



**ROBOTICARM**  
SOFTWARE

**Note**

This is a living document. To read the latest version, go to [www.RoboticArmSoftware.com](http://www.RoboticArmSoftware.com).

## **Make Better Games Faster!**

### *Achieving Pipeline Efficiency Through Tool Usability*

Dan Goodman, Founder & Director, Robotic Arm Software  
[www.RoboticArmSoftware.com](http://www.RoboticArmSoftware.com)

Consider the following scenario. Engineering develops a level design tool that most of the designers flounder with. The engineers point to the few designers that use it somewhat effectively and say, "There's obviously nothing wrong with it." The rest of the designers struggle for months, falling behind in their work, no matter how many extra hours they put in. The game ships to mixed reviews. The artwork looks good, they say, but the level design is terrible. The designers who floundered with the tool leave the company for greener pastures.

I didn't invent this scenario; I've seen it happen, and you've probably seen something similar. Maybe it wasn't a design tool. Maybe it was an effects tool, or a UI tool, or any of a dozen other tools developed internally at your company. More often than not, the tools developed by game studios don't meet the needs of the developers. They're slow and clumsy, and they make their users slow and clumsy too.

So how do we fix it? Well, there are many ways to attack the problem, but let's not get all gung ho just yet. First we need to understand why the problem occurs. Only then will we really know the best way to address it.

### **Where Game Companies Go Wrong**

Most game companies are not very good at making tools. The

Make Better Games Faster! Achieving Pipeline Efficiency Through Tool Usability  
Copyright Robotic Arm Software, LLC 2008  
Last Update: 01/14/09  
[www.RoboticArmSoftware.com](http://www.RoboticArmSoftware.com)

*A Game company's focus is on game development, not on tools development.*

focus is on game development. After all, games are the final product that ends up on the shelf of the local Wal-Mart. Tools are seen by many game developers as transitory. They believe that the tools they're using on their current project will not be the tools they'll be using on the next one. Because these tools don't receive the attention they deserve, they don't fulfill the needs of the end users, and must eventually be rewritten. And so the process begins again with tools that are little better than the ones before them.

There is a better way to develop tools – and games. Most of the industry just hasn't figured it out yet, because they are stuck in the old cycle, making the same mistakes over and over again.

### ***Leveraging the Wrong Technology***

Many game companies try to leverage existing technology to speed up the tool development process, taking shortcuts. Most people would applaud such efforts, considering typical time constraints on game development. But consider the consequences of taking a tool designed from the ground up for doing one thing and extending it to do something completely different. At one company I worked for, this is exactly what happened. Management decided to use Maya as a level design tool.

*Shortcuts may save time in the beginning, but you'll probably pay for it later.*

I was shocked to learn of the decision that had been made long before I joined one company. Our CEO, whom I greatly respect to this day, was also an engineer and had even given a talk at the Maya API Developers Conference about his ingenious marriage of Maya to C#. As a programmer, it made a lot of sense. After all, why reinvent the wheel? Much of the functionality needed in a design tool is already present in a modeling program like Maya. Now I cringe a little every time I hear of another company attempting to do the same thing.

Sure, it was technologically clever as an academic exercise, but we were making games. The interface for Maya is already cluttered enough to confuse the artists that use it. Adding another layer of complexity – a great deal of complexity, by the way – just confuses things further for a group that does not have a great deal of Maya expertise to begin with. Sure, much of the old interface can be removed with a little Mel scripting, but some things are static or just very difficult to change. By the time you reconfigured the entire interface, you could have implemented

the functionality you actually wanted yourself. And then there were the performance issues.

By the end of one project, Maya was taking longer and longer to load a level, and once it was loaded, it was a snail's crawl to navigate. Luckily, the idea of splitting the level into layers that could be disabled was core functionality that was implemented early on. Once the level did load, the first thing to do was turn off everything in the level that wasn't absolutely necessary. Even then, performance wasn't great. All in all, it was a highly unusable tool. One of the programmers downright refused to open it to change anything – no matter how simple. If he needed something changed to fix a problem, he made someone else do it. Since then, that company has wisely abandoned Maya as a design tool, due to its low usability. It took two less-than-stellar projects and a bunch of aggravated designers to figure this out.

