Applying Video Game Interaction Design to Business Performance

Round 2

by Ara Shirinian and Erik Dickelman

A basic division in the domain of interactive computer software exists between programs designed to help accomplish a business task (henceforth called enterprise tools, or just tools) and programs designed for entertainment purposes (games). From the software developer's perspective, video games and software tools share nothing significant in common. Ostensibly, the two subsets of the software world are mutually exclusive. It would be difficult to find a sentence referencing both *SAP*™ and *The Legend of Zelda* in the software literature.

On further consideration, however, several interesting parallels emerge with respect to goals. Tools are created to aid the user in accomplishing external business goals, while games are designed to accomplish goals that are internal to the software. Further, there are two fundamental goals pertaining to games: The first is completing a task that exists completely within the game's universe. For example, in the game *Pac-Man*, one must eat all the dots on the screen while stealthily avoiding capture by ghosts. The second, and perhaps more obvious one, is the real-world goal of entertaining the player. Because we consider the player to be integral to the software system, we consider this goal to be internal as well.

Because of the disparate nature of these software types, there is little communication or cooperation between those who make games and those who make tools. Game developers see tools as boring. Similarly, tool developers regard video games as trite and shallow amusements, compared to their domain of creating enterprise databases that are essential corporate assets. However, just because software might take the form of a game does not make it any less worthy of serious attention. In fact, games may be more deserving of attention because of the special psychological responses they elicit from their users: *Games are played because people want to play them*. Conversely, software tools, especially in an enterprise setting, are used because their results are required by the business. The game designer faces a greater challenge in designing software that individuals will choose to use. Enterprise systems are designed, built, and selected on the basis of technical and financial requirements of a larger corporate entity that is several times removed from the individual.

Nevertheless, we need to bear in mind that games and enterprise systems are both products. Although for different markets, they must follow the product development life cycle, ultimately reaching a compromise between engineering, user experience, and sales and marketing, to be successful (Norman, 1998).

Ultimately, the effectiveness of either software type largely depends on the quality of the interface between its human and computer components. Accordingly, and perhaps the most overlooked parallel between video games and enterprise systems, they share the *human* as the most significant component. It follows that human-centered design considerations that are effective in creating one type of software should apply equally to the other. It is no secret that the user-experience leg of the development triad for enterprise software gets the least attention, but at the greatest cost. Raising the user-experience bar for tools according to how games are developed is perhaps the answer.

Those designing user interfaces in the enterprise world can learn a great deal by studying the kinds of interfaces found in video games. Further, the video game domain naturally lends itself to experimentation and study of interface design, whereas interfaces for enterprise systems continue to take a back seat to system databases, middleware, and other engineering considerations. But user interface design problems are really the same regardless of the context or function of the software, because they share the fundamental human component. Thus, the lessons learned in video game design should be applied to software tools. In doing this, developers can more closely approach the ideal of creating enterprise systems that humans are actually motivated to use.

Progressive Functionality in Design

Productivity varies with an individual's motivation. Common sense tells us that when one is motivated to do something, he or she is more productive than when the motivation level is low or nonexistent. The interface between the human and the computer can create interactions that either encourage or hinder the user's motivation. For example, if a software interface (whether it be game or tool) is too complex, it discourages the user. In games, this human trait is *accommodated* by gradually introducing complexity. The idea is to keep the *perceived* level of complexity constantly at a low level or a level that matches the user's motivation. For example, when one is exposed to an environment that is complex but contains many familiar artifacts, the perceived complexity is low; that is, as one is more knowledgeable about the system environment, its perceived complexity decreases (Norman, 1993).

In tools we rarely see this concept applied. When a novice runs an application like SAP™ or PeopleSoft™, an entire suite of functionality and complexity is present from the very beginning, manifested in multiple nested menus comprising thousands of features with no apparent process support. What does the user do? He or she takes training courses or reads manuals to control how complexity is encountered and conquered within the system. Extrinsic and external supports are often included, such as online help, embedded tutorials, *bolt-on* guides, wizards, and the like. This so-called support adds more features and complexity, is little more than a bandage, and underscores how unusable, un-human-centered, or un-performance-centered the application is in the first place. A survey of 164 enterprise resource planning (ERP) system user companies in 62 US Fortune 500 companies reflected this, and the situation has not changed much since that time. The survey found that "ERP-enabled enterprises are now using a variety of bolt-ons and other complementary technologies" while attempting to attain business performance—and still failing in many important ways (*Financial Times*, 1999). ERP functionality may form the backbone of the business requirements, but to make them usable, functional, and maintainable, from business and human perspectives, they require more.

Suppose we had a special mode of using an application that enabled users to gradually gain competency. Initially, only the most basic functions would be available. As use continued, according to certain metrics that the software would keep track of, new functionality gradually would be unlocked, presented, and supported at the user's level of competency. Eventually, the complete functionality of the software would be exposed to a user who is confident and competent with all. The techniques to design such an interface are found, in part, in user-, human-, interaction-, and performance-centered design life cycles that match natural flows of work with human preferences manifested in *personas* (Cooper, 1999). Such techniques are not, however, applied to the same extent as in game design.

