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 joypad, 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 and the smash hits *Batman: Arkham Asylum* and *Batman: Arkham City* by Rocksteady Studios.

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

1.4.1 First-Person Shooters (FPS)

The first-person shooter (FPS) genre is typified by games like *Quake*, *Unreal Tournament*, *Half-Life*, *Counter-Strike* and *Battlefield* (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 shooter/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.

Figure 1.2. *Battlefield 4* by Electronic Arts/DICE (PC, Xbox 360, PlayStation 3, Xbox One, PlayStation 4). (See Color Plate I.)

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 up to 64 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/adven-

Figure 1.3. *Jak II* by Naughty Dog (Jak, Daxter, Jak and Daxter, and Jak II © 2003, 2013/™ SCEA. Created and developed by Naughty Dog, PlayStation 2). (See Color Plate II.)

Figure 1.4. *Gears of War 3* by Epic Games (Xbox 360). (See Color Plate III.)

ture games like *Dead Space 2*, *Gears of War 3* (Figure 1.4), *Red Dead Remption*, the *Uncharted* series, the *Resident Evil* series, *The Last of Us*, 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:

- 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 joypad stick (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:

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

- 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.

State-of-the-art fighting games like EA's *Fight Night Round 4* (Figure 1.6) have upped the technological ante with features like:

- high-definition character graphics, including realistic skin shaders with subsurface scattering and sweat effects;
- high-fidelity character animations; and
- physics-based cloth and hair simulations for the characters.

It's important to note that some fighting games like *Heavenly Sword* 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

Figure 1.6. *Fight Night Round 4* by EA (PlayStation 3). (See Color Plate V.)

fighting game can have technical requirements more akin to those of a third-person shooter or real-time 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 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. However, kart racers also devote significant rendering and animation bandwidth to the characters driving the vehicles. Figure 1.7 shows a screenshot from the next installment in the well-known *Gran Turismo* racing game series, *Gran Turismo 6*, developed by Polyphony Digital and published by Sony Computer Entertainment.

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

Figure 1.7. *Gran Turismo 6* by Polyphony Digital (PlayStation 3). (See Color Plate VI.)

- 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 Real-Time Strategy (RTS)

The modern real-time strategy (RTS) 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. For a discussion of this genre, see http://en.wikipedia.org/wiki/Real-time_strategy.

The RTS 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 an RTS game.

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 RTS *Age of Empires*.

Modern RTS 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, *Command & Conquer 3*, is shown in Figure 1.9.

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

Figure 1.9. *Command & Conquer 3* by EA Los Angeles (PC, Xbox 360). (See Color Plate VIII.)

Some other common practices in RTS 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.

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 1.10. *World of Warcraft* by Blizzard Entertainment (PC). (See Color Plate IX.)

Figure 1.11 shows a screen from Bungie's latest highly anticipated FPS game, *Destiny.* 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

Figure 1.11. *Destiny* by Bungie (Xbox 360, PlayStation 3, Xbox One, PlayStation 4). (See Color Plate X.)

only with the other players with whom they have been matched by the server. Also unlike a traditional MMO, the graphics fidelity in *Destiny* promises to be among the best of its generation.

1.4.7 Player-Authored Content

As social media takes off, games are becoming more and more colaborative in nature. A recent trend in game design is toward *player-authored content*. For example, Media Molecule's *Little Big Planet* and *Little Big Planet 2* (Figure 1.12) 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 instalment in this up-and-coming genre is *Tearaway* for the PlayStation Vita (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).

Figure 1.12. *Little Big Planet 2* by Media Molecule. © 2014 Sony Computer Entertainment Europe (PlayStation 3). (See Color Plate XI.)

Figure 1.13. *Tearaway* by Media Molecule, © 2014 Sony Computer Entertainment Europe (PlayStation Vita). (See Color Plate XII.)

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 buildings and machinery. Perhaps the biggest stroke of genious in *Minecraft* is *redstone*. This material serves as "wiring," allowing players to lay down

Figure 1.14. *Minecraft* by Markus "Notch" Persson / Mojang AB (PC, Mac, Xbox 360, PlayStation 3, PlayStation Vita, iOS). (See Color Plate XIII.)

