


2009

Automatic Graphics And Game Content Generation Through Evolutionary Computation

Erin Hastings
University of Central Florida

 Part of the [Computer Sciences Commons](#), and the [Engineering Commons](#)
Find similar works at: <https://stars.library.ucf.edu/etd>
University of Central Florida Libraries <http://library.ucf.edu>

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Hastings, Erin, "Automatic Graphics And Game Content Generation Through Evolutionary Computation" (2009). *Electronic Theses and Dissertations, 2004-2019*. 3997.
<https://stars.library.ucf.edu/etd/3997>

AUTOMATIC GRAPHICS AND GAME CONTENT GENERATION
THROUGH EVOLUTIONARY COMPUTATION

by

ERIN HASTINGS

M.S. University of Central Florida, 2005

B.S. University of Florida, 2001

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2009

Major Professor:
Kenneth O. Stanley and Ratan K. Guha

© 2009 by Erin Hastings

ABSTRACT

Simulation and game content includes the levels, models, textures, items, and other objects encountered and possessed by players during the game. In most modern video games and simulation software, the set of content shipped with the product is static and unchanging, or at best, randomized within a narrow set of parameters. However, ideally, if game content could be constantly and automatically renewed, players would remain engaged longer in the evolving stream of content. This dissertation introduces three novel technologies that together realize this ambition. (1) The first, NEAT Particles, is an evolutionary method to enable users to quickly and easily create complex particle effects through a simple interactive evolutionary computation (IEC) interface. That way, particle effects become an evolvable class of content, which is exploited in the remainder of the dissertation. In particular, (2) a new algorithm called *content-generating NeuroEvolution of Augmenting Topologies* (cgNEAT) is introduced that automatically generates graphical and game content while the game is played, based on the past preferences of the players. Through cgNEAT, the game platform on its own can generate novel content that is designed to satisfy its players. Finally, (3) the *Galactic Arms Race* (GAR) multiplayer online video game is constructed to demonstrate these techniques working on a real online gaming platform. In GAR, which was made available to the public and playable online, players pilot space ships and fight enemies

to acquire unique particle system weapons that are automatically evolved by the cgNEAT algorithm. The resulting study shows that cgNEAT indeed enables players to discover a wide variety of appealing content that is not only novel, but also based on and extended from previous content that they preferred in the past. The implication is that with cgNEAT it is now possible to create applications that generate their own content to satisfy users, potentially significantly reducing the cost of content creation and considerably increasing entertainment value with a constant stream of evolving content.

To my wife, Jaruwan and to my parents, Steve and Patricia Hastings.

ACKNOWLEDGMENTS

Thanks to my co-advisor Dr. Kenneth O. Stanley, who introduced me to Evolutionary Computation. With his guidance and attention to detail we've published valuable contributions to the field. Hopefully there will be many more.

Thanks to my co-advisor Dr. Ratan K. Guha, without whose support I would not be writing this dissertation today. He got me interested in graphics, and enabled me to apply research to gaming and simulations.

Thanks to dissertation committee members Dr. Michael Gourlay and Dr. Paul Wiegand for generously donating their time, knowledge, and suggestions.

Thanks to past and present members of the Evolutionary Complexity Research Group (Eplex) at University of Central Florida (<http://eplex.cs.ucf.edu>) for their valuable feedback on my various papers, projects, and presentations. Namely, Dr. Charles E. Bailey, David D'Ambrosio, J.T. Folsom-Kovarik, Jason Gauci, Amy Hoover, Joel Lehman, Yuan Li, Sebastian Risi, Jimmy Secretan, and Phillip Verbancsics.

Special thanks to the Galactic Arms Race (GAR) volunteer team: Nathan Sriboonlue (BOIDs and steering), Jaruwan Mesit (soft-body Space Blobs), Fabian Moncada (music and sound effects), John Martin (testing, balance, and additional server code), Derrick Janssen (additional design, testing, and game balance), Kristen Martin (additional database and

file processing), Eric Isles (additional music and sound effects), Gordon Hart (additional music and sound effects), Jonathan “Zarcath” Chan (additional design), FourTwoOmega (additional music), and JRWR (web stats and hosting).

