Computational Foundations of Artificial Life: A Systematic Analysis of Evolutionary Algorithms and Emergent Systems

The field of artificial life (ALife) represents a radical departure from traditional biological inquiry, transitioning from the analytic study of existing carbon-based organisms to the synthetic creation of lifelike behaviors in alternative media.[1] By abstracting the fundamental principles of life—such as self-organization, reproduction, and evolution—into mathematical rules and computational processes, researchers explore the domain of "life-as-it-could-be".[2] This discipline is not merely an exercise in mimicry; it provides a theoretical framework for understanding the essential logic of living systems regardless of their physical substrate.[1]

Historically, the drive to simulate life originated with mechanical automata, such as Jacques de Vaucanson's clockwork duck (1735), which attempted to replicate biological functions through physical gears and pulleys.[4] However, the modern computational era of ALife was inaugurated by John von Neumann in 1951, as he sought to identify the logical requirements for a machine to replicate itself.[4] His work established cellular automata as a foundational tool for the field.[1] Today, ALife spans a spectrum from discrete grid-based simulations and agent-based models to continuous biochemical reactions and neuro-evolutionary frameworks.[1]

The following report analyzes twenty of the most influential and well-known simulations in the history of artificial life and complex systems. Each section provides a detailed description of the model's mechanism, its historical and academic context, and specific implementation logic designed for integration into modern visualization engines.

Discrete Grid-Based Cellular Automata

Cellular automata (CA) are spatially and temporally discrete systems where the state of each cell in a grid is updated simultaneously based on a set of local transition rules.[8] These models demonstrate how complex, global order emerges from simple, decentralized interactions.

Conway's Game of Life

Conway's Game of Life is arguably the most famous cellular automaton, introduced by John Horton Conway in 1970.[6] It operates on a two-dimensional grid of cells that are either "alive" or "dead." The system's evolution is determined by the Moore neighborhood—the eight cells surrounding a central cell.[8]

Rule Name	Cell State	Condition (Neighbors)	Next State
Underpopulation	Alive	< 2 Alive	Dead
Survival	Alive	2 or 3 Alive	Alive
Overpopulation	Alive	> 3 Alive	Dead
Reproduction	Dead	Exactly 3 Alive	Alive

The implementation of the Game of Life requires two grids: the current state and the next state, to ensure all updates occur simultaneously.[11] The significance of the Game of Life lies in its Wolfram Class 4 classification, indicating its capacity for universal computation.[8] Structures such as "gliders" and "oscillators" serve as information carriers, allowing for the construction of logic gates and even entire computers within the simulation.[12] Implementations are widely available in the Golly repository and Daniel Shiffman's Nature of Code examples.[14]

Wolfram's Rule 110 and Elementary Cellular Automata

Stephen Wolfram's study of one-dimensional "elementary" cellular automata (ECA) revealed that even the simplest 1D systems can exhibit profound complexity.[9] Rule 110 is the most famous of these, as it was mathematically proven by Matthew Cook to be Turing-complete.[12]

Neighborhood Pattern	Resulting State (Rule 110)
111	0
110	1
101	1
100	0
011	1

010	1
001	1
000	0

The implementation involves treating a single row of cells as the current state and generating subsequent rows beneath it.[17] The decimal name "110" is derived from the binary string 01101110, which defines the transition table.[9] Rule 110 sits on the boundary between stability and chaos, producing localized "spaceships" that interact over an infinitely repeating background pattern.[12]

Langton’s Ant and Turmites

Langton’s Ant, created by Christopher Langton in 1986, is a two-dimensional universal Turing machine that operates with an agent (the "ant") rather than fixed grid rules.[6] The ant follows two simple rules: if it is on a white square, it turns 90° right, flips the color, and moves; if on a black square, it turns 90° left, flips the color, and moves.[6]

Despite these minimal rules, the ant exhibits three distinct emergent phases:

1. **Simplicity:** The first few hundred steps produce small, symmetric patterns.
2. **Chaos:** A long period (up to 10,000 steps) where the ant builds an irregular, pseudo-random "cloud" of colors.
3. **Order (The Highway):** The ant eventually enters a cycle of 104 steps that constructs a "highway"—a diagonal path that moves the ant indefinitely in one direction.[6]

Implementing Langton’s Ant requires only a grid and an agent state (position and orientation). It can be extended into "Turmites" (Turing-machine mites) by adding more states to the grid and more complex state transitions for the agent.[6]

Wireworld

Wireworld is a four-state cellular automaton designed to simulate digital logic and electronic circuits.[20] It is particularly effective for creative coding because it allows users to "draw" wires and gates that function realistically.

The states are as follows:

- **0: Empty (Black):** Serves as an insulator.
- **1: Electron Head (Blue):** The leading edge of a signal.
- **2: Electron Tail (Red):** The trailing edge of a signal, preventing the signal from flowing

backward.

- **3: Conductor (Yellow):** The wire itself.[20]

Current State	**Transition Condition**	**Next State**
Empty	Always	Empty
Electron Head	Always	Electron Tail
Electron Tail	Always	Conductor
Conductor	Exactly 1 or 2 neighbors are Heads	Electron Head
Conductor	Otherwise	Conductor

Wireworld's simplicity allows for the construction of XOR, AND, and OR gates, making it a favorite for visualizing computational structures.[10] Python and Nim implementations are available on GitHub, often using Tkinter or SDL2 for rendering.[20]

Brian’s Brain

Introduced by Brian Silverman, this CA is an extension of the "Seeds" rule and serves as a visual metaphor for neural firing and refractory periods.[6] It uses three states: **Off**, **On**, and **Dying**.

The rules are:

1. **Off to On:** An "off" cell turns "on" if exactly two of its neighbors are "on."
2. **On to Dying:** An "on" cell always turns "dying" in the next generation.
3. **Dying to Off:** A "dying" cell always turns "off" in the next generation.[22]

This refractory state (Dying) prevents immediate re-firing, leading to the emergence of complex, oscillating patterns that resemble a simplified model of brain activity or sparking electrical fields.[6] Implementation is straightforward using HTML5 Canvas and Easel.js, with a focus on color-coding (e.g., White for On, Red for Dying, Black for Off).[22]

Self-Replicating and Ecological Systems

Self-replication is a defining characteristic of life, and ALife researchers have developed several models to explore how informational patterns can copy themselves within a rule-based environment.

Langton’s Loops

Christopher Langton created Langton’s Loops in 1984 as a simplified alternative to von Neumann's massive self-replicating automaton.[16] The model uses 8 states and the von Neumann neighborhood (up, down, left, right).[16]

The loops consist of a sheathed wire containing "genetic information" that flows clockwise.[16] When the information reaches a T-junction, it is duplicated: one copy continues around the loop, while the other travels down an extension arm.[16] This arm eventually makes three left turns to close the loop, causing it to detach as a daughter organism.[24]

State Value	Color	Role
0	Black	Background / Empty
1	Blue	Core (Conductive)
2	Red	Sheath (Insulating)
3	Green	Temporary Marker
5	Magenta	Replication Trigger
7	Cyan	Extension Command

Implementing Langton’s Loops requires a large transition table (approximately 219 rules, often expanded to over 800 via rotational symmetry).[24] The "DNA" sequence 70-70-70-70-70-70-40-40 within the sheath directs the growth and turning of the extension arm.[16] Golly and various Python scripts provide reference implementations.[24]

Tierra

Developed by ecologist Thomas Ray, Tierra is a "digital soup" where programs (digital organisms) compete for two resources: CPU time and memory space.[27] Unlike other simulations, Tierra uses an evolvable machine code instruction set.[27]

Programs in Tierra can:

- **Self-replicate:** Copy their own code into new memory locations.

