Game Engine Architecture

Jason Gregory

Foreword by **Jeff Lander and Matt Whiting**

A K Peters/CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2009 by Taylor and Francis Group, LLC
A K Peters/CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4398-6526-2 (Ebook-PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the A K Peters Web site at
http://www.akpeters.com

Contents

of what we call "fun," just as a joke becomes funny at the moment we "get it" by recognizing the pattern.

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

1.2.1. Video Games as Soft Real-Time Simulations

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

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

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

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

1.3. What Is a Game Engine?

The term "game engine" arose in the mid-1990s in reference to first-person shooter (FPS) games like the insanely popular *Doom* by id Software. *Doom* was architected with a reasonably well-defined separation between its core software components (such as the three-dimensional graphics rendering system, the collision detection system, or the audio system) and the art assets, game worlds, and rules of play that comprised the player's gaming experience. The value of this separation became evident as developers began licensing games and re-tooling them into new products by creating new art, world layouts, weapons, characters, vehicles, and game rules with only minimal changes to the "engine" software. This marked the birth of the "mod community"—a group of individual gamers and small independent studios that built new games by modifying existing games, using free toolkits provided by the original developers. Towards the end of the 1990s, some games like *Quake III Arena* and *Unreal* were designed with reuse and "modding" in mind. Engines were made highly customizable via scripting languages like id's Quake C, and engine licensing began to be a viable secondary revenue stream for the developers who created them. Today, game developers can license a game engine and reuse significant portions of its key software components in order to build games. While this practice still involves considerable investment in custom software engineering, it can be much more economical than developing all of the core engine components in-house.

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

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

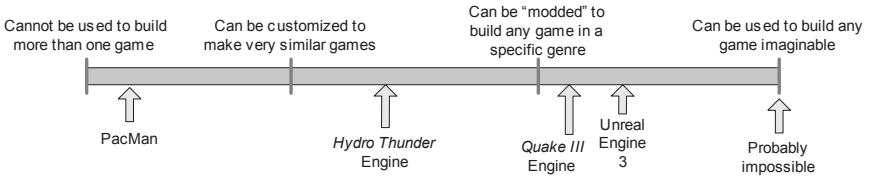

Figure 1.1. Game engine reusability gamut.

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

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

This phenomenon occurs because designing any efficient piece of software invariably entails making trade-offs, and those trade-offs are based on assumptions about how the software will be used and/or about the target hardware on which it will run. For example, a rendering engine that was designed to handle intimate indoor environments probably won't be very good at rendering vast outdoor environments. The indoor engine might use a BSP tree or portal system to ensure that no geometry is drawn that is being occluded by walls or objects that are closer to the camera. The outdoor engine, on the other hand, might use a less-exact occlusion mechanism, or none at all, but it probably makes aggressive use of level-of-detail (LOD) techniques to ensure that distant objects are rendered with a minimum number of triangles, while using high resolution triangle meshes for geometry that is close to the camera.

The advent of ever-faster computer hardware and specialized graphics cards, along with ever-more-efficient rendering algorithms and data structures, is beginning to soften the differences between the graphics engines of different genres. It is now possible to use a first-person shooter engine to build a real-time strategy game, for example. However, the trade-off between gener-