I developed the GAR game engine with the Microsoft XNA Game Studio SDK available from <http://creators.xna.com>. I developed the GAR multi-player networking code with Lidgren Network Library by Michael Lidgren which is open source and available from <http://code.google.com/p/lidgren-network>. Both are invaluable free software libraries for quickly creating professional networked gaming or simulation applications.

Galactic Arms Race is available online at <http://gar.eecs.ucf.edu> and the project’s official email address is gar@eecs.ucf.edu.

Erin J. Hastings

University of Central Florida

June 2009

TABLE OF CONTENTS

LIST OF FIGURES	xii
LIST OF TABLES	xxxii
CHAPTER 1 INTRODUCTION	1
1.1 Motivation	1
1.2 Approach	2
1.3 Contributions	4
1.4 Dissertation Overview	4
CHAPTER 2 FOUNDATIONS	6
2.1 Evolutionary Computation (EC)	7
2.2 Interactive Evolutionary Computation (IEC)	8
2.3 Artificial Neural Networks (ANNs)	9
2.4 Compositional Pattern-Producing Networks (CPPNs)	11
2.5 NeuroEvolution of Augmenting Topologies (NEAT)	13

2.6	Machine Learning in Commercial Games	15
2.7	Evolving Game Content	16
2.8	Particle Systems	17
2.9	Conclusion	19
CHAPTER 3	NEAT PARTICLES AND NEAT PROJECTILES	20
3.1	Motivation	21
3.2	Approach - NEAT Particles	23
3.3	Particle System Representation	24
3.4	CPPN Implementation	26
3.5	Activation Functions	31
3.6	Physics	31
3.7	Rendering	32
3.8	Evolution	33
3.9	NEAT Projectiles	35
3.10	Projectile Classes	36
3.11	Projectile Constraint	36
3.12	Projectile CPPNs	37
CHAPTER 4	EXPERIMENTS IN EVOLVING PARTICLE SYSTEMS	40

4.1	Comparison to Hand-Coded Particle Systems	42
4.2	User Study	44
4.3	Performance	45
4.4	Discussion and Future Work	45
4.5	Conclusion	46
 CHAPTER 5 CONTENT GENERATING NEUROEVOLUTION OF AUG-		
MENTING TOPOLOGIES (CGNEAT)		51
5.1	Algorithm Overview	52
5.2	Unique Features	54
5.3	Starter Pool, Spawning Pool, and Archive Pool	55
5.4	Conclusion	56
 CHAPTER 6 GALACTIC ARMS RACE (GAR)		57
6.1	Development	58
6.2	Game Mechanics Overview	58
6.3	Game Interfaces	60
6.4	Spatial Hashing	77
6.5	Soft Body Simulation	80
6.6	Particle System Weapon CPPNs	81

6.7	Physics	81
6.8	Particle Weapon Rendering	82
6.9	Calculating Weapon Fitness	83
6.10	Evolving New Weapons	84
6.11	Starter Weapons, Spawning Pool, and Archive Pool	85
 CHAPTER 7 AUTOMATIC CONTENT GENERATION RESULTS IN GAR		
92		
7.1	Hypotheses, Experimental Setup, and Hypothesis Validation	93
7.2	Automatic Content Generation Results in GAR Single Player Mode	95
7.3	Automatic Content Generation Results in GAR Multiplayer Mode	97
7.4	Validating Hypotheses	107
 CHAPTER 8 DISCUSSION AND FUTURE OPPORTUNITIES		125
 CHAPTER 9 CONCLUSIONS		128
 LIST OF REFERENCES		130

LIST OF FIGURES

2.1	IEC Evolution Example.	In this example a spaceship is interactively evolved with DelphiNEAT-based Genetic Art (DNGA) [Fag05, Sta06, Sta07]. The initial spaceship-like image (a) is evolved from an initial population of random images. An intermediate stage of evolution (b) suggests a tail section, wing section, and nose section. (c) As evolution proceeds the components become more defined and interesting details become apparent. By the final stage (d), a spaceship model evolves with elegant lines, a nose section, and tail stabilizers. This sequence illustrates how complex digital art can be evolved by user preference.	10
-----	-------------------------------	--	----