- **Mutate:** Randomly flip bits in their code.
- **Recombine:** Swap segments of code with other programs.[27]

The virtual machine for Tierra provides an endogenous fitness function; organisms only survive if they can successfully execute and replicate.[27] This led to the emergence of parasites—shorter programs that stole the replication code of larger hosts—and subsequent ecological cycles of defense and evasion.[27] Implementation is traditionally in C, using a custom MIMD (multiple instruction, multiple data) architecture.[27]

Daisyworld

Daisyworld is a zero-dimensional planetary simulation by James Lovelock and Andrew Watson designed to illustrate the Gaia hypothesis.[28] It models a planet populated by black and white daisies that regulate the global temperature through albedo (reflectivity).[28]

The mathematical core consists of coupled non-linear differential equations:

$$\frac{dA_w}{dt} = A_w(x\beta_w - \gamma)$$

$$\frac{dA_b}{dt} = A_b(x\beta_b - \gamma)$$

where A is the area covered, β is the growth rate (a function of temperature), and γ is the death rate.[28]

White daisies reflect light (cooling effect), while black daisies absorb light (warming effect).[28] The simulation demonstrates that even without conscious intent, biological feedback loops can stabilize a planet's environment against an increasing solar luminosity.[28] Web-based JavaScript implementations allow for adjusting parameters like solar luminosity and daisy albedo.[29]

Sugarscape

Sugarscape is an agent-based social simulation created by Joshua Epstein and Robert Axtell to model the emergence of artificial societies.[30] Agents move on a grid to collect and consume "sugar".[30]

Advanced rules include:

- **Metabolism and Vision:** Agents have varying rates of sugar consumption and ranges of scanning for resources.[31]
- **Pollution:** Agents leave "pollution" where they eat, which diffuses over time.[30]

- **Inheritance and Trade:** Wealth (sugar) can be passed to offspring or traded for other resources like "spice".[30]

Implementation in Python (using Tkinter or Matplotlib) or JavaScript reveals that simple survival rules lead to complex phenomena like wealth inequality and migration patterns.[31]

Collective Intelligence and Agent-Based Models

Collective behavior simulations focus on how groups of autonomous agents coordinate without a central leader, often inspired by biological swarms and colonies.

Boids (Flocking)

Craig Reynolds' Boids (1986) is the definitive model for simulating the flocking of birds or schooling of fish.[14] Each agent, or "boid," follows three steering behaviors [14]:

1. **Separation:** Steer to avoid crowding or colliding with neighbors.
2. **Alignment:** Steer toward the average heading of neighbors.
3. **Cohesion:** Steer toward the average position (center of mass) of neighbors.

The formula for steering is: $\text{steering force} = \text{desired velocity} - \text{current velocity}$.[14] For visualization, agents only perceive neighbors within a limited radius (e.g., 50 pixels).[14] Advanced implementations use bin-lattice spatial subdivision to handle large populations efficiently.[14]

Ant Colony Optimization (ACO)

ACO algorithms use artificial ants to solve combinatorial optimization problems, such as the Traveling Salesman Problem (TSP).[34] Ants leave "pheromone" trails on the edges of a graph, and the probability of an ant choosing a specific edge is determined by its pheromone density and heuristic length.[34]

The pheromone update rule is:

$$\tau_{ij}(t) = (1-\rho)\tau_{ij} + \sum \Delta\tau_{ij}^{k}$$

where ρ is the evaporation rate.[36] Over time, shorter paths accumulate more pheromones because ants can complete them more quickly, while evaporation prevents the colony from getting stuck in local optima.[34] Python implementations often visualize the darkening of trails to represent increasing pheromone concentration.[34]

Physarum (Slime Mold)

The Jeff Jones model of *Physarum polycephalum* simulates the growth of slime mold using a multi-agent system.[37] Agents move toward regions of high "trail" concentration and deposit their own trails behind them.[38]

The update cycle includes:

1. **Sensing:** Agents sample the grid at three sensors (front-left, front-center, front-right).[38]
2. **Rotation:** The agent turns toward the sensor with the highest value.[38]
3. **Movement:** The agent moves forward by a "step size" and deposits a value into the grid.[38]
4. **Diffusion/Decay:** The grid's values are blurred (Gaussian blur) and multiplied by a decay factor.[38]

This feedback loop generates intricate transport networks that approximate optimal graph structures like Voronoi diagrams.[39] Implementation is ideal for GLSL fragment shaders or compute shaders due to the massive number of agents required (often $> 100,000$).[37]

Continuous and Physicochemical Models

Moving beyond discrete grids, these simulations utilize continuous fields and complex mathematics to replicate organic growth and fluid dynamics.

