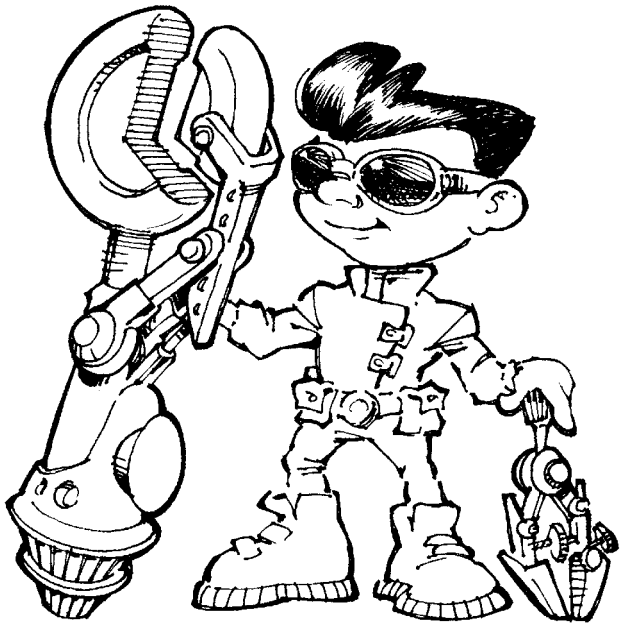
Chapter 21:

Designing Design Tools

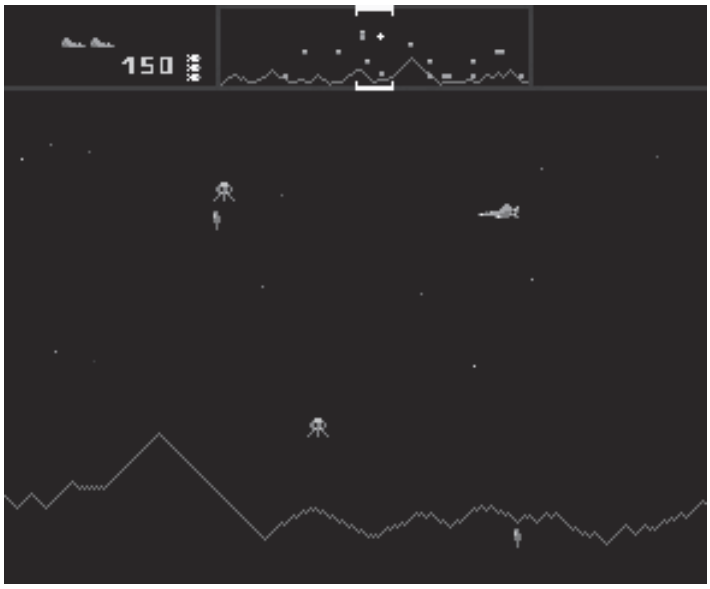


"Man is a tool-using animal . . . Without tools he is nothing, with tools he is all."
— Thomas Carlyle

An integral part of developing a good game is creating compelling content for that game. In order to create superior content, the design team will need to be equipped with well-designed, robust game creation tools. Therefore, one can conclude that designing a good game is about designing good game creation tools.

Other than the development environments the programmers use to compile the game's code and the graphics packages the artists use to make the game's art, the most commonly used game creation tool is the level editor. What distinguishes this tool from the others I mentioned is that it is typically built specifically for a project or, at least, for the engine the team is using to power the game. It is the responsibility of the development team to make this level editor as powerful as it can be in order to facilitate the job

of the level designers and allow them to make the best game-world possible.



The simple levels found in early games such as *Defender* did not require a sophisticated level editor to be created.

Of course, not every game has levels. Many of the classic arcade games from the early 1980s such as *Missile Command* or *Space Invaders* do not have levels as we think of them now. And the games that did, such as *Defender* or *Tempest*, certainly did not require sophisticated level editors to create their game-worlds. Sports titles have levels that are quite simple and mostly require the construction of visually pleasing stadiums to surround the gameplay. Games like *Civilization* and *SimCity* auto-generate the basis of a level using randomness combined with specific internal rules that will ensure the map will be fun to play. They then allow the players and AI to build the rest themselves, during the game. I discuss the nature of levels in games in more detail in Chapter 23, "Level Design." Many modern games employ sophisticated levels, levels that have a tremendous impact on the shape and form of the gameplay that takes place on them. These games demand that their development team create an editor with which the level designers can build the game-world.

Surprisingly, many development teams fail to invest enough programming time in making their tools as strong as possible. Often teams have no idea what is standard in other tools used in the industry. Frequently, not enough time is invested in preplanning and thoroughly designing how a level editor will work. As a result of all of these factors, it is often many months before the level design tools are reasonable to use. Frequently a programmer is stuck with implementing or improving the level editor as "extra" work on an already full schedule, and is forced to use the trusty "code like hell" method of implementation to get it done in time. Often, key timesaving features are not added until midway through a project, by which time the game's designers are already hopelessly behind in their own work. Up-front investment in the tools and their continued support throughout the project is certainly a lot of work, but in the end it is time well spent.

