Chapter 2: How to Make Virtual Worlds

Key Topics

- Development
- On Architecture
- Theory and Practice

Virtual worlds are implemented using complicated pieces of software, but, contrary to what many developers would like to believe, they are by no means the most sophisticated programs in existence. Modern operating systems comfortably beat them, and they're dwarfed by major projects, such as air traffic control networks. When you read the following, therefore, remember that it could all be much, much worse.

This book is written from the perspective of a virtual world designer. The fun part of design is the creativity; the boring part is what you have to learn to inform the creative process. It's not surprising that many designers therefore omit this step. This is a Bad Thing. It is *not* enough to have played or even coded other virtual worlds; to do a good job, you have to understand how they *work*. For example, a college student putting together a textual virtual world might try out different codebases to see which is the most appropriate. Well yes, that sounds only sensible. However, it would be like someone who knows how to drive taking a selection of cars for a spin before deciding which to use as the basis for designing a car of their own. There is more to designing cars than finding something that suits your driving style; there is more to designing virtual worlds than finding something that suits your playing style. Before you can make a start you need to be aware of how virtual worlds function, what the components are, how they fit together, what can go wrong, and a whole host of other things.

A student building a virtual world from a kit has the excuse that in doing so they might actually learn some of the important design principles involved. The student's next world will consequently be much improved. Professional virtual world designers can fall back on no such justification. There are some things that they simply ought to know beforehand, whether they want to or not.

It's this background knowledge that this chapter is intended to impart.

Development

Design is just one part of creating a virtual world. Designers like to think it's the most important part, but is it?

- Designers have wild, airy-fairy imaginings.

- Programmers do the actual work of building the virtual world.
- Artists are the magicians who imbue it with form.
- Sound engineers determine the moods and emotions.
- Operations staffs are the engineers who keep it running.
- Producers provide the resources.

Anyone can have wild imaginings. Only people with specialist skills can program, or draw, or compose, or run networks, or manage a project. Why are designers so important?

Because, if a designer screws up, the consequences for the virtual world can be devastating.

A piece of code that doesn't work may be hard to track down, but once discovered it's usually easy to correct. An odd-looking yet crucial texture may need to be painstakingly redrawn, but it's still only a single bitmap. However, if a designer makes a seemingly minor misjudgment, the effects could be so pervasive that they might paralyze a world for weeks.

If you find that hard to believe, consider a virtual world in which non-player shopkeepers sell goods at fixed prices. What happens if there is inflation in this virtual economy? Pretty soon, you can have whatever you want for peanuts. What happens if there's deflation? Even trivial items cost so much that only the very rich can afford them. What factors affect inflation/deflation? Oh, just about all of them—in hideously intertwined ways as determined by the actions of the designer.

Broken economies are not pretty. At one point, *Asheron's Call*'s currency became so worthless that players had to barter if they wanted to acquire goods from one another. This went on for months before it was finally brought under control; the debacle cost *AC* dearly.

So, that's why design is the most important thing about creating virtual worlds—it has the highest price of failure.

The Team

Design might be the most important cog in the machine that creates virtual worlds, but that isn't to say the other components are unimportant. Some are absolutely critical: If a server crashes, for example, every minute it stays down will be paid for in cancelled accounts. Designers have to know about these things, so they can account for them in

their virtual world design.

Designers should have not only a realistic idea of their own place in the system, but also a sound knowledge of the roles of the other people involved in the creation process. Composers don't have to know how to play every instrument in an orchestra, but it's essential that they know how all the instruments sound; designers can't be expected to know how programmers or artists do what they do, but they must be aware of any limitations. If you want every wall of your virtual palace to have a stunning, original fresco on it, you can think again.

To create a virtual world is to create a piece of software. That's not all it is, of course—it's creating a community, a service, a place—but these count for little if there isn't an engine to run the world.

A typical software engineering company is organized along functional lines that cover the following areas:

- Company leadership
- Sales and marketing
- Finance and accounting
- Software development, support, and quality assurance (QA)
- Operations and information technology (IT)
- Human resources (HR)

Some of these may be split into separate sections, for example sales might be distinct from marketing; on the whole, though, the preceding list is fairly uncontentious. Note that normally there is no specific group responsible solely for product specification; the task falls to whoever sources the software, which in many cases could well be the customer.

A typical computer games development company is organized in much the same way, but with some games-specific differences:

- An art and animation section is added.
- An audio (music, sound effects) section is added (unless outsourced).
- QA is expanded, and is formally separated from actual software development.
- A (usually small) design group is added; its members will be paid less, but get more fan kudos, than their coworkers.

For developers of massively multiplayer, graphical virtual worlds, the games development model is used except:

- The design group is expanded.
- The operations group (which maintains and supports the hardware on which the system will run) is expanded.
- The support group (which deals with players, both inside and outside the virtual world) is greatly expanded, and is formally separated from actual software development. It will usually reabsorb the QA section.

Designers are only occasionally bothered by the company leadership, HR, IT, and finance/accounting people. They have a dialogue with sales/marketing that may be in balance or lopsided ("This is the kind of world we want you to design" versus "This is the kind of world we want you to sell"[1]). They tell the operations, artwork, and audio experts what needs to be done, but generally leave them to it. They interact mostly with

- The programmers (because designers are never specific enough about what they want, except when they're so enthusiastic that they try to tell the programmers how to program[2]).
- QA (because testers spot more design flaws than they do programming bugs and operations problems).
- Support (because players spot more design flaws than QA people).
- Each other (because although this book keeps referring to "the designer" of a virtual world, there's usually a *design team*, led by a *lead designer*).

When work on a new virtual world begins, a *core team* is assembled. For a small world, this could be a single individual performing multiple tasks; indeed, it might never get any bigger. For a large-scale world, though, it is merely the nucleus about which a full-blown development effort will form. A core team consists of the

- Producer
- Lead designer
- Lead programmer(s) (server, client)
- Lead artist(s) (environment, inhabitants/characters)

There may be two lead programmers because client programming is something best done

by people with a background in computer games development, whereas server programming is best done by people with a background in software engineering. Programming is becoming a progressively more specialized field, and programmers expert in one area may need training to work in another.

There may be two lead artists because of the sheer quantity of artwork involved in a graphical virtual world (albeit not when development first starts). Strictly speaking, the "environment artist" is in charge of the concept art—defining the look of the virtual world. The "characters artist" is in charge of the technical side—interfacing with the programmers. Because this usually comes down to issues of animation, that's why they wind up being responsible for characters.

Increasingly, operations and customer service leads are being brought in to the core team, but because their work cannot begin until some time into the development process it's unusual if this occurs in a start-up company.

The Development Process

There are many steps to the development of a virtual world. For smaller worlds with fewer players and different functionality, some steps can be skipped or done in tandem with other steps. To highlight every aspect of the process, however, the description in this chapter is for a large, graphical world. Luckily, designers don't need to know every detail of this—that's the job of the producer—but they do need to have an idea of how it breaks down. Therefore, you'll be relieved to learn that only an overview will be presented here, rather than a how-to guide. If you want to find out more (and to understand why it is that producers are paid twice as much as designers), consult *Developing Online Games: An Insider's Guide*[3] by Bridgette Patrovsky and Jessica Mulligan (for virtual worlds) or *Game Architecture and Design*[4] by Andrew Rollings and Dave Morris (for games in general).

The development of online games has four distinct phases:

- Pre-production
- Production
- Roll out
- Operation

Let's look at these in turn.

Pre-Production

Pre-production can last as long as six months. The aim is to do all the concept evaluation and project planning necessary to reduce risk in the later stages of development. It's undertaken by members of the core team, in close consultation with one another. In many ways, it's the most exciting part of the project, but it's usually done under time pressure with inadequate resources available, which rather dulls the edge. In particular, a number of important deliverables will have been prepared by the end, all of which will almost certainly have needed more work on them than they actually received.

These deliverables are

- A visualization document. This is produced first, by the lead designer. Although only a few pages long, it sets the tone for the entire endeavor, asserting the project's mission statement, its philosophy, its goals, its main features, and its look and feel.

- A design document. Because the designer(s) creates this, I spend much of Chapters 3 through 5 of this book addressing the kind of material that goes into it. For the moment, though, suffice to say it defines things such as the world's background, its architecture, its mechanics (including gameplay), its control mechanisms (how players interact with it), and its integral community support systems[5]. Specifics will be added constantly as development continues.

- A technical design review, assessing hardware and software requirements. What technologies are needed? What tools (both bespoke and middleware)? The technical design review is often folded into the design document.

- An art bible, describing the stylistic conventions to be used along with examples illustrating the range of material required. This is so that artists can produce work that is consistent with a single overall look[6].