2.2	ANNs and CPPNs. Unlike traditional ANNs (a), which typically only have sigmoid or Gaussian activation functions, CPPNs (b) may have sigmoids, Gaussians, and many other activation functions in the same network. These additional activation functions makes CPPNs encode variety more quickly and help them discover certain patterns that are difficult to obtain with standard ANNs (e.g. straight lines). Additionally, CPPNs are activated across an entire input range to represent a complete pattern. The ability of CPPNs to encode patterns makes them ideal for representing graphical content.	13
2.3	Illustrating Collaborative Content Evolution (CCE) in Games. The main idea in CCE is that content evolution in games begins with a diverse population of randomized content (left). As players explore the world and discover new content, they likely keep content with which they are satisfied and discard that with which they are not (center). As evolution continues, content that is widely disliked filters out of the game (right) and content that players enjoy becomes the parents of new generations of content. In this way, players continually explore a stream of changing content. Note that, at the time of writing, no published game has implemented CCE evolution except Galactic Arms Race, which is detailed in this dissertation.	18

3.1	Generation Shapes. A particle system’s generation shape defines the region in which new particles spawn. (a) Spherical generation produces area effects such as smoke and explosions. (b) Point generation facilitates effects that are attached to specific points on objects, such as vehicle thrust and muzzle flash. (c) Line generation commonly produces effects attached to characters or melee weapons, such as glowing swords. (d) Circular generation enables effects that surround objects, such as energy fields.	26
3.2	Particle System Classes. Predefined classes constrain the search space for designers. (a) The generic particle system models effects such as fire, smoke, and explosions. (b) The plane system warps and stretches individual particles for flashes, lens flares, and other effects. (c) The beam system simulates beam, laser, or electricity effects. (d) The rotator system models effects based on orbital rotation common in explosions, energy, and magic. (e) The trail system is similar to the generic system; however each particle drops a trail of smaller particles. Trail systems commonly implement magic, energy, weapon, and exhaust effects.	26

- 3.3 Update Function.** Every frame of animation, each particle passes through an update function to compute velocity and color for that frame. Suppose animation for a particle is being computed at frame t (on the right). The particle's position and distance from center in the previous frame $t - 1$ (on the left) are input into the particle system CPPN. After the CPPN is activated, its outputs are interpreted as velocity and color at frame t . The high frame rate of real-time animation produces small position changes; thus animation and color change is fluid. Over the long term; however, position changes are large, producing a variety of patterns and behaviors. 28
- 3.4 Particle System CPPNs.** To produce a specific range of effects, each particle class CPPN uses different inputs (i.e. position and distance from center) and outputs (i.e. velocity, color, and rotation), which are shown for (a) the generic and trail particle system, (b) the beam particle system, (c) the plane particle system, and (d) the rotator particle system. The beam system CPPN appears similar to the generic and trail system CPPNs; however a generic system CPPN controls individual particles, whereas a beam system CPPN controls Bezier curve control points. The plane system CPPN controls four corners of a warped quad, and the rotator system CPPN controls individual particle rotation. Each CPPN is evolved in NEAT Particles to connect the inputs to the outputs of each class. 29

3.5	CPPN Activation Functions. In NEAT Particles all CPPN hidden nodes and output nodes are randomly assigned one of eight activation functions: (a) sine, (b) cosine, (c) tangent, (d) bipolar sigmoid ($\frac{1-\exp(-x)}{1+\exp(-x)}$), (e) hyperbolic ($\frac{e^x - e^{-x}}{e^x + e^{-x}}$), (f) Gaussian ($[\frac{1}{\sqrt{0.5*PI}}] * e^{(-x^2)}$), (g) ramp ($x = \begin{cases} -1 & \text{if } (x < -1) \\ 1 & \text{if } (x > 1) \end{cases}$), or (h) step ($x = \begin{cases} -1 & \text{if } (x < 0) \\ 1 & \text{if } (x \geq 0) \end{cases}$).	31
3.6	NEAT Particles Interface. In the main interface (a), the user is presented with nine particle systems. System parameters such as generation shape and inputs are displayed on the bottom of the screen. In zoom mode (b), a single particle system and its CPPN can be inspected.	34
3.7	NEAT Projectiles Interface. In the main interface (a), the user chooses among nine projectile systems of any weapon class. The weapons can be rotated (as in this figure) and the refire rate adjusted to display their full behavior. In zoom mode (b), a single projectile system and its CPPN can be inspected.	35