### ***Lack of Design***

Achieving success requires an understanding of what success means, and having a plan to achieve it. Too often in game development, we pick a vague direction and slingshot ourselves forward, without any idea where – or how – we're going to land. The act of design is like painting a mark on a target, so that the entire team can fly toward that goal instead of randomly flailing about in the sky.

The “seat of your pants” development style often finds its way into our technology teams. There's a game to be made and tools needed today – this very minute, in fact – and there's no time to waste “designing” them. So seems to be the attitude of some managers.

What they don't realize is this: software always gets designed, whether time was allocated or not. Unfortunately, designing as you go, which is what programmers will do when not given a choice, does not achieve the best results.

Once coding begins, software has a kind of inertia. Every decision made is affected by every decision that has come before it. If a stumbling block presents itself, a workaround is put in place, and as coding continues, it becomes more and more brittle.

When requests for new features come in, as they always do,

*Design that isn't scheduled still occurs – just poorly.*

engineers find the implementation that is least likely to break the entire system. The smallest change, after all, can cause unforeseen consequences. The new feature may not be able to work as the end user had needed. If only a design process had been in place from the beginning, the foundation could have been solid enough to account for such requests.

### ***Programmers as Interface Designers***

The programmer developing the tool might not even be the best person to design its interface. He has a stake in the tool's success, but success for the programmer may not be the same as for the end user. Alan Cooper discusses this issue at great length in his book, "The Inmates Are Running the Asylum."

He calls one of the problems of programmer-designers "scarcity thinking". As software engineers, we have been trained to fit our programs into constraints that, for many applications, no longer apply. Cooper says that most software is designed to serve the machine instead of the user in a false belief that the CPU is overworked. However, computers have gotten so incredibly powerful over the last few decades, that the CPU is actually idle more often than not.

This mindset is more relevant to the game industry than most sectors of software development, since we are constantly trying to squeeze performance from our games. We are limited in what we can do with the processors of current generation consoles, so we take performance into account on a daily basis. It would be unnatural for us not to do the same with our tools. Unfortunately, that means less focus on the end user. This is merely part of a larger problem, and is just one way an engineer's desires counter the needs of end users.

Engineers want total control, while end users are likely to give up that control for simplicity. Have you ever seen an engineer supply a feature-rich tool to an artist or designer, only to have half the features go unused? I have. How many days or weeks of the engineer's time were wasted on features that the end users invariably didn't understand, and ultimately didn't need?

Programmers also concern themselves with edge cases that most end users will never experience. Cooper stipulates that most end users are willing to accept a rare setback as long as the most common use cases work well.

*Most programmers don't think like end users and are unsuited to design interfaces.*

Lastly, the engineer's understanding of the underlying technology actually hinders the ability to create an interface to match the end user's goals. Engineers often fall back to an implementation model of the interface – an interface that exposes the underlying data structures directly to the user. Although it makes sense to an engineer to match the interface to the underlying system, an end user, with little or no understanding of the implementation, will often find it bewildering.

### ***Low Tool Usability***

All of the problems mentioned above inevitably lead to low tool usability, which is why Cooper asserts the need for what he calls “interaction designers”, also known as user experience (UX) designers. UX designers, acting as usability experts, are slowly usurping the programmer's role as interface designer in other software based industries.

Usability is defined by the Usability Professionals' Association's (UPA) website as “the degree to which something - software, hardware or anything else - is easy to use and a good fit for the people who use it.” According to the UPA, usability measures how “efficient, effective and satisfying” a product is.

Outside the game industry, improved usability has been linked with increased productivity, decreased training and support costs, increased sales and revenues, reduced development time and costs, reduced maintenance costs and increased customer satisfaction, according to the UPA's “Business Benefits of Usability”.