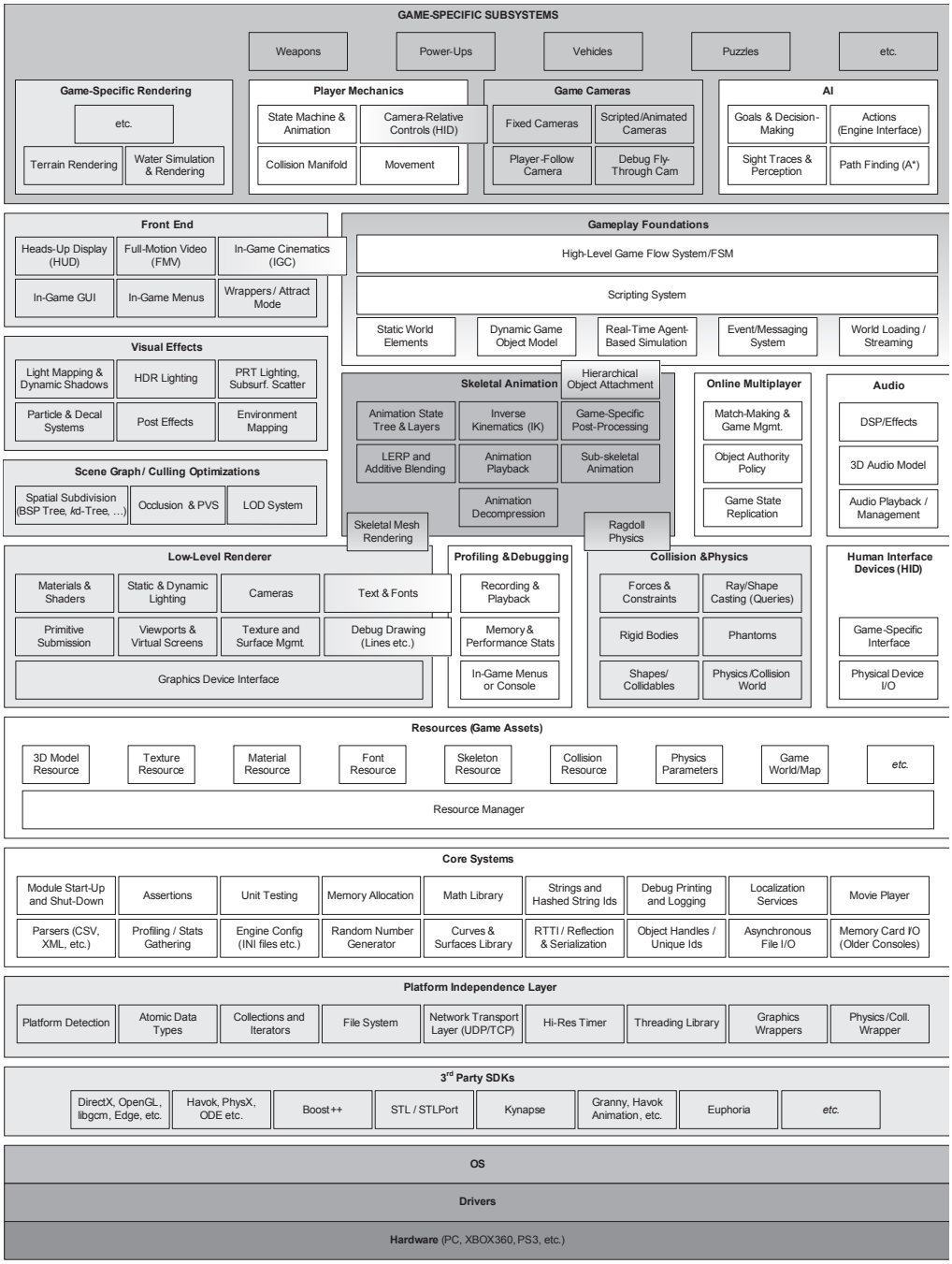

Figure 1.11. Runtime game engine architecture.

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

1.7. Tools and the Asset Pipeline

Any game engine must be fed a great deal of data, in the form of game assets, configuration files, scripts, and so on. Figure 1.31 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 pre-existing 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.