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.

1.4.8 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);
- *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 The Unreal Family of Engines

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 Kismet. Many games are being developed with UE4 lately, including of course Epic's popular *Gears of War.*

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).

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 smash hit *Half-Life 2* and its sequels *HL2: Episode One* nad *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* and *Dragon Age.* Frostbite boasts a powerful unified asset creation tool called FrostEd, a powerful tools pipeline known as Backend Services, and a powerful runtime game engine. At the time this was written, the latest version of the engine is Frostbite 3, which is being used on DICE's popular title *Battlefield 4* for the PC, Xbox 360, Xbox One, PlayStation 3 and PlayStation 4, along with new games in the *Command & Conquer*, *Dragon Age* and *Mass Effect* franchises.

1.5.5 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*, *Warface* and *Ryse: Son of Rome.* Over the years the

engine has evolved into what is now Crytek's latest offering, CryENGINE 3. 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 3 can be used to make games targeting a wide range of platforms including Xbox One, Xbox 360, PlayStation 4, PlayStation 3, Wii U and PC.

1.5.6 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 *flOw*, *Flower* and *Journey*, VectorCell's *AMY*, and From Software's *Demon's Souls* and *Dark Souls*. PhyreEngine now supports Sony's PlayStation 4, PlayStation 3, PlayStation 2, PlayStation Vita and PSP platforms. PhyreEngine 3.5 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.

1.5.7 Microsoft's XNA Game Studio

Microsoft's XNA Game Studio is an easy-to-use and highly accessible game development platform 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.

XNA is based on Microsoft's C# language and the Common Language Runtime (CLR). The primary development environment is Visual Studio or its free counterpart, Visual Studio Express. Everything from source code to game art assets are managed within Visual Studio. With XNA, developers can create games for the PC platform and Microsoft's Xbox 360 console. After paying a modest fee, XNA games can be uploaded to the Xbox Live network and shared with friends. By providing excellent tools at essentially zero cost, Microsoft has brilliantly opened the floodgates for the average person to create new games.

1.5.8 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 (Apple iOS, Google Android, Windows phone and BlackBerry 10 devices), consoles (Microsoft Xbox 360 and Xbox One, Sony PlayStation 3 and PlayStation 4, and Nintendo Wii and Wii U) and desktop computers (Microsoft Windows, Apple Macintosh and Linux). It even supports a Webplayer for deployment on all the major web browsers.

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, *Chop Chop Runner* by Gamerizon and *Zombieville USA* by Mika Mobile, Inc.

1.5.9 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 iOS, Android, Flash, Java and XNA 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.

1.5.10 Other Commercial 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 C4 Engine by Terathon Software (http://www.terathon.com), a company founded by Eric Lengyel in 2001. Documentation for the C4 Engine can be found on Terathon's website, with additional details on the C4 Engine wiki.

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 latest hit, *The Last of Us*, and it will continue to evolve as Naughty Dog transitions onto the PlayStation 4. And of course, most commercially licensed game engines like Quake, Source, Unreal Engine 3, CryENGINE 3 and Frostbite 2 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" typi-

cally 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.

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 game engines.

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 1.15 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

