EMERGENCE

THE CONNECTED LIVES OF
ANTS, BRAINS, CITIES AND SOFTWARE

STEVEN JOHNSON

PENGUIN BOOKS

ABOUT THE AUTHOR

Steven Johnson has written for *The New York Times*, *Lingua Franca*, the *Wall Street Journal* and the *Guardian*. He co-founded the award-winning Online magazine *FEED* and is author of the book *Interface Culture*.

5

Control Artist

On the screen, the pixels dance: bright red dots with faint trails of green, scurrying across a black background, like fireflies set against the sky of a summer night. For a few seconds, the movement on-screen looks utterly random: pixels darting back and forth, colliding, and moving on. And then suddenly a small pocket of red dots gather together in a pulsing, erratic circle, ringed by a strip of green. The circle grows as more red pixels collide with it; the green belt expands. Seconds later, another lopsided circle appears in the corner of the screen, followed by three more. The circles are unlike any geometric shape you've ever seen. They seem more like a life-form—a digital blob—pulsing haphazardly, swelling and contracting. Two blobs slowly creep toward each other, then merge, forming a single unit. After a few minutes, seven large blobs dominate, with only a few remaining free-floating red pixels ambling across the screen.

Welcome to the world of Mitch Resnick's tool for visualizing self-organizing systems, StarLogo. A descendant of Seymour Papert's legendary turtle-based programming language, Logo, StarLogo allows you to model emergent behavior using simple, English-like commands—and it displays that behavior in vivid, real-time animations. If decentralized systems can sometimes seem counterintuitive or abstract, difficult to describe in words, StarLogo makes them come to life with dynamic graphics that are uniquely suited for the Nintendo generation. If a calendar is a tool for helping us think about the flow of time, and a pie chart is a tool for thinking about statistical distributions, StarLogo is a tool for thinking about bottom-up systems. And, in fact, those lifelike blobs on the screen take us back to the very beginnings of our story: they are digital slime molds, cells aggregating into larger clusters without any "pacemaker" cell leading the way.

"Those red pixels are the individual slime mold cells," Resnick says, pointing at the screen, sitting in his Cambridge office. "They're programmed to wander aimlessly around the screen space, and as they wander, they 'emit' the green color, which quickly fades away. That color is the equivalent of the c-AMP chemical that the molds use to coordinate their behavior. I've programmed the red cells to 'sniff' the green color and follow the gradient in the color. 'Smelling' the green pixels leads the cells toward each other."

Like Gordon's ant colonies, Resnick's slime mold simulation is sensitive to population density. "Let's start with only a hundred slime mold cells," he says, adjusting a slider on the screen that alters the number of cells in the simulation. He presses a start button, and a hundred red pixels begin their frenetic dance—only this time, no clusters appear. There are momentary flashes of green as a few cells collide, but no larger shapes emerge at all.

"With a hundred cells, there isn't enough contact for the aggregates to form. But triple the population like so," he says, pulling the

slider farther to the right, "and you increase the contact between cells. At three hundred cells, you'll usually get one cluster after a few minutes, and sometimes two." We wait for thirty seconds or so, and after a few false starts, a cluster takes shape near the center of the screen. "Once they come together, the slime molds are extremely difficult to break apart, even though they can be very fickle about aggregating in the first place."

Resnick then triples the population and starts the simulation over again. It's a completely different system this time around: there's a flash of red-celled activity, then almost immediately ten clusters form, nearly filling the screen with pulsing watermelon shapes. Only a handful of lonely red cells remain, drifting aimlessly between the clusters. More is *very* different. "The interesting thing is," Resnick says with a chuckle, "you wouldn't have necessarily predicted that behavior in advance, just from looking at the instructions. You might have said, the slime mold cells will all immediately form a giant cluster, or they'll form clusters that keep breaking up. In fact, neither is the case, and the whole system turns out to be much more sensitive to initial conditions. At a hundred cells, there are no clusters at all; at three hundred, you'll probably get one, but it'll be pretty much permanent; and at nine hundred cells, you'll immediately get ten clusters, but they'll bounce around a little more." But you couldn't tell any of that just by looking at the original instruction set. You have to make it *live* before you can understand how it works.

StarLogo may look like a video game at first glance, but Resnick's work is really more in the tradition of Friedrich Froebel, the German educator who invented kindergarten, and who spent much of his career in the early nineteenth century devising ingenious toys that would both amuse and entertain toddlers. "When Froebel

designed the first kindergarten," Resnick tells me, "he developed a set of toys they called Froebel's gifts, and he carefully designed them with the assumption that the object he'd put in the hands of kids would make a big difference in what they learned and how they learned. We see the same thing carried through today. We see some of our new technology as the latter-day versions of Froebel's gifts, trying to put new sorts of materials and new types of toys in the hands of kids that will change what they think about—and the *way* they think about it."