- A production management assessment, which uses the other deliverables to gauge the project's demands. It will include a schedule (with milestones), resource requirement details and some risk assessment. The schedule will be continually updated in the light of how things actually proceed, as opposed to how they're supposed to proceed.

- Prototypes to provide proof of concept and to show that potential technical difficulties can be overcome.

Pre-production is primarily a planning phase, therefore the construction of (limited) prototypes might seem to be out of place. Prototypes are necessary partly for commercial reasons—they demonstrate to investors that the team can produce the goods—but they also benefit the team itself. They ensure standards for design, programming, and art have been set, and that source control works. They show that the basic principles will work, and (hopefully) can be integrated. For companies that produce a steady stream of material

for different projects, assembly line style, the basic pathways for communicating with the various production centers will also have been tested.

The technical design review addresses basic issues, such as how the server code will be modularized, what network transport layer protocols will be used (TCP/IP versus UDP), how background content will be trickled to clients, and how multiple access options will be incorporated (PC/console, web browser, mobile phone). Additionally, it has to consider topics not directly related to the virtual world at all, primarily back-end systems for login, billing, and so on. The necessary software development tools (including ones for system testing and debugging) must be acquired at this stage, in addition to as many pieces of middleware as are suitable. In particular, even with the typically huge license fees involved, it is usually more cost-effective for a company to buy a database, 3D engine, and billing system than to write its own from first principles[7].

Whether or not you can acquire useful middleware for world creation and AI scripting is project-dependent, however, because it requires great flexibility. Developers usually like to have their own tame programmers available to make any necessary changes expediently, rather than having to rely on someone else's people to do so at short notice.

The production management assessment covers a wider brief than its name might suggest. The term comes from the computer games industry, where typical products don't need a great deal of support after they hit the stores; their management assessment therefore only needs to cover production. Virtual worlds (whether or not games) do, however, need to be managed after launch—immensely so! The production management assessment for them must also extend through the rollout and into the operation phase. This means it has to consider things such as quality assurance, live team management, community maintenance, content creation, and patching. It's also the place where the battle with the Marketing department starts over the handling of the product's launch.

Production

The *production* phase[8], which lasts between two and three years[9] for a large, commercial virtual world, is when the bulk of the programming and data creation takes place. Code must be produced for the client, the server, and for tools. It all has to be done in order, according to a production schedule[10] set by the producer. Tools are usually written first, because other activities are dependent on them and because some of the code can usually be re-used for the server or client. Tools are required for things such as world generation, artificial intelligence (AI) scripting, and customer service support. It's also a good idea to build some analysis tools, too, so that once the world is running it will be possible to determine what the players, the software, and the hardware are doing without having to ask.

The amount of server-side code needed depends on the chosen architecture. It includes

driver functionality at the level of LAN networking (to connect the server to its peers) and communications modules (to connect the server cluster to the Internet). It usually includes a mudlib layer, to support the world physics and AI system. Whether it includes a world model layer depends on what the scripting tools produce. If it is to run in multiple incarnations, it will not include any instantiation-specific detail (it would be too hard to make general updates otherwise).

The client-side code will be the home of a major 3D engine plus support for music and audio effects. Communications protocols to connect with the server are obviously necessary, as are software update mechanisms for when the client needs patching (which is an inevitability).

While the programmers are busy programming, the artists are busy creating object models (static and animated) and texture maps, along with other miscellaneous images (for example, for manuals, intro movies, and web sites). The volume of artwork required is so high that it will normally be stored in its own database so that the artists can keep track of it all. Scalability and maintainability issues also arise for graphics[11].

The world itself is constructed using the building tools that the programmers have created, to the specifications of the design document. There is a fair degree of creative freedom involved in this activity[12], which is why specialist designers usually undertake it rather than programmers; it's analogous to the way that animation is generally done by artists, even though programmers created the necessary tools.

Roll Out

Roll out is the most critical phase of development, when all the technologies and assets created are brought together to form a virtual world experience. It formally begins with the *open beta* (test), but has its roots much earlier in the development process.

Testing takes place all the way through development, of course. Programmers will test individual pieces of code, animators will test animations, even designers will run data through models to ensure that what they think will happen has a good chance of being what *will* happen after players are let loose in their handsome creation.

When enough of a virtual world is available as an integral environment, a *test server* can be set up and *alpha testing* can begin. This is undertaken by the designers, programmers, and artists themselves, looking for bugs mainly in their own areas of responsibility but also reporting anything else they discover that seems Somehow Wrong. Around now, enlightened developers might invite independent design consultants to take a look, but most aren't enlightened and don't. Folks, the opinions of knowledgeable people from outside the team who haven't been living and breathing it for two years *are* worth having, and worth paying to have. Of course, this does also assume that you'll listen to what they

say rather than simply check the "hire consultant" box on the production schedule then move on.

Alpha testing is also the stage at which trained customer support staff can begin their learning process, subjecting the virtual world to the kind of punishment that real players are likely to mete out as they do so.

QA specialists may be brought in (externally or from elsewhere in the company) to perform platform testing—seeing whether the client runs on a representative variety of home computer configurations—but they won't hang around afterwards as virtual worlds are typically much greater in scope than regular computer games and take longer to play through. Given that customer service representatives need to have an in-depth knowledge of the virtual world anyway, it makes sense to provide them with enough QA training that they can perform this task instead, while building their playing skills.

During alpha testing, anything and everything goes as bugs are found, fixed, and their solutions reintegrated into the whole. Eventually, however (hopefully at a point previously scheduled by the producer), the world is stable enough to allow people into it who are not directly involved in the development process. In other words, players.

Initially, only a few outsiders are allowed into the virtual world. The first ones will be those the developers specifically ask to play, either because they are friends[13] or because they are influential yet responsible (or sounded that way on the community message boards). A few others will be signed up from a general call for play-testers, so as to disguise the fact that most of their peers get in through the back door. Thus begins *beta testing*.

At this stage, it's a *closed beta*, because the world is invitation-only. As stability increases, player numbers can be gradually increased by letting in more wannabes from the general call (a technique known as *ramping*). When the barriers are lifted high enough that the testers begin acting like real players, the world is said to be in *live beta*; this may or may not coincide with the moment the world is officially opened up to all-comers for stress-testing—that is, when it enters *open beta*. Because this final stage of testing marks the point of no return, this is when the roll out truly begins.

Usually, computer games go into beta testing as late as possible. Virtual worlds, not really being computer games (despite what many of their developers seem to think), go into beta testing as early as possible. This allows for bugs and exploits to be discovered well before paying customers can leave over them, all the while forging strong community bonds between the beta testers. Some of these people may even come up with decent ideas for improvements[14].

Roll out ends after the launch, when its legacy is passed to the marketing department (which will have had considerable involvement in it already). Later expansion modules

may have their own roll outs, of course, as they do the other phases of development.

To summarize, the aim of the roll out period is to launch a virtual world with

- A seeded community
- A primed market
- Balance
- No bugs

All but the last of these are possible.

Operation

The operation phase[15] begins when people start paying to enter the virtual world. It ends when people stop paying, or when the resources needed to support them would be better employed (that is, make more money) elsewhere.

During the operation phase, the original design and development team (the *dev team*) typically hands over control to a new set of developers (the *live team*). The rationale is that the battle-hardened dev team can move on to other projects (say, creating the next expansion), leaving the less experienced live team responsible for the maintenance and long-term improvement of the virtual world. This is not always the case, however; *Dark Age of Camelot*, for example, retained its dev team for the operation phase, rather than putting the very people who knew the project best to work elsewhere.

So what exactly does a live team do? Its tasks include the following:

- Customer and community support.
- Network and technical support.
- Feature development and enhancement.
- Maintaining overall quality of gameplay in response to player cunning.
- Keeping in step with technology (for example, new platforms, new video cards).
- Occasionally, marketing (the virtual world and its intellectual properties).

The size of the development team for a commercial virtual world varies; generally speaking, the further into the project, the more people are involved. Although some companies may claim that they can produce a world capable of handling 100,000 players

with only a designer, a programmer, and a clip art package, the reality is somewhat different. As a rough idea, a year or so into the production phase there will typically be around 30 people in the dev team split 5:10:15 for designers, programmers, and artists/animators.

Why mention this now? Because the live team will be three to four times *larger* than the dev team! It'll have similar numbers of designers and programmers (maybe fewer artists[16]), but add a hundred or more people in customer and community support.

For virtual worlds, the work only *really* begins at the operation phase. Time and time again, this is something that developers fail to understand—especially if they have long-time exposure to the fire-and-forget approach of the regular computer games industry. Virtual worlds, despite their origins, are *not* regular computer games—or necessarily any kind of game at all (what they are instead is discussed in Chapter 6, "It's Not a Game, It's a...").