Gradually matching and increasing complexity, knowledge, and motivation to attain a satisfying user experience is far superior to elements we find in today's tools, like random tips-of-the-day or invasive and inaccurate user-interface agents that try to infer user intention from action. Wouldn't it be far more effective to pave the smoothest and simplest (that is, most *natural*) road to becoming an expert instead of having the tool continually try to pre-empt one's actions?

The software tool itself is potentially a much more powerful medium than a manual or training seminar for enabling competency. Simple, intrinsic human-centered elements can supplant all the roundabout extrinsic and external techniques that people use today to achieve competency.

Many games have players learn by doing to achieve competency. It can be thought of as an embedded tutorial phase of

the game that is indistinguishable from the rest of the game. The whole point of gradual increases of complexity is to introduce new play mechanics (for example, functionality) in a manner where the player can learn and master them with the least amount of frustration. If the mechanics of the game include maneuvers A, B, and C, then first the A maneuver is presented and the game is built around that only. Then, when the player has mastered the A maneuver, the B maneuver is presented, and the game revolves around using maneuvers A and B in various ways. And so on. The great thing about a game is that the designer can devise elements of interaction that ensure that the player has mastered maneuver A before maneuver B is encountered. When the player reaches a certain point in the game, the designer is able to assume certain things about his or her knowledge and skill. But all of this is achieved by virtue of playing the game, not learning how to play it in advance.

Figure 1. Metroid (©1986, Nintendo of America, Inc.).

A great example is the classic game Metroid (see Figure 1). At the very beginning of the game, the player's character's actions are relatively limited. She can jump, move, and shoot in various directions in a two-dimensional world that is represented in a side view. First, few enemies threaten the player. The ones that are around are slow and easy to destroy. At this point, the game is just a toy that allows the player to experiment. It's an opportunity to develop an understanding of how the mechanics work without any pressure.

The first several screens are like this. Even though the functional purpose of these screens is that of a tutorial, there is nothing to suggest that the player is in a tutorial per se. It doesn't *feel* like a tutorial, but at the same time, the player is learning and mastering the interface as a secondary consequence of playing through this section of the game.

Close by, within the space of just a few screens, there's an ingenious and simple design that enables the player to master a number of more complex gameplay elements. First, there is a large structure that the character must jump over if she is to proceed beyond it. By jumping on and over this structure, the player learns how the jumping mechanics work. At the same time, the designer knows how much the player understands about jumping if he or she is able to get over the structure.

At the bottom of the same structure, there is a narrow horizontal opening that's much too small for the player to fit through. Beyond that is a dead end; furthermore it's impossible for the player to return over the structure in the same manner because the ledge on one side is much higher than the other. In the dead end area, the player obtains an item that slightly increases the complexity of the interface: The character now can curl up into a ball that's just small enough to fit through the aforementioned opening. The player is also forced to use this new ability to proceed any further in the game. This entire sequence of actions takes place within the space of only a minute or two, and at the end the game has successfully taught the player several basic actions involved in game play while ordering their presentation in a gradual manner to ensure that the player is not overwhelmed. At the same time, the sequence of events flows intuitively.

Traditionally, a player might spend time learning how to play a game before he or she starts playing it. These two phases are usually thought of as sequential and disjointed. In a game like Metroid, the player learns how to play the game and plays at the same time. Efficiency and enjoyment are increased simultaneously. The same principles can be applied to the design of enterprise systems to achieve not only greater productivity, but also more satisfied users.

Inside-Out Design

One of the unique qualities of video game design is that because there are no real standards (though there are some practiced conventions) and fewer restrictions and limits are imposed, there are a wide variety of interfaces. The designer can more readily get a sense of which kinds of interfaces work well and which do not. In contrast, software created for an operating system such as Windows® is subject (at least by convention) to a large set of standards that bind the interface. But just because there is a set of standards does not mean that all the associated conventions are good ones.

Of course, over time, the user becomes accustomed to such standards and they gradually become intuitive, meaning that things like radio buttons become familiar virtual artifacts even though the original artifact (a button on a 1930s radio) is completely unfamiliar to the user. For example, developers have strong preferences for certain operating system standards and conventions in which they work. Traditional disagreements arise as one developer has a working history with the Windows environment while another has experience with the MacOS environment. Implementing aspects of either interface type may be good or bad, but the difference in design is neither good nor bad, just different. Each has become accustomed to the artifacts of his or her respective

operating system and how it handles interactions. In the absence of alternatives, each developer subconsciously assumes that his or her choice is best.

In this context, graphic operating system standards are really a trick to compensate for a less-than-ideal one-size-fits-all interface by forcing it on users in as many forms as possible and for as long as it takes for it to become familiar and seemingly more intuitive. To make matters even worse, graphic operating systems developers (Windows XP, Mac OS X, etc.) seem compelled to change many of their established conventions every five years or so.

It is the purity of the video game, because of the minimal imposition of standards, that creates the best environment for effective testing of its interface. For the same reason, it is the most difficult type of interface to design and can be more effective than even a well-designed standard interface for a Windows application.

