Game DESIGN Workshop

A PLAYCENTRIC APPROACH TO CREATING INNOVATIVE GAMES

by Tracy Fullerton

with a foreword by Eric Zimmerman

MORGAN KAUFMANN

Catastrophic Prototyping and Other Stories

by Chaim Gingold

Chaim Gingold was part of Spore's original concept team and is the design lead of Spore's creature creator and other in-game creativity tools. Prior to working with Will Wright, Chaim studied with Janet Murray at Georgia Tech, where he earned an MS in digital media. He has spoken around the world on game design, prototyping, and player creativity. His current obsessions include improv, trains, and Japanese aesthetics.

My hard drive was full of failures. Twelve years after learning to program, I looked back on all my software: Almost none of it was finished, and what was, wasn't ambitious enough. The projects that started out ambitiously always seemed to fall back to Earth, like failed rockets lacking the power to propel their own weight into orbit. Sure, there were interesting ideas in there, lots of wacky toys, and I had even attempted a few large projects, but none of them ever came together like the cool games and software I had always admired.

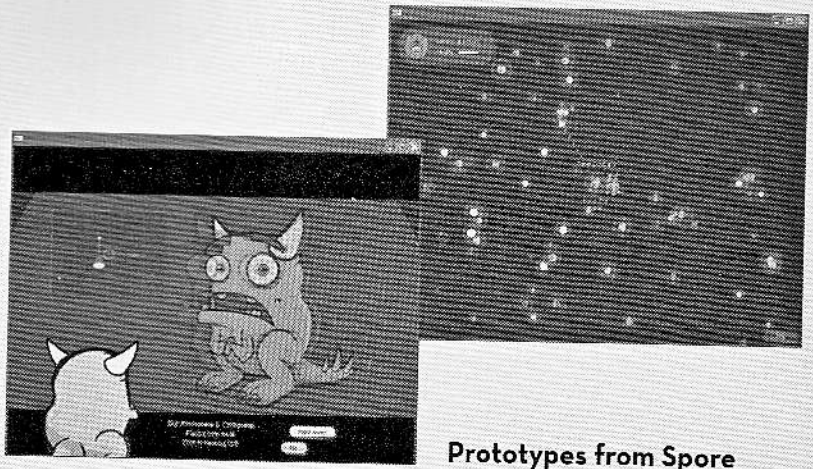
Sure, I had become a pretty good programmer and learned to make cool stuff, but clearly none of it would ever amount to anything. I just didn't have what it took.

I went to graduate school at Georgia Tech and read some Chris Crawford. I learned that he had the same problem. But he didn't think of it as failure. For him, this was an organic part of the development process. The failures filling his head were actually prototypes that helped him decide which ideas were worth pursuing. For each good idea, there were a large number of stupid ones that didn't work out. Failing, for this successful designer, was a way to find the good ideas. The revelation hit me like a ton of bricks. Maybe I had a chance after all.

Ken Perlin came to Georgia Tech and gave a talk on his work with emotional software actors. His work blew my mind. He had an infinite series of cool little toys, which he considered to be sketches or studies. Master artists like Escher or van Gogh don't just sit down and crank out a finished piece. Artists create numerous sketches and studies before they undertake finished paintings, let alone masterpieces. Ken's larger demos clearly built on top of what he had learned in previous ones. It all formed one long line of inquiry and research. In Ken's world, my failures, which I was now calling *prototypes*, were like an artist's studies, a necessary part of any major undertaking.

All of my software failures, which I was now thinking about as prototypes, sketches, and studies, had taught me a thing or two about design and programming. If you want to learn to draw, you have to make a ton of bad drawings first. The difference between practice and failure is simply a matter of attitude. One thing led to another, and my experience, plus being in the right place at the right time, led to an internship with Will Wright. He was working on a new game at the time, code named Spore, and had a small team working on prototypes.