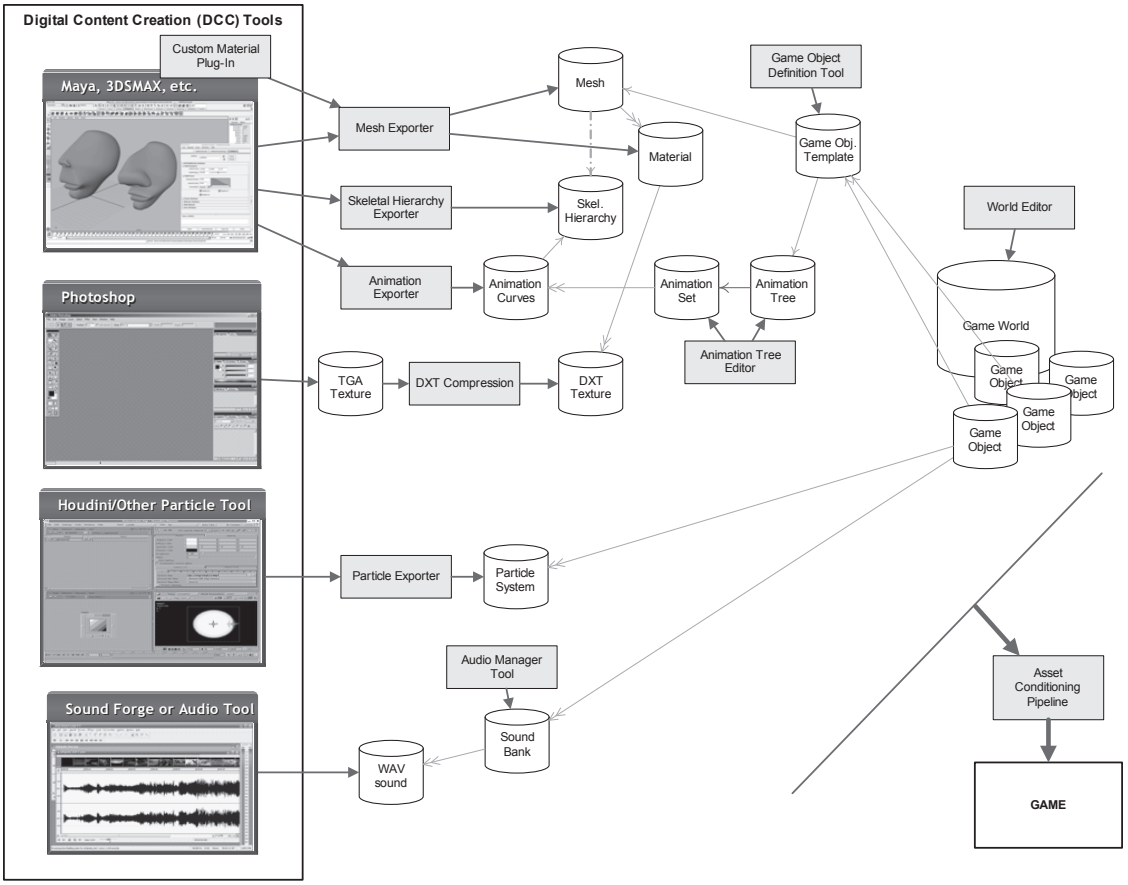

Figure 1.31. Tools and the asset pipeline.

1.7.2. 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

virtually all game engines have some form of tool-side object model and a corresponding runtime implementation of that object model.

13.3. Data-Driven Game Engines

In the early days of game development, games were largely hard-coded by programmers. Tools, if any, were primitive. This worked because the amount of content in a typical game was miniscule, and the bar wasn't particularly high, thanks in part to the primitive graphics and sound of which early game hardware was capable.

Today, games are orders of magnitude more complex, and the quality bar is so high that game content is often compared to the computer-generated effects in Hollywood blockbusters. Game teams have grown much larger, but the amount of game content is growing faster than team size. In the most recent generation, defined by the Wii, the Xbox 360, and the PLAYSTATION 3, game teams routinely speak of the need to produce ten times the content, with teams that are at most 25% larger than in the previous generation. This trend means that a game team must be capable of producing very large amounts of content in an extremely efficient manner.

Engineering resources are often a production bottleneck because high-quality engineering talent is limited and expensive and because engineers tend to produce content much more slowly than artists and game designers (due to the complexities inherent in computer programming). Most teams now believe that it's a good idea to put at least some of the power to create content directly into the hands of the folks responsible for producing that content—namely the designers and the artists. When the behavior of a game can be controlled, in whole or in part, by *data* provided by artists and designers rather than exclusively by *software* produced by programmers, we say the engine is *data-driven*.