3.8	Projectile System CPPNs.	Projectile models are designed to minimize undesirable behaviors (e.g. firing backward). (a) In the offset constrained model, projectile movement is constrained by an offset cone in front of the projectile. The offset cone is computed by adding two vectors <i>up-down</i> (o_{ud}) and <i>left-right</i> (o_{lr}). CPPN inputs include the current position of the particle, distance from center of the weapon, and a bias. The outputs are left-right offset, up-down offset, and color. (b) In the force constrained model, projectile motion is perturbed by an additional push force applied to the projectile after CPPN processing. Inputs are the current position of the particle, the distance from the system center, and a bias. The outputs are the particle velocity and color. Both models constrain projectile behavior while allowing sufficient evolutionary variety.	38
-----	---------------------------------	---	----

3.9 Projectile Constraint Mechanics. To ensure that weapons behave as projectiles, particle velocities are constrained to the 90° cones shown above. In the figure, W is the weapon and T is the target. (a) Dumb weapon particles are not target-aware so they are constrained in velocity to a fixed 90° cone in front of the weapon at the moment of discharge. (b) Directed weapon particles are not target-aware but may be influenced while in flight by the weapon. Thus directed particle velocity is constrained to the 90° cone in which the weapon is currently facing. (c) Smart particles are target-aware; therefore they are constrained to a 90° cone around a vector from the particle to the target. 39

4.1 Particle System Evolution. This figure illustrates evolution of a *flame shield* effect with NEAT Particles, in which red and yellow particles are evolved to orbit a player at the center. (a) Evolution begins with a ring shaped rotator system with an appropriate red and yellow color scheme. (b) After a few generations particles begin to detach from the ring. (c) Several generations later a prominent orbital behavior becomes apparent. (d) Evolution concludes with a full orbital pattern and a brighter color scheme, producing a convincing flame shield effect suitable for use in a video game. . 41

4.2	NEAT Projectiles Evolution.	A double-arc beam weapon that seeks a target is evolved with NEAT Projectiles. The weapon emits particles from the left side of each frame and the target is marked with a cross on the right side. (a) Evolution begins with a single arc that connects to the target. (b) After several generations, the beam begins to split in the middle. (c) Continuing evolution, the double arc becomes more pronounced. (d) Eventually the arcs become fully disjoint and the intended projectile behavior is achieved.	41
4.3	Evolved Particle Animations.	Key frames from animations of two evolved particle systems are presented in this example. Figures (a) through (d) depict an expanding vortex or explosion-like effect of an evolved plane system. Figures (e) through (h) depict a realistic billowing smoke cloud or explosion effect produced by an evolved trail system. Such effects illustrate that NEAT Particles can evolve effects appropriate for graphics and games.	42
4.4	Evolved Projectile Animations.	Key frames of evolved single projectiles are displayed above. The projectiles are fired from the left side of the screen towards the right side. The trailing lines mark motion over time. Figures (a) through (d) display a solid yellow projectile with a smooth curving behavior. Figures (e) through (h) depict a spiraling projectile that changes colors. Multiple projectiles evolved within the constraints combine to form complex weapon effects in NEAT Projectiles.	43

4.5	Sample Evolved Particle Systems. The images in this figure are single animation frames from effects evolved with NEAT Particles. These images demonstrate the variety of effects evolved through IEC.	48
4.6	Sample Evolved Projectile Systems. Single animation frames are shown of evolved NEAT Projectile systems fired from the left side of the screen toward targets on the right. The trailing lines plot motion over time. These images demonstrate the variety of weapon behaviors evolved by NEAT Projectiles.	49
4.7	Traditional Particle System Comparison. To compare IEC-generated particle systems to traditional ones, two hand-coded particle emitters were implemented within the same renderer as NEAT Particles. The expanding ring emitter (a) supports explosion effects and the simple ring emitter (b) can convey a variety of magical and force effects. Both systems display similar visual quality to NEAT Particles. However, they are capable of comparatively much less behavioral complexity. This example demonstrates the dependence on math and programming of traditional particle system implementations. .	50
4.8	User-Evolved Particle Effects. The images in this figure depict effects evolved with NEAT Particles by participants in the user study. The variety and complexity of these examples demonstrate that the IEC approach of NEAT Particles enables users to quickly evolve compelling particle system effects.	50