I went to Walnut Creek for the summer, joined the impossibly small Spore team, and it all finally started to click into place. Maxis was in the thick of The Sims Online, and the other intern and I were placed in the hallway outside of Will's office, next to the Elvis shrine, on folding tables. Under my "desk" was one of Will's old Macs, a fancy machine from the mid-1990s, which, to my delight, we hooked up. Spending the summer at Maxis was like going to Santa's workshop at the North Pole and finding out how the elves made the toys.



Prototypes from Spore

The old Mac was like a treasure cave, a historical archive of blueprints, prototypes, projects, and concepts. I could study the source code to some of my favorite games, like SimAnt, SimCity, and SimCity 2000. But that wasn't even the best part. I found an ambitious Maxis project about tribal civilization from the early 1990s that was never completed. It was like a murder mystery. Why had this project died? The hard drive was full of prototypes for a secret project, which turned out to be The Sims. Apparently, Maxis had been working on the game for a long time, and many aspects of it had been prototyped in isolation, including a 2.5d character animation system and editor, the motive- and decision-making AI, and a house editor. The code to the last prototype was clearly a hacked version of the SimCity 2000 engine. I found a program that used genetic algorithms to procedurally generate SimCity-style buildings with a blind watchmaker style interface. That program had clearly been written as efficiently as possible, not from a run time point of view, but from an implementation standpoint. It was using the SimCity 2000 code base as a host organism for some rapid experimentation. Will's imagination had clearly been running faster than proper software engineering practice allowed. All of this, plus the awesome array of prototypes the Spore concept team had been cranking out, made a big impression on me. I joined in and contributed some of my own wacky prototypes to Spore.

What was going on here? What did all of this mean? Thinking back, I realized I had made two classic mistakes. First, my eyes were bigger than my stomach. The ambitious projects I had undertaken in the past "failed" because I made the mistake of not proving out the core ideas in prototypes. You can't send a rocket to the moon if you haven't first experimented with launching simple toy rockets. My sense is that the tribal civilization game died for similar reasons: Its author had launched into an ambitious finished project without doing the proper research, sketches, and prototypes.

Secondly, my success/failure evaluation function had been wrong. While the code to the The Sims prototypes wasn't in the final game, they had clearly informed the final product. All of my small "failures" were actually a series of small successes that had improved my design skills, and they were in fact studies I could incorporate into larger projects. I had it backward the whole time. My "successful," but incomplete, large projects were the real failures. I had invested too much energy into large projects that would fail because I hadn't done my homework. It's a hard lesson to take and one that most people probably have to learn the hard way.

So I dusted off my ACM programming competition skills, which taught me to make tightly focused programs in minimal time and with minimal frills. I became a better designer, really fast. I gained tons of design experience points by slaying so many gremlins.

I finished at Georgia Tech and joined Spore. My vast collection of tiny student projects, bite-sized personal projects, and work prototypes I had made added up to a huge amount of experience and intuition. Compared to my peers, I had a tremendous amount of design experience, simply from writing, evaluating, and throwing away so many ideas. I witnessed good prototypes move mountains. I like to think of good prototypers as powerful ninjas who can drop into hard design challenges, or tedious design debates, and cut them to shreds with one swift movement of their prototyping blade.

Here are a few prototyping rules of thumb. Even with years of experience, I often find a prototype going nowhere and can usually trace the problem to not following one of these rules:

- **Always Ask a Question.** Always ask a question, which will give you purpose, and have a hypothesis, which is a specific idea you are testing out. For example, you might be thinking about mouse-based control schemes for a school of fish. Your question is: How do I control these fish with a mouse? A hypothesis might be: Flocking will make the fish move together, and every mouse click will drop an invisible "bomb" that will act as a repulser upon every fish's steering AI, and it will take a few seconds to complete exploding. A good way to make sure you aren't going to waste time implementing ideas you don't actually have, which happens to me more often than I'd like, is to diagram the idea on paper first and work out as many details with a pen as possible. This also speeds up writing the prototype.
- **Stay Falsifiable.** Just like good science, you must validate the results of your experiment. Did your hypothesis work? Does your fish flock control scheme feel good to you? Do your friends find that it feels good? Does it work in the context of your game idea? You can never user test and playtest an idea too early. I have seen many cool ideas go down in flames because its owner was overprotective, didn't think it was ready, didn't believe the feedback they were getting, explained away people's responses, or thought that only their opinion mattered. Eventually users will play with your work, and by then it will be much harder to fix the design. Incorporate the user into the design process as early as possible. Be honest with yourself and your players, and you will be richly rewarded. This one is easy for me because as a designer, my main intent is to entertain and transform other people, so I'm always interested in what effect my work has on others. Watching people use what you make will also make you a smarter designer.
- **Persuade and Inspire.** We're making entertainment and art—your prototype should be cool, fun, and excite people. If you and your peers are compelled, your players will be too. On the flip side,

if something isn't resonating with other people, perhaps your idea or approach should be reconsidered. Prototypes can be powerful persuasive devices. Keita Takahashi, the designer of Katamari Damacy, couldn't convince anyone that rolling around a giant sticky ball would be fun. Until they played the prototype.

- **Work Fast.** Try to minimize time to your first "failure" (rejecting a hypothesis), and don't be afraid to push the eject button. A classic error is to spend months working on an engine, architecture, or something else that has nothing to do with proving out your core design idea. Prototypes don't need engines. Prototypes are slipshod machines held together by bubble gum and leftover bits of wire that test and prove simple ideas as quickly as possible. If you find yourself weeks or months into a project with only an engine, you've failed. Perhaps you need to articulate a specific gameplay idea to validate. For me, the ideal window of time to start and finish a prototype (including design, implementation, testing, and iteration) is two days to two weeks. Anything longer than that sets off alarm bells.
- **Work Economically.** You're making something small and beautiful, so invest development effort wisely. To work fast, you must stay small: Don't do too much at once or you'll never make progress. Be realistic. Here are some questions to ask yourself when you are considering how much effort to spend on proper engineering, art, interface design, or any aspect of your prototype. What's the purpose of this prototype? Who will use it? What's important? Look? Kinesthetics? Load time? Run time? Usability? Persuading your peers? Be a cheap, lazy, slothful programming bum. Just make it work so you can test your idea. Don't go above and beyond the call of duty in programming, art, or any other aspect of your prototype.
- **Carefully decompose problems.** Don't bite off more than you have to at once. If you prototype all systems simultaneously you will fail because you can't work fast or reach any kind of conclusion. To build it all at once is to build the actual game, which is hard. The prototype designer's job, like a good Go player, is to cut and separate the enemy stones (your design problem) into small, weak groups that can be killed or manipulated at will. Wisely divide your problem into manageable pieces. You must be careful because problems are sometimes connected in nonobvious ways and bite you later. Through practice, your designer's intuition and experience will help you see the connected nature of the problem you are trying to subdivide and make the most judicious cuts.

PROTOTYPING FOR GAME FEEL

by Steve Swink

Steve Swink is a game designer, teacher, and unicycle enthusiast. Having done stints at Neversoft and the now defunct Tremor Entertainment, he now designs for Flashbang Studios, a small development studio in Tempe, Arizona. He is also an instructor in Game and Level Design at the Art Institute of Phoenix, and he is currently writing a book entitled Game Feel: a Game Design Guide to Virtual Sensation *to be published by Morgan Kaufmann in summer 2008.*

What is good game feel? Among other things, it might mean that the feel of controlling the game is intrinsically pleasurable. The feel of Super Mario 64 fills me with thoughtless joy, enhancing every aspect of the game. From the first few seconds, I'm hooked, sold, ready to spend endless hours discovering the endless challenges and permutations implied by this tantalizing motion. Every interaction I have with the game will have this base, tactile, kinesthetic pleasure. How was this sensation designed? What's behind the curtain? Wherein does the "magic" of game feel lie?