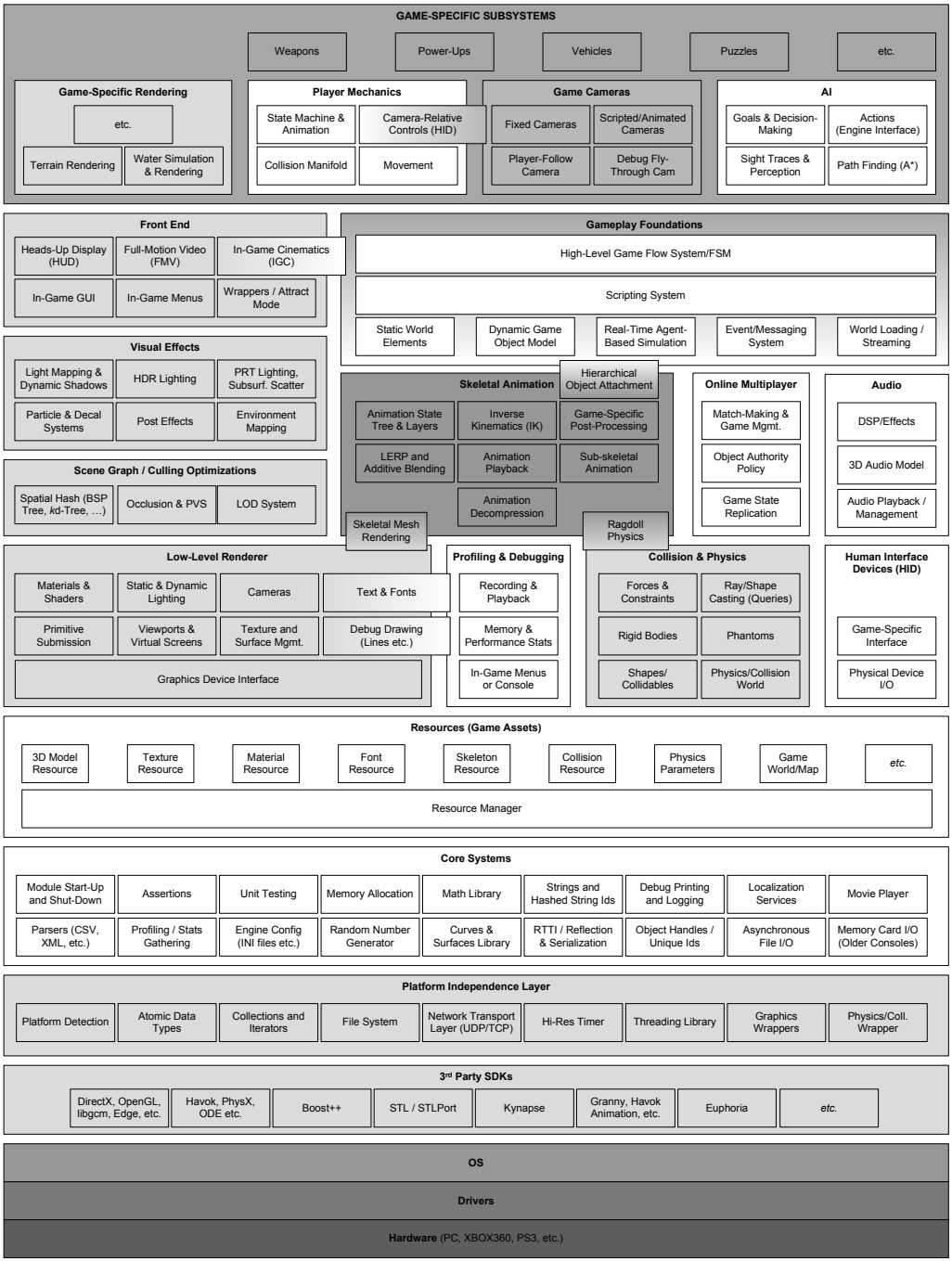

Figure 1.15. Runtime game engine architecture.

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 1.15. 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, shown in isolation in Figure 1.16, 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 and Wii U. 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.

Figure 1.16. Hardware layer.

1.6.2 Device Drivers

As depicted in Figure 1.17, 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 1.17. Device driver layer.

Figure 1.18. Operating system layer.

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. The OS layer is shown in Figure 1.18. 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 a console, the operating system is often just a thin library layer that is compiled directly into your game executable. On a console, the game typically "owns" the entire machine. However, with the introduction of the Xbox 360 and PlayStation 3, this was no longer strictly the case. The operating system on these consoles and their successors, the Xbox One and PlayStation 4 respectively, 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 PS3's Xross Media Bar or the Xbox 360's dashboard, for example. 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 1.19. 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 1.19. Third-party SDK layer.