Gray-Scott Reaction-Diffusion

The Gray-Scott model describes the chemical reaction between two species, U and V, which diffuse and react on a grid.[41]

The governing equations are:

$$\frac{\partial U}{\partial t} = D_u \nabla^2 U - UV^2 + F(1-U)$$

$$\frac{\partial V}{\partial t} = D_v \nabla^2 V + UV^2 - (F+k)V$$

where F is the feed rate and k is the kill rate.[11]

Pattern Name	Feed (F)	Kill (k)	Visual Description
Gliders	0.014	0.054	Moving bubbles that replicate

Maze	0.029	0.057	Labyrinthine stripes
Spots	0.030	0.062	Static or pulsating dots
Worms	0.058	0.065	Meandering lines

Implementing this requires calculating the discrete Laplacian ∇^2 using a five-point stencil.[11] Python (NumPy) and Swift (Metal) implementations are highly effective for visualizing these patterns in real-time.[42]

SmoothLife and Lenia

SmoothLife (Rafler, 2011) and Lenia (Chan, 2015) are generalizations of the Game of Life to continuous space and time.[6] Instead of discrete cells, they use a continuous field where state values are in the range $$.[45]

Lenia utilizes a "kernel" (a bell-shaped function) and a "growth function" to determine how the field updates.[6] It is capable of producing thousands of "species" of digital organisms with complex, lifelike movements.[6] Particle Lenia is a variation that treats these organisms as particles, enabling 3D simulations.[6] Implementation often requires Fast Fourier Transforms (FFT) for efficient convolution.[47]

Diffusion-Limited Aggregation (DLA)

DLA models the process where randomly moving (Brownian) particles aggregate onto a seed to form fractal, branch-like structures.[48] It mimics phenomena like frost, coral growth, and crystal formation.[48]

The algorithm logic:

1. **Seed:** Place a stationary particle in the center.
2. **Walker:** Release a particle at a random point on a perimeter.
3. **Walk:** Move the walker randomly until it hits the seed or another stuck particle.[49]
4. **Aggregate:** The walker becomes stationary, and a new walker is released.[49]

For high performance, DLA is implemented on the GPU (using CUDA or WebGL) to handle millions of particles.[50] Creative versions, such as "dlaf," achieve blistering speeds by using spatial indexing to bypass brute-force collision detection.[48]

Particle Life

Particle Life is a multi-particle system where different "species" of particles exert attractive or repulsive forces on one another.[6] Based on Jeffrey Ventrella's "Clusters," it produces emergent, fluid-like ALife.[6]

Implementation involves calculating the force between two particles i and j: $F_{ij} = G(r_{ij}, \text{type}_i, \text{type}_j)$, where G is a distance-dependent function.[6] This leads to particles forming stable, rotating, or swimming structures that resemble multi-cellular organisms.[6]

Evolutionary and Morphological Algorithms

These models focus on the evolution of form and behavior through genetic algorithms and neural network optimization.