StarLogo, of course, is designed to help kids—and grown-ups, for that matter—think about a specific type of phenomenon, but it is by no means limited to slime molds. There are StarLogo programs that simulate ant foraging, forest fires, epidemics, traffic jams—even programs that generate more traditional Euclidean shapes using bottom-up techniques. (Resnick calls this "turtle geometry," after the nickname used to describe the individual agents in a StarLogo program, a term that is itself borrowed from the original Logo language, which Papert designed to teach children about traditional programming techniques.) This knack for shape-shifting is one of the language's great virtues. "StarLogo is a type of modeling environment where kids can build models of certain phenomena that they might observe in the world," Resnick says. "Specifically, it enables them to build models of phenomena where lots of things interact with each other. So they might model cars on a highway, or they might model something like a bird flock, where the kids design behavior for lots of individual birds and then see the patterns that form through all the interactions.

"One reason that we're especially interested in building a tool like this is that these phenomena are common in the everyday world," he continues. "We see bird flocks and traffic jams all of the time. On the other hand, people have a lot of trouble understanding these types of phenomena. When people see a flock of birds,

they assume the bird in the front is the leader and the others are just following. But that's not the way the real birds form flocks. In fact, each bird just follows simple rules and they end up together as a group."

At its core, StarLogo is optimized for modeling emergent systems like the ones we've seen in the previous chapters, and so the building blocks for any StarLogo program are familiar ones: local interactions between large numbers of agents, governed by simple rules of mutual feedback. StarLogo is a kind of thinking prosthetic, a tool that lets the mind wrap itself around a concept that it's not naturally equipped to grasp. We need StarLogo to help us understand emergent behavior for the same reason we need X-ray machines or calculators: our perceptual and cognitive faculties can't do the work on their own.

It's a limitation that can be surprisingly hard to overcome. Consider the story that Resnick tells of artificial-intelligence guru Marvin Minsky encountering the slime mold simulation for the first time. "One day shortly after I developed the first working prototype of StarLogo, Minsky wandered into my office. On the computer screen he saw an early version of my StarLogo slime mold program. There were several green blobs on the screen (representing a chemical pheromone), with a cluster of turtles moving around inside each blob. A few turtles wandered randomly in the empty space between the blobs. Whenever one of these turtles wandered close enough to a blob, he joined the cluster of turtles inside."

Minsky scanned the screen for a few seconds, then asked Resnick what he was working on. "I explained that I was experimenting with some self-organizing systems. Minsky looked at the screen for a while, then said, 'But those creatures aren't self-organizing. They're just moving toward the green food.'"

"Minsky had assumed that the green blobs were pieces of food, placed throughout the turtles' world. In fact, the green blobs were

created by the turtles themselves. But Minsky didn't see it that way. Instead of seeing creatures organizing themselves, he saw the creatures organized around some preexisting pieces of food. He assumed that the pattern of aggregation was determined by the placement of food. And he stuck with that interpretation—at least for a while—even after I told him that the program involved self-organization."

Minsky had fallen for the myth of the ant queen: the assumption that collective behavior implied some kind of centralized authority—in this case, that the food was dictating the behavior of the slime mold cells. Minsky assumed that you could predict where the clusters would form by looking at where the food was placed when the simulation began. But there wasn't any food. Nor was there anything dictating that clusters should form in specific locations. The slime mold cells were self-organizing, albeit within parameters that Resnick had initially defined.

"Minsky has thought more—and more deeply—about self-organization and decentralized systems than almost anyone else," Resnick writes. "When I explained the rules underlying the slime mold program to him, he understood immediately what was happening. But his initial assumption was revealing. The fact that even *Marvin Minsky* had this reaction is an indication of the powerful attraction of centralized explanations."

Of course, on the most fundamental level, StarLogo is itself a centralized system: it obeys rules laid down by a single authority—the programmer. But the route from Resnick's code to those slime mold clusters is indirect. You don't program the slime mold cells to form clusters; you program them to follow patterns in the trails left behind by their neighbors. If you have enough cells, and if the trails last long enough, you'll get clusters, but they're not something you can control directly. And predicting the number of clusters—or their longevity—is almost impossible without extensive trial-and-error experimentation with the system. Kevin Kelly called his

groundbreaking book on decentralized behavior *Out of Control,* but the phrase doesn't quite do justice to emergent systems—or at least the ones that we've deliberately set out to create on the computer screen. Systems like StarLogo are not utter anarchies: they obey rules that we define in advance, but those rules only govern the micromotives. The macrobehavior is another matter. You don't control that directly. All you can do is set up the conditions that you think will make that behavior possible. Then you press play and see what happens.

That kind of oblique control is a funny thing to encounter in the world of software, but it is becoming increasingly common. Programming used to be thought of as a domain of pure control: you told the computer what to do, and the computer had no choice but to obey your orders. If the computer failed to do your bidding, it inevitably had to do with a bug in your code, and not the machine's autonomy. The best programmers were the ones who had the most control of the system, the ones who could compel the machines to do the work with the least amount of code. It's no accident that Norbert Wiener derived the term *cybernetics* from the Greek word for "steersman": the art of software has from the beginning been about control systems and how best to drive them.