On Architecture

Chapter 1, "Introduction to Virtual Worlds," described the way that virtual world servers are constructed in terms of four different layers of functionality (driver, mudlib, world model, incarnation). This is the breakdown of most interest to designers. However, an understanding of how the rest of the system hangs together is required, at least at an abstract level. That's what's discussed next.

Overall Architecture

Actual architectures differ from developer to developer, but they can almost all be regarded as variations of a single generic approach that emphasizes reliability, scalability, and maintainability. Figure 2.1 illustrates this overall architecture.

Figure 2.1
Overall architecture.

Here's how it works.

Central to the system is the *user database*. This is a powerful piece of software running on its own, fast machine with lots of storage. It contains records for all the players registered for the virtual world. Before anyone can access content, they must first log in; this means checking with the user database. Other parts of the complete system (not shown in Figure 2.1) for billing, customer service management, and patching also have access to the user database. It's a very important, industrial-strength system, and is therefore usually bought in rather than programmed in-house.

Players have a number of options for connecting to the virtual world. The main one is to

use a client from a PC (or console or Macintosh or Linux box—the clients will all present the same interface, so the virtual world neither knows nor cares what operating system they're running under). Players may also connect to the host by using a browser, but not to the same extent as with a bespoke client; although there are text-based virtual worlds that do have good browser-based (Java) clients, no-one has yet been reckless enough to try the same kind of thing for a full-blown 3D graphics-based one. Similarly, mobile phones fall way short of being usable for actual play. However, both browsers and phones can be used to obtain information from a virtual world (for example, news, the virtual weather) or to make changes to standing orders (for example, training regimes, the prices of goods offered for sale). Phones and email can be used to inform players of unfolding events, but they don't usually offer the chance to participate in them.

When an incoming connection is established, it is handled by a *front end*. Front ends can communicate with both their target platform (client, browser/email, mobile phone) and the user database. Figure 2.1 shows the various front ends as separate entities because they're separate processes; however, in practice they may be consolidated onto a single machine or be split across several (for example, by real-world geography).

Individual incarnations of virtual worlds run on server clusters (also known as *shards*[17]). The architecture of shards is described in detail shortly, but for the moment a shard can be regarded simply as a unitary server entity. Having approved a player for access to content, the front end will either query the server to which the content relates (for browser or phone connections), or pass control to the server itself (for client connections). The server does not need to access the user database in order to support the virtual world, because servers deal with characters rather than players. Customer support staff, however, deal with players rather than characters, so they do need to be able to check the user database from within the virtual world. This is sometimes done using a separate database tool that they can invoke while simultaneously using the regular client, but not always. In particular, if the client owns[18] the player's PC, a special *admin client* is needed with the database tools built in.

If this is so, then either the client has to be able to maintain two connections (one to the server, one to the user database) or—more easily—the server itself can issue database requests when required, passing the results back to the client.

Server Architecture

Server clusters implement instances of the virtual world. There may be programmed differences between them (attacking other players might be allowed on one but not on another, for example), but these are generally minimal in form if not effect. The number of servers present in a cluster varies from implementation to implementation, but it's usually around half a dozen or so. The number of clusters also varies, with more clusters being added as a product's popularity increases (*EverQuest* hit 40 server clusters in 2000,

averaging around 1,500 players simultaneously on each one at peak time).

Figure 2.2 shows how a server cluster is typically configured.

Figure 2.2
Server cluster architecture.

Individual clients are connected to individual sub-servers. Ideally, each sub-server does the same amount of work (which in practice means it handles the same number of players) as every other sub-server. Sub-servers have access to a shared character database[19] that stores the persistent data relating to players' characters on this server[20]. This may or may not be part of a systemwide accounts database for managing player access. For maintainability, a large-scale world generally keeps a separate database for environment data, which may be partitioned into a template database, a scripting language database, and an instantiation database (as described in Chapter 1).

Furthermore, whether these world/environment databases are shared (as shown here) or local to each sub-server depends on how load-balancing works. *EverQuest*'s zoning system, for example, can get away with having smaller environment databases that are controlled by individual sub-servers; only the character database needs to be shared with the other sub-servers, for when characters move between zones. This is paid for in other ways, of course, which we will discuss shortly.

The hardware implementation of a server cluster is as a bunch of PCs or beefier hardware (typically running some flavor of Unix) connected over a LAN. Figure 2.2 shows the usual pathways between elements of the server, but in fact any sub-server can talk to any other sub-server should the need[21] arise. Thus, passing players between sub-servers can be achieved either formally (through the databases) or informally (by direct negotiation between machines). Individual sub-servers may also be in contact with the user database, as shown in Fig. 2.1 (but not in Fig. 2.2, so as to prevent its suffering from dashed line death).

I should point out that academics are experimenting with other architectures, particularly the distributed kind much loved by Science Fiction (*Idoru* and *Otherland* both use it, for example). These, and those involving multicast, are not, however, likely to be used beyond academia due to the security and liability issues they raise.

Load Balancing

Fortunately, the arcane subject of load balancing is not something with which designers need concern themselves directly. However, they do need to be aware of the consequences of whatever solution is adopted by the technical experts, especially because they are likely to be consulted on the matter.

Ideally, a virtual world would run on a single, very powerful computer. For textual worlds, this is already the case[22]. Graphical worlds may go the same way[23], but for the moment there are still plenty of things on which newly available computational resources could be spent at the server side:

- Increasing the number of players present in each incarnation (100,000 players on 50 servers is one thing, but 100,000 players on 1 server is something else).
- Increasing the level of detail at which the virtual world functions (for example, leaving footprints in the snow that fade as it melts or as new snow falls).
- Improving the AI of the virtual world's denizens (both in quality and quantity).

It is therefore likely that virtual worlds will soak up whatever additional computing power is thrown their way for quite some time. Inexpensive machines will still be clustered, because it will always cost less to use eight computers of power *X* than to use one computer of power 8**X* (although management overheads degrade overall efficiency as new machines are added, which is why no one would use 64 computers of power *X*/8) 24.

So, given that distributed servers are here to stay, how does this affect the designer?

Well, the virtual world is too big to fit on one computer; therefore it must be partitioned over several computers.

Sort of....

If the virtual world were inert, that is, nothing ever happened in it, it wouldn't need any computers at all, it would just sit on a disk pack. It only needs computers when things happen in it. The issue is therefore one of ensuring that *activity* is spread across sub-servers such that they can all cope with the work they have; in other words, the computational load must be shared in a balanced way (hence *load balancing*).

The greatest source of activity in a virtual world is the player community. Every moment that they are in the virtual world, players are interacting with it. Merely moving from one location to another entails informing every other character that can (in the virtual world) see you do so. Yes, some activities are more of a drain on resources than others, but they tend not to be specific to particular playing styles. The issue is the sheer quantity of actions being performed, not the efficiency of individual actions. Load balancing in virtual worlds therefore generally means ensuring that roughly the same number of players is connected to each sub-server in the cluster.

The obvious way to do this would be to assign each incoming player to whichever sub-server has the fewest players. It turns out that it's quite difficult to do this without

introducing big overheads, though. To update the instantiation database to reflect an action, records need to be locked to prevent other sub-servers from also changing them at the same time (for example, if two characters attempted to pick up the same object simultaneously). The sub-server needs to lock all records that it could need during an action, perform the tests to ensure that the action is possible, make any necessary update requests, and unlock the records. This is a lot of locking/unlocking. It would be really handy if there were some way to block-book records in the instantiation database for long periods without relinquishing control of them. Are there any types of records for which this could easily be done?

Statistically, most actions performed by players involve movement[25].

The player wants to move their character from A to B, so the sub-server has to lock location B, check if it's empty, if so then move the player into location B, then unlock it. Location A must also be locked, so anyone wanting to do anything to the character that assumes it is in location A (for example, teleport it to location C) will not inadvertently screw up things. Many other common commands (particularly get/drop and those to do with communication) are also location-based.

For these reasons, servers typically partition responsibility for the virtual world along (virtual) geographic lines. A sub-server can lock location records in the instantiation database for extended periods; indeed, if the system is programmed correctly, it doesn't have to lock them at all—it has implicit control by mere virtue of the fact that none of the other sub-servers do.

To summarize the argument so far: We want to spread the players fairly evenly across sub-servers, but the obvious way would introduce too many overheads on database access; a far more efficient way to do it is to partition by geographical location. The question is: Would using this partitioning model give us load balancing?

The answer is that yes, it would. How, exactly, depends on the virtual world.