6.1	GAR Client 1.0. Players in GAR pilot their space ship (screen center) from a third-person perspective. This picture demonstrates a player destroying enemies with an evolved corkscrew-shaped weapon. Left of the player ship is a weapon pickup dropped from a destroyed enemy base. A particle system preview emits from the weapon pickup (i.e. “neuralium isotope,” left of player) to visually indicate how the weapon will function before the player picks it up. GAR is designed to look and feel like a near-commercial quality video game to effectively demonstrate the potential of automatic content generation in mainstream games. The GAR Client software is available online at http://gar.eecs.ucf.edu and runs on any Windows PC.	63
6.2	GAR Server 1.0. The GAR Servers 1.0 software provides a simple GUI through which users can host and monitor their own GAR game world with up to 32 players. At startup and periodically while running, GAR servers contact the GAR Master server to advertise the server name and IP. Players from around the world may then join the game server after retrieving the name and IP from the GAR Master Server. Each GAR Server has a separate persistent database for players that have joined that server; thus cgNEAT-evolved player weapons and progress are saved between online sessions. The GAR Server software is freely available online at http://gar.eecs.ucf.edu for those who wish to run their own game world on any Windows or Windows Server PC.	64

6.3	GAR Master Server 1.0. The GAR multiplayer mode enables players engage with up to 32 other players online. When joining multiplayer mode, GAR Clients first query the GAR Master Server for eligible GAR Servers. Players then select from a list of available servers to join a game. There is only a single GAR Master Server that indexes all GAR Servers, which is hosted by the Evolutionary Complexity Research Group at UCF.	65
6.4	GAR Multiplayer Architecture. This diagram illustrates the networked communication between the GAR Client, Server, and Master Server. (1) When GAR Servers are online, they periodically advertise to the Master Server which indexes all currently active Game Servers. (2) When a player wishes to join an Internet game, a list of all active game servers is retrieved from the Master Server. (3) The player then chooses from the list of active Internet games and connects to the desired Game Server.	66
6.5	GAR Main Menu. The main menu is displayed when GAR starts. It has an animated space scene and background music meant to convey the atmosphere of the game.	67

6.6	GAR Multiplayer Menu.	The GAR multiplayer menu enables players to connect to Internet or LAN games. Name and password are set so that game servers can retrieve online progress at a later date. A list of available multiplayer servers retrieved from the GAR Master Server is displayed. There is an additional button at the bottom that connects to a local host (i.e. when the GAR server is on the same machine as the GAR Client) or LAN game. .	68
6.7	GAR Head-Up Display (HUD).	The GAR HUD is always displayed while the player is in game. The lower left panel displays the player's current status. The numbered vertical panels at center-left display which of the player's three weapons is currently active. The center-right panel displays a portrait and information on the player's current target, and the lower-right panel is a radar showing objects in the local system. The seven small buttons above the lower panels give access to the other in-game screens described in this chapter. . .	69
6.8	GAR Weapons Screen.	The GAR weapons screen displays information on the player's three current weapons, including fitness and number of shots fired. The actual CPPNs for the weapons are displayed (known in the game as "neuralium isotopes"). A three-dimensional rendering scheme for the CPPNs displays all inner nodes in a circle, which eliminates the need for special methods to clearly display CPPN graphs in two dimensions.	70

6.9	GAR Ship Mods Screen.	GAR offers ships modifications to players for upgrading their ships. Ship mods include upgrades to armor, shields, hull, weapons, and teleport systems. Mods can be installed at any friendly station. Players earn one point for additional mod installs for each level they gain by defeating enemies and completing missions.	71
6.10	GAR Buy Ships Screen.	Players can access the buy ships screen by right-clicking on a friendly station. Stations in each system offer different ships, which are purchased with credits that players earn by defeating enemies and completing missions.	72
6.11	GAR Galaxy Map.	The map screen displays a three-dimensional graph of the GAR galaxy. Colors denote the faction that owns the system, and connections designate routes between systems. Players can travel between systems through a network of jump gates. Additional information displayed on the map screen includes system level, which indicates the relative challenge of the system, and the number of players currently in the system in multiplayer mode.	73