But that control paradigm is slowly giving way to a more oblique form of programming: software that you "grow" instead of engineer, software that learns to solve problems autonomously, the way Oliver Selfridge envisioned with his Pandemonium model. The new paradigm borrows heavily from the playbook of natural selection, breeding new programs out of a varied gene pool. The first few decades of software were essentially creationist in philosophy—an almighty power wills the program into being. But the next generation is profoundly Darwinian.

* * *

Consider the program for number sorting devised several years ago by supercomputing legend Danny Hillis, a program that undermines all of our conventional assumptions about how software should be produced. For years, number sorting has served as one of the benchmark tests for ingenious programmers, like chess-playing applications. Throw a hundred random numbers at a program and see how many steps it takes to sort the digits into the correct order. Using traditional programming techniques, the record for number sorting stood at sixty steps when Hillis decided to try his hand. But Hillis didn't just sit down to write a number-sorting application. What Hillis created was a recipe for learning, a program for creating another program. In other words, he didn't teach the computer how to sort numbers. He taught the computer to figure out how to sort numbers *on its own.*

Hillis pulled off this sleight of hand by connecting the formidable powers of natural selection to a massively parallel supercomputer—the Connection Machine that he himself had helped design. Instead of authoring a number-sorting program himself—writing out lines of code and debugging—Hillis instructed the computer to generate thousands of miniprograms, each composed of random combinations of instructions, creating a kind of digital gene pool. Each program was confronted with a disorderly sequence of numbers, and each tried its hand at putting them in the correct order. The first batch of programs were, as you might imagine, utterly inept at number sorting. (In fact, the overwhelming majority of the programs were good for nothing at all.) But some programs were better than others, and because Hillis had established a quantifiable goal for the experiment—numbers arranged in the correct order—the computer could select the few programs that were in the ballpark. Those programs became the basis for the next iteration, only Hillis would also mutate *their* code slightly and crossbreed them with the other promising programs. And then the whole process

would repeat itself: the most successful programs of the new generation would be chosen, then subjected to the same transformations. Mix, mutate, evaluate, repeat.

After only a few minutes—and thousands of cycles—this evolutionary process resulted in a powerful number-sorting program, capable of arranging a string of random numbers in seventy-five steps. Not a record breaker, by any means, but impressive nonetheless. The problem, though, was that the digital gene pool was maxing out at the seventy-five-step mark. Each time Hillis ran the sequence, the computer would quickly evolve a powerful and efficient number sorter, but it would run out of steam at around seventy-five steps. After enough experimentation, Hillis recognized that his system had encountered a hurdle often discussed by evolutionary theorists: the software had stumbled across a local maximum in the fitness landscape.

Imagine the space of all possible number-sorting programs spread out like a physical landscape, with more successful programs residing at higher elevations, and less successful programs lurking in the valleys. Evolutionary software is a way of blindly probing that space, looking for gradients that lead to higher elevations. Think of an early stage in Hillis's cycle: one evolved routine sorts a few steps faster than its "parent" and so it survives into the next round. That survival is the equivalent of climbing up one notch on the fitness landscape. If its "descendant" sorts even more efficiently, its "genes" are passed on to the next generation, and it climbs another notch higher.

The problem with this approach is that there are false peaks in the fitness landscape. There are countless ways to program a computer to sort numbers with tolerable efficiency, but only a few ways to sort numbers if you're intent on setting a world record. And those different programs vary dramatically in the way they tackle the problem. Think of those different approaches as peaks on the

fitness landscape: there are thousands of small ridges, but only a few isolated Everests. If a program evolves using one approach, its descendants may never find their way to another approach—because Hillis's system only rewarded generations that *improved* on the work done by their ancestors. Once the software climbs all the way to the top of a ridge, there's no reward in descending and looking for another, higher peak, because a less successful program—one that drops down a notch on the fitness landscape—would instantly be eliminated from the gene pool. Hillis's software was settling in at the seventy-five-step ridges because the penalty for searching out the higher points was too severe.

Hillis's stroke of genius was to force his miniprograms out of the ridges by introducing predators into the mix. Just as in real-world ecosystems, predators effectively raised the bar for evolved programs that became lazy because of their success. Before the introduction of predators, a miniprogram that had reached a seventy-five-step ridge knew that its offspring had a chance of surviving if it stayed at that local maximum, but faced almost certain death if it descended to search out higher ground. But the predators changed all that. They hunted down ridge dwellers and forced them to improvise: if a miniprogram settled into the seventy-five-step range, it could be destroyed by predator programs. Once the predators appeared on the scene, it became more productive to descend to lower altitudes to search out a new peak than to stay put at a local maximum.

Hillis structured the predator-prey relationship as an arms race: the higher the sorting programs climbed, the more challenging the predators became. If the system stumbled across a seventy-step peak, then predators were introduced that hunted down seventy-step programs. Anytime the software climbers decided to rest on their laurels, a predator appeared to scatter them off to find higher elevations.

After only thirty minutes of this new system, the computer had

evolved a batch of programs that could sort numbers in sixty-two steps, just two shy of the all-time record. Hillis's system functioned, in biological terms, more like an environment than an organism: it created a space where intelligent programs could grow and adapt, exceeding the capacities of all but the most brilliant flesh-and-blood programmers. "One of the interesting things about the sorting programs that evolved in my experiment is that I do not understand how they work," Hillis writes in his book *The Pattern on the Stone.* "I have carefully examined their instruction sequences, but I do not understand them: I have no simpler explanation of how the programs work than the instruction sequences themselves. It may be that the programs are not understandable."