In game industry terms, tools with high usability mean faster, better, and happier artists, designers, and programmers, a shorter learning curve, and less technical support from the tools team. Tools with low usability, of course, have the opposite effect. Often, game development tools fall into the latter category, due to lack of design or understanding of the end users' needs.

Making a concerted effort to improve usability in your development tool chain will reap real, long-term rewards. Just over the course of a single project, significant savings can be had, meaning more time for polish – something many teams intend on, but never seem to have enough time to actually do.

*High tool usability allows your developers to spend time on polish.*

Having that extra time translates directly into better games, better press, better sales, and so on.

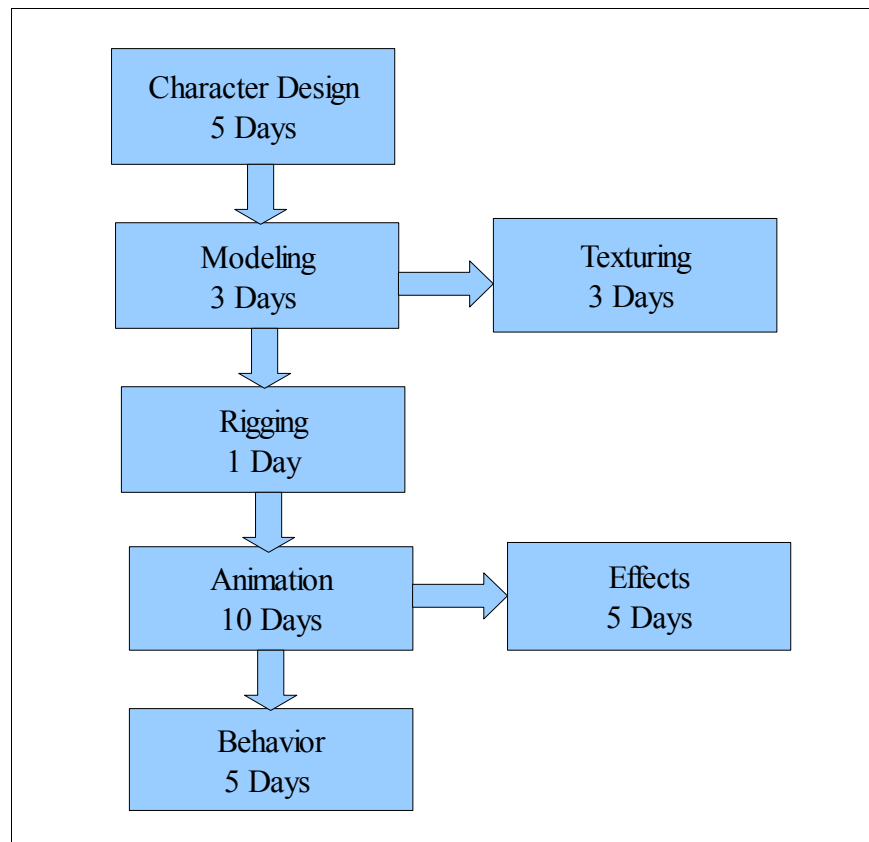
Of course, as game developers we are trained to get “the most bang for the buck.” To that end, we need to determine where in the tool chain we can benefit the most from improving usability.

## Finding the Hot Spots

When optimizing game code, the key is to find the biggest bottlenecks. Profiling software helps engineers determine which functions to optimize. Generally speaking, the functions that use the most time per frame are targeted first. In the development pipeline, the steps that take the most total time should be considered for optimization.

Consider the following illustration showing a potential workflow for character creation, from concept to finished character. Estimated times are in days per character.

*Optimizing your development pipeline requires knowing the problem areas (hot spots).*



The figure above represents a very small, incomplete piece of

the game development pipeline. Your actual pipeline will be much more complex, including more areas of development (level design, UI, gameplay programming, and so on), resources performing multiple processes, and multiple dependencies for a single task. If you can determine which parts are holding up development the most, you have a good start in finding what to optimize.

Assuming that each phase of character development needs to be complete before continuing to the next, and one person per phase, the first character could be fully implemented in 24 days. We get this by adding up the number of days in the longest pathway, from character design to completion.

