Better Scripts, Better

Walker White, Christoph Koch, Johannes Gehrke, and Alan Demers, Cornell University

Smarter, more powerful scripting languages will improve game performance while making gameplay development more efficient.

The video game industry earned $8.85 billion in revenue in 2007, almost as much as movies made at the box office. Much of this revenue was generated by blockbuster titles created by large groups of people. Though large development teams are not unheard of in the software industry, game studios tend to have unique collections of developers. Software engineers make up a relatively small portion of the game development team, while the majority of the team consists of content creators such as artists, musicians, and designers.

CONTENT CREATION IN GAMES

Since content creation is such a major part of game development, game studios spend many resources developing tools to integrate content into their software. For example, entry-level programmers typically make tools to allow artists to manage assets or to allow designers to place challenges and rewards in the game. These tools export information in a format usable by the software engineers, either as auto-generated code or as standardized data files.

This content-creation "pipeline" is not very well understood, and each studio has its own philosophy and set of tools. Many tools are taken from, or developed in coordination with, the film industry. Unlike film, however, games need to be interactive. Player


Games

Better Scripts, Better Games

actions require visual feedback; game characters should react to player choices. Adding interactive features typically requires some form of programming. These features are also a form of artistic content, and game studios would prefer they be created by designers—developers who understand how the player will interact with the game, and what makes it fun—rather than software engineers.

The idea of game software as artistic content has led many game studios to split their software developers into two groups. *Software engineers* work on technical aspects of the game that will be reused over multiple titles. They work on core technology such as animation, networking, or motion planning, and they build the tools that make up the content-creation pipeline. *Gameplay programmers*, on the other hand, create the behavior specific to a single game. Part designer, part programmer, they implement and tune the interactive features that challenge and reward the player.

The gameplay programmer should produce fun, not complex, algorithms. Game studios design their programming workflow to relieve gameplay programmers of any technical burdens that keep them from producing fun. Often this involves an iterative process between the gameplay programmers and the engineers. The gameplay programmers develop feature prototypes to play-test before adding them to the game. The software engineers then use these feature prototypes to design support libraries, which are used to build another round of prototypes. This is an effective workflow, but game companies are always looking for ways to speed up or even automate this process.

In addition to supporting the interaction between gameplay programmers and software engineers, the studios are always looking for ways to integrate the designers into the programming process. Designers often have very little programming experience, but they have the best intuitions for how the game should play. Thus, studios want tools that allow designers, if not actually to program behavior, at least to fine-tune the parameters behind it.

THE ROLE OF SCRIPTING LANGUAGES

Many game studios rely on scripting languages to enable gameplay programmers and designers to program parts of their games. These languages allow developers to easily specify how an object or character is supposed to behave, without having to worry about how to integrate this behavior into the game itself. Scripting languages are particularly important for massively multiplayer games where any piece of code must interact with multiple subsystems, from the application layer to the networking layer to the database.

User-created content is another reason for games to support scripting. Open-ended virtual worlds such as Second Life have made player scripting a common topic of conversation. Even before that, games had a long tradition of player-developed *mods*. Given tools—either official or third party—to modify the data files that came with the game, players have been able to create completely new experiences. Generally, modding has been seen as a way to extend the lifespan of older games. In some cases, however, it can create completely new games: the commercially successful Counter-Strike was a player modification of the game Half-Life and relied heavily on scripting features present in its parent game.

Scripting languages allow players to modify game behavior without access to the code base. Just as important, they provide a sandbox that—unlike a traditional programming language—limits the types of behavior the player can introduce. If the game has a multiplayer component, the game developers do not want players creating scripts to give themselves an undue advantage. Overly powerful scripting languages have facilitated many of the bots—automated players performing repetitive tasks—that currently populate massively multiplayer games. Sandboxing can even be useful in-house. By limiting the types of behaviors that their designers can create, the studios can reduce the number of bugs that they can introduce—bugs that cost valuable time to find and eliminate.