Data-driven architectures can improve team efficiency by fully leveraging all staff members to their fullest potential and by taking some of the heat off the engineering team. It can also lead to improved *iteration times*. Whether a developer wants to make a slight tweak to the game's content or completely revise an entire level, a data-driven design allows the developer to see the effects of the changes quickly, ideally with little or no help from an engineer. This saves valuable time and can permit the team to polish their game to a very high level of quality.

That being said, it's important to realize that data-driven features often come at a heavy cost. Tools must be provided to allow game designers and artists to define game content in a data-driven manner. The runtime code must

be changed to handle the wide range of possible inputs in a robust way. Tools must also be provided in-game to allow artists and designers to preview their work and troubleshoot problems. All of this software requires significant time and effort to write, test, and maintain.

Sadly, many teams make a mad rush into data-driven architectures without stopping to study the impacts of their efforts on their particular game design and the specific needs of their team members. In their haste, such teams often dramatically overshoot the mark, producing overly complex tools and engine systems that are difficult to use, bug-ridden, and virtually impossible to adapt to the changing requirements of the project. Ironically, in their efforts to realize the benefits of a data-driven design, a team can easily end up with significantly lower productivity than the old-fashioned hard-coded methods.

Every game engine should have some data-driven components, but a game team must exercise extreme care when selecting which aspects of the engine to data-drive. It's crucial to weigh the costs of creating a data-driven or rapid iteration feature against the amount of time the feature is expected to save the team over the course of the project. It's also incredibly important to keep the KISS mantra ("keep it simple, stupid") in mind when designing and implementing data-driven tools and engine systems. To paraphrase Albert Einstein, everything in a game engine should be made as simple as possible, but no simpler.

13.4. The Game World Editor

We've already discussed data-driven asset-creation tools, such as Maya, Photoshop, Havok content tools, and so on. These tools generate individual assets for consumption by the rendering engine, animation system, audio system, physics system, and so on. The analog to these tools in the gameplay space is the *game world editor*—a tool (or a suite of tools) that permits game world chunks to be defined and populated with static and dynamic elements.

All commercial game engines have some kind of world editor tool. A well-known tool called *Radiant* is used to create maps for the *Quake* and *Doom* family of engines. A screen shot of Radiant is shown in Figure 13.4. Valve's *Source* engine, the engine that drives *Half-Life 2*, *The Orange Box* and *Team Fortress 2*, provides an editor called *Hammer* (previously distributed under the names *Worldcraft* and *The Forge*). Figure 13.5 shows a screen shot of Hammer.

The game world editor generally permits the initial states of game objects (i.e., the values of their attributes) to be specified. Most game world editors also give their users some sort of ability to control the *behaviors* of the dynamic objects in the game world. This control might be via data-driven configuration

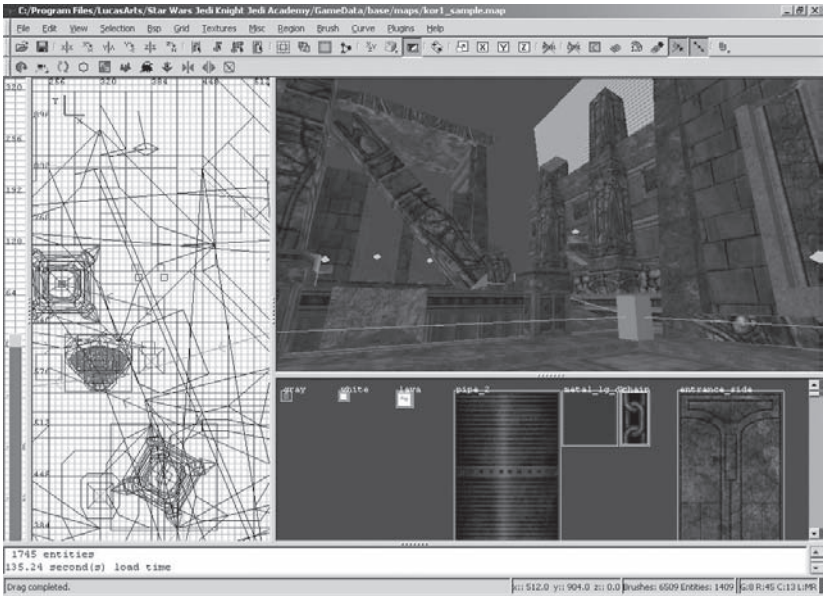

Figure 13.4. The Radiant world editor for the Quake and Doom family of engines.