Character Design + Modeling + Rigging + Animation + Behavior  
= 24 Days

Texturing is left out in this case because no other tasks are dependent on it, and the time it takes is less than the other pathway from modeling to behavior. Either effects or behavior could have been excluded since they are both endpoints and take the same amount of time. In that case, I just picked one.

That doesn't mean that it takes 24 calendar days to complete every character. This is an assembly line; four other characters are being worked on while the current one is being finished – one at each stage in the pipeline. So how many calendar days does it take to complete a new character after the first? Exactly the time it takes for the longest stage – 10 days.

That sounds great! Ten characters can be complete in just under 6 calendar months. Of course, if just the animation time were cut in half, we could double the character output by creating a new character every 5 days.

In any assembly line, efficiency is achieved by breaking down tasks into component parts so that every stage of manufacturing takes an equal amount of time. That would also be the most efficient model for game development. Unfortunately, each stage in the game development pipeline requires unique skills, and it is often difficult to predict the exact timing for each stage. So, work backs up behind one developer, while another sits idly by waiting for another's work to be completed.

There are many tactics for getting around this issue, all of which

*It's easier to manage efficiency than inefficiency.*

are employed by game companies in one extent or another. Managerial overhead and individual efficiency is often traded for development pipeline bandwidth. Still, if we can increase efficiency in the areas of the pipeline that cause slowdown, the team would be capable of much, much more, and the overhead would be much, much less.

There are several ways to do this. Training is one obvious example. Even the most seasoned animator could learn a few shortcuts for the latest version of Max or Maya. Still, one of the most effective way of increasing efficiency across all members of a particular discipline is to increase tool usability. Often, it is the unforeseen technical issues with the tools which lead to wasted time, and shorter learning curves allow new team members to start contributing much more quickly.

## **Determining Usability**

Once you understand where the bottlenecks in your pipeline are occurring, it's time to decide where improving tool usability can help. To do that, you need a way to determine the current usability of the tool used for a particular process and how much it can be improved. Whether or not you believe that these (or other) methods can truly measure something as subjective as usability, it's still a starting point.

Very little work has been done in the area of game development tool usability. Current usability techniques, which come from other industries, don't address the unique nature of game development. For the most part, they have been applied to non-development, consumer products, and ignore major issues faced by game developers.

One of the biggest issues, by far, is iteration. Because of the interactive nature of gaming, there are often edge cases found at runtime that just weren't obvious in the tool. Going back and forth between tool and game is a hassle for developers, especially when tracking down errors of logic or syntax.

The manner in which these methods are applied to internally developed game tools differs from their consumer product equivalents. These methods are normally applied to users who have as little as an hour of experience with the software. Applying the same techniques to users with months of experience will likely indicate much better results than could be

expected from a new user. Still, some foundation with the type of software may be required. You wouldn't test the usability of Maya by having a programmer try to model something with it, for example. To get the most accurate comparative results, the methods should be performed early in the software's release with individuals unfamiliar with the specifics of it, but familiar with the concepts behind it.

## SUS

The first method uses the System Usability Scale (SUS), developed at Digital Equipment Corporation (DEC). John Brooke (formerly of DEC) wrote a paper on it titled, "SUS - A quick and dirty usability scale". It asks the participant in the survey to rate a series of statements from 1 – 5, where 1 is equivalent to "strongly disagree" and 5 is equivalent to "strongly agree". Use the following statements to conduct the survey of tool users.