Evolved Virtual Creatures (Karl Sims)

Karl Sims' 1994 simulation is a landmark in ALife, involving the evolution of 3D creatures in a physically realistic environment.[52] Creatures are defined by a directed graph (genome) where each node represents a rigid part and each edge represents a joint.[52]

The evolutionary loop:

1. **Genome to Phenotype:** The graph is used to build a 3D body and a neural network controller (the brain).[52]
2. **Evaluation:** The creature is tested for a task, such as walking, swimming, or competing for a cube.[53]
3. **Selection:** The most successful creatures are chosen to reproduce.[53]
4. **Reproduction:** Their "virtual genes" are combined via crossover and mutated.[53]

While the original source code is proprietary, modern replicas like "evolving-creatures" use Box2D or Bullet physics engines to recreate these results.[55]

BoxCar2D

BoxCar2D is a popular genetic algorithm that evolves the shape of 2D cars to navigate bumpy terrain.[58] The DNA consists of vertices and wheel parameters.[59]

Gene Index	Property	Description
0	Distance	Distance of vertex from center

1	Angle	Angle of vertex
2	Wheel Spawn	Boolean (60% probability)
3	Wheel Radius	Size of the wheel

Implementation requires a physics engine (like Box2D) and a procedural terrain generator (often using Perlin Noise).[59] Fitness is defined as the horizontal distance traveled before the car becomes stationary.[59] Repositories like "HTML5_Genetic_Cars" provide complete canvas-based examples.[57]

Neural Cellular Automata (NCA)

Neural Cellular Automata (Mordvintsev et al., 2020) are CA where the update rules are learned by a neural network.[7] Each cell has a 16-channel state, and the system is trained to "grow" a specific image (like a lizard or an emoji) from a single pixel seed.[7]

The update process:

1. **Perception:** Each cell uses Sobel filters to sense its neighbors' states.[7]
2. **Inference:** A small MLP (multi-layer perceptron) processes this perception to propose an update.[7]
3. **Stochastic Update:** To prevent over-fitting, only a random fraction of cells are updated in each step.[7]

NCAs are famously regenerative; if a portion of the "organism" is cut away, the remaining cells coordinate to regrow the missing parts.[61] Python (PyTorch/TensorFlow) and JavaScript implementations are widely available in the Distill.pub tutorial and on Google Colab.[7]

L-Systems (Lindenmayer Systems)

L-systems are recursive string rewriting systems used to model the growth of plants and trees.[14]

Production rules look like: F -> FF+[+F-F-F]-[-F+F+F].[14] When combined with "Turtle Graphics" (where F is forward and +/- are rotations), these rules produce self-similar, fractal shapes.[14] Stochastic L-systems add randomness to the rules to create more "natural" and non-identical trees.[14]

Synthesis of Implementation Strategies

For a creative coder looking to visualize these simulations, the primary challenge is managing the trade-off between computational complexity and real-time performance.

Neighborhood and Grid Management

Most grid-based CA (Life, Wireworld, Brian's Brain) use two standard neighborhood types:

- **Von Neumann:** 4 adjacent cells (North, South, East, West).[8]
- **Moore:** 8 adjacent cells (includes diagonals).[8]

For 2D grid updates in languages like JavaScript or Python, the "Double Buffering" technique is essential:

Python

```python
# Pseudo-code for simultaneous update
next_grid = current_grid.copy()
for x, y in grid:
    next_grid[x][y] = apply_rules(current_grid, x, y)
current_grid = next_grid
```

In high-performance visualization, this is handled via "ping-ponging" between two textures in a GPU fragment shader.[37]

Genetic Algorithm Flow

When implementing evolutionary models (Sims, BoxCar2D), the following architecture is standard [14]:

Phase	Action	Implementation Detail
Generation	Population size P	Typically 20-100 individuals.
Simulation	Physics Step	Run for T frames or until stagnation.
Selection	Mating Pool	Add individuals to pool proportional to fitness.

Crossover	DNA Merging	Split parent arrays at a random midpoint.
Mutation	Randomize Gene	Apply a small delta to 1-5% of genes.