1.6.4.1 Data Structures and Algorithms

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

- *STL.* The C++ standard template library provides a wealth of code and algorithms for managing data structures, strings and stream-based I/O.
- *STLport*. This is a portable, optimized implementation of STL.
- *Boost.* Boost is a powerful data structures and algorithms library, designed in the style of STL. (The online documentation for Boost is also a great place to learn a great deal about computer science!)
- *Loki.* Loki is a powerful generic programming template library which is exceedingly good at making your brain hurt!

Game developers are divided on the question of whether to use template libraries like STL in their game engines. Some believe that the memory allocation patterns of STL, which are not conducive to high-performance programming and tend to lead to memory fragmentation (see Section 5.2.1.4), make STL unusable in a game. Others feel that the power and convenience of STL outweigh its problems and that most of the problems can in fact be worked around anyway. My personal belief is that STL is all right for use on a PC, because its advanced virtual memory system renders the need for careful memory allocation a bit less crucial (although one must still be very careful). On a console, with limited or no virtual memory facilities and exorbitant cache-miss costs, you're probably better off writing custom data structures that have predictable and/or limited memory allocation patterns. (And you certainly won't go far wrong doing the same on a PC game project either.)

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.

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.
- *Edge.* The Edge library produced for the PS3 by the ICE team at Naughty Dog, the Tools and Technology group of Sony Computer Entertainment America, 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 recently 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 1.20. 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 or replacing the most commonly used standard C library functions, operating system calls and other foundational application programming interfaces (APIs), the platform independence layer ensures consistent behavior across all hardware platforms. This is necessary because there is a good deal of variation across platforms, even among "standardized" libraries like the standard C library.

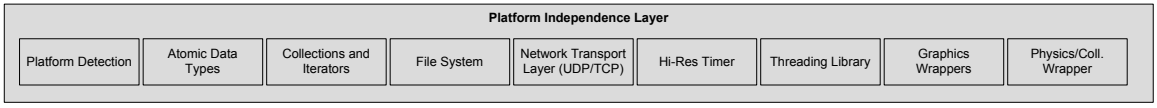

Figure 1.20. Platform independence layer.

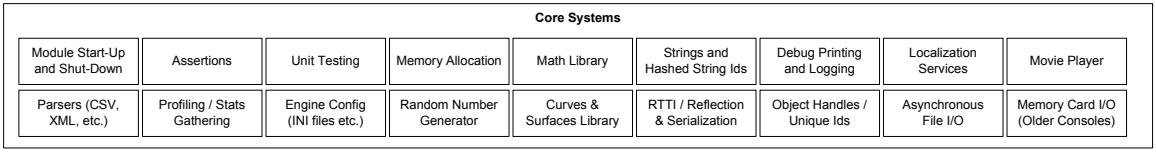

Figure 1.21. Core engine systems.

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 1.21. 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.
- *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 5.2.1.4).
- *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 a third-party package such as STL, 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).

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 `ResourceManager` 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 1.22.

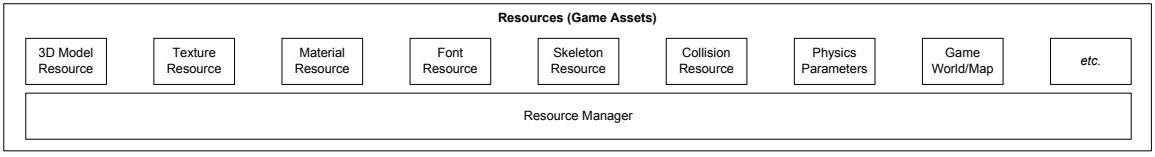

Figure 1.22. Resource manager.

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 1.23, 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.

Graphics Device Interface

Graphics SDKs, such as DirectX and OpenGL, 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

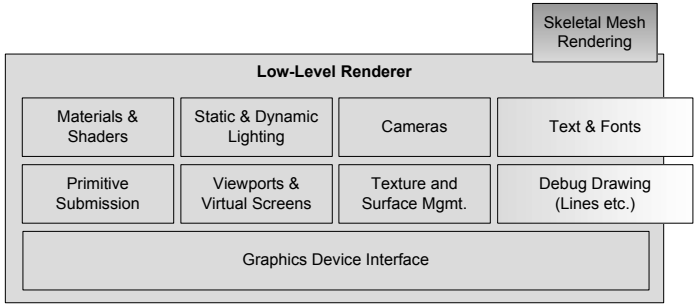

Figure 1.23. Low-level rendering engine.