There are essentially two approaches: fixed load balancing and dynamic load balancing. The former, exemplified by *EverQuest*, assigns a predefined geographical location (a zone) to one sub-server[26]; the latter, exemplified by *Asheron's Call*, moves responsibility for geographical locations between sub-servers.

Fixed load balancing:

- Is easier to implement.
- Can partition the instantiation database and keep it local to the sub-server, for greater efficiency.

- Allows the client to figure out in advance what texture maps will be needed and preload them into graphics card memory.

Dynamic load balancing:

- Has seamless terrain (you can see to the horizon[27]).
- Has boundaries that are not physical (monsters chasing you don't get stuck at zone edges).
- Balances the load better.

What does all this mean for design?

It's an example of where technology imposes constraints. For fixed load balancing, zones can be created with greater individuality: The 'physical' barriers between them allow for radical change. Players can cross from one to another and *expect* to see something different on the other side. For a seamless system, sudden changes have to make more sense or they'll seem out of place.

The fact that content such as monsters can't cross zone boundaries means that players will use different tactics in such a world than they would in one where there was always the possibility that a creature they had royally annoyed could pursue them relentlessly. Victorian London's police forces couldn't (legally) cross precinct boundaries: Jack the Ripper would commit a murder in one police district and then run into another where the police were not allowed to follow; if they could have followed, he would have had to rethink his getaway strategy.

Zoned worlds have something of a problem with *flash crowds* (people appearing instantly in the same vicinity in response to interesting news; the term comes from a 1973 Larry Niven short story[28]). Most of the time, each sub-server will be handling similar numbers of players. However, sometimes something happens that causes everyone to want to be in the same general locale[29]. Maybe it's a rare spawning of an impressive dragon, or a social event such as a wedding or guild rally. Whatever, all of a sudden more people want access to a server than it can handle. The designer has to decide what to do when this happens. Do they simply show a "zone full" message if people try to enter it? Do they let them in and leave Customer Service to handle the resulting complaints about lag? Do they organize the virtual world such that it would be counter to its fiction for everyone to want to be in the same zone?

Dark Age of Camelot partitions its player base into three "realms." Members of one realm can't enter territory belonging to members of another realm. They can enter a no-man's land between two realms, however, which is where realm versus realm combat takes

place. The upshot of this is that unless there is a serious skewing of the *DAoC* player base, only a third of the players online will usually be present in any one realm. That's excellent for load balancing. What's more, realms are aggregations of zones, they're not zones in themselves. Each realm is made up of 13 outdoor zones (64K by 64K squares) plus five dungeons plus one city. There's no reason why a sub-server has to handle zones from only one realm; if (for some reason) 75% of the players are all in one realm, the sub-servers handling that realm will automatically have less load from the other (sparser) realms they're controlling.

Players of *EQ* notice how the sub-servers take responsibility for zones, but players of *DAoC* don't. Why not? Because of the latter game's world *design*.

Other Things Happen

Although players are the main source of the load on virtual worlds' supporting hardware, they are not the only one. Things can happen whether or not players are present. Some of it is mechanistic: The virtual world's sun rises and sets, its weather comes and goes, all irrespective of whether there are players around. For a highly detailed world, this could amount to considerable work (a breeze rustles individual leaves on a tree, one of which falls off to land in a stream that carries it lazily to a river and thence to the sea). Virtual worlds of this complexity are some way off at the moment, though.

What's more of an issue is the presence in the virtual world of virtual creatures. These are commonly known as *mobiles*[30] (*mobs* for short), and they represent the monsters and non-player characters that inhabit the virtual world. They are discussed in more detail in Chapter 4, "World Design," but what concerns us about them right now is that they need to behave in a believable manner. This requires artificial intelligence techniques, which gobble up computational resources like nothing else. Even simple path-finding is an insatiable consumer of CPU cycles. It would be great to have a virtual city with 100,000 virtual inhabitants, each making real-time decisions as to how to spend their virtual lives. We may have to wait some time before we get this, though.

If a designer wants more mobiles in the virtual world than the servers can handle, they have to offer solutions for managing these mobiles. The classic answer is to suspend processing of those mobiles whose actions would not be witnessed directly by players. What causes AI load is not the *number* of mobiles on a sub-server, but the number that are *active* at any one time.

Consider a group of goblins in a village. With no players in the vicinity, there's no point in having them do anything. Sure, a sub-server can move them around a bit when it's not unduly loaded, but players get priority. Only when a group of adventurers shows up is it time to activate the goblins so they can behave intelligently and give the players a run for their money. When the players leave, the dead goblins can respawn and wait for the next batch of adventurers.

Designers who want more mobiles than the programmers tell them they can have might be tempted to use this proximity activation approach. They have to realize, though, that every decision they make has consequences.

In this particular case, the consequences are on causality. If a tree falls in a desert, does it make a noise? Using proximity activation, it would never fall in the first place.

Consider a second goblin raiding party. It emerges from its camp, kills some villagers' sheep, and then returns home with the spoils. The villagers get angry and offer to pay players to kill the goblins.

It's an evening's quest.

It's an evening's quest that would never happen if the goblins stayed in their camp until a player happened upon them.

Yes, of course, plenty of ways around this instantly present themselves, but that's not the point: What's important is that the designer has to recognize that there may be a problem in the first place. Would your proposed solutions to the problem come with problems of their own? Would you have thought about that if I hadn't asked?

Virtual world design is about *consequences*.

The Client/Server Model

The server embodies the virtual world; the client translates it into a form the player can comprehend. Because the client is the player's window on the virtual world, designers have a lot to say about its look and feel. Much of this is a matter of taste and convention, though, and will therefore not be discussed here. Clearly, it does make a difference if your proposed spell-casting system is too complicated to be implemented for mouse and keyboard[31], but if this kind of thing isn't obvious to you, then you've no business being a designer anyway.

There are, however, engineering considerations specific to the implementation of virtual worlds that impact directly on the virtual world itself. These are unavoidable, and designers must be aware of them.

To illustrate: In theory, the same virtual world could be presented using different *skins*, so whereas one player might see a defense robot discharging an energy weapon, another might see a wizard wielding a wand. This is something that players would find novel, but it presents a major challenge to designers. Few genres map onto each other to this extent, so compromises would have to be made.

As it happens, this particular example is a red herring because the amount of artwork

necessary to support just one genre, let alone two, is considerable. The benefits aren't worth the cost. Textual worlds are easier to reskin than graphical ones in this respect, of course, but have less reason to want to do it[32]. In practice, using skins for graphical clients means altering the look and layout of the client's interface, not the look of what it displays of the virtual world. Hanging different curtains doesn't change the view through the window.

It turns out that there are only two issues relating to the client/server relationship that have concrete effects on the design of virtual worlds, but both of them are very important: synchronization and security.

Synchronization

It takes time for information to travel between the client and the server. It takes time for the server to execute commands. During this time, the client has to maintain its display of the virtual world. What it shows may not, therefore, be a true representation of what the server defines to be the current world state: The two aren't synchronized. In the real world, the sun might have spontaneously exploded 19 seconds ago, but you're not going to find out for another eight minutes. In a virtual world, someone may have quit four seconds before you loosed an arrow at him.

Lag due to server load can be addressed by buying in faster hardware and by optimizing code. There's little that can be done about lag due to communications, though. Even with a perfect connection, the speed of light through glass[33] is such that someone in Sydney playing in a virtual world with servers in San Diego[34] would experience a delay of over 0.06 seconds in each direction with a direct cable connection. The fact that their communication has to go through routers and isn't in a straight line brings it up to more like a third of a second. Throw in an analogue modem and you can add another third of a second. Lag happens.

Designers have to account for this by making nothing too time-sensitive. In a regular computer game, players can be expected to make timed runs through windmill sails or giant steam pistons or pendulum scythes, but in a virtual world there's no guarantee that when the player sees a gap on their client the server is actually implementing one. For this reason, players can never be called on to make reflex actions (although their *characters* can be), which means virtual worlds have little or no *twitch*.

Timed actions are possible, but the window for success needs to be at least four seconds in duration or some players are going to miss it while believing they hit it. Nevertheless, improvements in Internet reliability have led to attempts to bring the kind of response times common in first-person shooters to virtual worlds: *Planetside*[35] is the best-known pioneer.

Precise timing issues aside, virtual worlds are fairly robust in the face of lag. The exact moment that a player initiates an action is rarely important; when players agree to do something "at the same time," they know it's a fuzzy concept. As long as designers avoid doing anything deliberately that requires speed, there's rarely a problem.