Desired Functionality

So what sort of functionality should a level editor include? Many might suggest an important part of any level editor is having hot keys hooked up to all the important functionality. Others would recommend plenty of configurable settings that allow different designers to turn on and off the features they prefer, when they need to use them. It goes without saying that a level editor should be stable enough that a designer can use it for a number of hours without it locking up. But these suggestions are all obvious; they are the bare minimum that an editor should do to be useful. What sorts of features should be included to allow an editor to truly shine, to empower designers to do the best work possible?

Visualizing the Level

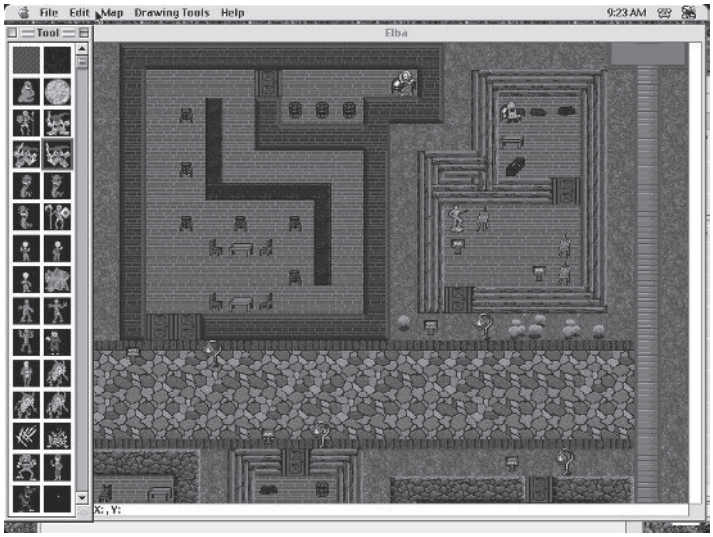
The most important objective for a world creation tool must be to allow the designer to see the world he is creating while simultaneously enabling him to make modifications to it. This is often called What You See Is What You Get (WYSIWYG) in the domain of word processors and desktop publishing packages, but is not something that level editors are universally good at. I will call such a WYSIWYG view the "player's view" since it represents what players will see when they play the game. The world the designer is crafting should be seen in this player's view window using the same rendering engine the game itself will employ, whether this means 2D or 3D, sprites or models, software driven or hardware accelerated. This seems to be the most important feature of any level editor. How can a designer hope to create a good looking world if he must first tweak the world's settings in the editor and then run a separate application to see how it looks in the game?

The designer should be easily able to move the camera in this player's view so that he can quickly maneuver it to whatever section of the map he needs to see in order to work on the level. This movement is probably best accomplished with a simple "flight" mode in which the designer can control the camera's position using simple movement and turning keys. In this mode the camera should move without colliding with geometry or other game-world objects. Though one may also want to provide a mode for the player's view where the designer can maneuver through the game-world as players will in the final game, the editor should always allow the designer to move around the level unconstrained. In order to finely edit a level, the designer must be able to look closely at whatever he wants without having to worry that a tree blocks his way.

Every difference that exists in what the designer sees in the editor and what will show up in the game will make the levels look that much worse. Suppose only the unlit view is available in the editor, while the game itself has fancy lighting that creates pronounced shadows. This will create frustration for the designer, since he will not be able to easily tell how the level will appear in the game. Sure, the level looks great when playing, but while in the editor he has to guess what the lighting will be like and how the changes he makes will affect the final level. A limitation like this would be especially unacceptable if the gameplay relied more on lighting and shadows.

Of course, the world as it will appear in the game is not always the best view from which to edit that world. For this reason, level editors often need to include an "editing view" in addition to the player's view. The editing view is often top-down, but may also

consist of a rotatable wire-frame view or multiple views. The last option is particularly useful for the editing of 3D game-worlds. For instance, the popular *Quake* engine editing tool Worldcraft, which was used to create all the levels in *Half-Life*, provides the designer with the popular "tri-view" setup, with which the designer can see top-down (along the Y axis), from one side (along the X axis), and from another side (along the Z axis) simultaneously in three separate windows. The three side views appear in addition to a 3D "player's view" window. Having multiple views is of particular importance for editing complex, overlapping 3D architecture, such as one finds in *Quake* levels. In contrast to the player's view window, which exists in order to show the designer exactly what the level will look like in the game, the editing view's purpose is to allow the designer to easily modify and shape what he sees in the player's view window. Of course, the editor should allow editing views and a player's view to be all up on the screen simultaneously, and the changes made in one window should be instantly reflected in all the views. Also, just because I refer to one view as the "player's view" and one as the "editing view" does not mean users shouldn't be able to edit all the game content in all the views.



The view provided in the Zoner level editor for *Odyssey* was perfectly suited to editing a 2D world.