Visualization Foundations

Visual exploration of ALife often leverages libraries like p5.js, which simplifies the rendering of thousands of agents or grid cells.[14] For complex systems like Gray-Scott or DLA, compute shaders (via WebGPU or Unity) are preferred as they allow for millions of calculations per frame.[50] The "Nature of Code" by Daniel Shiffman serves as the central pedagogical resource for these implementations, offering ports in languages ranging from Processing to Rust.[14]

Conclusion and Future Outlook

The twenty simulations analyzed in this report represent the pinnacle of artificial life and evolutionary computing research. From the discrete mathematical certainty of Rule 110 to the learned, regenerative complexity of Neural Cellular Automata, these models provide a window into the mechanisms of emergent behavior.[3]

The significance of these systems extends beyond academic curiosity. They offer practical applications in:

- **Optimization:** Ant Colony Optimization and Genetic Algorithms solve complex logistical problems.[34]
- **Design:** L-systems and DLA generate procedural architectures and organic textures.[49]
- **Resilience:** Neural CAs provide a framework for creating self-healing software systems.[61]

As computational power increases, the boundary between "soft" ALife and real biological complexity continues to blur. The transition toward continuous, differentiable simulations like Lenia and NCA suggests a future where artificial life is not just a simulation of biology, but a fundamental expansion of what life is capable of becoming.[3] By mastering these twenty algorithms, the researcher or creative coder gains the tools necessary to architect entire universes from simple local rules, contributing to our collective understanding of complexity and the very nature of existence.[2]

Works cited

1. Artificial life - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Artificial_life
2. artificial life christopher g. langton - ARS Electronica, accessed on January 27,

2026, https://webarchive.ars.electronica.art/en/archiv_files/19931/E1993_025.pdf
3. Artificial Life - ais.uni-bonn.de, accessed on January 27, 2026, https://www.ais.uni-bonn.de/SS09/skript_artif_life_pfeifer_unizh.pdf
4. The Past, Present, and Future of Artificial Life - Frontiers, accessed on January 27, 2026, https://www.frontiersin.org/journals/robotics-and-ai/articles/10.3389/frobt.2014.00008/full
5. 20th WCP: Synthetic Biology: The Technoscience of Artificial Life, accessed on January 27, 2026, https://www.bu.edu/wcp/Papers/Tech/TechSull.htm
6. Math Art Part 1: Cellular Automata | Fractals | Artificial Life ..., accessed on January 27, 2026, https://marcusvolz.com/mathart1/
7. Growing Neural Cellular Automata - Distill.pub, accessed on January 27, 2026, https://distill.pub/2020/growing-ca/
8. Cellular automaton - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Cellular_automaton
9. Elementary cellular automaton - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Elementary_cellular_automaton
10. Functional cellular automatons, accessed on January 27, 2026, https://ivan1931.github.io/functional-cellular-automatons/
11. gray_scott.ipynb - benmaier/reaction-diffusion - GitHub, accessed on January 27, 2026, https://github.com/benmaier/reaction-diffusion/blob/master/gray_scott.ipynb
12. Rule 110 - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Rule_110
13. What Can We Learn about Engineering and Innovation from Half a Century of the Game of Life Cellular Automaton? - Stephen Wolfram Writings, accessed on January 27, 2026, https://writings.stephenwolfram.com/2025/03/what-can-we-learn-about-engineering-and-innovation-from-half-a-century-of-the-game-of-life-cellular-automaton/
14. Examples / Nature of Code, accessed on January 27, 2026, https://natureofcode.com/examples/
15. nature-of-code/noc-examples-processing - GitHub, accessed on January 27, 2026, https://github.com/nature-of-code/noc-examples-processing
16. Langton's loops - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Langton%27s_loops
17. Rule 110 Cellular Automaton - Blake Crosley, accessed on January 27, 2026, https://blakecrosley.com/blog/rule-110
18. Universality in Elementary Cellular Automata Matthew Coo: Department of Computation and Neural Systems* Caltech, Mail Stop 136-9, accessed on January 27, 2026, https://www.dna.caltech.edu/courses/cs191/paperscs191/Cook_Rule110_Full_Unpublished.pdf
19. finite-width elementary cellular automata, accessed on January 27, 2026, https://www.whitman.edu/documents/Academics/Mathematics/SeniorProject_Ian