Although most of the lag that players endure is fairly constant across a connection, not all of it is. Sometimes, lag can be intermittent: Things can work fine then suddenly halt for no apparent reason. This kind of lag is actually due to bandwidth issues that cause a service provider to invoke some kind of resource allocation scheme. It doesn't matter how good your broadband connection is, if your ISP isn't sending you the packets, then you're not going to see them. This kind of lag can last several seconds, whereupon all the packets that are buffered up are dispatched at once and service resumes as normal.

Clients expect regular packets of data from the server to update their local state. Most of the time, these arrive in a timely manner. However, it only takes someone using the same router as you to start downloading copious quantities of pornography and packets will inevitably be delayed. What does the client do in such circumstances?

The easiest solution is to do nothing. Just sit and wait until packets arrive, then update them in sequence. The problem with this is that the virtual world effectively freezes for the player until the updates arrive. The player can try to do things, but gets no response until the packets start flowing again. This isn't always a problem in textual worlds, but it looks very disturbing in graphical ones.

Most graphical clients therefore use a predictive model, whereby they continue moving objects along whatever course they were taking the last time information was available. If a character is running east, the chances are that when an update packet finally arrives it will still have the client running east, so the screen will be right without ever having frozen. Predictive models work well when their predictions are correct, which (fortunately) is most of the time. However, there are problems when they're wrong. If the character who was running east had stopped and turned north, there could be a serious discrepancy between their actual position and the position the delayed client is displaying them at.

To correct failed predictions, there are two approaches. The traditional way is to use the new information and just forget about the predictions. This results in an effect called *warping*, which originated in *Air Warrior*. Players would be on the tail of an enemy plane when suddenly it would disappear from their sights to rematerialize instantly a short distance away like it had made a hyperspace jump. Players even found ways to induce it, so they could plan the reappearance to gain a tactically superior position.

More modern clients apply gradual translations to the displayed position of an object so that it moves to its correct position smoothly (if a little more slowly and still looking highly suspicious).

For designers, this means that not only can't they use relative time (in three seconds) but they can't use relative space (dead ahead) either. The client might think a particular coordinate is slightly to the left of the player whereas the server knows it's slightly to the right. Commands that need coordinates therefore have to use absolute ones rather than relative ones. Again, when designers are aware of the problem and don't call for players to follow complex instructions in mazes or anything, virtual worlds are usually sturdy enough to cope; a little error in absolute positioning is fine. They do have to be aware first, though.

There is a special case, however, in that sometimes players want to do things to other players at a distance, for example, shoot an arrow at them. In this situation, not only might the archer's position be at odds with the server's definitive version, but so might the target's. The client knows that the player wants to shoot an arrow *at* a character, but doesn't know for sure that character's coordinates. The designer must decide whether the command is "shoot at a target" or "shoot at a location" (which hopefully contains the target). If the arrow has a timed flight (and if it's modeling a real arrow it certainly ought to), the potential for error increases even more.

Bleah!

Graphical virtual worlds are presented as being continuous. Although characters might actually occupy integral coordinates, they are animated such that they move smoothly between them, thereby giving the impression that they at times occupy the spaces "between" the coordinates.

Textual worlds can be continuous too, but they are usually contiguous. Locations can represent an area rather than a point, and several characters can occupy such locations[36] without causing an anomaly. Restricting horizontal movement to eight compass points means that using relative direction in such virtual worlds is a definite possibility. True to form, there are textual worlds that allow players to use both absolute (north, northeast, east, and so on) and relative (forward, ahead right, right, and so on) directions for movement, with the room description format (absolute or relative) also under player control. Graphical games can take relative coordinates from a client, but they have to transmit them as absolute ones.

A common mistake among inexperienced client authors is to take all this with a pinch of salt. So what if the client and server have slightly different ideas as to what is where pointing in whatever direction? There isn't going to be *that* much divergence. Okay, so maybe *occasionally* you see someone run through a solid windmill sail because there is a gap on their client, but that's hardly a show-stopper. Let the *client* decide if an arrow hits, rather than putting the burden on the server. There might be a few hits that should be misses and misses that should be hits, but it'll all even out over time.

This brings us neatly to the issue of security.

Security

Your client software *will* be hacked. For some virtual worlds (such as those with no game aspect to them), this won't necessarily matter. For the rest, it matters a great deal.

At the very least, it means that all packets sent from the client to the server have to be checked to see if they make sense. Error-correction at the hardware and transport layers should ensure that what arrives at the server is what was sent by the client, so why waste time checking for nonsense that isn't going to come? Well, the fact is that the server may not be talking to a *bona fide* client. It could be talking to a piece of code a player has written to masquerade as a client or to insert data into the client's packet stream. If the server receives nonsense but has no way of handling it, what happens then? Has someone acquired an ability to crash the server?

That's if it's even your server. Sometimes, groups of players will write their own server and persuade clients to speak to that instead. They can then design and play their own world in preference to yours—and all for free[37]!

Virtual world programmers don't have to make life easy for hackers—they can use encryption, own the screen, make very infrequent identity checks that packet-sniffing software may miss—but eventually their code will be reverse-engineered and people will figure out what's going on. Then they'll change it[38].

Important: Absolutely *no* decisions with regard to what happens in a virtual world can be delegated to a client. *No* decisions. That's *no* decisions.

Air Warrior's first client performed the necessary calculations to determine if a shot hit or missed. It was considered unfair for players to line up their sights exactly on a target, pull the trigger, but miss because their client was showing the target to be somewhere the server didn't think it was. All very laudable, until someone wrote a hack for the client such that whenever you fired, it sent a packet claiming you'd hit whichever plane was *closest* to your sights—irrespective of whether it was actually *in* them. Kesmai had to bring out a patch to fix it.

Programmers should *never* put world-critical code in the client. If they do, it can mean major, big-time fraud. Designers only need worry about this if they have to come up with a strategy for repairing the damage should it go undetected for too long (for example, groups of players giving themselves money and spending it in inflation- causing amounts).

Designers do have to worry about things such as automated play. Anything that requires similar actions to be performed repeatedly is usually easy to automate—it doesn't even need a client hack in many cases, as there are off-the-shelf tools that will do it. *Ultima*

Online's craft system was so boring that players were overjoyed when they discovered macro software intended for typists that enabled them to save their index fingers from repetitive stress injury. Computers can generally issue commands faster than players, too: Someone wrote their own client for *MUD1* that stuffed commands down the line so quickly that 30 seconds after having started to play, the automated character would be standing with arms full of vicious weapons and other useful kit, while everyone else was still pretty well empty-handed.

Designers should therefore avoid calling for anything that involves doing something again and again and again with no respite (and that can include movement). If you really want an action to take a lot of time, let characters do it as a background task while their players are offline. If speed is occasionally important (for example, for dramatic reasons), insist that the server programmers institute delays between processing the commands from any one player[39]. If some level of searching or exploration is required (for example, for a puzzle), put in moving obstacles or traps so a program can't easily find a solution using a brute-force method.

People usually want to automate tasks that are tedious. If you design something that you think many players might like to automate, consider the possibility that it could be intrinsically uninteresting. For example, players will come to understand a virtual world far better if they make their own maps, so a designer might want to encourage them to do so. However, the actual mechanics of mapping are so mind-numbing that there are auto-mapping programs around (for textual worlds) that try every exit from every room until they have produced a complete map that can then go on a web site. If you were hoping to stimulate a climate of exploration by holding back on embedded mapping tools in the client, it didn't work.

There are other ways in which wily players can subvert a client to put themselves at an advantage. Consider the case of a character executing a 360° turn. The client must be in close, rapid contact with the server to ensure that the necessary information is at hand to display the world as the sweep progresses. If it weren't, then the player could be looking at a half-blank screen or seeing characters pop up in the middle of the view that weren't there an instant earlier. However, as was pointed out earlier, the client can't usually rely on a super-fast connection to the server.

To solve this problem, the *EverQuest* server was configured to send the client more information than it strictly needed; if a player wanted to turn, then the positions of nearby objects would already be known and could be displayed without lag effects. To this end, the client was told the locations of all objects and players in the vicinity—not just those that character could "see"—along with other handy tidbits, such as what they were carrying and how many hit points they had. The geography also was kept permanently available on the player's PC.

This was an efficient approach, in that anyone could rotate or otherwise change the

camera angle with impunity without suffering staggered images. However, a number of players soon realized that if the information was present, it could be displayed whether or not it was in actual use. They wrote programs to give radar-like readouts of everyone and everything in the neighborhood. No longer did mobiles attack them from behind; no longer did they need to guess which were carrying loot. One such program, *ShowEQ*, was so useful a tool that Verant felt compelled to ban its use (leading to one of its many public relations disasters[40]).