In some cases there may not be a need for separate editor and player views. For instance, in a 2D world such as was found in my first game, *Odyssey: The Legend of Nemesis,* the player's view of the world may be perfectly suited to editing the levels. While I worked on the many levels for that game, not once did I wish for another view of the game-world. Similarly, in *StarCraft*, the representation of the world as it appears in the game is sufficiently clear to allow the designer to make modifications to it directly. For this reason, the *StarCraft* Campaign Editor provides only a player's view window for the designer to edit in. However, for the *StarCraft* editor, it might have been beneficial to provide a separate editing view. Because of the isometric view the game uses, a view which can sometimes be confusing to look at, a strictly top-down view in which the designer could edit his level could have been quite useful in the placing and manipulating of units and other game elements. The *StarCraft* Campaign Editor does include a

top-down "mini-map" of the level being created, but the designer cannot actually change the level using that view, nor is the mini-map large enough to allow for easy editing.

The Big Picture

I have argued that it is important for a game's level editor to allow the designer to see the level exactly as he will see it in the final game, but the player's view window does not always need to represent exactly what players will see. It can be quite useful if the level editor can also show the designer various extra information about the level that will assist in that level's creation. For instance, suppose that the game being developed involves various monsters maneuvering the level on predetermined paths. Being able to see exactly where these paths go is key to understanding how the level functions and to making sure the paths are set up properly.

In many level editors, this sort of level functionality information is communicated in the editing view but not in the player's view, but it makes sense to display this data in both places. Certainly the player's view window should not always be filled up with this sort of level functionality information, but the ability to turn on and off the rendering of different data can be quite useful in setting up the level's behaviors. This is especially true for 3D games. Returning to the path example, why should the designer have to extrapolate in his head from the 2D top-down or side editing view exactly where a path will end up in the 3D view? Instead, the editor should just draw it for him, so there is no guesswork.

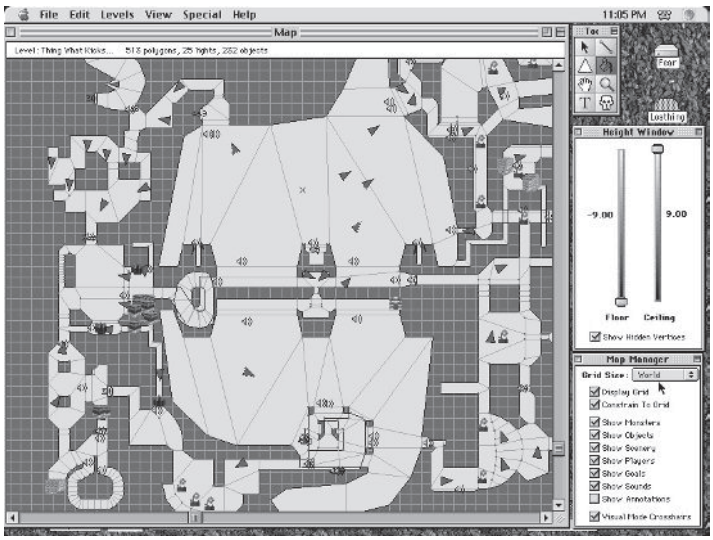
When working on *Centipede 3D*, a programmer was adding code that would prevent players from traveling up slopes that were too steep. In order to debug this new slope-restriction code, he added functionality to the level editor that allowed it to toggle on and off lines that separated the different triangles that made up the landscape. These lines would change color depending on whether or not a given edge could be crossed by players. The triangles themselves were marked with a red X if they were too steep for players to rest on. The programmer added this functionality primarily to aid in his debugging of the slope-restriction code, never realizing what a boon it would be to the level designers. Now the designers could see exactly where players could and could not travel on the level. An even better side effect was the rendering of the triangle boundaries, which created a sort of wire-frame view of the landscape, functionality that had not previously been available in the editor. This then vastly simplified the editing of geometry, for now the designers could see exactly which triangles created which slopes and then modify the level accordingly. The addition of the wire-frame view and the slope-restriction markers led directly to better, more refined geometry in the final game. And the beauty of this functionality was that it could be turned on and off in the editor, so if the designer wanted to see how the level looked he could turn it off, and if he wanted to see how it functioned he could turn it on.

As with paths, it may also be useful if the designer can turn on and off the rendering of objects such as triggers and other normally invisible objects. Similarly, it can be enormously helpful to display the bounding information for the objects in the world (which often does not exactly match the visual composition of the object's sprite or model), so the designer can easily observe how the bounding information will impact the ability of players and NPCs to navigate the game-world. Marking off where players can and

cannot go can be quite useful as well. And again, each part of this functionality data should be easily toggled on or off via hot key, menu, or button, so that the designer has the choice of seeing exactly the data he needs for the problem he is working on. And the data should absolutely be rendered in the player's view window in addition to the editing view, so that the designer can see exactly how the trigger, path, slope restriction, or other object is placed in the game-world, without having to guess from a top-down view. By using a visually authentic view of the game-world that can also display game behavior data, the designer is able to work on a level's aesthetic qualities just as well as its gameplay attributes.