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.

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, which is covered in depth in many excellent books on computer graphics, including [14], [44] and [1].

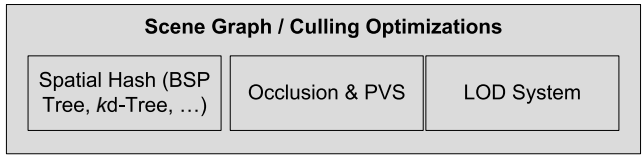

Figure 1.24. A typical scene graph/spatial subdivision layer, for culling optimization.

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 1.24.

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 *k*d-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 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 1.25, including:

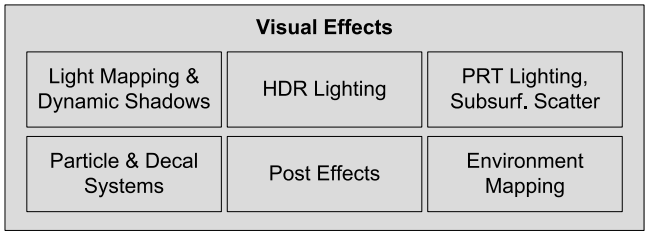

Figure 1.25. Visual effects.

- 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 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

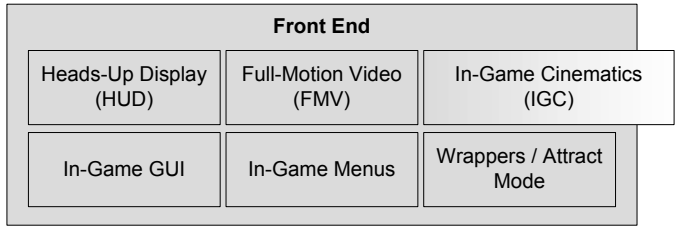

Figure 1.26. Front end graphics.

- 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 1.26. 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.

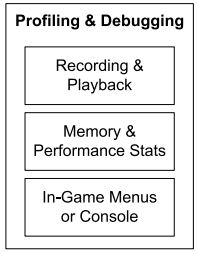

Figure 1.27. Profiling and debugging tools.

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 1.27, 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), and
- Compuware's *Bounds Checker*.

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
- 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.

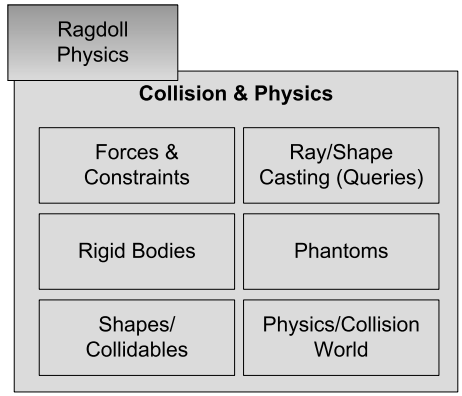

Figure 1.28. Collision and physics subsystem.

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 1.28.

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 new 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 informa-

tion, 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 1.29.

You'll notice in Figure 1.15 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

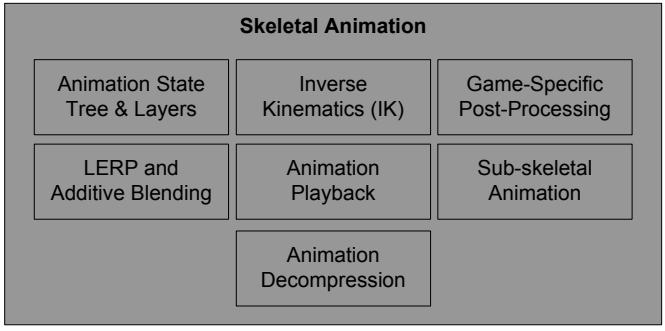

Figure 1.29. Skeletal animation subsystem.