THE NEED FOR GAME-SPECIFIC SCRIPTING LANGUAGES

The foremost criterion for a scripting language is that it should make gameplay development fast and efficient. Often game objects—rocks, plants, or even intelligent characters—share many common attributes. Game scripting languages are often part of IDEs (such as the one shown in figure 1) that provide forms for quickly modifying these attributes. The scripting languages themselves, however, are fairly conventional. Many companies use traditional scripting languages such as Lua or Python for scripting. Even companies that design their own lan-

guages usually stick with traditional format and control structures. Little effort has been spent tailoring these scripting languages for games.

One of the major problems with traditional scripting languages is that the programmer must be explicitly aware of low-level processing issues that have little to do with gameplay. Performance is a classic example of such a low-level issue. Animation frame rate is so important to developers that they optimize by counting the number of multiplies or adds in their code. This type of analysis is beyond the skill of most designers, however. Furthermore, existing languages provide almost no tools to help designers improve script performance.

Designers must also take performance into account when creating content. If the game runs too slowly, they may be forced to reduce the number of objects in the game, which in turn can significantly alter the playing experience. This is what occurred when The Sims was ported to consoles. In this game, a player indirectly controls a character (Sim) by purchasing furniture or other possessions for it. Each piece of furniture is scripted to advertise its capabilities to the Sim periodically. The Sim then compares these capabilities with its needs in order to determine its next action. Furniture does not exist in isolation, however; a couch in front of a television is much more versatile than one alone in a room. Therefore, pieces of furniture also periodically poll the other furniture in the room to update their capabilities. As each piece of furniture may communicate with other pieces of furniture, the cost of processing a room can grow quadratically with the number of objects in the room. When the title was ported to consoles, the performance issue became so pronounced that the designers had to introduce a "feng shui meter" to prevent players from filling rooms with too many possessions.

Game developers have many techniques available to them for improving performance. Spatial indexes are one popular way of handling interactions between game objects at less than quadratic cost. Parallel execution is another possibility; many games are embarrassingly parallel, and developers leverage this fact for multicore CPUs and distributed multiplayer environments. These techniques are beyond the skill of the typical game designer, however, and are left to the software engineers.

Another low-level issue with scripting languages is the lack of transaction support for massively multiplayer games. Individual scripts are often executed concurrently, particularly in massively multiplayer games, so designers need some form of transaction to avoid inconsistent updates to the game state. Indeed, script-level concurrency violations are one of the major causes of bugs in multiplayer environments.

To make scripting easier for designers, we have to provide them with simple tools for addressing these low-level issues. None of these problems is really new; many programming languages have been developed over the years to address them, but most of these languages make programming more difficult, not easier. Fortunately, designers do not need an arbitrary scripting language; they just need a language that helps them write games.

FIGURE 1 **Neverwinter Nights 2**

The Neverwinter Nights 2 toolset is an extensive IDE that allows users to create new content for the game.

FROM PATTERNS TO LANGUAGE FEATURES

Despite these problems, games are being developed. Game developers have come up with many ideas that, if not complete solutions, do ameliorate the problems. These ideas typically come in the form

Better Scripts, Better Games

of programming patterns that have proven over time to be successful. Though developers use these programming patterns in creating game behavior, the scripting languages usually do not support them explicitly. One of the reasons object-oriented programming languages have been so successful is that object-oriented programming patterns existed long before the languages that supported them. Similarly, by examining existing programming practices in game development, we can design scripting languages that require very little retraining of developers. The challenge in developing a scripting language is identifying those patterns and creating language features to support them most effectively.

THE STATE-EFFECT PATTERN

One popular pattern in game development is the *state-effect pattern*. Every game consists of a long-running simulation loop. The responsiveness of the game to player input depends entirely on the speed at which the simulation loop can be processed. In the state-effect pattern, each iteration of the simulation loop consists of two phases: *effect* and *update*. In the effect phase, each game object selects an action and determines individually the effects of this action. In the update phase, all the effects are combined and update the current state of the game to create the new state for the next iteration of the simulation loop.