*Methods used by other industries to measure usability can be applied to the game industry as well.*

*The 10 statements adapted from the original SUS survey.*

1. I think that I would like to use this tool frequently
2. I found the tool unnecessarily complex
3. I thought the tool was easy to use
4. I think that I would need the support of a technical person to be able to use this tool
5. I found the various functions in this tool were well integrated
6. I thought there was too much inconsistency in this tool
7. I would imagine that most people would learn to use this tool very quickly
8. I found the tool very cumbersome to use
9. I felt very confident using the tool
10. I needed to learn a lot of things before I could get going with this tool

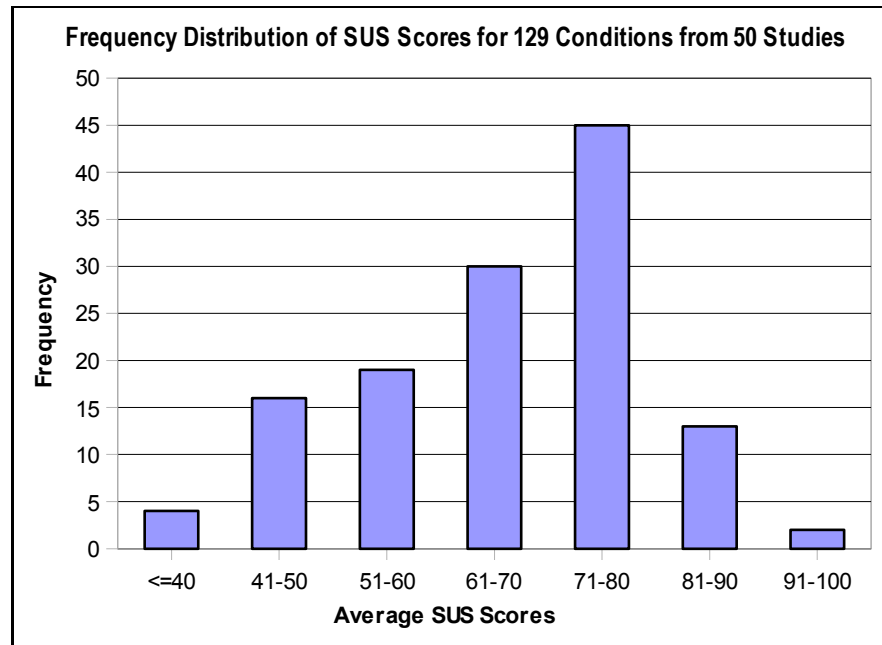
*SUS is a subjective method that measures the user's perception of the software.*

The score value of each item ranges from 0 to 4, and is calculated as follows:

- For each odd numbered item, subtract 1 from the value.
- For each even numbered item, subtract the value from 5.

Now, to calculate the total score, multiply the sum of the individual item scores by 2.5. The total score will range somewhere between 0 and 100.

Tom Tullis and Bill Albert, authors of “Measuring the User Experience”, presented a presentation at the UPA-Boston's Seventh Annual Mini UPA Conference titled “Usability and User Experience 2008”. At this presentation, they showed the following graph of SUS scores taken from 50 studies.



This chart shows typical SUS scores, and could be used to determine comparative usability of your own tools to other applications. A particularly low score could mean that usability improvement will be of great benefit. Of course, SUS is just one “quick and dirty” way to measure usability.

## **SUM**

The Single Usability Metric (SUM) uses task completion rates, task time, error counts and satisfaction to generate a combined usability score. It is discussed at length in “A Method to Standardize Usability Metrics Into a Single Score” by Jeff Sauro and Erika Kindlund.

Users of the software are asked to complete a series of tasks. For example, an individual testing a level design tool may be

asked to complete the following tasks:

- 1) Open the level named Castle\_Level
- 2) Place a new Knight character on the ground at x=345, y=437
- 3) Move the character Ogre1 to coordinates x=962,y=578 z=Ground
- 4) Change ElfWarrior5's health property to 250
- 5) Save Castle\_Level as Desert\_Level

Task completion is measured by the ratio of successfully completed tasks to attempts, so if the user completes 4 of the 5 tasks above, the user has achieved an 80% completion rate.

Error rate measures how much of each task was completed successfully by breaking down individual tasks into subtasks, called "error opportunities". In the example above, moving Ogre1 might be broken down into these sub-tasks:

- 1) Change the current manipulator to the move tool
- 2) Select Ogre1 in the level
- 3) Change Ogre1's x-coordinate to value 962
- 4) Change Ogre1's y-coordinate to value 578
- 5) Snap Ogre1's z-coordinate to the terrain

If Ogre1 ends up at the correct XY coordinates, but is floating above the terrain, it means that subtask 5 failed, and the user completed only 4 of the required steps. The error rate in this case is 1/5.