6.12	GAR Mission Screen. The player’s current mission objectives are shown on this screen. Missions award both experience towards level advancement and credits with which ship upgrades, ship mods, and new ships may be purchased. The early missions in GAR function as an in-game tutorial by directing the player to perform objectives integral to the game such as picking up new weapons, installing mods, buying ships, and using jump gates.	74
6.13	GAR Logoff Screen. The option is given on this screen to exit the current game. In multiplayer mode, the logoff screen displays a leaderboard with statistics on the highest ranked players currently online.	75
6.14	GAR Help Screen. The GAR help screen displays all keyboard and mouse commands along with helpful hints on game play.	76
6.15	GAR Spatial Hashing. The spatial hashing methods in GAR are based on two papers I co-authored [MG05, HMG04]. To optimize rendering, collision, picking, sound effects, particle special effects, and AI routines in GAR, each system in GAR is divided into grid cells called sectors (shown above). Every frame, all objects in the game are placed into grid cells by a hash function. This spatial hashing routine optimizes nearly all aspects of the GAR game engine, including rendering, collision, NPC decision routines, picking, sound effects, and special visual effects.	87

6.16	Space Blob.	The “Space Blob” in GAR is a large green blob-like enemy creature that is animated and rendered based on a real-time soft body modeling techniques outlined in a paper I co-authored ([MHG06]). The space blob functions as a powerful “boss” enemy like those commonly seen in other games.	88
6.17	How CPPNs Represent Particle Weapons.	(a) Each frame of animation, each particle separately inputs the position (p_x, p_z) and distance (d_c) from where it was <i>initially</i> fired into the CPPN (p_y is ignored because the game is situated entirely on the $y = 0$ plane). (b) The CPPN is activated and particle velocity (v_x, v_z) and color (r, g, b) are obtained from CPPN outputs. This method provides GAR with smooth particle animations and a wide variety of possible evolved weapons.	89
6.18	Weapon Evolution Examples.	As weapons evolve over the course of the game, players are likely to find weapons with qualities similar to those they favored in the past. In this example from actual single-player gameplay, the player often fired a spread weapon (a). Later in the game, new spread gun variations (b,c) evolved. Another interesting spread gun (d) fires two slower-firing outer projectiles and a fast inner projectile. Later descendants of this weapon (e,f) exaggerated the speed difference between the inner and outer projectiles, diversified the color pattern, and modified the spread width. These examples illustrate how cgNEAT evolves novel content based on past user preferences.	90

6.19	GAR Starter Weapons. When the game begins, players are equipped with straight-shooting starter weapons that do not contribute to evolution. Starter weapons ensure that players begin the game with effective weapons (which would not be guaranteed by randomization) and additionally act as a control to which the effectiveness of evolved weapons is compared. Starter weapons do earn fitness by being fired; however, if a starter weapon is selected by a roulette roll during evolutionary selection, a spawning pool weapon is created instead.	91
7.1	Weapons Evolved During Single Player Gameplay. GAR players discovered many useful and aesthetically pleasing weapons. The number of generations of reproduction taken to evolve each weapon is shown next to its name. The <i>multispeed</i> (a) fires two slow outer projectiles, which are useful for blocking incoming enemy fire, and a fast center projectile for quickly striking distant targets. The <i>ultrawide</i> (b) and <i>three prong</i> (c) emit wide particle patterns that are effective for fighting many enemies at once. The <i>corkscrew</i> (d) emits a pattern that is initially wide, for blocking, but later converges for concentrated damage at a distance. Two version of the <i>ladder gun</i> (e,f) fire a wide wave-like pattern that can swivel around obstacles like asteroids. Additionally results are presented in figure 7.2.	98