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)

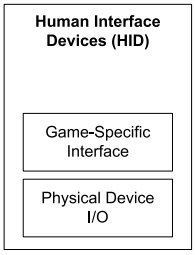

Figure 1.30. The player input/output system, also known as the human interface device (HID) layer.

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

- the keyboard and mouse,
- a joypad, 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 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 1.30.

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

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 1.31.

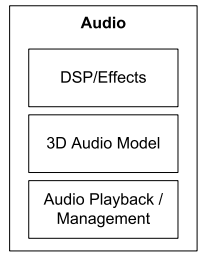

Figure 1.31. Audio subsystem.

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 [40]), 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 audio tool suite called XACT, supported at runtime by their feature-rich XAudio2 and X3DAudio APIs. Electronic Arts has developed an advanced, high-powered audio engine internally called SoundR!OT. In conjunction with first-party studios like Naughty Dog, Sony Computer Entertainment America (SCEA) provides a powerful 3D audio engine called Scream, which has been used on a number of PS3 titles including Naughty Dog's *Uncharted 3: Drake's Deception* and *The Last of Us.* 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 1.32.

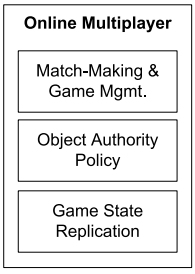

Figure 1.32. Online multiplayer networking subsystem.

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 preexisting 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 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 1.33, 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

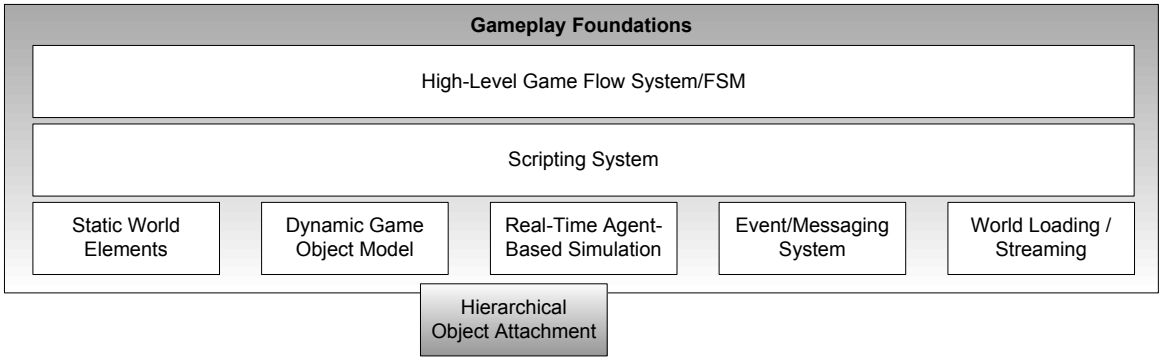

Figure 1.33. Gameplay foundation systems.

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 15.2.

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.

A company called Kynogon developed a middleware SDK named Kynapse, which provided 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. Autodesk also offers a visual programming system and runtime engine called Gameware Cognition, which together with Gameware Navigation aims to make building ambitious game AI systems easier than ever.

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 1.34, 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

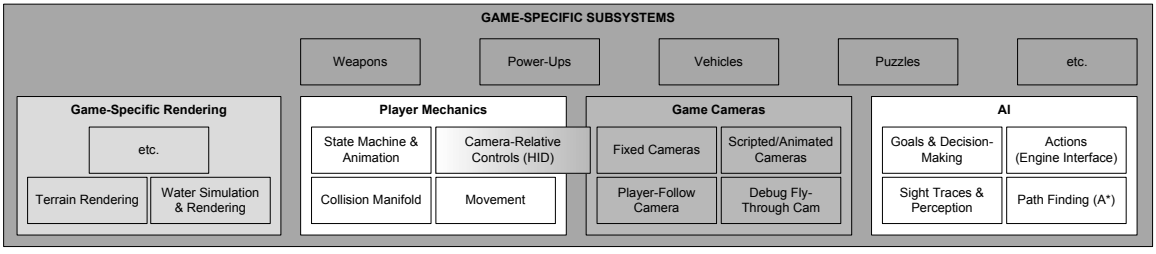

Figure 1.34. Game-specific subsystems.