Satisfaction rate is the most subjective of the four metrics used in SUM. After each task is completed, a simple three-question survey is given to the user to determine perceived difficulty, task time, and overall user satisfaction with the tool for that task. The average value of the three questions is subtracted from 4 – a typical rating for software with good usability.

*SUM combines several metrics – error counts, completion rates, task time, and user satisfaction – into one usability score.*

How would you describe how difficult or easy it was to complete this task?				
Very Difficult				Very Easy
1	2	3	4	5
How satisfied are you with using this application to complete this task?				
Very Unsatisfied				Very Satisfied
1	2	3	4	5
How would you rate the amount of time it took to complete this task?				
Too Much Time				Very Little Time
1	2	3	4	5

Scoring task time is a simple matter of subtracting the actual time from the ideal time for the task. The difficult part is in identifying the “ideal” time. Without data from a previous version of the tool, it is very difficult, but Sauro and Kindlund suggest a method in their paper “How Long Should a Task Take? Identifying Specification Limits for Task Times in Usability Tests” that uses the satisfaction rate, above to derive such a value. In this method, times of failed tasks and tasks with a satisfaction rate of less than 4 are removed, and the task time at the 95<sup>th</sup> percentile of remaining times is used as the ideal.

Once scores are collected, normalize the individual scores to z-values using the mean and standard deviation for each metric. To obtain the final, combined score, simply average the 4 components for all tasks. You can find a SUM calculator (and a ton of other info on measuring usability with SUM) on Jeff Sauro's website: <http://www.measuringusability.com>.

SUM is useful, not only as an overall score of product usability, but also because it gives usability scores at task level. This may tempt you to attack the usability problem by starting at the bottom of the task usability list and working your way upward. Though this may help somewhat, improving usability of features does not necessarily equal improving usability of the whole tool. Not all tasks are created equal, or more specifically, not all tasks are performed equally by end users. In general, software that meets the goals of the user with as little extraneous functionality

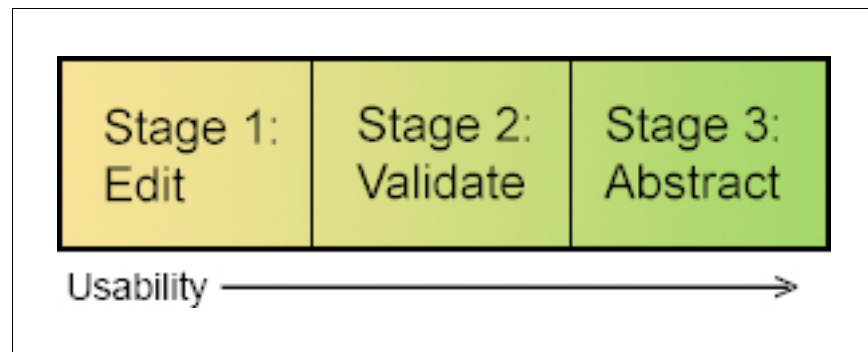
*The per task nature of SUM gives insight into usability of specific aspects of the software.*

as possible will be more usable than software that clutters the UI with buttons and menu options the user doesn't understand.

## Improving Usability

If you've been around the industry for a while, you may have heard programmers claim that Notepad is a tool. You may even remember a time when notepad (or another text editor) was the primary tool for design and possibly some aspects of art. It's probably still in use in your current development pipeline. While Notepad is a tool, it is a poor one in the context of most game development tasks.

There are three basic levels of tool functionality, with respect to the way data is presented to the user, on the wide spectrum of usability, seen below.



### **Stage 1: Edit**

The simplest tools give the user the ability to edit data. Notepad fits into this level of tool usability. Not coincidentally, it also has the widest scope, since any type of data that can be represented as text can be edited. Within this stage are other editors that reduce scope by supplying text boxes for specific data points without supporting validation.