Proponents of emergent software have made some ambitious claims for their field, including scenarios where a kind of digital Darwinism leads to a simulated intelligence, capable of open-ended learning and complex interaction with the outside world. (Most advocates don't think that such an intelligence will necessarily resemble *human* smarts, but that's another matter, one that we'll examine in the conclusion.) In the short term, though, emergent software promises to transform the way that we think about creating code: in the next decade, we may well see a shift from top-down, designed programs to bottom-up, evolved versions, like Hillis's number-sorting applet—"less like engineering a machine," Hillis says, "than baking a cake, or growing a garden."

That transformation may be revolutionary for the programmers, but if it does its job, it won't necessarily make much of a difference for the end users. We might notice our spreadsheets recalculating a little faster and our grammar checker finally working, but we'll be dealing with the end results of emergent software, not the process itself. (The organisms, in Darwinian terms, and not the environment that nurtured them.) But will ordinary computer-users get a chance to experiment with emergent software firsthand, a chance

to experiment with its more oblique control systems? Will growing gardens of code ever become a mainstream activity?

In fact, we can get our hands dirty already. And we can do it just by playing a game.

It's probably fair to say that digital media has been wrestling with "control issues" from its very origins. The question of control, after all, lies at the heart of the interactive revolution, since making something interactive entails a shift in control, from the technology—or the puppeteers behind the technology—to the user. Most recurring issues in interactive design hover above the same underlying question: Who's driving here, human or machine? Programmer or user? These may seem like esoteric questions, but they have implications that extend far beyond design-theory seminars or cybercafé philosophizing. I suspect that we're only now beginning to understand how complicated these issues are, as we acclimate to the strange indirection of emergent software.

In a way, we've been getting our sea legs for this new environment for the past few years now. Some of the most interesting interactive art and games of the late nineties explicitly challenged our sense of control or made us work to establish it. Some of these designs belonged to the world of avant-garde or academic experimentation, while others had more mainstream appeal. But in all these designs, the feeling of wrestling with or exploring the possibilities of the software—the process of mastering the system—was transformed from a kind of prelude to the core experience of the design. It went from a bug to a feature.

There are different ways to go about challenging our sense of control. Some programs, such as the ingenious Tap, Type, Write—created by MIT's John Maeda—make it immediately clear that the user is driving. The screen starts off with an array of letters; hitting

a specific key triggers a sudden shift in the letterforms presented on-screen. The overall effect is like a fireworks show sponsored by Alphabet Soup. Press a key, and the screen explodes, ripples, reorders itself. It's hypnotic, but also a little mystifying. What algorithm governs this interaction? Something happens on-screen when you type, but it takes a while to figure out what rules of transformation are at work here. You know you're doing something, you just don't know what it is.

The OSS code, created by the European avant-punk group Jodi.org, messes with our sense of control on a more profound—some would say annoying—level. A mix of anarchic screen-test patterns and eclectic viral programming, the Jodi software is best described as the digital equivalent of an aneurysm. Download the software and the desktop overflows with meaningless digits; launch one of the applications, and your screen descends instantly into an unstable mix of static and structure. Move the mouse in one direction, or double click, and there's a fleeting sense of something changing. Did the flicker rate shift? Did those interlaced patterns reverse themselves? At hard-to-predict moments, the whole picture show shuts down—invariably after a few frantic keystrokes and command clicks—and you're left wondering, Did I do that?

No doubt many users are put off by the dislocations of Tap, Type, Write and OSS, and many walk away from the programs feeling as though they never got them to work quite right, precisely because their sense of control remained so elusive. For me, I find these programs strangely empowering; they challenge the mind in the same way distortion challenged the ear thirty-five years ago when the Beatles and the Velvet Underground first began overloading their amps. We find ourselves reaching around the noise—the lack of structure—for some sort of clarity, only to realize that it's the reaching that makes the noise redemptive. Video games remind us that messing with our control expectations can be fun,

even addictive, as long as the audience has recognized that the confusion is part of the show. For a generation raised on MTV's degraded images, that recognition comes easily. The Nintendo generation, in other words, has been well prepared for the mediated control of emergent software.

Take as example one of the most successful titles from the Nintendo64 platform, Shigeru Miyamoto's Zelda: Ocarina of Time. Zelda embodies the uneven development of late-nineties interactive entertainment. The plot belongs squarely to the archaic world of fairy tales—a young boy armed with magic spells sets off to rescue the princess. As a control system, though, Zelda is an incredibly complex structure, with hundreds of interrelated goals and puzzles dispersed throughout the game's massive virtual world. Moving your character around is simple enough, but figuring out what you're supposed to do with him takes hours of exploration and trial and error. By traditional usability standards, Zelda is a complete mess: you need a hundred-page guidebook just to establish what the rules are. But if you see that opacity as part of the art—like John Cale's distorted viola—then the whole experience changes: you're exploring the world of the game and the rules of the game at the same time.

