Game Programming Gems

Edited by Mark A. DeLoura

CHARLES RIVER MEDIA, INC.

Rockland, Massachusetts

Publisher: Jenifer Niles
Production: Publishers' Design and Production Services, Inc.
Cover Design: The Printed Image
Cover Image: Mark Peasley
Printer: InterCity Press, Rockland, MA.

CHARLES RIVER MEDIA, INC.
P.O. Box 417
403 VFW Drive
Rockland, Massachusetts 02370
781-871-4184
781-871-4376 (FAX)
chrivmedia@aol.com

This book is printed on acid-free paper.

Game Programming Gems
edited by Mark DeLoura
ISBN: 1-58450-049-2
Printed in the United States

Printed in the United States of America
00 01 5 4 3

1.0

The Magic of Data-Driven Design

Steve Rabin

Games are made up of two things: *logic* and *data*. This is a powerful distinction. Separate, they are useless, but together, they make your game come alive. The logic defines the core rules and algorithms of the game engine, while the data provides the details of content and behavior. The magic happens when logic and data are decoupled from each other and allowed to blossom independently.

Obviously, game data should be loaded from files, not embedded inside the code base. The genius comes from knowing how far to run with this concept. This article gives seven ideas that will revolutionize the way you make your games, or at least confirm your suspicions.

Idea #1: The Basics

Create a system that can parse text files on demand (not just at startup). This is essential to putting data-driven design to work. Every game needs a clean way to read in general-purpose data. The game should eventually be read in binary files, but the ability to read in text files during development is crucial. Text files are dead simple for editing and making changes. Without altering a single line of code, your whole team, including testers and game designers, can try out new things and experiment with different variations. Thus, something that is trivial to implement can quickly become an indispensable tool.

Idea #2: The Bare Minimum

Don't hard-code constants. Put constants in text files so that they can be easily changed without recompiling code. For example, basic functionality such as camera behavior should be exposed completely. If this is done properly, the game designer, the producer, and the kid down the street will all be able to alter the behavior of the camera with nothing more than Notepad. Game designers and producers are often at the mercy of programmers. By exposing algorithm constants, non-programmers can tune and play with the values to get the exact behavior they desire—without bothering a single programmer.

Idea #3: Hard-Code Nothing

Assume that anything can change, and probably will. If the game calls for a split screen, don't hard-code it! Write your game to support any number of viewports, each with its own camera logic. It isn't even any more work if it's designed right. Through the magic of text files, you could define whether the game is single-screen, split-screen, or quad-screen. The files would also define all the starting camera values, such as position, direction, field of view, and tilt. The best part is that your game designers have direct access to all elements within the text files.

When core design decisions are flexible, the game is allowed to evolve to its full potential. In fact, the process of abstracting a game to its core helps tremendously in the design. Instead of designing to a single purpose, you can design each component to its general functionality. In effect, designing flexibly forces you to recognize what you should really be building instead of the limited behavior outlined in the design document.

For example, if the game calls for only four types of weapons, you could program a perfectly good system that encompasses all of them. However, if you abstract away the functionality of each weapon, using data to define its behavior, you'll allow for the possibility of countless weapons that have very distinct personalities. All it takes is a few changes in a text file in order to experiment with new ideas and game-play dynamics. This mindset allows the game to evolve and ultimately become a much better game.

Did You Believe Me When I Said "Nothing"?

The truth is that games need to be tuned, and great games evolve dramatically from the original vision. Your game should be able to deal with changing rules, characters, races, weapons, levels, control schemes, and objects. Without this flexibility, change is costly, and every change involves a programmer—which is simply a waste of resources. If change is difficult, it promotes far fewer improvements to the original design. The game will simply not live up to its full potential.

Idea #4: Script Your Control Flow

A *script* is simply a way to define behavior outside of the code. Scripts are great for defining sequential steps that need to occur in a game or game events that need to be triggered. For example, an in-game cut-scene should be scripted. Simple cause-and-effect logic should also be scripted, such as the completion conditions of a quest or environment triggers. These are all great examples of the data-driven philosophy at work.

When designing a scripting language, branching instructions require some thought. There are two ways to branch. The first is to keep variables inside the scripting language and compare them using mathematical operators such as equals (=) or less than (<). The second is to directly call evaluation functions that compare variables that exist solely inside the code, such as `IsLifeBelowPercentage(50)`. You could always use a mix of these techniques, but keeping your scripts simple will pay off. A game designer will have a much easier time dealing with evaluation functions than declaring variables, updating them, and then comparing them. It also will be easier to debug.

Unfortunately, scripts require a scripting language. This means that you need to create an entirely new syntax for defining your behavior. A scripting language also involves creating a script parser and possibly a compiler to convert the script to a binary file for faster execution. The other choice is to use an existing language such as Java, but that requires a large amount of peripheral support as well. In order not to sink too much time into this, it pays off to design a simple system. Overall, the tendency is to make the scripting language too powerful. The next idea explains some pitfalls of a complicated scripting language.

Idea #5: When Good Scripts Go Bad

Using scripts to data-drive behavior is a natural consequence of the data-driven methodology. However, you need to practice good common sense. The key is remembering the core philosophy: Separate logic and data. Complicated logic goes in the code; data stays outside.