The tools used at this stage are very unforgiving, since there is no protection of the user from himself. Iteration is the key for users of the tool, since it is easy to make mistakes. When errors occur during the runtime, the bad data may be extremely difficult to find. The users may spend hours looking for the number with the wrong decimal place or the text with the incorrect capitalization.

## **Stage 2: Validate**

The next stage, which is often the first and last stop for many game development tools, is one that adds a layer of protection from user errors by limiting the possible input to valid values. This is a major step forward, since developers are able to create game assets that are less prone to error. Designers and artists can begin to feel safe in the knowledge that “at least it will run”.

Still, there is little feedback on what effect the data will have on the game, and therefore there is still a great deal of iteration time necessary to test changes, tweak values, and so on. This is due to the fact that this level of tool development is still based on the implementation model of design. That is, the data that is edited directly by the user is equivalent to the data used by the underlying game system, and is therefore more complex than the user can fully understand. Iteration is still necessary, even though the tool has spared the user from typographical mistakes that could have caused serious errors (crashing). Typos have been eliminated, but errors of logic have not.

Abstraction begins to take place in tools at this stage, albeit on a small scale. Anytime a text box is replaced by a drop down list of enumerated values, the internal representation of the data (a number) is being replaced with something better understood by the user. Color pickers, graphs, calendars, progress bars, and just about any graphical representation of data imaginable all serve this purpose. Still, the individual bits of data do not represent a cohesive whole.

## **Stage 3: Abstract**

Usability and scope are inherently bound together. As the scope of the interface tightens, usability increases. If a tool has too much functionality exposed in the UI (a wide scope), it becomes unwieldy. It follows that the easiest tool to use is the one with the least interaction, but it must give the user the ability to still achieve his goals. So how do you reduce the scope of the interaction, while still editing all of the necessary data? To reduce scope further, data abstraction must occur at the price of precision.

Consider a 3D model. The actual data is a list of vertex positions, weighting, UV mapping coordinates, RGBA values, and so on. Editing a text file that directly manipulates the value

*Not allowing users to shoot themselves in the foot is an important first step toward usability.*

of all of these data points would offer extreme precision, but would take much longer than using a modeling program to generate the same type of data with much better results. The modeling program takes the complex data and abstracts it into something more easily understood by the user, showing what the data represents instead of the data itself. This, finally, is where we leave the implementation model of design behind forever and move to what Cooper calls the represented model.

*Abstraction allows data to be represented to users in a way that makes sense to the end user.*

The abstraction that takes place in the validation stage gives users a more appropriate view of individual bits of data. To achieve true data abstraction is to provide a more appropriate view of what the combined data represents. Individual data points are absorbed into a larger, more conceptual picture.

When a tool gets to the abstraction stage, iteration times are significantly reduced because both typos and logic errors are all but eliminated. The abstract view of the data gives the user the ability to understand its purpose. The final data still needs to be tested in game, obviously, but the data created from the tool should be sound.

Representing the data in some randomly chosen abstract way will not achieve the best results. You must find a way to represent the data in a way that closely resembles how the end user understands it. The closer that representation, the easier the tool will be to use. The only way to create an abstract model of the data that represents the typical user's understanding of it, is to first understand the typical user.

*Understanding users should be the first step in designing for them.*

### **User Models**

Sometimes, we talk about users in a way that suits our own desires. "I think the player will really like this," is often heard as an excuse to include some feature in a game. I'm sure you've heard, and probably even said something similar. Actually, we usually invoke the end user only when it suits us. That's where user models come in.

In "About Face", Alan Cooper describes personas – user models – as concrete, fictional end users that have their own goals, desires, hobbies, families and so on. They aren't necessarily the "average user," but they are representative of a typical user. A single piece of software may have several user models that need to use the software, but the key is to design an interface

so that it satisfies one of those users very well. Therefore, the list of user models must be narrowed to a primary user model – the person the software needs to work best for. It's possible to have more than one primary, but doing so means having multiple interfaces (one for each). If that isn't possible due to budget or time constraints, then it may be best to cater the interface to the least technical user, as others are more suited for jumping through hoops to get at “advanced functionality”.