7.2	Additional Weapons Evolved During Single Player Gameplay. This figure displays additional results to those presented in figure 7.1. The <i>double bolt</i> (a) demonstrates that weapons similar to those in typical space shooters can evolve. The <i>trident</i> (b) launches a single projectile forward and two perpendicular projectiles that can block enemy fire from the sides. <i>Subatomic heat</i> (c) fires a chaotic multi-colored stream resembling bouncing subatomic particles. Two types of <i>wallmaker</i> (d,e) literally create defensive walls of particles in front of the player. The <i>tunnelmaker</i> (f) creates a defensive line of particles as well, but on both sides of the player, yielding a defensive sheath. These results demonstrate the ability of cgNEAT to generate a tactically and aesthetically diverse and genuinely useful array of weapons. Furthermore, useful weapons appear in early generations and continue to elaborate over successive generations.	99
7.3	Corkscrew Weapon Trend. Corkscrew guns fire twisting corkscrew patterns that are effective for blocking and travel quickly for hitting distant targets.	106
7.4	Double Shot Weapon Trend. Double shot weapons fire two parallel shots, demonstrating that weapons similar to those in typical space shooters can evolve in cgNEAT. Double shot weapons travel quickly and are effective for hitting distant targets.	107

7.5	Fork Weapon Trend. Fork guns fire a wide triple-shot that is effective at firing into crowds of enemies or for hitting distant fast-moving enemies. The projectiles issues from fork gun (d) alternately spread widely and converge toward the center.	108
7.6	Goop Weapon Trend. Goop guns drop animated clouds of particles resembling liquids. They create effective “space mines” that can block incoming bullets and that can be dropped as obstacles while fleeing.	109
7.7	Multispeed Weapon Trend 1. Multispeed guns, which proved highly popular on the test server, have a fast center projectile and slower outer projectiles that move in a variety of patterns. Examples (a) and (d) are also called “tunnelmakers” because they create a defensive tunnel of near-stationary particles.	111
7.8	Multispeed Weapon Trend 2. This figure presents additional Multispeed guns. Example (a) has traits of both multispeed and “wallmakers”; since it acts generally like a multispeed, but creates an intermittent wall of particles at a fixed distance from the player.	112
7.9	Plasma Weapon Trend 1. Plasma guns, which also were popular on the server, fire chaotic streams of colorful particles resembling plasma. Plasma guns fire fast and erratic particles that are excellent for blocking incoming projectiles and are difficult for other players to dodge in PVP mode.	113
7.10	Plasma Weapon Trend 2. This figure presents additional Plasma weapons.	114

7.11	Shield Weapon Trend. Shield guns are excellent defensive weapons that create a particle shield completely encasing the player ship.	115
7.12	Spread Weapon Trend. Spread guns fire a tight stream that widens as it travels; thus these weapons deliver concentrated fire at close range but spread later to make distant targets easier to hit. Some spread guns (b) are also called <i>tunnelmakers</i> because of the lines of stationary particles created on either side of the player ship.	116
7.13	Squiggle Weapon Trend. Squiggle guns create diverse curved patterns that resemble hand-written script. Squiggle guns display some of the unique color and shape patterns possible through cgNEAT weapon evolution.	117
7.14	Triple Weapon Trend 1. Triple shot weapons fire straight patterns with various widths and colors. These weapons demonstrate that GAR, in addition to producing exotic weapons, is also capable of creating weapons similar to those found in typical space shooters.	118
7.15	Triple Weapon Trend 2. This figure presents additional triple shot weapons.	119
7.16	Vortex Weapon Trend. Vortex weapons produce spinning patterns similar to wind tornadoes. The seemingly random wide patterns are effective for blocking incoming projectiles and make it easy to hit targets.	120

7.17 Wall Gun Weapon Trend.	Wall guns create a literal wall of defensive particles in front of the player, useful for blocking or dropping behind while fleeing. Some wall guns such as (b) and (d) create multiple lines of particles.	121
7.18 Zig Zag Weapon Trend.	Zigzag guns produce jagged linear patterns, often in very wide formations, making them effective for both blocking and hitting targets.	122
7.19 Miscellaneous Evolved Weapons.	These weapons from the GAR server archive are not easily categorized. (a) The <i>slime gun</i> fires a large green mass resembling slime. (b) The <i>rainbow gun</i> issues colorful curving arcs. (c) The <i>stealth gun</i> produces a very faint projectiles. (Note that, due to the rendering methods in GAR, it is impossible to produce completely invisible projectiles.) (d) Several <i>mine layer</i> weapons proved popular to drop as traps for enemies pursuing the player.	123
7.20 PVP Archive Weapons.	This example displays PVP archive weapons with which players scored PVP kills on the GAR Official 32-player server. Overall, the popular PVP weapons display as much variety as those used solely against NPCs; thus players employed many evolved weapon tactics to defeat each other.	124