Think about the ten-year-olds who willingly immerse themselves in Zelda's world. For them, the struggle for mastery over the system doesn't feel like a struggle. They've been decoding the landscape on the screen—guessing at causal relations between actions and results, building working hypotheses about the system's underlying rules—since before they learned how to read. The conventional wisdom about these kids is that they're more nimble at puzzle solving and more manually dexterous than the TV generation, and while there's certainly some truth to that, I think we lose something important in stressing how talented this generation is with their joysticks. I think they have developed another skill, one

that almost looks like patience: they are more tolerant of being out of control, more tolerant of that exploratory phase where the rules don't all make sense, and where few goals have been clearly defined. In other words, they are uniquely equipped to embrace the more oblique control system of emergent software. The hard work of tomorrow's interactive design will be exploring the tolerance—that suspension of control—in ways that enlighten us, in ways that move beyond the insulting residue of princesses and magic spells.

With these new types of games, a new type of game designer has arisen as well. The first generation of video games may have indirectly influenced a generation of artists, and a handful were adopted as genuine objets d'art, albeit in a distinctly campy fashion. (Tabletop Ms. Pac-Man games started to appear at downtown Manhattan clubs in the early nineties, around the time the Museum of the Moving Image created its permanent game collection.) But artists themselves rarely ventured directly into the game-design industry. Games were for kids, after all. No self-respecting artist would immerse himself in that world with a straight face.

But all this has changed in recent years, and a new kind of hybrid has appeared—a fusion of artist, programmer, and complexity theorist—creating interactive projects that challenge the mind and the thumb at the same time. And while Tap, Type, Write and Zelda were not, strictly speaking, emergent systems, the new generation of game designers and artists have begun explicitly describing their work using the language of self-organization. This too brings to mind the historical trajectory of the rock music genre. For the first fifteen or twenty years, the charts are dominated by lowest-common-denominator titles, rarely venturing far from the established conventions or addressing issues that would be beyond the reach of a thirteen-year-old. And then a few mainstream acts begin

to push at the edges—the Beatles or the Stones in the music world, Miyamoto and Peter Molyneux in the gaming community—and the expectations about what constitutes a pop song or a video game start to change. And that transformation catches the attention of the avant-garde—the Velvet Underground, say, or the emergent-game designers—who suddenly start thinking of pop music or video games as a legitimate channel for self-expression. Instead of writing beat poetry or staging art happenings, they pick up a guitar—or a joystick.

By this standard, Eric Zimmerman is the Lou Reed of the new gaming culture. A stocky thirty-year-old, with short, club-kid hair and oversize Buddy Holly glasses, Zimmerman has carved out a career for himself that would have been unthinkable even a decade ago: bouncing between academia (he teaches at NYU's influential Interactive Telecommunications Program), the international art scene (he's done installations for museums in Geneva, Amsterdam, and New York), and the video-game world. Unlike John Maeda and or Jodi.org, Zimmerman doesn't "reference" the iconography of gaming in his work—he openly embraces that tradition, to the extent that you have to think of Zimmerman's projects as games first and art second. They can be fiendishly fun to play and usually involve spirited competition between players. But they are also self-consciously designed as emergent systems.

"One of the pleasures of what I do," Zimmerman tells me, over coffee near the NYU campus, "is that you get to see a player take what you've designed and use it in completely unexpected ways." The designer, in other words, controls the micromotives of the player's actions. But the way those micromotives are exploited—and the macrobehavior that they generate—are out of the designer's control. They have a life of their own.

Take Zimmerman's game Gearheads, which he designed during a brief sojourn at Phillips Interactive in 1996. Gearheads is a pure-

bred emergent system: a meshwork of autonomous agents following simple rules and mutually influencing each other's behavior. It is a close relative of StarLogo or Gordon's harvester ants, but it's ingeniously dressed up to look like a modern video game. Instead of spare colored pixels, Zimmerman populated the Gearhead world with an eclectic assortment of children's toys that march across the screen like a motley band of animated soldiers.

"There are twelve windup toys," Zimmerman explains. "You design a box of toys by choosing four of them. You wind up your toy and release it from the edges of the game board, and the goal of the game is to get as many toys as possible across your opponent's side of the screen. Each of the toys has a unique set of behaviors that affect the behavior of other toys." A skull toy, for instance, "frightens" toys that it encounters, causing them to reverse direction, while an animated hand winds up other toys, allowing them to march across the screen for a longer duration. As with the harvester ants or the slime mold cells, when one agent encounters another agent, both agents may launch into a new pattern of behavior. Stumble across your hundredth forager of the afternoon, and you'll switch over to midden duty; stumble across Zimmerman's skull toy and you'll turn around and go the other way.

"The key thing is that once you've released your toys, they're autonomous. You're only affecting the system from the margins," Zimmerman says. "It's a little chaos machine: unexpected things happen, and you only control it from the edges." As Zimmerman tested Gearheads in early 1996, he found that this oblique control system resulted in behavior that Zimmerman hadn't deliberately programmed, behavior that emerged out of the local interactions of the toys, despite the overall simplicity of the game.