The problem arises when the desire to data-drive the game goes too far. At some point, you'll be tempted to put complicated logic inside scripts. When a script starts holding state information and needs to branch, it becomes a *finite state machine*. When the number of states increases, the innocent scriptwriter (some poor game designer) has the job of programming. If the scripting becomes sufficiently complex, the job reverts to the programmer who must program in a fictional language that's severely limiting. Scripts are supposed to make people's jobs easier, not more difficult.

Why is it so important to keep complicated logic inside the code? It's simply a matter of functionality and debugging. Since scripts are not directly in the code, they need to duplicate many of the concepts that exist in programming languages. The natural tendency is to expose more and more functionality until it rivals a real language. The more complicated scripts become, the more debugging information is needed to figure out why the scripts are failing. This additional information results in more and more effort devoted to monitoring every aspect of the script as it runs.

As you probably guessed, non-trivial logic in scripts can get very involved. Months of work can be wasted writing script parsers, compilers, and debuggers. It's as though programmers didn't realize they had a perfectly good compiler already in front of them.

The Fuzzy Line

There is no doubt that the line between code and scripts is fuzzy. Generally, it's a bad idea to put artificial intelligence (AI) behavior in scripts, whereas it's generally a good idea to have a scripted trigger system for making the world interactive. The rule should be: If the logic is too complicated, it belongs in the code. Scripting languages need to be kept simple, so they don't consume your game (and all of your programming resources).

However, some games are designed to let players write their own AI. Most commonly, these games are first-person shooters that allow the creation of bots. When this is the goal, it's inevitable that the scripting language will resemble a real programming language. An example of this situation is Quake C. Since bot creation was a requirement of the design, resources and energy had to be put into making the scripting language as useful as C. A scripting language of this magnitude is a huge commitment and shouldn't be taken lightly.

Above all, remember that you don't want your game designers or scriptwriters programming the game. Sometimes programmers are trying to shirk responsibility when they create scripting languages. It's all too easy to lure game designers into programming the game. Ideally, programmers should be boiling down the problem and exposing the essential controls in order to manipulate the logic. That's why programmers get paid the big bucks!

Idea #6: Avoiding Duplicate Data Syndrome

It's standard programming practice to never duplicate code. If you need the same behavior (for example, a common function) in two different spots, it needs to exist in only one place. This idea can be applied to data by using references to global chunks of data. Furthermore, by taking a reference to a chunk of data and modifying some of its values, you end up with a concept very close to inheritance.

Inheritance is a great idea that should be applied to your data. Imagine that your game has goblins that live inside dungeons. In any particular dungeon, your data defines where each goblin stands, along with its properties. The right way to encapsulate this data is to have a global definition of a goblin. Each dungeon's data simply has a reference to that global definition for every instance of a goblin. In order to make each goblin unique, the reference can be accompanied by a list of properties to override. This technique allows every goblin to be different while eliminating duplicate data.

This idea can be taken to multiple levels by allowing each chunk of data to have a reference. Using this technique, you can have a global definition of a goblin along with another global definition of a fast goblin that inherits from the basic goblin. Then inside each dungeon definition, regular goblins or fast goblins can be instanced trivially. Figure 1.0.1 shows this inheritance concept using referencing and overriding of values.

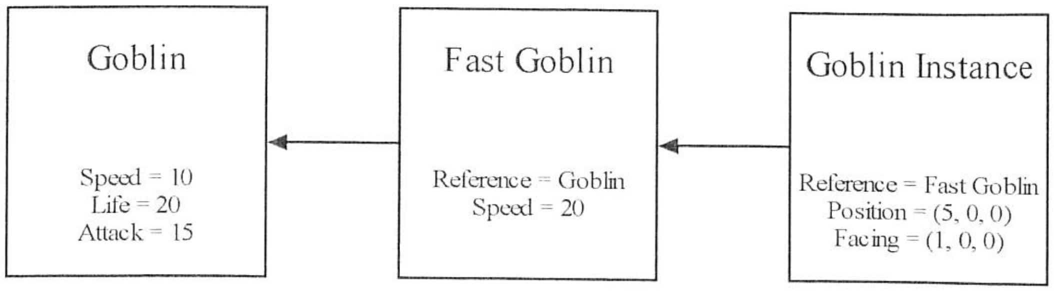

FIGURE 1.0.1. Data inheritance.

Idea #7: Make the Tool That Makes the Data

With any large game, text files eventually become unruly and hard to work with. The real solution is to make a tool that writes the text files. Call this tool a game editor, a level editor, or a script editor, but you'll speed up the game development process by building the right tools. Having a tool doesn't change the data-driven methodology; it merely makes it more robust and efficient. The time you save always makes the extra tool development time worth it.

Conclusion

It's easy to buy into the data-driven methodology, but it's harder to visualize the dramatic results. When everything is data driven, amazing possibilities unfold.

An example of this rule is the game Total Annihilation. The designer, Chris Taylor, pushed data-driven design to the limit. Total Annihilation was an RTS that featured two distinct races, the Arm and the Core. Although the entire game was centered on these two factions, they were never hard-coded into the game. Theoretically, data could have been added to the game to support three races, even after the game shipped. Although this possibility was never exploited, Total Annihilation took full advantage of its flexibility. Since all units were completely defined by data, new units were released on a weekly basis over the game's Web site. In fact, many people created their own units with functionality that shocked even the game's developers.

The data-driven design helped Total Annihilation maintain a committed following in a crowded genre. Since Total Annihilation, other games, such as The Sims, have employed the same idea by providing new data content over their Web sites. Without developers' serious commitment to the data-driven philosophy, this unprecedented expandability wouldn't be possible.