Because of these two phases, we can separate the attributes of game objects into *states* and *effects*. State attributes represent the snapshot of the world after the last iteration of the simulation loop. They are altered only in the update phase and are read-only in the effect phase. Effect attributes, on the other hand, contain the new actions of the game objects, and the state of the game is updated with effects during the update phase. Because interactions between game objects are logically simultaneous, effect values are never read until the update phase. Hence, effect values are, in some sense, write-only during the effect phase.

Game physics provides many examples of this pattern. At the beginning of the simulation loop, each game object has a current position and velocity recorded as state attributes. To compute the new velocity, each object computes the vector sum of all of the forces acting upon it, such as collisions, gravity, or friction. In other words, the force attribute may be written to multiple times during the simulation loop, but it is never read until all of the force values have been summed together at the end of the loop. The example in figure 2 illustrates the use of the state-effect pattern to simulate objects moving about in a potential field. The variable force is an effect in this calculation. During the effect phase we only increment its value and never read it to determine control flow. Whereas most implementations would read the old value of force to perform this increment, this is not necessary; we could also gather all of

FIGURE 2 **Example of the State-Effect Pattern**

```
// Outer simulation loop
for each timestep {

  // Compute effects for all
  for each particle o {
    o.effectPhase();
  }


  // Update state for all
  for each particle o {
    o.updatePhase() ;
  }

}
```

```
// State variables
vector position, velocity;
scalar q, damping, mass;

// Effect variables
vector force;

// Read state, write effects
effectPhase() {
  for each particle p {
    r = position-this.p.position;
    s = ((this.q*p.q)/(r.magnitude))^3;
    force += s*r;
  }
}

// Read and write state, read effects
updatePhase() {
  velocity = damping*velocity+force/mass;
}
```

these force values in a list and add them together at the end of the effect phase.

Most of the time, game developers use the state-effect pattern to manually design high-performance algorithms for very specific cases. That is because it has several properties that allow them to significantly enhance the performance of the simulation loop. The effect phase can be parallelized since the effect assignments do not influence each other. The update phase can also be parallelized since it consists only of the aggregation of effects and updates to state variables. This does not need to be done by hand; if the scripting language knew which attributes were state attributes and which were effect attributes, it could perform much of this parallelization automatically, even in scripts written by inexperienced designers. This is similar to what Google achieves with its Sawzall language and the MapReduce pattern; special aggregate variables perform much the same function as effect attributes, and the language allows programmers at Google to process data without any knowledge of how the program is being parallelized.[2]

Automatic parallelization is an example of an alternative execution model; the game runs the script using a control flow that is different from the one specified by the programmer. Since the simulation loop logically processes all of the game objects simultaneously, we can process them in any order, provided that we always produce the same outcome. Thus, alternative execution models are among the easiest ways of optimizing game scripts. Another unusual execution model is used by the SGL scripting language, which is being developed at Cornell University.[1] This language is based on the observation that game scripts written in the state-effect pattern can often be optimized and processed with database techniques. The script compiler gathers all of the scripts together and converts them into a single in-memory query plan. Instead of using explicit threads, it constructs a data pipeline that allows the code to be parallelized in natural ways. Many of these data pipelines are similar to the ones that game programmers create when they program on the graphics processing unit, except that these are generated automatically.

THE RESTRICTED ITERATION PATTERN

Iteration is another common source of problems in game development. Allowing arbitrary iteration can quickly lead to significant performance degradation of the simulation loop. Iteration can be even more dangerous in the hands of inexperienced designers. During the development of City of Heroes, Cryptic Studios discovered that many of the scripts had interdependencies that produced hard-to-find infinite loops. To prevent this, the developers removed unbounded iteration from the scripting language.