It doesn't even have to be the client that's hacked. *Ultima Online* used the PC graphics card to control brightness, which meant that opportunistic players could override the client (by turning up the gamma correction) to get full daylight when they were supposed to be in total darkness. Great for ambushes!

If at all possible, designers should be adamant that no items of data are sent to the client that convey any information beyond what the player's character is entitled to know. Client programming being the inexact science that it is, however, chances are some additional information will have to be sent in advance, even if it's only texture maps on a CD-ROM. Designers should be appraised of this in advance, so they can adjust their designs accordingly. If you know that players are going to figure out where all nearby objects are anyway, give that information to them officially and be sure it's never of much use. Have new mobiles that teleport in from nowhere, if you want to keep them on their toes.

Okay, so, your programmers assure you that nothing is in the client that should be in the server, except for a few elements that you can design around. That's the security issue sorted, then!

Aw, you know it isn't. Your design itself could have security problems.

For example, suppose (having decided that fixed prices are a bad idea) you make your economy determine the price of goods based on local supply and demand. Suppose also that it allows people to buy and sell in bulk at the price for a single item. Normally, this wouldn't be a problem: If there are 300 swords for sale, it doesn't matter whether you buy one at 20 UOC41 or 10 at 20 UOC each. The price will rise the fewer swords are left, which is what you want—having bought 10 swords, the new price might be 22 UOC. You'd make the buy-back price be much lower than the sale price, so that anyone trying to sell back 10 swords they just bought for 200 UOC would receive maybe only 110 UOC for them instead of 220 UOC.

So far, so good.

What happens, though, for high-cost, low-production items? Maybe the local diamond mine unearths only five diamonds a week. When all five are available, the price at the diamond mart is 1,000 UOC; if only one is available, the price soars to 4,000 UOC. If

none are available, the supply is exhausted. In that situation, how much could a single diamond be sold back to the diamond mart for? Even with a 50% mark-down on the purchase price, it would still be at least 2,000 UOC. So, if someone were to buy all five diamonds at once for 1,000 UOC each, then sell them back all at once for 2,000 UOC each, they'd make 5,000 UOC each transaction. Your design has given them an engine for generating however much money they like! Augh!

When something is allowed by the virtual world but the designers wished it wasn't, it's known as an *exploit*. Exploits—design bugs—can ruin a virtual world[42] overnight.

Exploits aside, there are plenty of other ways that players can subvert a designer's well-meaning intentions. Identity theft—pretending to be someone else in real life—is fairly easy but is hardly the responsibility of the designer[43]. Character theft—pretending to be someone else's character in a virtual world—should not be easy, because it *is* the responsibility of the designer. In particular, if two players can use the same name for their characters then it's partly the designer's fault if one of them subsequently successfully pretends to be the other. This "name problem" is discussed in some depth in Chapter 3.

So, your design will have bugs. Players will find these and wring every advantage from them that they can. Even if they report them immediately, they'll feel you owe them for their honesty. So what do you do?

You can't prevent exploits, but you can take steps to minimize their number and impact. Detection and recovery are of critical importance. If possible, *log everything*. In *MUD2*, players would regularly complain that line noise[44] had severed their connection to the game and led to their character's demise. Instituting a "log everything" policy (player input/output transcripts and all server decisions) solved the problem at a stroke—90% of the time, players were exposed as having been active right up until the moment the dragon incinerated them or the wolf bit off their head or whatever, and therefore their impassioned pleas for resurrection amounted to cheating.

This degree of logging is not always possible for virtual worlds that use a lot of bandwidth, for example graphical game worlds. There are two approaches to it, but neither is particularly satisfactory: store events (from which exact circumstances can be reconstituted) or store client communications (which gives the player's actual viewpoint). Both of these create huge quantities of hard-to-search data. Customer service administrators may be able to *snoop*[45] on players as situations unfold, which can partially alleviate the problem, but if they arrive at a scene too late then the only alternative to logs is the presence of impartial witnesses (like there'll be many of *those*).

Detecting possible bugs and exploits is important, but it isn't itself enough. When it transpires that something really has gone unfortunately wrong, the ability to correct the consequences of it should absolutely *always* be available (even if it means a wholesale

reinstallation of the character database from a timed back-up). An incomplete ability to discover when things are going awry is inconvenient; an incomplete ability to recover damaged data is incompetent[46].

Part of the satisfaction of virtual world design lies in seeing your creation evolving, with things occurring within it that you hadn't anticipated but which make perfect sense. One of the less fortunate consequences, though, is that some of what happens you'll almost certainly wish hadn't, and will have to fix. The more contingency plans you have in place, the better, but you'll never be able to cover everything.

Still, if you want the unpredictable, you can't complain when you get it.

Theory and Practice

How virtual worlds *ought* to be put together and how they *are* put together are two different things. You can spend 30 hours in a classroom learning how to drive a car, but fifteen minutes at the wheel is going to teach you a whole lot more.

At times, the practice is more useful than the theory.

This section discusses some of the things that look like they should be important, but aren't, along with those that look like they shouldn't be important, but are. It's a bit of a mixed bag, but is fairly representative of the kind of hidden depth (or lack of same) that only becomes apparent when you actually develop a virtual world. There are plenty more, but I'll leave them for you to discover them for yourself; as I said, you'll learn more by doing than by reading about it.

Modes

When you start up your web browser, what page does it point at? Surprisingly, for most people the answer is "whatever my ISP set it to when I installed its software." Their first view of the World Wide Web every morning is what their ISP shows them.

Virtual worlds have many ways to let players customize their experiences. In a textual world, for example, some people want full room descriptions the whole time, and some want short ones the whole time. There are commands that let players choose for themselves which of these to go with. However, one option will be the default, and that's the one that newbies will use. In *MUD1*, the first time you entered a room you saw its full description, and on later visits you saw the short version. This helped stop newbies from getting lost. Later, they would use exclusively verbose descriptions when making accurate maps and exclusively brief descriptions at all other times, but later is later. When they started, they got the default, and the default said, "Explore!"

Defaults set the tone of virtual worlds, because all newbies play under them. As they

become more experienced, they'll inevitably customize some of the settings; most options, however, will stay at the default. Thus, the designer's choice of defaults can have long-term influences on how a virtual world is perceived. Defaults are more important than they look.

To illustrate this, let's take a look at *modes*, because in a sense they establish what the world is "about."

All virtual worlds assume a hardware device for entering freeform data; it's usually a keyboard. The player types some text, hits return, and—what? It depends on the current mode. In a textual world, the line is usually interpreted as a command.

```
open door
north
get book
```

This would be *command mode*.

In a graphical world, the line is usually interpreted as speech.

```
Follow me!
What happened? You were supposed to be following!
Hey! That's mine!
```

This would be *conversation mode*. Chat rooms usually default to conversation mode, too.

So where's the hidden depth here? It's not like you can't act in conversation mode or speak in command mode.

When a newbie enters a world for which the default is command mode, the message that the world is sending them is that this is a place where you can *do* things: It emphasizes freedom to act on the world. If the default is conversation mode, the message is this is a place where you can *communicate*: It emphasizes freedom to interact with other players. You might expect, therefore, that players of textual worlds do more and players of graphical worlds say more.

Actually, for graphical virtual worlds the default is to use the less-than-freeform mouse most of the time, forcing a greater distinction between limited doing and unlimited saying. If you want to talk or attempt anything complicated, you have to stop playing to do so. This is sound advice for crowd control, but somewhat dissatisfying for the individuals in the crowd so controlled.

The designer sets the default mode. The default mode shapes the style of play. The designer can therefore encourage or discourage a style of play by changing the default mode. Thus, a simple, almost throwaway design decision can have a long-term influence

on a virtual world's ethos.

Note that although the main choice is between command mode and conversation mode, there are other modes used in virtual worlds. The convention has evolved that the first character in a line is used to switch modes for the remainder of that line. There will often be an option to turn a mode on/off until further notice, but no standard syntax for this has yet emerged.

Table 2.1 shows the most common modes in use, along with the (sometimes conflicting) options for the leading characters used to switch them on.

Table 2.1 Common Modes>

Mode	Leading Characters	Explanation
Command	> / .	Input is a direct command to the server.
Conversation	' " ´	Input is a parameter to the say command.
Coding	@	Input is a scripting language command.
Acting	; :	Input is a parameter to the pose/act/emote command.
Help	?	Input is a parameter to the help command.
Switch	/ \ $	Input is for the client or front-end, rather than the server.

Virtual Reality

Read a selection of Science Fiction stories about virtual worlds, and you'll soon discover that the following are inevitabilities:

- True intelligence will emerge from the hideously complex machinations performed by the virtual world engine, with unnerving consequences for human morality.
- Unscrupulous people will transfer their consciousnesses into hardware in order to live forever, muahahaha.