"Two toys reverse the direction of other toys—the skull, and the Santa toy, who's called Krush Kringle," Zimmerman says. "He walks for a few steps and then he pounds the ground, and all the toys near

him reverse direction. During our testing, we found a combination where you could release one Krush Kringle out there, then the walking hand that winds up toys, then another Krush Kringle. The hand would run out and wind up the first Krush, and then the Krush would pound the floor, reversing the direction of the hand, and sending it back to the second Krush, which it would wind up. Then the second Krush would stomp on the ground, and the hand would turn around and wind up the first Krush. And so the little system of these three toys would march together across the screen, like a small flock of birds. The first time we saw it happen, we were astonished."

These unexpected behaviors may not seem like much at first glance, particularly in a climate that places so much emphasis on photo-realistic, 3-D worlds and blood-spattering combat. Zimmerman's toys are kept deliberately simple; they don't simulate intelligence, and they don't trigger symphonies of surround sound through your computer speakers. A snapshot of Resnick's slime molds looks like something you might have seen on a first-generation Atari console. But I'll put my money on the slime molds and Krush Kringles nonetheless. Those watermelon clusters and autowinding flocks strike me as the very beginning of what will someday form an enormously powerful cultural lineage. Watching these patterns emerge spontaneously on the screen is a little like watching two single-celled organisms decide to share resources for the first time. It doesn't look like much, but the same logic carried through a thousand generations, or a hundred thousand—like Hillis growing his gardens of code—can end up changing the world. You just have to think about it on the right scale.

Most game players, alas, live on something close to day-trader time, at least when they're in the middle of a game—thinking more about their next move than their next meal, and usually blissfully

oblivious to the ten- or twenty-year trajectory of software development. No one wants to play with a toy that's going to be fun after a few decades of tinkering—the toys have to be engaging *now,* or kids will find other toys. And one of the things that make all games so engaging to us is that they have rules. In traditional games like Monopoly or go or chess, the fun of the game—the play—is what happens when you explore the space of possibilities defined by the rules. Without rules, you have something closer to pure improv theater, where anything can happen at any time. Rules give games their structure, and without that structure, there's no game: every move is a checkmate, and every toss of the dice lands you on Park Place.

This emphasis on rules might seem like the antithesis of the open-ended, organic systems we've examined over the preceding chapters, but nothing could be further from the truth. Emergent systems too are rule-governed systems: their capacity for learning and growth and experimentation derives from their adherence to low-level rules: ants choosing to forage or not, based on patterns in their encounters with other ants; the Alexa software making connections based on patterns in the clickstream. If any of these systems—or, to put it more precisely, the agents that make up these systems—suddenly started following their own rules, or doing away with rules altogether, the system would stop working: there'd be no global intelligence, just a teeming anarchy of isolated agents, a swarm without logic. Emergent behaviors, like games, are all about living within the boundaries defined by rules, but also using that space to create something greater than the sum of its parts.

Understanding emergence should be a great boon for the videogame industry. But some serious challenges face the designers of games that attempt to harness the power and adaptability of self-organization and channel it into a game aimed at a mass audience. And those challenges all revolve around the same phenomenon: the

capacity of emergent systems to suddenly start behaving in unpredictable ways, sorcerer's-apprentice style—like Zimmerman's flock of Krush Kringles.

Consider the case of Evolva, a widely hyped game released in mid-2000 by a British software company called Computer Artworks. The product stood as something of a change for CA, which was last seen marketing a trippy screen-saver called Organic Art that allowed you to replace your desktop with a menagerie of alien-looking life-forms. That program came bundled with a set of prepackaged images, but more adventurous users could also grow their own, "breeding" new creatures with the company's A-Life technology. While the Organic Art series was a success, it quickly became clear to the CA team that *interacting* with your creatures would be much more entertaining than simply gazing at snapshots of them. Who wants to look at Polaroids of Sea-Monkeys when you can play with the adorable little critters yourself?

And so Computer Artworks turned itself into a video-game company. Evolva was their first fully interactive product to draw upon the original artificial-life software, integrating its mutation and interbreeding routines into a game world that might otherwise be mistaken for a hybrid of Myth and Quake. The plot was standard-issue video-game fare: Earth has been invaded by an alien parasite that threatens world destruction; as a last defense, the humans send out packs of fearless "genohunters" to save the planet. Users control teams of genohunters, occupying the point of view of one while issuing commands to the others. A product of biological engineering themselves, genohunters are capable of analyzing the DNA of any creature they kill and absorbing useful strands into their own genetic code. Once you've absorbed enough DNA, you can pop over to the "mutation" screen and tinker with your genetic makeup—adding new genes and mutating your existing ones, expanding your character's skills in the process. It's like suddenly

learning how to program in C++, only you have to eat the guy from tech support to see the benefits.

That appetite for DNA gives the A-Life software its entrée into the gameplay. "As the player advances through the game, new genes are collected and added to the available gene pool," lead programmer Rik Heywood explained to me in an e-mail conversation. "When the player wants to modify one of their creations, they can go to the mutation screen. Starting from the current set of DNA, two new generations can be created by combining the DNA from the existing genohunter with the DNA in the collected gene pool and some slight random mutations. The new sets of DNA are used to morph the skin, grow appendages all over the body, and develop new abilities, such as breathing fire or running faster."