The problems of game feel quickly become intertwined with the problems of the design as a whole, but it is possible to separate out the relevant components of game feel to make them a bit more manageable:

- *Input*: How the player can express their intent to the system
- *Response*: How the system processes, modifies, and responds to player input in real time
- *Context*: How constraints give spatial meaning to motion
- *Polish*: The impression of physicality created by layering of reactive motion, proactive motion, sounds, and effects, and the synergy between those layers
- *Metaphor*: The ingredient that lends emotional meaning to motion and that provides familiarity to mitigate learning frustration
- *Rules*: Application and tweaking of arbitrary variables that give additional challenge and higher-level meaning to constrained motions

Note: In the interest of brevity, the following discussion focuses on input, response, and context. The concepts of polish, metaphor, and rules are equally important.

Input

Input is the player's organ of expression in the game world, and the potential for expression is deeply affected by the physical properties of the input device. Consider the difference between a button and a computer mouse. A typical button has two states: on or off. It can be in one of two positions, which is about the minimum you can get sensitivity-wise. A mouse, on the other hand, has complete freedom of movement along two axes. It is unbounded; you can move it as far as the surface underneath allows. So an input device can have an inherent amount of sensitivity, what I call "input sensitivity."

An input device can also provide opportunities for natural mappings. That is, what kinds of motion are implied by the constraints, motions, and sensitivity of the input device? My favorite example is Geometry Wars for Xbox 360. Look at Geometry Wars, and then look at the Xbox 360 controller. Notice the way that the joystick is formed and how that transposes almost exactly to the motion in Geometry Wars. It's almost one for one: The joystick sits in a circular plastic housing that constrains its motion in a circular way. That means that pushing the control stick against the edge of the plastic rim that contains it and rolling it back and forth creates these little circles, which is almost exactly the motion that gets produced on-screen by Geometry Wars. This is what Donald Norman would refer to as a "natural mapping." There's no explanation or instruction needed because the position and motion of the input device correlates exactly to the position and motion of the thing being controlled in the game.

The controls of Mario 64 have this property; the rotation of the thumbstick correlates very closely to the rotation of Mario as he turns, twists, and abruptly changes direction.

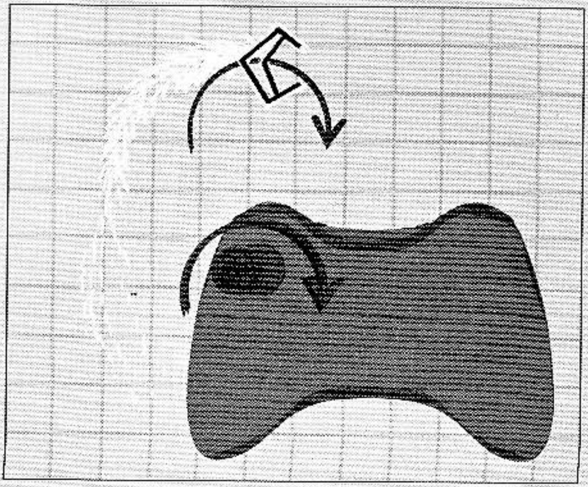


Figure 1 Natural mapping

The overall implication for game feel prototyping is to consider the overall sensitivity of your system and add or remove sensitivity to get a feel that is sufficiently, but not overly, expressive. The sweet spot is difficult to pin down, but it can be achieved with a high or low sensitivity input device, depending on how the system responds to a given input.

Response

A very simple input device with very little sensitivity can, by virtue of a nuanced reaction from the game, be part of a very sensitive control system. I call this "reaction sensitivity": sensitivity created by mapping user input to game reaction.