Although this was a fairly drastic solution, most games do not need arbitrary iteration in their scripts. The scripts just need to perform a computation over a *finite set of objects*; such scripts follow the *restricted iteration pattern*, which obviously guarantees termination on all loops. In addition, it may enable code analysis and compile-time code transformations that improve performance. For example, SGL can take nested loops that produce quadratic behavior and generate an index structure from them[1]; it then replaces the nested loops with a single loop that performs lookups into that index.

Examples of the restricted iteration pattern appear throughout the scripts in Warcraft III, a realtime strategy game that has to process armies of individual units. The NudgeObjectsInRect script in figure 3 appears in the

FIGURE 3

Example of the Restricted Iteration Pattern

```
//===========================================================================
// Nudge items and units within a given rect, so that they can find
// locations where they can peacefully coexist
function NudgeObjectsInRect takes rect nudgeArea returns nothing
  local group g

  set g = CreateGroup()
  call GroupEnumUnitsInRect(g, nudgeArea, null)
  call ForGroup(g, function NudgeUnitsInRectEnum)
  call DestroyGroup(g)

  call EnumItemsInRect(nudgeArea, null, function NudgeItemsInRectEnum)
endfunction
```

Better Scripts, Better Games

Blizzard.j file. This function takes a rectangle and loops through all of the military units that appear in that rectangle; in that loop, it uses the function NudgeUnitsInRectEnum to push units apart so that there is a minimum distance between pairs of units.

All the operations in this script are external functions provided by the software engineers. The scripting language is not aware that these functions implement the equivalent of a for-each loop (a loop over a fixed set of objects); otherwise, the compiler would be able to perform loop optimizations on it. Given the number of times this pattern appears in the Warcraft III scripts, this could result in significant performance improvements.

CONCURRENCY PATTERNS

Iteration is not the only case in which developers could benefit from alternative control structures. Many games execute scripts in parallel, which requires scriptwriters to be cognizant of concurrency issues. As an example, consider inventory management in online games, a notoriously problematic scenario, with consistency violations resulting in lost or duplicated objects. Consider the following simple script written to put an item in a container such as a sack or a backpack:

```
// Test a container, and insert an object if okay
success = TestPutItem(me, container, item)
if (!success):
  Bail()
else:
  PutItemInContainer(item, container)
```

This script tests if a container has the capacity to hold an item, then adds the item if there is space. Nothing in the script says that this action must be executed atomically, so in a distributed or concurrent setting, the container could fill up between the time it is tested and the time the item is added to the container. Obviously, this could be eliminated by the addition of locks or synchronization primitives to the scripting language. Locks can be expensive and error-prone, however, so game developers like to avoid them if at all possible. They are particularly dangerous in the hands of designers.

Additionally, lock-based synchronization is incompatible with the state-effect pattern. In the state-effect pattern, the state of the container consists of the contents at the end of the last iteration of the simulation loop, while an effect attribute is used to gather the items being added to the container. Effect variables cannot be read, even with locks, so the script cannot test for conflicting items being added simultaneously.

Instead of trying to solve this problem with traditional concurrency approaches, it is best to step back and understand what the programmer is trying to do in this pattern. The programmer wants to update an object, but under some conditions this update may result in an inconsistent state. The function TestPutItem defines which states are consistent. If the language knew this was the consistency function for PutItemInContainer, it could delay the check to ensure consistency without a lock. The language could first gather all of the items to be added to the container and then use the consistency check to place as many as the container can hold. In some cases, the language could even place multiple objects with a single consistency check.

Of course, this approach does not solve arbitrary problems with parallel execution, but game companies use languages with almost no concurrency support, and they rely on coding conventions to limit consistency errors. Adding features that provide concurrency guarantees for the more common design patterns in games would allow the game developers to trust their scriptwriters with a wider variety of scripts, increasing their artistic freedom.