Coleman.pdf
20. An implementaton of the Wireworld cellular automaton - GitHub, accessed on January 27, 2026, https://github.com/yackx/wireworld
21. An implementation of Wireworld (Cellular Automata) in Nim using SDL2 - GitHub, accessed on January 27, 2026, https://github.com/JHonaker/wireworld-nim
22. davidhu2000/brians_brain: A cellular automaton simulation based on the rules of Brian's Brain. - GitHub, accessed on January 27, 2026, https://github.com/davidhu2000/brians_brain
23. Building a Self-Replicating Cellular Automaton - - Programming for Lovers, accessed on January 27, 2026, https://programmingforlovers.com/chapter-3-building-a-self-replicating-cellular-automaton-with-top-down-programming/langtons-loops/
24. Yet Another Implementation of Langton's Loops Self-Replicating Cellular Automata Using Python - James D. McCaffrey, accessed on January 27, 2026, https://jamesmccaffrey.wordpress.com/2024/09/20/yet-another-implementation-of-langtons-loops-self-replicating-cellular-automata-using-python/
25. Source code for Langton's loops CA in MCell - GitHub Gist, accessed on January 27, 2026, https://gist.github.com/Sgeo/ead48728917153cb8fe2c2c2221f2e75
26. ruletablerepository/GollyAccessPoint.html at gh-pages - GitHub, accessed on January 27, 2026, https://github.com/GollyGang/ruletablerepository/blob/gh-pages/GollyAccessPoint.html
27. Tierra (computer simulation) - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Tierra_(computer_simulation)
28. Daisyworld - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Daisyworld
29. btschwertfeger/Daisyworld-Model-Website - GitHub, accessed on January 27, 2026, https://github.com/btschwertfeger/Daisyworld-Model-Website
30. Sugarscape - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Sugarscape
31. Sugarscape - Joshua Palicka - GitHub Pages, accessed on January 27, 2026, https://joshuapalicka.github.io/sugarscape.html
32. mw2000/SugarScape: A lean implementation of SugarScape in Python, optimized for experimentation - GitHub, accessed on January 27, 2026, https://github.com/mw2000/SugarScape
33. Sugarscape in JavaScript! - GitHub, accessed on January 27, 2026, https://github.com/bragin/sugarscape
34. Ant colony optimization algorithms - Wikipedia, accessed on January 27, 2026, https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms
35. Ant colony optimization - Scholarpedia, accessed on January 27, 2026, http://www.scholarpedia.org/article/Ant_colony_optimization
36. Application of an improved ant colony optimization algorithm of hybrid strategies using scheduling for patient management in hospitals - PubMed Central, accessed on January 27, 2026, https://pmc.ncbi.nlm.nih.gov/articles/PMC11693920/