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 1.35 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 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 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.

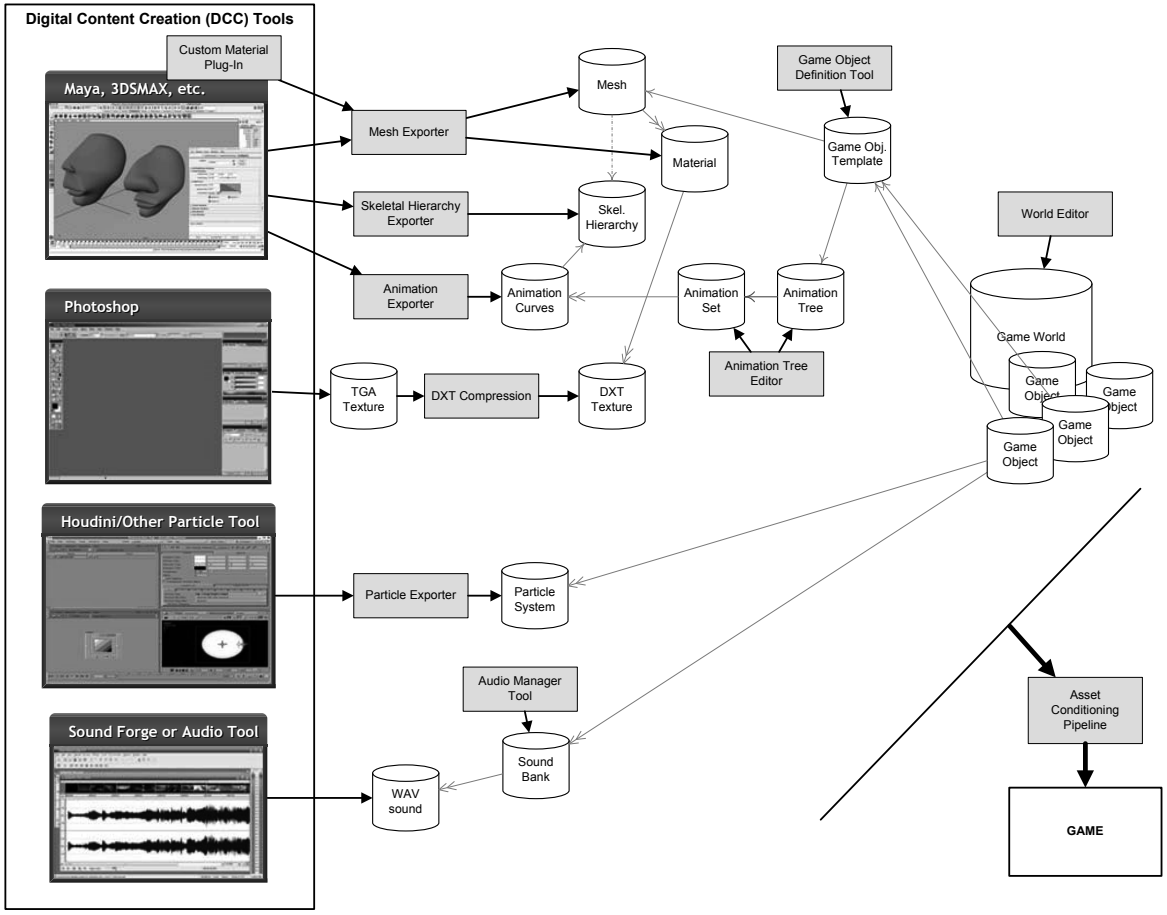

Figure 1.35. Tools and the asset pipeline.

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 10.

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.

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.
- 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 6.2.1.3.