The NES controller was just a collection of buttons, but Mario had great sensitivity across time, across combinations of buttons, and across states. Across time, Mario sped up gradually from rest to his maximum speed and slowed gradually back down again, more commonly known as dampening. In addition, holding down the jump button longer meant a higher jump, another kind of sensitivity across time. Holding down the jump and left directional pad buttons simultaneously resulted in a jump that flowed to the left, providing greater sensitivity by allowing combinations of buttons to have different meanings from pressing those buttons individually. Finally, Mario had different states. That is, pressing left while "on the ground" has a different meaning than pressing left while "in the air." These are contrived distinctions that are designed into the game but that lend greater sensitivity to the system as a whole so long as the player can correctly interpret when the state switch has occurred and respond accordingly.

The result of all these kinds of nuanced reactions to input was a highly fluid motion, especially as compared to a game such as Donkey Kong, in which there was no such sensitivity.

The comparison in Figure 2, between Super Mario Brothers and Donkey Kong, shows very clearly just how much more expressive and fluid Mario's controls are. The interesting thing to note is that Donkey Kong used a joystick, a much more sensitive input than the NES controller. No matter how simple the input, the reaction from a system can always be highly sensitive.

Context

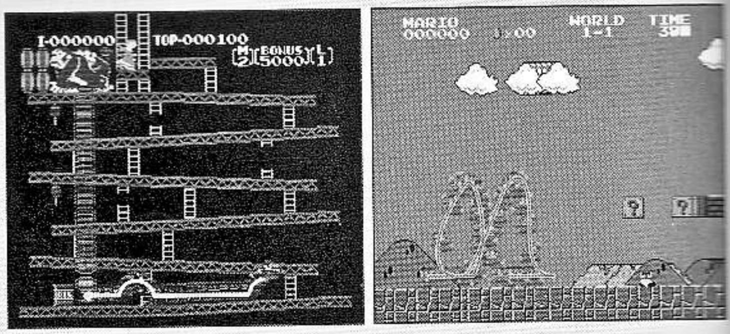


Figure 2 Comparison of character motion in Donkey Kong and Super Mario Bros.

Returning to Mario 64, imagine Mario standing in a field of blank whiteness, with no objects around him. With nothing but a field of blankness, does it matter that Mario can do a long jump, a triple jump, or a wall kick?

If Mario has nothing to interact with, the fact that he has these acrobatic abilities is meaningless. Without a wall, there can be no wall kick. At the most pragmatic level, the placement of objects in the world is just another set of variables against which to balance movement speed, jump height, and the other parameters that define motion. In game feel terms, constraints define sensation. If objects are packed in, spaced tightly relative to the avatar's motion, the game will feel clumsy and oppressive, causing anxiety and frustration. As objects get spaced farther apart, the feel becomes increasingly trivialized, making tuning unimportant and numbing thoughtless joy into thoughtless boredom.

Concurrently to the implementation of your system, you should be developing some kind of spatial context for your motion. You should put in some kind of platforms, enemies, some kind of topology that will give the motion meaning. If Mario is running along with an endless field of blank whiteness beneath him, it will be very difficult to judge how high he should be able to jump. So you need to start putting platforms in there to get a sense of what it will be like to traverse a populated level.

Constraint is also the mother of skill and challenge. Think of a football field: There are these arbitrary constraints around the sides of the football field that limit it to a certain size. If those constraints weren't there, the game of football would have a very different skill set and would arguably be a lot less interesting because you could run as far as you want in one direction before bringing the football back. The skills of football are defined by the constraints that bound it.

Conclusion

There is an aesthetic beauty possible with game feel. That is, something beautiful is created at the intersection of player and game. The act of play can create something aesthetically beautiful, aurally, visually, and/or tactilely.

Before you dive in and start coding, consider the overall sensitivity of the system, the affordances of the input device, and the sensitivity of the response from the game. Concurrently, develop some kind of spatial context for your motion. The idea is to create a "possibility space" that will, through tweaking the variables you've exposed, give rise to the game feel you want, the thoughtless joy that will hook players, engage them, and keep them playing.