- Virtual worlds will be experienced through virtual reality interfaces so good that the virtual will be indistinguishable from the real.

The first two of these are not of immediate interest to the designers of virtual worlds, being somewhat distant prospects at the moment. Virtual reality (VR), however, while as yet nowhere near the quality envisaged in speculative novels, is far more accessible. Why not create a virtual world with a virtual reality interface? It would attract media attention, if nothing else.

For a virtual world with a closed user base, a VR interface is indeed a reasonable proposition. A small academic research community or a large industrial or military training establishment would be able to experiment with the idea and get fruitful, worthwhile results.

For an open user base, though, VR isn't yet an option. It's simply not value-added enough for developers at the moment, that is, the costs of putting it in outweigh the benefits. Until the technology improves[47] and the installed base reaches some critical mass (whether because of computer games, 3D movies, 3D video cameras, or something else), VR would be available to but a few, lucky or wealthy players.

There is an argument, though, that VR access to a major virtual world could *itself* be enough of a draw that people would be willing to acquire the necessary hardware to play. Of course, for this to happen VR would have to bring something quite special to the virtual world experience.

So what would that be? It depends on the set-up, of course, but in the first instance the chances are that a basic VR kit would mean full-vision headsets with surround sound and orientation detection, plus gloves incorporating movement sensors and some degree of positive tactile feedback. Using such an interface, the virtual world could

- Let you see it in 3D[48]
- Let you hear it in 3D
- Tell in what direction you're looking
- Tell what your hand is doing
- Deliver sensation to your fingers
- Accept speech input

What advantages would accessing a virtual world through such a VR interface confer over the traditional mouse and/or keyboard approach?

There's some convenience in being able to control what you see and do by means of head and hand movements, countered only by the slight inconvenience in having to sport the equipment needed to support it. Nevertheless, it's unlikely that a player using a keyboard and mouse would be at a serious disadvantage compared with someone kitted out in the modest VR set-up described here, at least in terms of their ability to control their character in a virtual world.

Speech as an input form seems, at first glance, to be an attractive proposition. It wouldn't be all that great in command mode, because speech recognition software still has a long way to go before it could be of genuine utility. In conversation mode, though, it would be much more effective—but only if it could be completely disguised. Again, although voices can easily be distorted using today's technology, it's likely to be some time before they can be altered so well that they don't sound altered. But why is some form of disguise necessary anyway?

One of the main attractions of virtual worlds is the ability to be whoever you want to be; anything in the virtual world that anchors you to who you are in the real world is a disincentive. When you hear an elf in the middle of a sylvan wood speaking a New York accent, or a mighty-thewed, bare-chested barbarian who sounds like a schoolmarm, sadly reality is intruding a little too much. If voices aren't disguised, players aren't disguised, and then the virtual world is just another aspect of the real one.

3D vision and sound, and to some extent tactile feedback, are the real gold for VR. They make the virtual world more persuasive, which helps players immerse themselves in their (virtual) surroundings. The change won't be to everyone's taste—some people are always going to prefer text, for example, on account of how it speaks to the imagination rather than to the senses—but it'll help anyone who gets on well with graphics[49]. That said, flat images are already quite capable of immersing players into a virtual world, and they don't have to cut off great swathes of the real world to do it. Players *like* being immersed in virtual worlds, and will happily ignore negative cues if they get sufficient positive ones that they can will themselves to suspend their disbelief. You don't have to trick them into it; they'll go for it anyway. Giving them more 'realistic' visual and auditory stimulation will provide additional signals, but are they mere luxury? Designers can and do encourage immersion by many other means[50].

So, would VR merely amount to a marketing ploy to attract newbies?

Cynics might suggest that this is all that adding graphics to virtual worlds ever did, but even they would have to concede that it worked. The allegation is unfair anyway, in that a graphical interface to a virtual world does actually make a tangible difference. If, for example, in a textual virtual world you happened on a gathering of 50 characters, you'd have to read 50 names to see if there was anyone there with whom you wanted a chat. Spotting a friend from among 50 faces in a graphical virtual world is far, far quicker[51].

Would a VR interface add some genuinely useful feature that flat graphics and two-point stereo don't?

Frankly, probably not. Unless the virtual world has very counter-intuitive physics (for example, the further away an object is, the bigger it appears), you're not going to learn much more about it from experiencing it in 3D than you can already figure out from the perspective and motion conventions of 2D.

Unless...

Unless designers can exploit the extra capabilities that a VR interface imparts. Instead of clicking where you want your arrow to land, you raise and point your bow, then release the string—with true depth to your vision, you can now judge trajectories. Instead of running to attack the lightly armed man, you flee—you can now see he's a giant. Instead of walking in the gaps between well-spaced trees, you push through dense forest—the tree trunks no longer merge to look like fences when they're close together. VR really does offer new prospects.

Unfortunately, virtual world design is not yet sufficiently developed to make the best of this. Game designers still have to make more use of sound as a gameplay element, so it seems unlikely that virtual world designers will successfully embrace VR the moment it becomes available. There may be some centerpieces that show off the technology, but that's likely to be all. When you catch a movie on TV and notice that people seem to be pointing or throwing things toward the camera a lot, pretty soon you realize you're watching an old 3D movie in 2D. Directors never really came to grips with 3D in movies. Will designers fare better when VR comes to virtual worlds? Or will it be a case of, "Aww, man, not the guys with pikes again!"?

The theory is good, but the practice may take some time to measure up to it.

Extensibility

Virtual worlds are designed such that they can be extended over time.

Why? To add content, to correct imbalances, to allow for more simultaneous players—there are lots of reasons.

Who extends the world? The live team.

Who created the world? The dev team.

Is the dev team a subset of the live team? At some point, the answer has to be "no."

For large-scale, graphical virtual worlds, the answer is usually "no" immediately, because

the live team is a separate entity from the dev team. Even for small-scale worlds, or those for which the live team is built from the dev team, the original designers aren't going to be around indefinitely. Students who create virtual worlds eventually leave college; professional designers who create successful worlds are offered new opportunities[52]; people move, their circumstances change: Ultimately, no one lives forever.

The live team may therefore differ in attitude to the dev team[53] when it comes to assessing how the virtual world "should" work. There are many opportunities for divergence:

- The live team may fail to understand aspects of the design, seeing flaws where there are none.
- The live team may misunderstand the dev team's intentions, believing they're doing the right thing when they're not.
- The live team may have a different overall philosophy, and "correct" the design where it runs counter to this.
- The dev team's design might fail when exposed to real players.
- The dev team may have higher quality staff than the live team (or *vice versa*), with a consequently better handle on things.

Because the live team is in control, there is great scope for a virtual world to shift away from the designers' original vision over time. This isn't necessarily a bad thing—adapting to circumstances is how systems evolve, after all. Neither, though, is it necessarily a good thing—survival of the fittest is great when you're one of the fittest, but not so great otherwise.

At this level, it's a classic conflict between theory (what the dev team wants) and practice (what the live team gets). It's a little more complicated than that, though. The live team has to deal with players, every single one of whom believes they know just as much about virtual world design (if not more) than anyone in the live team—and are prepared to argue the point.

The bad news is that players know nothing about virtual world design. Nothing whatsoever.

Well, that's not *strictly* true. A very small fraction of them do[54], but these are generally indistinguishable from normal players except in the benighted eyes of people who actually *do* know about virtual world design. Message boards are full of erudite arguments by players able to put their opinions cogently, politely, and convincingly. That

doesn't mean they're *right*, though. It's like listening to a religious discussion between people of a religion different than your own: They obviously know exactly what they're talking about, in great and profound detail, but from your point of view they're at least misinformed and at most completely misguided. Player discussions are frequently like that: Designers can recognize some truths in what is being said, but these are so mixed with dogma, rhetoric, and downright falsehoods that the conclusions they reach are often bizarre and irrelevant (whenever they reach conclusions at all, that is).

Yes, I realize I've just insulted about three million people, here.

It's not that most players *don't* know about virtual world design, but that their knowledge is too personal. As mentioned in Chapter 1, players tend to view all virtual worlds in the context of the one they "grew up" playing. If a new idea is suggested, many players will immediately consider how it would fit into their preferred virtual world, whether or not the virtual world for which it is intended is remotely similar. If the debate actually concerns "their" virtual world, they'll figure out the short-term repercussions of their own playing style and use that as a basis to decide whether they're for or against. They'll only refer to long-term effects or other playing styles when they're trying to win allies or to convince the live team that they are responsible people whose opinions should carry weight.