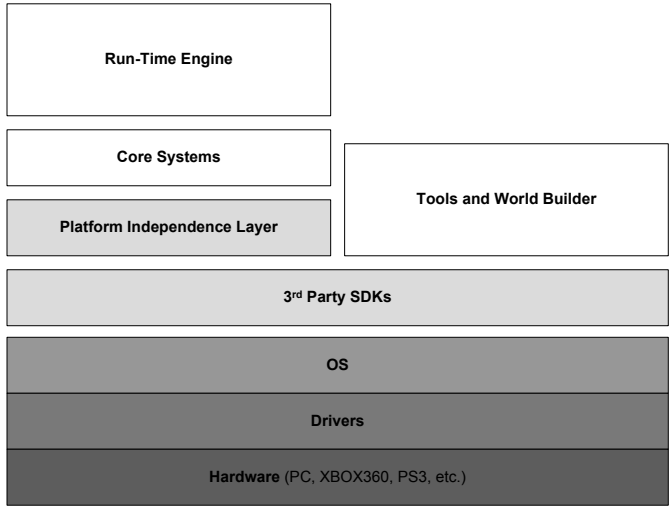

Figure 1.36. Stand-alone tools architecture.

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 1.36. Some tools may be built on top of some of the lower layers used by the runtime engine, as Figure 1.37 illustrates. Some tools might be built into the game itself. 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.

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 1.38. 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

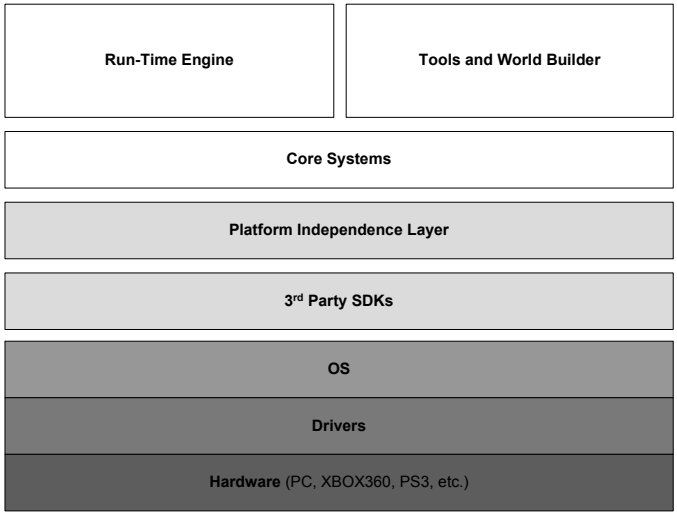

Figure 1.37. Tools built on a framework shared with the game.

become unusable as well. Hence a tight coupling between engine and asset creation tools can tend to slow down production.

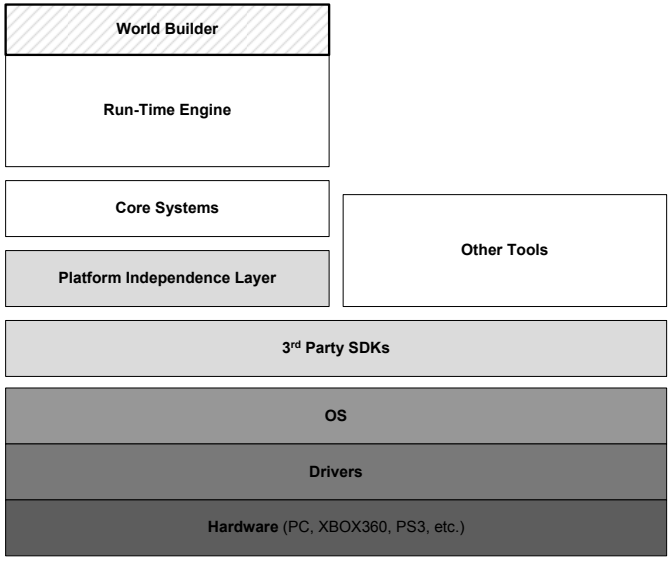

Figure 1.38. UnrealEngine's tool architecture.

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.