Jumping into the Game

For games where players are manipulating a character through a world, it is important for the designer to be easily able to know how the level "feels" to navigate. To this end, in addition to having the player's view of the world represent what players will see in the game, it can be quite useful to allow the designer to actually maneuver in this view as he would in the actual game. With this sort of addition, the designer is able to test whether players will be able to make a certain jump, how it will feel to navigate a particular "S" curve, and whether or not the players' character moves smoothly up a set of stairs. In addition to this "gameplay" mode, the level editor should retain the unconstrained "flight" mode I mentioned previously.



Bungie's Forge level editor for the *Marathon* engine included a "visual mode" where the designer could actually maneuver through his level exactly as a player would in the game.

The Vulcan editor for Bungie's *Marathon* engine was particularly well suited to allowing the designer to test the "feel" of the level while constructing it. The *Marathon* technology was similar but a bit better than *Doom*'s, and was licensed for use in a number of other games, including my game *Damage Incorporated.* Vulcan was subsequently revised, renamed Forge, and released with the final game in the series, *Marathon Infinity.* Vulcan/Forge allowed for a "visual mode," which functioned as a player's view window. In visual mode the designer could navigate the world just as players would in the final game. The shortcoming of this was that the designer was unable to edit the

world, aside from texture and lighting placement, while in this view. This was no doubt due to the speed of processors available when the editor was created, and the comparatively small size of affordable monitors at the time. Today, it would definitely be expected that this view could be opened simultaneously with other views, allowing users to switch back and forth from it trivially with a mouse click or hot key press. Similarly, designers would expect to be able to edit more than just the lighting and texturing in this view. Nonetheless, the visual mode in Vulcan was quite useful, and the switch from editing mode to gameplay mode was fast enough to allow the designer to make a change, see how it felt, and then switch back to make more changes as necessary.

Of course, one might conclude that the next logical step is to allow the designer to actually play the game in the player's view. In this way the designer can see how well different mechanisms function, and what sort of a challenge different adversaries will present. However, this opens the programmer up to a large amount of implementation difficulties. In order for game-world objects to function as they do in the game, many objects will move from the position they start out in when players begin the level. For instance, an aggressive troll might run toward the player character and attack. Do these moving objects then actually move in the level editor as well? And what happens if the designer saves the level in this new state? Surely that is a bad idea, since all of the locations in which the entities have been carefully placed will be changed. What a designer wants is to be able to quickly test a level at any given location, and once he is done playtesting have the level revert to its "unplayed" state. This may best be accomplished by allowing the designer to quickly enter a "test mode" and then exit it just as quickly, instantly returning him to level editor functionality. The quicker this transition the better, for the faster and easier it is, the more likely the designer will want to go back and forth to test and retest the playability of his level. If the designer has to wait a minute or longer to playtest, he will not be able to try as many different changes to the level before he gets behind schedule. For this reason, it makes sense to have a programmer focus on smoothing out and speeding up this transition as much as possible.

Any seasoned game designer will tell you that whether a game succeeds or fails is largely dependent on how well it is playtested and balanced. Even the most brilliant initial game design can be completely destroyed if the implemented game is not playtested thoroughly. I do not mean just for bugs, but for gameplay — for how the game feels to play and for how it captivates players. Playtesting is an iterative process that involves trying a type of gameplay, then modifying it, then trying it again, and repeating this loop until the game is fun. It can be very hard, then, to properly iterate through playtesting if the level editor does not facilitate the modification of the game's levels, and then easily allow the designer to try out what has been changed. The easier it is for designers to jump into the game, try something, jump back out of the game, make a content change, and then jump back in again, the more likely they are to repeat the playtesting cycle again and again until the game is as perfect as possible. If the level editor does not facilitate such testing, designers are likely to become frustrated or simply not have the time they need to sufficiently balance the game.

Editing the World

The best development tools for a game are composed of a delicate mix of off-the-shelf programs and proprietary editors. A good team will know just how much of each to use so that they are neither wasting the time of their programmers by having them develop overly sophisticated tools when a good commercial package is better suited, nor unreasonably restraining the efforts of their designers by not allowing them to refine the game's content from within a custom level editor. Though no team should be forced to develop a game without a level editor, it is equally foolhardy to force the team to do all of the game's content creation from within proprietary tools.

It is important that the level editor actually allow the designer to modify all gameplay-critical aspects of a level. This would seem to me to be an obvious prerequisite for an editor, but I have heard so many stories of teams working with 3D Studio MAX and "entity editors" that it bears mentioning. Often teams think they can get away with using an off-the-shelf tool such as MAX to create all of their world geometry, and then create a level editor only for importing the meshes from MAX and positioning the items, NPCs, and other game-world entities. Though this is workable given enough time and tenacity, it will not lead to as good levels as will a fully featured editor. As the designer is placing creatures in the map, he needs to be able to simultaneously change the geometry to fit the placement of that creature. If a designer must exit the editor and then run a 3D modeling application (which are seldom known for their speed), modify the geometry in that program, and then re-import the level into the proprietary editor before he can test out his modifications, he will certainly be discouraged from making too many "tweaks" to the geometry. As a direct result, the geometry will not look as good or play as well in the final game. Not allowing a designer to edit the level's gameplay-critical architecture in the editor itself is tantamount to tying one arm behind his back. It is my experience that designers work best with both hands free.