Imagine that the software tools currently employed to perform work...were also designed to provide the user with visibility into his or her own performance metrics in real time.

For better or worse, enterprise software is essentially data centric. In the simplest view, these are applications with only two real functions: the storage and retrieval of corporate data. As such, the primary concerns in the design of such software are in the technologies and structural designs that ensure the integrity of the data, allow for its storage and retrieval, and provide system interfaces to other enterprise applications. Generally speaking, enterprise systems are built from the inside out and the user interface is derived from the structure of the data to offer only the basic functional user requirements.

Enterprise software often contains intentional redundancy and ambiguity in the user interface to make it as generic as possible to support a variety of implementations. This may make sense from the point of view of versatility, but when put into a specific context, this excess of functionality only serves to make the software complicated and difficult to understand. To be usable, the interface needs to reflect only the functionality that is needed to support the business goal. But there is no inherent way to accomplish this, given the generic interface, and it is not practical to design a new interface for each implementation of the system. The associated costs of custom enterprise software are prohibitive.

In recent years enterprise system vendors have added other user interface customization capabilities, such as filtered views based on job requirements, but this is a far cry from the user-centered approach of video game interface design. In the ideal outside-in implementation, there should be no need to rely on wizards or other types of wrapper functionality that attempt to patch the problem with a higher level interface instead of allowing the real human need to drive the process from the beginning.

Engaging the User

In any kind of interface design, feedback is an effective technique to show the user the results of his or her actions. Direct and continuous feedback is even more effective. In the domain of video games, quantitative usage and play metrics are a special kind of feedback otherwise collectively referred to as *score*.

In its simplest form, score measures the player's skill level in the game and is represented by a number that increases in various ways, depending on the actions taken. Because it is important for this feedback device to never discourage the player, score starts at zero when play begins and should never decrease by any amount. The challenge to simply obtain the highest score possible is a highly effective psychological technique. Even if the gameplay is boring, concentrating on maximizing the score often makes the experience more exciting for the player. Interestingly, the game then becomes wholly an attempt to increase the value of this number and any other result of the player's actions becomes a side effect of this attempt.

This principle can be quite useful when applied to tools. If the score metric is implemented properly, the resulting software is one where the user's performance is directly correlated with score. The user then doesn't need to think about performing well; he or she only needs to think about obtaining the highest score possible. At the same time, the mechanism provides additional motivation. This feedback also helps the user increase his or her productivity in ways that may not have been possible with other methods. In the corporate world, *score* might mean compensation, privileges, status, or other motivating perks.

We have already seen this idea applied to certain kinds of software, albeit in a limited and rudimentary manner. For example, in some message board systems, users get ratings and titles based on how many times they have posted. On the popular auction site eBay, users get a rating number and graphic icon based how many successful transactions they have completed.

In an enterprise setting, a worker's compensation is tied to his or her productivity. Unfortunately, this metric is frequently applied in the form of a review summarizing

performance, and often with the negative objective of corrective action rather than of rewarding achievement. But imagine that the software tools currently employed to perform work and measure productivity in this environment were also designed to provide the user with visibility into his or her own performance metrics in real time. Eureka! The user suddenly has an intrinsic score to strive to increase on a continuous basis and from which to perhaps foster a healthy sense of competition in the workplace.

To this end, just how game-like can we make enterprise software?

References

Cooper, A. (1999). *The inmates are running the asylum: Why high-tech products drive us crazy and how to restore the sanity*. Indianapolis, IN: SAMS.

Financial Times. (May 26, 1999). *Systems fail to fulfill promises.*

Norman, D.A. (1993). *Things that make us smart: Defending human attributes in the age of the machine.* Reading, MA: Addison-Wesley Publishing Company.

Norman, D.A. (1998). *The invisible computer: Why good products can fail, the personal computer is so complex, and information applicances are the solution*. Cambridge, MA: MIT Press.

Ara Shirinian is the Senior Editor at *Tips & Tricks* magazine, a video gaming publication, where he is attempting to "shake up" the industry from the inside with his unique specialties in interface design. Ara first cultivated these interests as a Research Assistant for the Human-Computer Interaction Laboratory at the University of Maryland, College Park, where he developed user interfaces from a human factors engineering philosophy.

Later, as a Software Engineer for WPI, Inc., he further developed these skills by designing and implementing custom performance support applications for clients such as the Federal Deposit Insurance Corp. Currently, Ara applies his technical background specifically to the video game industry, developing analyses and methodologies that bring the concepts of human factors, interface design, and performance-centered design to the process of creating better video games. He may be reached at ara@housenet.org.

Erik Dickelman is a Project Leader for RedPrairie, one of the leading providers of integrated supply chain solutions. In his formal role, Erik specializes in the design, implementation, and integration of high-volume enterprise logistics applications (warehouse, transportation, and productivity management systems) for tier-1 companies such as Hershey Foods Corporation, Stanley Tool Works, and Tyson Foods, Inc. He is a graduate of the University of Maryland, College Park, where he developed a strong interest in performance-centered techniques. Erik describes himself as working professionally on the back ends of systems but having his spirit and interests on the human end. He may be reached at ejd@housenet.org.