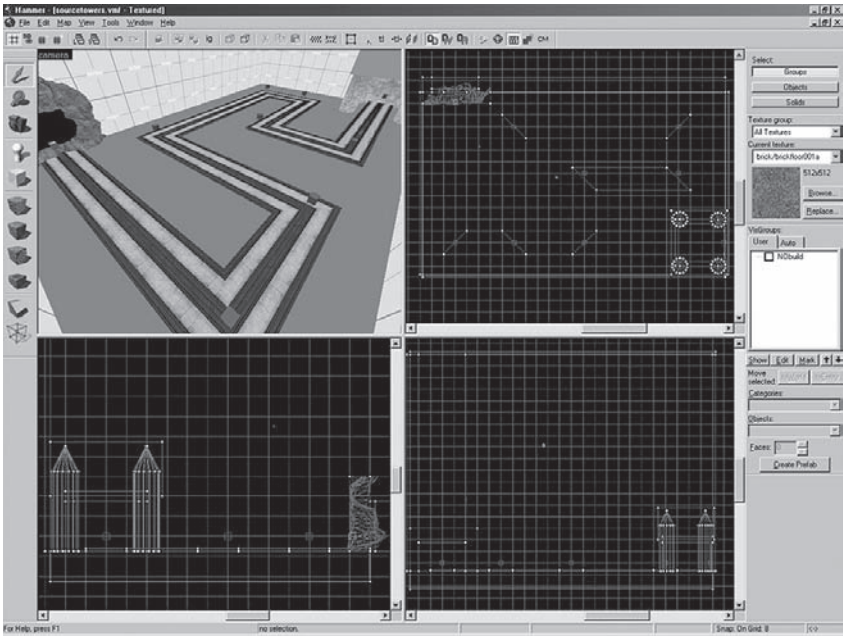

Figure 13.5. Valve's Hammer editor for the Source engine.

parameters (e.g., object A should start in an invisible state, object B should immediately attack the player when spawned, object C is flammable, etc.), or behavioral control might be via a scripting language, thereby shifting the game designers' tasks into the realm of programming. Some world editors even allow entirely new types of game objects to be defined, with little or no programmer intervention.

13.4.1. Typical Features of a Game World Editor

The design and layout of game world editors varies widely, but most editors provide a reasonably standard set of features. These include, but are certainly not limited to, the following.

13.4.1.1. World Chunk Creation and Management

The unit of world creation is usually a chunk (also known as a level or map—see Section 13.1.2). The game world editor typically allows new chunks to be created and existing chunks to be renamed, broken up, combined, or destroyed. Each chunk can be linked to one or more static meshes and/or other static data elements such as AI navigation maps, descriptions of ledges that can be grabbed by the player, cover point definitions, and so on. In some engines, a chunk is defined by a single background mesh and cannot exist without one. In other engines, a chunk may have an independent existence, perhaps defined by a bounding volume (e.g., AABB, OBB, or arbitrary polygonal region), and can be populated by zero or more meshes and/or brush geometry (see Section 1.7.3.1).

Some world editors provide dedicated tools for authoring terrain, water, and other specialized static elements. In other engines, these elements might be authored using standard DCC applications but tagged in some way to indicate to the asset conditioning pipeline and/or the runtime engine that they are special. (For example, in *Uncharted: Drake's Fortune*, the water was authored as a regular triangle mesh, but it was mapped with a special material that indicated that it was to be treated as water.) Sometimes, special world elements are created and edited in a separate, standalone tool. For example, the height field terrain in *Medal of Honor: Pacific Assault* was authored using a customized version of a tool obtained from another team within Electronic Arts because this was more expedient than trying to integrate a terrain editor into Radiant, the world editor being used on the project at the time.

13.4.1.2. Game World Visualization

It's important for the user of a game world editor to be able to visualize the contents of the game world. As such, virtually all game world editors provide