When I started working on *Centipede 3D*, the level editor we had was really more of a game entity manipulator than a proper level editor. The geometry for a given level was derived from a grayscale, square height-map, with those used in *Centipede 3D* all consisting of 32 x 32 pixels. Each pixel therein represented a height value on the landscape. These height-maps, which could be created in Photoshop or any other pixel-pushing tool, were a good way to create an initial version of a level's geometry. Unfortunately, in the version of the editor used at the start of the project, the height-maps could only be modified in a paint program; they could not be edited in the editor itself. This was a shame, since looking at a top-down 2D representation of a 3D level is not exactly the best way to get an idea of how the level will end up looking. As a result, the levels that were created early in the project were simple and a bit flat. It was not that the level designers were not working hard to make the levels attractive, merely that there was only so much that could be accomplished with the tools provided.

However, midway through the project, functionality was added to the tool to allow the designer to edit the height-maps while in the level editor. The height-maps could still be created in Photoshop and brought into the game, and this remained the best way to make a first pass on the level's architecture. After that first pass, however, the geometry was easily manipulated in the level editor, where the designer was able to see the level in 3D while modifying the height-map. As a result, the designers were able to

tweak the geometry until it was perfect. The change in the quality of the levels was dramatic. As always, time did not allow for us to go back and redo the earlier levels. Since the levels were made in the order they appeared in the game, anyone playing *Centipede 3D* will be able to tell at what point the level designers were given the new and improved tool. It was not that the designers could not create levels with the previous incarnation of the editor, it was just that level editing was so much more difficult that the levels failed to look as good as the designers wanted.

There is a lot to be said for being able to create fancy level geometry in a fully featured 3D package, and even level editors with sophisticated geometry editing capabilities would benefit from the ability to import externally created architecture. The key to creating quality game art assets, whether they are 2D sprites or 3D models, is being able to import from commercial packages. I do not know that anyone was ever forced to create 2D sprite artwork for a game using only an in-house tool. Yet, it seems that many unfortunate artists have only been allowed to model characters or other objects using proprietary modeling tools. I have discussed how important it is to allow the level designers to manipulate a level's architecture in the editor. But certainly forcing game designers or artists to model every game-world element in the level editor is a big mistake. Artists should be able to create game-world objects such as trees, weapons, or trash cans in their favorite modeling package and import them into the game. Simply put, there is no way a game's programming team is going to be able to code up an art editing package with all the power, robustness, and stability of a Photoshop, 3D Studio MAX, Maya, Softimage, or any of a number of other popular off-the-shelf products. Without the many features found in these packages, artists will simply be unable to create the best quality art possible. Furthermore, most artists are already familiar with one or more of these packages, and so when they come on to the project they will be that much closer to being "up to speed."

At the same time, the team will need to be able to manipulate this art using proprietary tools. Having an in-house editor with which to set up animations, nodes on a skeleton, collision data, or other information is essential to making the art function properly within the game. Teams who attempt to avoid setting up any sort of art editing software will frustrate their artists, designers, or whoever gets stuck with configuring the art and its animations to work in the game. A proprietary art manipulation tool that does exactly what the game engine needs it to do is a key ingredient in a bearable game development experience.

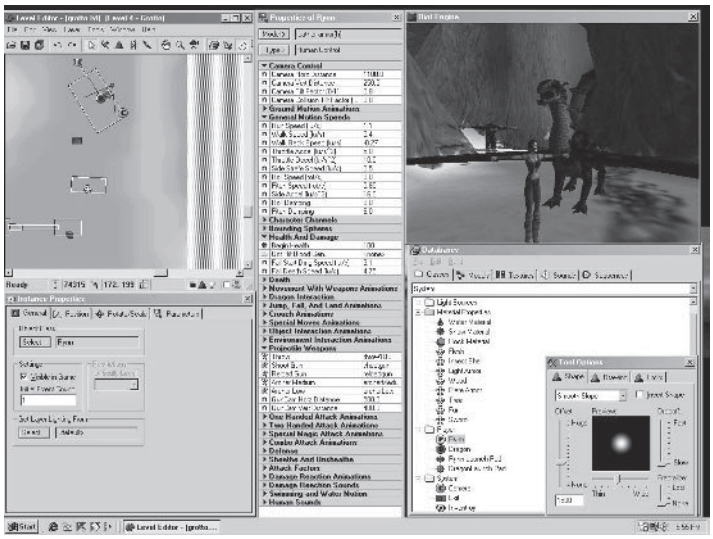
Scripting Languages and Object Behaviors

It seems to have become the norm for games to use a system where designers can set up and balance the enemy, weapon, and other game behaviors exactly as they need them, without involving a programmer. Many games now include scripting languages which, though relatively simple, allow for complex entity creation without requiring the game engine itself to be recompiled. These scripting languages provide many benefits to game development. Probably most important is that they encourage the creation of more unique behaviors in the game, whether these are reusable in-game entities such as NPCs or unique events for a specific level such as NPCs carrying on a particular conversation while the player watches in *Half-Life*.