GAME-AWARE RUNTIMES

Language features provide the runtime with clues on how best to execute the code, but some games have properties outside of the scripting language that the runtime can also leverage. For example, the right optimization strategy for a set of scripts depends on the current state of the game. If the game is controlling a large army marching toward an enemy, then the game should optimize movement of soldiers; on the other hand, if the army is guarding against an attack, the game should optimize individual perception. Games often have a small number of these high-level states, and changes between them happen relatively slowly. If the runtime can recognize which state the game is in, it can switch to an optimized execution plan and improve performance.

To some degree, game developers already take advantage of this fact in their performance tuning. Currently they log runs of the game during play-testing, and later

data-mine these logs for recurring patterns. If these patterns are easy to detect, developers can take advantage of them. This type of optimization, however, is very difficult for designers or for players developing user-created content. Ideally, a game-aware runtime would have some knowledge of common patterns and be able to adjust for them automatically.

Performance is not the only reason for the runtime to monitor how the game changes over time; it is also useful for debugging. Debugging a game is not as simple as stepping through a single script. Each object is scripted individually, and these scripts can interact with one another in subtle ways. An incorrect data value in one script may be the result of an error in a completely different script. In addition, many errors are the result of user input that is not always easy to reproduce. A script designer needs some way of visualizing which scripts modify which objects and how these objects change over time. This is an application of data provenance, which is an active area of development in the field of scientific computation. Like designers, the scientists targeted by data provenance tools often have little programming experience; instead, the provenance techniques model the way they naturally think about the data. As yet, no game scripting language supports data provenance.

Data provenance is even more important if the script runtime has an unusual execution model. In the previous script to place items in a container, efficient execution involved reordering portions of the script. Instead of having the programmer debug the scripts in an execution model that is different from the one in which the bug appeared, it is best to give him or her a higher-level visualization of how that bug might have occurred.

Game-aware runtimes are more difficult to implement than language features. Language features can often be implemented piecemeal; as programming patterns are identified, new language features can be added without adversely affecting the old. Runtimes, once architected, can be very interdependent and difficult to change. For example, any changes to the order in which operations are processed will affect the debugger. Thus, while languages can have an attitude of "see what works," runtimes need to be well understood from the beginning.

CONCLUSION

Scripting languages are an integral part of both game development and modding, and their design has huge impact on both correctness and performance of the resulting game. Game developers earn money from the titles that they publish, not the engineering problems that they solve. Therefore, anything that reduces technical challenges for the developers and allows them to create more content is a welcome innovation. Advances in design patterns and scripting languages will influence the way games are programmed for years to come. Q

REFERENCES

1. White, W., Sowell, B., Gehrke, J., Demers, A. 2008. Declarative processing for computer games. In *Proceedings of the 2008 ACM SIGGRAPH Sandbox Symposium*. http://doi.acm.org/10.1145/1401843.1401847.

2. Dean, J., Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 51(1): 107-113. http://doi.acm.org/10.1145/1327452.1327492.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

WALKER WHITE is the director of the Game Design Initiative, an interdisciplinary undergraduate program training students in the design and development of computer games, at Cornell University. He has been actively working with game companies to identify ways in which database technology can be better integrated with computer games. His current research interests include data-driven design and data management for computer games.

CHRISTOPH KOCH is an associate professor of computer science at Cornell University. He is interested in both the theoretical and the systems-oriented aspects of data management, and currently works on managing uncertain data, community data-management systems, data-driven games, and Web information extraction and management.

JOHANNES GEHRKE is an associate professor in the department of computer science at Cornell University. His research interests are in the areas of data mining, data privacy, scalability, and computer games and virtual worlds. He coauthored the undergraduate textbook *Database Management Systems* (McGraw Hill, 2002), currently in its third edition.

AL DEMERS is a principal research scientist in the department of computer science at Cornell University. He has done seminal work on data replication and ubiquitous computing, and he holds several patents in these areas. His current research interests are in data management for computer games and virtual worlds.