37. ISF shader to simulate dynamics of physarum slime mold - GitHub, accessed on January 27, 2026, https://github.com/nwhetsell/isf-physarum
38. Understanding the Physarum Simulation - Deniz Bicer, accessed on January 27, 2026, https://denizbicer.com/202408-UnderstandingPhysarum.html
39. Biological Computation of Physarum - CumInCAD, accessed on January 27, 2026, https://papers.cumincad.org/data/works/att/ecaade2018_303.pdf
40. Applications of Multi-Agent Slime Mould Computing Author=Jeff Jones - arXiv, accessed on January 27, 2026, https://arxiv.org/pdf/1511.05774
41. Pattern formation in the Gray-Scott reaction-diffusion equations - Polymathic AI, accessed on January 27, 2026, https://polymathic-ai.org/the_well/datasets/gray_scott_reaction_diffusion/
42. Examples of the Gray-Scott model - GitHub, accessed on January 27, 2026, https://github.com/wigging/gray-scott
43. kaityo256/python_gs: Python implemenation of Gray-Scott Model - GitHub, accessed on January 27, 2026, https://github.com/kaityo256/python_gs
44. BlagojeBlagojevic/smooth-life: implementation of a conway game of life - GitHub, accessed on January 27, 2026, https://github.com/BlagojeBlagojevic/smooth-life
45. A simple version of SmoothLife in Haskell - GitHub, accessed on January 27, 2026, https://github.com/travisbrown/smoothlife
46. xuset/SmoothLife: An artificial life simulator - GitHub, accessed on January 27, 2026, https://github.com/xuset/SmoothLife
47. SmoothLife Implementation in C - GitHub, accessed on January 27, 2026, https://github.com/tsoding/SmoothLife
48. 2d-diffusion-limited-aggregation-experiments/README.md at master - GitHub, accessed on January 27, 2026, https://github.com/jasonwebb/2d-diffusion-limited-aggregation-experiments/blob/master/README.md
49. MRAC-IAAC/Diffusion-Limited-Aggregation - GitHub, accessed on January 27, 2026, https://github.com/MRAC-IAAC/Diffusion-Limited-Aggregation
50. zentralwerkstatt/dla-gpu: Diffusion limited aggregation on the GPU - GitHub, accessed on January 27, 2026, https://github.com/zentralwerkstatt/dla-gpu
51. fogleman/dlaf: Diffusion-limited aggregation, fast. - GitHub, accessed on January 27, 2026, https://github.com/fogleman/dlaf
52. Evolving Virtual Creatures, accessed on January 27, 2026, https://team.inria.fr/imagine/files/2014/10/sims_siggraph94.pdf
53. Evolved Virtual Creatures - Karl Sims, accessed on January 27, 2026, https://www.karlsims.com/evolved-virtual-creatures.html
54. Evolving Virtual Creatures - Karl Sims, accessed on January 27, 2026, https://www.karlsims.com/papers/siggraph94.pdf
55. Evolving Creatures - A Simulation of Evolutionary Process using Genetic Algorithm - GitHub, accessed on January 27, 2026, https://github.com/hanzholahs/evolving-creatures
56. A virtual creatures model for studies in artificial evolution - channon.net, accessed on January 27, 2026, http://www.channon.net/alastair/papers/cec2005.pdf

57. red42/HTML5_Genetic_Cars: A genetic algorithm car evolver in HTML5 canvas. - GitHub, accessed on January 27, 2026, https://github.com/red42/HTML5_Genetic_Cars
58. HTML5 Genetic Algorithm 2D Car Thingy - Chrome recommended, accessed on January 27, 2026, https://rednuht.org/genetic_cars_2/
59. ad71/Genetic-Algorithms - GitHub, accessed on January 27, 2026, https://github.com/ad71/Genetic-Algorithms
60. MECLabTUDA/NCA-tutorial - GitHub, accessed on January 27, 2026, https://github.com/MECLabTUDA/NCA-tutorial
61. Fun With Neural Cellular Automata | nca – Weights & Biases - Wandb, accessed on January 27, 2026, https://wandb.ai/johnowhitaker/nca/reports/Fun-With-Neural-Cellular-Automata--VmlldzoyMDQ5Mjg0
62. Growing Neural Cellular Automata (A Tutorial) : r/neuralnetworks - Reddit, accessed on January 27, 2026, https://www.reddit.com/r/neuralnetworks/comments/1ldds7j/growing_neural_cellular_automata_a_tutorial/
63. terkelg/awesome-creative-coding: Creative Coding: Generative Art, Data visualization, Interaction Design, Resources. - GitHub, accessed on January 27, 2026, https://github.com/terkelg/awesome-creative-coding
64. Day 35: Evolution Beyond Biology: Using Genetic Algorithms for Creative Art and Design, accessed on January 27, 2026, https://www.woodruff.dev/day-35-evolution-beyond-biology-using-genetic-algorithms-for-creative-art-and-design/
65. Attention-based Neural Cellular Automata - NeurIPS, accessed on January 27, 2026, https://papers.neurips.cc/paper_files/paper/2022/file/361e5112d2eca09513bbd266e4b2d2be-Paper-Conference.pdf
66. The Nature of Code - ITP - NYU, accessed on January 27, 2026, https://itp.nyu.edu/itp/the-nature-of-code/