One great benefit of a properly designed scripting system is that it is completely portable to other systems. This means that when the game is ported from the PC to the Dreamcast, for instance, all of the enemy behaviors that have been scripted and debugged on the PC will be equally functional on the Dreamcast, provided the script interpreter and its associated functions are properly ported as well. In that vein, a robust scripting language is also more stable to work with than programming in C. The scripting language gives the script's author less opportunity to thoroughly crash the game, and when a script does something illegal, the game can spit out a properly informative message instead of just locking up. Often scripting languages are not as complex as actual C programming, and thereby allow designers with some programming savvy to take on the creation of unique world behaviors, thus freeing up harder-to-find programmers for more complex tasks. In most systems, scripts can also be loaded on demand, which means only the scripts that a particular section of the game uses will need to be resident in memory, thus freeing up more code overhead. An added bonus of a game having a scripting language is that it allows for complex user modification of that game. A well-designed and appropriately powerful scripting system will empower motivated players to make their own "mods" for the game for distribution to friends.

Scripting languages have their downside as well. First is the time involved in implementing a scripting system. If the language is to be actually useful to the game as described above, it will need to be very stable and provide its user with a lot of power, which is certainly non-trivial to implement. Debugging a problematic script can also be quite a lot of trouble, since no game developer is going to have the time to implement a symbolic debugger as nice as the one that comes with Visual Studio or CodeWarrior. Most of the time, the scripts are compiled at run time, and as a result can be significantly slower than C/C++ code. Again, no matter what the developer does in terms of optimizing performance of the scripts, he will not be able to match the compiling power of the C++ compilers made by Watcom, Microsoft, or Metrowerks. And finally, though one of the big advantages to scripting languages is supposed to be that they can be used by non-programmers, it often turns out that, if the scripting language is actually powerful enough to create AI for an NPC, it is going to be so complex that it requires a programmer to use it effectively. And if a programmer's time is being tied up in the creation of scripts, why stop him from just doing his coding work in C?

Of course, one of the main advantages of scripts is that they greatly simplify the balancing of gameplay. Instead of a programmer tweaking a number in the code and then waiting for the game to recompile, a designer can adjust a value in a script and just run the game. But what if one wants to achieve this benefit of scripts without having to implement a scripting system? What if, instead, the designers were able to adjust behavior parameters in the level editor itself? This is the approach taken by Surreal Software's Riot Engine. In Surreal's Level Editor, designers are given access to all the settings or "behavior variables" for a given AI, weapon, or other game-world entity. The behaviors themselves are coded in C++, with the programmers leaving "hooks" to all the crucial settings that determine how the game-world object will behave, such as how fast it moves, what its detect radius is, what it does when it is destroyed, and so forth. This provides much of the game-balancing benefit of scripting languages by empowering the designers to endlessly tweak the game while still taking advantage of the speed of a powerful C++ compiler and debugger. This functionality makes the



Surreal Software's Riot Engine Level Editor allows the designer to tweak all sorts of settings for different game-world entities.

level editor not just a tool for modifying the game's levels, but turns it into more of a gameplay editor, where the designer is able to change much of the game's content on the fly. Of course, you don't always need a fancy editor to pull off easily tweakable variables; many games have allowed designers to modify gameplay values via text files that can be edited in any text editor and that are then read in by the game at run time. Of course maintaining a number of text files can be less user friendly than an editor specifically designed for modifying this type of data. But at the same time, if you don't have enough time to invest in your tools, editing text files may be preferable to data that is only modifiable in an especially clunky or broken editor.

"Scripted events" in levels are another thing that game scripting languages do well. Each level in the game can have a unique script that sets up and triggers various unique behaviors on that level. Having complex, unique behaviors has recently become a much bigger concern of game developers. An early and influential example was Valve's excellent use of scripted events integrated into the more dynamic gameplay proper in *Half-Life*. Of course, there is a key difference between "scripted events" and the "scripting language" one uses to set them up. *Half-Life* had great scripted events, but apparently a difficult-to-use method for setting them up. Creating a solid and simple scripting system is the best way to ensure that the designers will make use of it. Instead of involving a separately compiled, text-based scripting language, level editors can include the ability to empower designers to easily set up complex game events. Surreal Software's STOMP editor, used in the *Drakan* games as well as *The Suffering*, though not the most bug-free piece of software ever written, allows designers and animators to set up complex scripted sequences relatively easily in the editor and view them in real-time 3D as they are built. The sequences, which can be used both as cut-scenes and in-game scripted events, can be easily edited in real-time using an interface reminiscent of Adobe Premiere. *StarCraft*'s Campaign Editor is an especially good example of a more gameplay-oriented scripting system with a well-conceived and user-friendly interface. Its "Triggers" editor allows designers to use a very familiar point-and-click interface to set up complex scripted events. Pop-up menus provide lists

of all the commands available, and then further pop-ups show the designer all of the different parameters that can be passed to those commands. The whole system is easily comprehensible to someone looking at it for the first time, with commands written in plain English. Thus, the Campaign Editor allows unique events to occur in *StarCraft* levels without involving the overhead of a full-blown scripting language.

