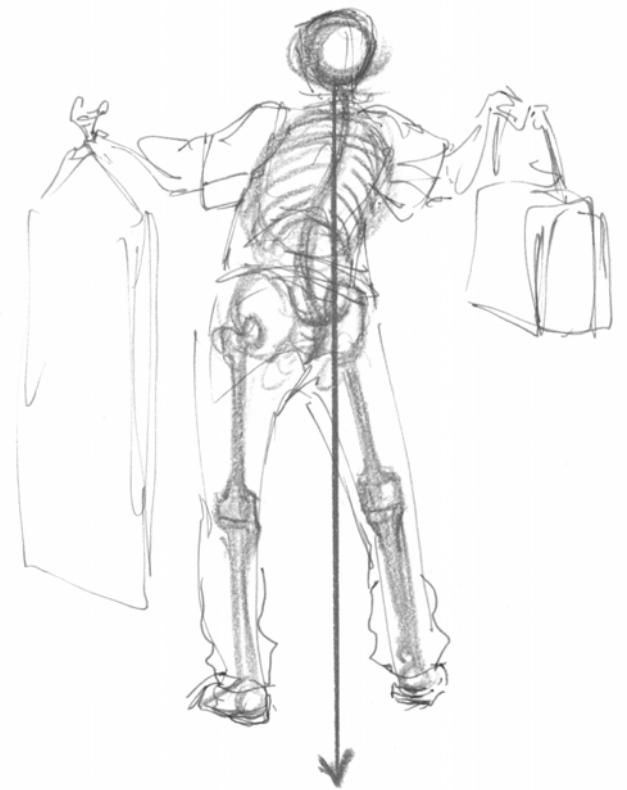
Balance/balance in motion—right arm moves with left leg and vice versa







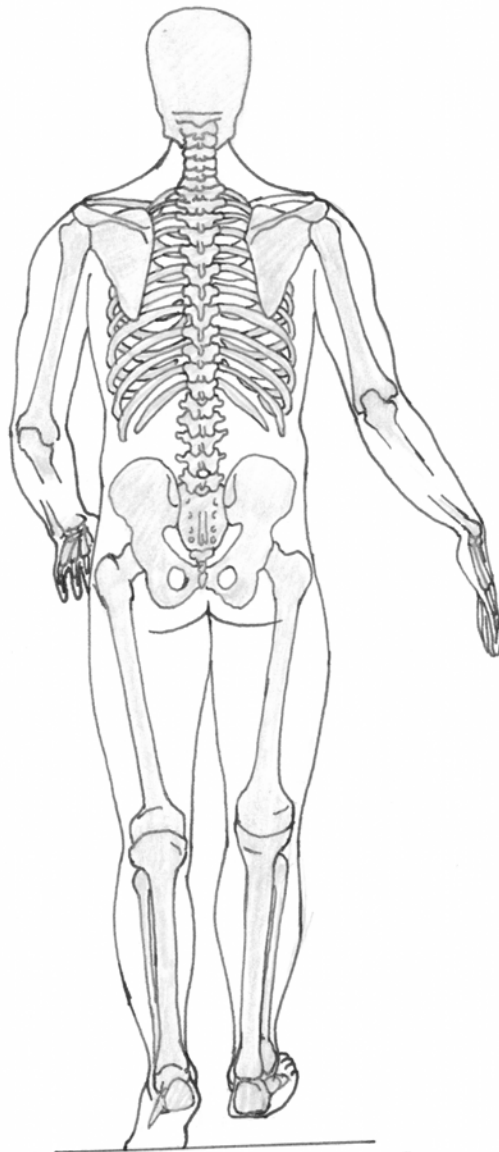
BALANCE OF HEAD OVER FOOT



SKELETON UNDERNATH

Artists must become people watchers, recognizing the difference in those who make up the human race and analyzing body types. (Muscle, fat, and the clothes on top of the skeleton display body type.) Remember that a skeleton is the same for all body types.

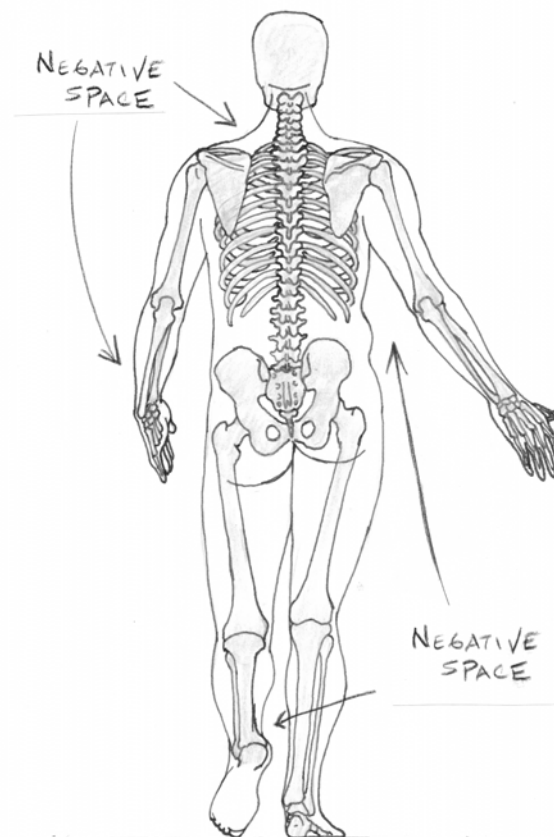
Observation is the key to analyzing any action or activity. Take walking, for instance. Everyone has their own unique gait and often their walk is influenced by their physical makeup (tall, short, thin, or heavy set), and/or their attire (high heels, no shoes, etc.). These factors all influence the walk. Alter the surface or even the weather conditions and the possibilities are endless of how one traverses the terrain. The challenge as an artist is to quickly capture that on paper.



A good drawing is one that utilizes perspective, proportion, weight, and balance to convey to the viewer its own unique story. In order to accomplish this, it is important to have a working knowledge of the skeleton and muscles of the human body.

Think about the negative space around the body parts. This will help you make the drawing interesting and with clarity. Remember you have to be your own editor, with the freedom to move a limb ever so slightly as to not lose the essence of the gesture, but enough to show more clearly in the pose.

Don't lose sight of the goal; to put enough lines on paper to answer the *who*, *what*, *when*, *where*, *why*, and *how* questions. These questions may be clear to you, but do they communicate to your viewing audience?



Summary

This chapter has focused on introducing you to quick sketching and the basics of identifying and utilizing the four basic shapes when analyzing and capturing your subjects in drawing. Anatomy and body types were discussed, leading into the importance of weight, proportion, and balance. Bringing all these elements together will truly help you along the way as you begin the art of quick sketching.