This is because actually playing a virtual world adds a *subjective* element to all discussion. Designers have to be *objective*. If you can play a virtual world for fun, it's very hard to be a designer; every decision you make is related to your own experiences as a player. Designers *can't* play virtual worlds for fun. When I enter a virtual world, all I see is the machinery, the forces at work, the interactions—it's intellectually interesting and can be artistically exciting, but it isn't *fun*. Other designers are the same: The price you pay for being able to deconstruct a virtual world is that of being unable *not* to deconstruct it. Magic isn't magic when you know how the trick is done.

That's why most players aren't good at design. They still sense the magic.

Unfortunately for the live team, that's not quite how the players themselves see it. Players want improvements made to their virtual world, and most of the time they are appreciative—even fanatical—of the live team's efforts. When they *aren't*, though, oh boy, do they ever let the live team know! The pressure can be phenomenal. It can reach the stage where it's more gainful to implement the change that everyone is screaming for than it is to answer all the emails they'd send if you continued to hold out.

At this point, the live team often surrenders. Top-down design gives way to bottom-up experiment.

Whether this fills the original dev team with pride or despair depends on the extent to which they'd planned for its occurrence.

Footnotes

1. Marketing people consider themselves to have expert knowledge of what players like and dislike. They may indeed have this knowledge. The friction comes from when they try to tell designers what features should be added/removed/changed to exploit such knowledge.

2. It's particularly important that programmers don't feel that they can *ad lib* features of their own.

3. Bridgette Patrovsky and Jessica Mulligan: *Developing Online Games: An Insider's Guide*. Indianapolis, New Riders Publishing, 2003.

4. Andrew Rollings and Dave Morris: *Game Architecture and Design*. Scottsdale AZ, Coriolis, 2000.

5. In other words, it looks a lot like the strategy guide that will be sold when the product ships, except with the most boring parts removed.

6. The artwork bible is often built up incrementally (and even informally) during the production phase, as it isn't always needed this early. This state of affairs isn't likely to last for much longer, though.

7. Programmers may resist using a third-party graphics engine, because every programmer who ever worked in the field thinks they can do it better than what's already out there. Strangely, this self-belief rarely extends to the less fun domains of databases and billing systems.

8. Also known as the *implementation* phase.

9. Or less, if the investment money runs out.

10. Or *pipeline*.

11. This is what's supposed to happen, anyway. Unfortunately, artists are often proudly nontechnical and will look for any excuse not to use such a system.

12. The task is known as *level design* in conventional games development terminology, but there's no real industry-standard name for it in virtual world creation. "World-building" is coming into fashion, but is somewhat ambiguous.

13. Yes, developers have friends. The rationale for getting them into the beta is that honest opinions are needed from people who can be trusted. Trusted not to mind nepotism, this would be....

14. They will, however, be minuscule in number compared to the vast hordes that *think* they've had a brilliant idea but are sadly mistaken.

15. Also known as the "commercial exploitation phase" in business school language.

16. For non-3D virtual worlds, such as *Ultima Online*, this could mean no artists at all.

17. This term is an *Ultima Online* fiction to explain how come there are multiple copies of a supposedly single world. It's as if a mirror that reflected the world was shattered into a myriad of tiny pieces, each such shard reflecting the original world but in a slightly different way.

18. In the sense of allowing no other processes to run while the client is running, thereby making life difficult for hackers. It also makes life difficult for non-hackers, though, so clients may settle for merely owning the screen. Players still don't generally appreciate the gesture.

19. This was known as the *persona file* in *MUD1*.

20. For virtual worlds where characters can move between incarnations, a single-character database may be necessary that is shared among all virtual worlds (like the user database). On the other hand, if transfers have to be done manually then developers can reasonably charge for the service (this is *EverQuest*'s approach; they'd made over a million dollars from it by mid-2002).

21. The precise definition of "need" here depends on the virtual world.

22. Processor speed finally ceased to be an issue for *MUD2* when I replaced its 33MHz server with a swanky new 50MHz one.

23. *Meridian 59* has a single-server architecture, which limited it to 200 players per incarnation at launch. *Shadowbane* also has a single-server architecture, but runs on somewhat more powerful hardware.

24. Actually, no one has yet tried. Although this kind of parallel processing architecture frequently runs into problems for many business computing applications, it may be that for certain partitionings of virtual worlds it's fine.

25. It's over 50% for *MUD2*; graphical virtual worlds have an even higher figure—90% or more—because players have to take more steps to get anywhere. This is changing with the arrival of click interfaces, in which you click where you want to go rather than point where you want to go.

26. It should be pointed out that sub-servers can handle more than one zone at once.

27. Invisible cross-server boundaries are also possible with tessellated worlds such as *Ultima Online* that have fixed load balancing. However, odd things may happen when interactions occur over server boundaries (for example, shooting arrows across them). In *Asheron's Call*, which has dynamic load balancing, characters that interact are moved to the same server so as to reduce interserver communication confusion.

28. Larry Niven, *Flash Crowd*. Larry Niven, *The Flight of the Horse*. New York, Ballantine, 1973.

29. If this is very focused, the same problem can afflict seamless worlds, too. It's less frequent than for zoned worlds, but because of this it can be harder to handle when it does happen.

30. From *MUD1*, "mobile objects." I called them that because creatures moving in a controlled but unpredictable way are like the kind of "mobiles" that hang from ceilings. Well, I was in kind of a hurry...

31. Well, only for spell-casters. There's no theoretical reason why different character classes can't have interfaces customized for what they spend most of their time doing.

32. Translation schemes for moving functionally equivalent objects between differently themed parts of a virtual world (and even between virtual worlds) do exist, however.

33. Approximately 197,000 kilometers per second.

34. Approximately 12,083 kilometers from Sydney at sea level.

35. http://planetside.station.sony.com/

36. Which are called *rooms*.

37. Examples include *EQemu* (http://www.eqemu.net/) and *EthernalQuest* (http://www.ethernalquest.com) for *EverQuest*, and *Sphere* (http://www.sphereserver.com/), *UOX* (http://www.uox3dev.net/) and *Epsilon* (http://www.epsilon.escend.net/) for *Ultima Online*. See *The Smithys Anvil* (http://www.smithysanvil.com/) for more.

38. This is what developers mean by the term *arms race*.

39. Actually it's more fashionable not to enforce delays, but have the server—or even other clients—perform spot checks to see if someone is cheating. Needless to say, if clients get to report who is cheating, sooner or later they'll be hacked so as to accuse innocent players of doing it.

40. Verant changed the end user license agreement to give them the right to search your

home PC for programs that in their view could interfere with the proper running of *EverQuest*. The ensuing full-scale revolt helped them to reconsider.

41. Units of Currency (UOC)

42. They can, of course, occur in any kind of simulation software, they're just at their worst in virtual worlds. I found an exploit similar to the diamond example in a single-player trading game (Ascaron's *The Patrician*) that finally allowed me to beat it after five years of trying. They took it out for *The Patrician II*.

43. In 2000, I spent several months building a reputation in *EverQuest* despite not having played the game at any point during that period. Someone made out they were me, and other people believed them.

44. In the old days, modems did not have hardware error-correction. A crackly phone line meant crackly data.

45. Copy output from a player's screen to their own screen, not necessarily with the knowledge of that player.

46. And if players realize you don't have accurate snapshots of events, that's when the problems *really* begin.

47. Minimally, it mustn't induce blinding headaches in its users.

48. Stereoscopically. As pointed out in Chapter 1, for virtual worlds the term "3D graphics" usually means that the graphics are displaying something that is 3D, not that the image so displayed is itself 3D. A VR headset could be expected to present separate images to each eye, giving a much truer sense of 3D than does a flat plane.

49. Well, almost anyone. If you only have sight in one eye, 3D visual effects aren't going to impress you. Similarly, if (like me) you can't tell where sounds are coming from, expensive 3D auditory set pieces are going to pass unnoticed.

50. Chapter 3 features an in-depth discussion of the concept of immersion in virtual worlds.

51. I should point out that long-term players of textual worlds can acquire the ability to do something similar—glance at a list of names and pick out a friend without having to read (or even speed-read) anything. It's a skill that comes only with time, however, whereas face-recognition is something humans learn in infancy (if indeed it's not already hardwired into the brain at birth). Of course, this point about facial recognition would carry more weight if people didn't all choose the same faces and outfits so everyone ends up relying on the names above characters' heads anyway.

52. It’s quite a different story for professional designers who create unsuccessful worlds, but the effect is the same: They end up not working on the project.

53. Indeed, they may differ in attitude to the live team of a few months earlier.

54 Hopefully, a fraction that will increase as more and more players read this book.