The *StarCraft* Campaign Editor allows for easy scripting using triggers and a simple point-and-click interface.

Us Versus Them

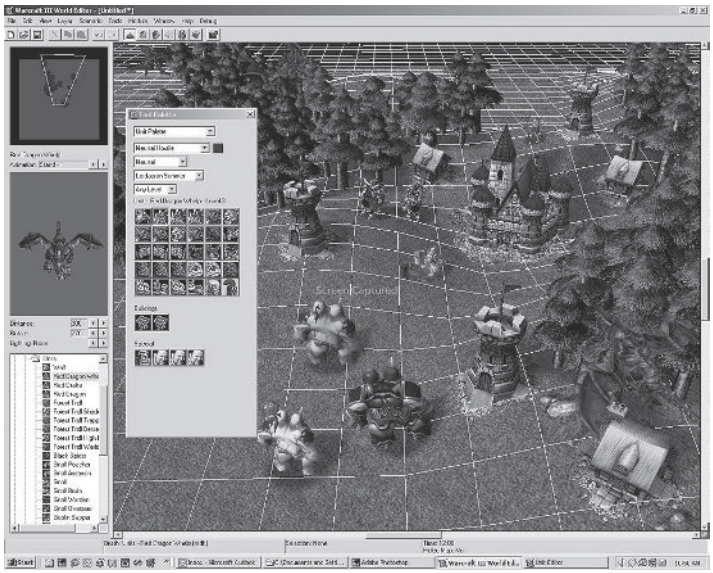
Unfortunate as it may be, the development of the tools for a project often comes down to a battle between the programmers and the designers. Game programmers are often loath to work on tools for a variety of reasons. First, many of the programmers who wanted to get into gaming did so because they did not want to program databases, spreadsheets, or 3D modeling packages. They wanted to make games, and tools often seem too much like "real programming." There's also a perception that getting one's code in the game is more important than getting it in the tools. If the title is a big hit, the game will be played by millions of people. The tools for a given project, however, will be used by perhaps ten or twenty people. When a programmer's friends ask him what he worked on while he was at that wacky game company, most programmers do not want to have to answer, "I worked on the tools." There is a certain lack of glamour there.

Further complicating matters is the perception that a programmer's time is more valuable than a designer's. So if a designer has to spend five times as long making a level because a programmer does not have the time to make the level editor better, well, that's OK. The level still gets made, right?

As I have stated previously, game developers should not be asking themselves the question, "Do the tools allow for the game's content to be created?" Instead, they should ask, "Do the tools allow for the game's content to be made well?" If a designer is constantly fighting with the level creation tools, he is not going to be able to invest time into truly refining the level. In fact, he may be so irritated at perceived programmer

laziness that he throws his hands up in disgust and does not work on the level as much as he might otherwise. A good level designer will be inspired by a good tool set to do the best work he can, because he can see direct results. The example I used before about the level design tool and the resultant quality of the levels in *Centipede 3D* is a good lesson for game developers. With the creation of a superior level editing tool, level quality will improve dramatically.

A tools programmer should be able to take pride in having worked on a really good tool that facilitates the designer's work. The programmer responsible for a well-conceived and well-implemented level editor that greatly facilitates the creation of beautiful levels should feel that he played a vital role in the creation of those levels. For without the features of the level editor, the designer would not have been able to create the landscapes or structures he did. The designer must always make it a point to remember the programmer who made possible the creation of such levels and be suitably appreciative of his efforts.



Blizzard's *WarCraft III* World Editor automatically sets up transitions between different types of landscape textures, thereby saving designers a lot of work.

At one point I added a texturing feature to the Riot Engine Level Editor. At the time, the Riot Engine employed tiling textures for its landscape, with transition textures available for when a grass texture meets a rock texture, for example. I added the functionality that allowed the editor to automatically place the proper transitions between two different texture types. Interestingly, this was a feature included in the level editor for my first published game of five years prior, *Odyssey: The Legend of Nemesis*. Indeed, this auto-transitioning functionality is found in many 2D terrain level editors, such as Blizzard's *StarCraft* Campaign Editor or *WarCraft III* World Editor. Before I added the feature, the level designers at Surreal had to pick by hand the transition texture that was needed. Certainly the auto-transitioning feature was not absolutely necessary for the creation of levels. All of the levels for the game *Drakan* had been made without the use of the auto-transitioning tool, and certainly they were very beautiful levels with transitions in all the right places. The key difference is that

those transitions took a lot of designer time to set up. Once I added the auto-transitioning tool the designers were delighted, since now a large and tedious part of their jobs had been all but eliminated. One even said, "Richard could take off the next month and we could keep paying him." He was appreciative of the feature I had added and was thoughtful enough to communicate his thanks to me. With praise like that, tools programmers are much more likely to keep adding nifty features to the editor.