The promotional material for Evolva makes a great deal of noise about this open-endedness. Some 14 billion distinct characters can be generated using the mutation screen, which means that unless Computer Artists strikes a licensing deal with other galaxies, players who venture several levels deep in the game will be playing with genetically unique genohunters. For the most part, those mutations result in relatively superficial external changes, more like a new paint job than an engine overhaul. The more sophisticated alterations to the genohunters' behavior—fire-breathing, laser-shooting, long-distance jumping, among others—are largely discrete skills programmed directly by the CA team. You won't see any genohunters spontaneously learning how to play the cello or use sonar. The bodies of your genohunters may end up looking dramatically different from where they started, but those bodies won't let their hosts adopt radically new skills.

These limitations may well make the game more enjoyable. For a sixteen-year-old Quake player who's just trying to kill as many parasites as possible on his way to the next level, suddenly learning how to read braille is only going to be a distraction. Anyone who

has spent time playing a puzzle-based narrative game like Myst knows nothing is more frustrating than spending two hours trying to solve a puzzle that you don't yet have the tools to solve, because you haven't stumbled across them in your explorations of the game space. Imagine how much more frustrating to get stumped by a puzzle because you haven't evolved gills or lock-picking skills yet. In a purely open-ended system—where the tools may or may not evolve depending on the whims of natural selection—that frustration would quickly override any gee-whiz appeal of growing your own characters. And so Heywood and his team have planted DNA for complex skills near puzzles or hurdles that require those skills. "For example, if we wanted to be sure that the player had developed the ability to breath fire by a particular point in the game," he explains, "we would block the path with some flammable plants and place some creatures with a fire-breathing ability nearby."

The blind watchmaker of Evolva's mutation engine turns out to have some sight after all. Heywood's solution might be the smartest short-term move for the gamers, but it's worth pointing out that it also runs headlong against the principles of Darwinism. Not only are you playing God by deliberately selecting certain traits over others, but the DNA for those traits is planted near the appropriate obstacles. It's like some strange twist on Lamarckian evolution: the giraffe neck grows longer each generation, but only because the genes for longer necks happen to sprout next to the banana trees. The space of possibility unleashed by an open-ended Darwinian engine was simply too large for the rule-space of the game itself. A game where anything can happen is by definition not a game.

Is there a way to reconcile the unpredictable creativity of emergence with the directed flow of gaming? The answer, I think, will turn out to be a resounding yes, but it's going to take some trial and error.

One way involves focusing on traditional emergent systems—such as flocks and clusters—and less on the more open-ended landscape of natural selection. Evolva is actually a great example of the virtues of this sort of approach. Behind the scenes, each creature in the Evolva world is endowed with sensory inputs and emotive states: fear, pain, aggression, and so on. Creatures also possess memories that link those feelings with other characters, places, or actions—and they are capable of sharing those associations with their comrades. As the web of associations becomes more complex, and more interconnected, new patterns of collective behavior can evolve, creating a lifelike range of potential interactions between creatures in the world.

"Say you encounter a lone creature," Heywood explains. "When you first meet it, it is maybe feeling very aggressive and runs in to attack your team. However, you have it outnumbered and start causing it some serious pain. Eventually fear will become the dominant emotion, causing the creature to run away. It runs around a corner and meets a large group of friends. It communicates with these other creatures, informing them of the last place it saw you. Being in a large group of friends brings its fear back down, and the whole group launches a new attack on the player." The *group* behavior can evolve in unpredictable ways, based on external events and each creature's emotional state, even if the virtual DNA of those creatures remains unchanged. There is something strangely comforting in this image, particularly for anyone who thinks social patterns influence our behavior as readily as our genes do. Heywood had to restrict the artificial-life engine because the powers of natural selection are too unpredictable for the rules-governed universe of a video game. But building an emergent system to simulate collective behavior among characters actually improved the gameplay, made it more lifelike without making it impossible. Emergence trumps "descent with modification": you may not be able to

use Evolva's mutation engine to grow wings, but your creatures can still learn new ways to flock.

There is a more radical solution to this problem, though, and it's most evident in the god-games genre. Classic games like SimCity—or 1999's best-selling semi-sequel The Sims, which lets game players interact with simulated personalities living in a small neighborhood—have dealt with the unpredictability of emergent software by eliminating predefined objectives altogether. You define your own goals in these games; you're not likely to get stuck on a level because you haven't figured out how to "grow" a certain resource, for the simple reason that there are no preordained levels to follow. You define your own hurdles as you play. In SimCity, you decide whether to build a megalopolis or a farming community; whether to build an environmentally correct new urbanist village or a digital Coketown. Of course, you may find it hard to achieve those goals as you build the city, but because those goals aren't part of the game's official rules, you don't feel stuck in the same way that you might feel stuck in Evolva, staring across the canyon without the genes for jumping.