### **Goal-Driven Design**

Often, we think of the work done by developers as tasks. The schedule is full of tasks to implement features, and we (very wrongly) believe that when those tasks are complete, the software must be complete. The falsehood of this thinking should be obvious to anyone who's worked on a game. Once the features have been implemented, there's still something missing – the part that ties everything together, the *je-ne-sais-quoi*, the fun.

*Achieving goals is the true purpose of tools, not performing tasks.*

Truly, the way to make better software is to target specific users based on user models and focus on their goals – what Cooper calls goal-directed design. Tasks are the means to the end, but goals are the end. Users should be given exactly what they need. Less means they can't do the job; more means the job becomes overly complex. Every design decision should go through the goal filter: “Does this meet the goals of our user?” Only features which meet this requirement should be included in the design.

When goals supersede tasks and features, it is possible to find better ways to achieve success through fewer tasks for the end user. Understanding the goals of the user model also makes it possible to find a way to represent the data in a way that suits that person best. Thus overall usability is achieved, giving the users the ability to achieve much better results in a shorter time-frame.

### **Our Team**

We offer our expertise on game development tools through design and implementation, as well as consulting on tool usability to improve the game development pipeline. We work hard to deliver tools based on end users' needs by using sound

design principles and building strong relationships with our clients.

Our mission is to make better tools for you, so you can make better games. To find out more about how our team can help yours, contact us today.

## Contact Us

Email: [biz@RoboticArmSoftware.com](mailto:biz@RoboticArmSoftware.com)

Phone: (415)692-1472

### About the Author

Dan Goodman spent over a decade as an engineer and lead in the game industry, developing tools, technology and gameplay for a dozen published titles before founding Robotic Arm Software. You can contact him about this paper or just to say hi at [dan@RoboticArmSoftware.com](mailto:dan@RoboticArmSoftware.com).

## References

- A. Cooper. (2004). *The Inmates Are Running The Asylum*. Sams.
- Usability Professionals' Association. (2008) *What Is Usability?* Retrieved December 22, 2008 from Usability Professionals' Association Web Site:  
[http://www.upassoc.org/usability\\_resources/about\\_usability/index.html](http://www.upassoc.org/usability_resources/about_usability/index.html)
- Usability Professionals' Association. (2008) *Business Benefits of Usability*. Retrieved December 23, 2008 from Usability Professionals' Association Web Site:  
[http://www.upassoc.org/usability\\_resources/usability\\_in\\_the\\_real\\_world/benefits\\_of\\_usability.html](http://www.upassoc.org/usability_resources/usability_in_the_real_world/benefits_of_usability.html)
- J. Brooke. (1996) *SUS: A quick and dirty usability scale*, pages 189–194. *Usability Evaluation in Industry*. Taylor and Francis.
- T. Tullis & B. Albert. (2008) *Tips and Tricks for Measuring the User Experience*. Presentation from the UPA-Boston's Seventh Annual Mini UPA Conference, May 28, 2008. Retrieved December 17, 2008, from Measuring User Experience Web site :  
<http://measuringuserexperience.com/Tips&Tricks-Boston-UPA-2008.pdf>
- J. Sauro & E. Kindlund (2005) *A Method to Standardize Usability Metrics Into a Single Score*. Retrieved December 17, 2008, from Measuring Usability Web site :  
<http://www.measuringusability.com/papers/p482-sauro.pdf>
- J. Sauro & E. Kindlund (2004) *How Long Should a Task Take? Identifying Spec Limits for Task Times in Usability Tests*. Retrieved December 17, 2008, from Measuring Usability Web site :  
[http://www.measuringusability.com/papers/hcii2005task\\_times.pdf](http://www.measuringusability.com/papers/hcii2005task_times.pdf)
- A. Cooper. (2007). *About Face 3: The Essentials of Interaction Design*. Wiley.