The Best of Intentions

However, one must be careful. Sometimes when programmers are tasked with adding functionality to the editor, they may end up adding features that no one really needs. It is difficult for a programmer who, most of the time, does not make the game's levels and therefore does not spend a lot of time working with the level editor, to properly understand what that editor is lacking. Indeed, what a programmer may see as a cool feature can turn out to be functionality no designer will ever want to use. When a programmer goes to a lot of trouble to implement a feature for the editor and then the designers fail to use it, resentment tends to grow in the programmer. Then when a designer comes to the programmer requesting a more practical and necessary feature be added to the editor, the programmer is likely to ignore him, thinking, "He never used the vertex-warping tool that I worked so hard on, so why should I work on this model-aligner for him? Forget it."

Anyone who has worked in the industry knows that, in a lot of ways, designers and programmers think differently. For this reason, it is very important for the designers and programmers to be in constant communication about what features the editor needs and how they can best be implemented. When developing an in-house tool set, the programmer has the tremendous advantage of having his user base down the hall. He does not have to guess what they want from the program; instead he can go ask them. Similarly, the designers have the advantage of being able to go to the editor's developer and make suggestions on how the tool should function. With a good flow of information between the parties involved, the tools cannot help but improve.

One possible technique for facilitating the creation of a good tool is to assign one programmer to be primarily responsible for the maintenance and improvement of the level editor, instead of passing editor tasks off to "whoever has time." This one programmer can then become quite familiar with the workings of the tool and can take pride in what a good application it is. If one programmer does most of the editor work, the designers will know which programmer they can turn to with their suggestions for improvements to the tool. That programmer will get a better sense of what the designers like and do not like. Of course, if the programmer assigned to working on the tool really wishes he was working on lighting effects or AI, the tool is going to suffer as a result. Finding a programmer who really wants to work on the tool is important if this strategy is to succeed.

Another useful tactic is to actually have a programmer make a complete, simple level using the tool. That way, the programmer can easily spot areas for improvement in the editor, and can finally understand what the designers have been complaining about for so many weeks. If the level is of sufficient quality and fits the needs of the project, you may even want to consider shipping this programmer-created level with the game. But even if you don't, the understanding the programmer gains through using the

editor as it is really used will be invaluable. Without actually having to sit down and fully use the application they are creating, the programmer is likely to conclude that the designers are overemphasizing the problems with the editor (known in industry parlance as "whining"). But by actually having to use the tool he is working on, a programmer is likely to easily identify editor shortcomings that can be easily fixed through a few hours of coding. Designers frequently fail to understand the complexity of different programming tasks, and as a result make requests for nearly impossible features in the level editor, while thinking easily remedied problems are unfixable. Perhaps the best solution of all is to have a designer who is also a programmer, and thereby spends a lot of time working with the editor. This designer/programmer is directly motivated toward improving the tool he must work with every day, and is likely to do whatever he can to make it the best tool possible. Ten years ago I am sure this was not that uncommon, but for full-scale projects in development today it is fairly rare. Programming a level editor and designing levels have each become tasks that fully consume an individual developer's time, and unfortunately the days of the designer/programmer seem to be mostly a thing of the past.

A Game Editor for All Seasons

A level editor does not actually need to be bug free. Bug-free software is the stuff one buys in stores, if one is lucky. Really great in-house tools can have plenty of bugs in them. What is important is that these tools be buggy in predictable ways. The bugs should occur in patterns that the designers can learn how to predict and teach themselves to avoid. Once a designer becomes adept at the tools he will know what not to do and will be able to easily work around the trouble spots. Proprietary level editor tools are one place in software development where the old joke "Doctor, it hurts when I do this!" "Then don't do that!" really rings true.

Of course, if the tools used on a project are good enough, marketing may catch on and can come up with the bright idea, "Hey, we can release the tools with the game!" Indeed, shipping a game with its level editor and having users create add-on levels for your title can help to keep interest alive in a game long after it has been released. Hard-core fans will love to make "mods" for the game to circulate among their friends or the general public. For the tools to be released, they really will need to be relatively bug free, or at least much more stable than when they were only being used in-house. The possibility of releasing the level editor to the fans should function as an incentive to encourage the programming team to create the best tools possible. Of course, some publishers still fail to see the logic of having the fan community build add-ons and refuse to release the tools used for the game's creation. The argument they often give is that if users can build more levels themselves, who will want to buy the sequel? Of course, id Software, the company that popularized releasing level editors to the public, continues to do quite well financially, suggesting that protectionist thinking in terms of level editors is somewhat foolish.

In the end, it all comes down to what should be recognized as an axiom in the gaming industry: a game can only be as good as the tools used in its creation. A well-conceived level design tool can make the difference between a great game and a mediocre one. One can think of the ideal level editor as a place where the designer has

total control of the game-world: of its architecture (where players can go), of its aesthetic appearance (lighting, texturing, and sounds), and of its gameplay (NPC, item, and other entity placement, movement, and behavior). Of course, the best level editor in the world is not going to make up for a subpar engine, a fundamentally flawed game design, or a demoralized development team. But those are topics for another chapter.