There's a catch here, though. "The challenge is, the more autonomous the system, the more autonomous the virtual creatures, the more irrelevant the player is," Zimmerman explains. "The problem with a lot of the 'god games' is that it's difficult to feel like you're having a meaningful impact on the system. It's like you're wearing these big, fuzzy gloves and you're trying to manipulate these tiny little objects." Although it can be magical to watch a Will Wright simulation take on a life of its own, it can also be uniquely frustrating—when that one neighborhood can't seem to shake off its crime problem, or your Sims refuse to fall in love. For better or worse, we control these games from the edges. The task of the game designer is to determine just how far off the edge the player should be.

Nowhere is this principle more apparent than in the control

panel that Will Wright built for The Sims. Roll your cursor along the bottom of the screen while surveying your virtual neighborhood, and a status window appears, with the latest info on your characters' emotional and physical needs: you'll see in an instant whether they've showered today, or whether they're pining for some companionship. A click away from that status window is a control panel screen, where you can adjust various game attributes. A "settings" screen is by now a standard accoutrement of any off-the-shelf game: you visit the screen to adjust the sound quality or the graphics resolution, or to switch difficulty levels. At first glance, the control panel for The Sims looks like any of these other settings screens: there's a button that changes whether the window scrolls automatically as you move the mouse, and another that turns off the background music. But alongside these prosaic options, there is a toggle switch that says, in unabashed Cartesian terms, "Free will."

If you leave "Free will" off, The Sims quickly disintegrates into a nightmare of round-the-clock maintenance, requiring the kind of constant attention you'd expect in a nursery or a home for Alzheimer's patients. Without free will, your Sims simply sit around, waiting for you to tell them what to do. They may be starving, but unless you direct them to the fridge, they'll just sit out their craving for food like a gang of suburban hunger artists. Even the neatest of the Sims will tolerate piles of rotting garbage until you specifically order them to take out the trash. Without a helpful push toward the toilet, they'll even relieve themselves right in the middle of the living room.

Playing The Sims without free will selected is a great reminder that too much control can be a disastrous thing. But the opposite can be even worse. Early in the design of The Sims, Wright recognized that his virtual people would need a certain amount of autonomy for the game to be fun, and so he and his team began developing a set of artificial-intelligence routines that would allow

the Sims to think for themselves. That AI became the basis for the character's "free will," but after a year of work, the designers found that they'd been a little too successful in bringing the Sims to life.

"One of our biggest problems here was that our AI was too smart," Wright says now. "The characters chose whichever action would maximize their happiness at any given moment. The problem is that they're usually much better at this than the player." The fun of The Sims comes from the incomplete information that you have about the overall system: you don't know exactly what combination of actions will lead to a maximum amount of happiness for your characters—but the software behind the AI can easily make those calculations, because the happiness quota is built out of the game's rules. In Wright's early incarnations of the game, once you turned on free will, your characters would go about maximizing their happiness in perfectly rational ways. The effect was not unlike hiring Deep Blue to play a game of chess for you—the results were undeniably good ones, but where was the fun?

And so Wright had to dumb down his digital creations. "We did it in two ways," he says. "First, we made them focus on immediate gratification rather than long-term goals—they'd rather sit in front of the TV and be couch potatoes than study for a job promotion. Second, we gave their personality a very heavy weight on their decisions, to an almost pathological degree. A very neat Sim will spend way too much time picking up—even after other Sims—while a sloppy Sim will never do this. These two things were enough to ensure that the player was a sorely needed component—ambition? balance?—of their world." In other words, Wright made their decisions local ones and made the rules that governed their behavior more intransigent. For the emergent system of the game to work, Wright had to make the Sims more like ants than people.

I think there is something profound, and embryonic, in that "free will" button, and in Wright's battle with the autonomy of his

creations—something both like and unlike the traditional talents that we expect from our great storytellers. Narrative has always been about the mix of invention and repetition; stories seem like stories because they follow rules that we've learned to recognize, but the stories that we most love are ones that surprise us in some way, that break rules in the telling. They are a mix of the familiar and the strange: too much of the former, and they seem stale, formulaic; too much of the latter, and they cease to be stories. We love narrative genres—detective, romance, action-adventure—but the word *generic* is almost always used as a pejorative.

It misses the point to think of what Will Wright does as storytelling—it doesn't do justice to the novelty of the form, and its own peculiar charms. But that battle over control that underlies any work of emergent software, particularly a work that aims to entertain us, runs parallel to the clash between repetition and invention in the art of the storyteller. A good yarn surprises us, but not too much. A game like The Sims gives its on-screen creatures some autonomy, but not too much. Emergent systems are not stories, and in many ways they live by very different rules, for both creator and consumer. (For one, emergent systems make that distinction a lot blurrier.) But the art of the storyteller can be enlightening in this context, because we already accept the premise that storytelling *is* an art, and we have a mature vocabulary to describe the gifts of its practitioners. We are only just now developing such a language to describe the art of emergence. But here's a start: great designers like Wright or Resnick or Zimmerman are *control* artists—they have a feel for that middle ground between free will and the nursing home, for the thin line between too much order and too little. They have a feel for the edges.