LIST OF TABLES

7.1	Official Server Player Accounts. A summary of a snapshot of player accounts on the GAR 1.1 Official 32-player server taken on July 30th, 2009 is shown. Over 73% of players progressed past level 20, indicating significant time invested. A large number of players progressed to higher levels, which could take several days in-game, suggesting that the evolving content mechanic enhances replay value.	102
7.2	Official Server Kill Counts. Aggregate kill counts and the maximum kill counts for a single player, for the 1,007 player accounts on the GAR 1.1 Official 32-player server snapshot taken on July 30th, 2009, are shown. Such significant totals demonstrate the playability of the weapons evolved by cgNEAT in GAR.	103

7.3 Official Server Weapon Evolution. Aggregate weapon evolution data is shown for the GAR 1.1 Official 32-player server snapshot taken on July 30th, 2009. Given that players fired over 23.6 million shots and looted over 130,000 weapons (about one third of weapons evolved), by sheer volume (and the relatively low number of starter gun kills) it can be inferred that players found the evolving weapon mechanic engaging, and the weapons useful against NPCs and other players. The average weapon generation of 16 suggests that effective weapons can be obtained quickly, and the maximum generation of 98 indicates that weapons continue to be effective many generations later. . . . 104

CHAPTER 1

INTRODUCTION

Creating the models, levels, textures, and other content that players encounter and possess in games is time consuming and expensive [Edw06, Irw08]. In part to address this problem and to provide additional replay value, it is increasingly popular for developers to distribute tools that enable players to create their own content [Sof07b, Gam07, Sof07a] or to randomize content (e.g. random map generators). However, content creation tools usually require significant effort to master and specialized knowledge beyond that of most players. Moreover, *content randomization* only works if it is tightly constrained to avoid generating undesirable content, and provides no means to deduce the kind of content that players prefer. Thus a more intriguing potential solution is to automatically generate content during the game, as it is played, based on actual player behavior.

1.1 Motivation

Evolutionary computation (EC) methods have proven effective at evolving diverse media such as two-dimensional art images [Fag05, SBD08], simple three-dimensional forms [HGM96,

NTC01], and even music [HS09]. However, the feasibility of such methods for creating video game content is little explored. Currently, machine learning techniques are beginning to be applied in mainstream games, but only for non-player character (NPC) controllers [SBM05].

The proven ability of evolutionary methods to evolve media and NPC controllers provides the basis for the idea of Collaborative Content Evolution (CCE) for games, which is first realized in this dissertation. The idea of CCE is that the system automatically generates graphical and game content that is extended from and elaborates upon content players preferred in the past. Content that players like (i.e. use often) is evolved to produce new content, whereas content that players dislike is not evolved. Thus, a continuous stream of novel content is produced that potentially (1) keeps players engaged longer and (2) reduces the content creation burden on developers. The first implementation of CCE is in the Galactic Arms Race video game, described in this dissertation.

1.2 Approach

To make such content generation possible, this dissertation presents three novel technologies: (1) NEAT Particles and NEAT Projectiles, (2) the *content-generating NeuroEvolution of Augmenting Topologies* (cgNEAT) algorithm, and (3) Galactic Arms Race (GAR), a multiplayer video game in which novel weapons are automatically evolved based on content users preferred in the past.

NEAT Particles and NEAT Projectiles are both interactive systems for evolving complex particle systems. NEAT Particles enables users to evolve general purpose particle system effects, whereas NEAT Projectiles is specialized to evolve experimental particle weapon effects for video games. Techniques developed in both systems pave the way for automatic content generation through cgNEAT and GAR.

In particular, the content-generating NeuroEvolution of Augmenting Topologies algorithm aims to automatically generate complex graphic and game content. This approach creates new content in real-time through an evolutionary algorithm based on the content players liked in the past.

To show that automatic content generation is genuinely possible in mainstream games, cgNEAT is implemented in this dissertation in a novel video game called *Galactic Arms Race*. In GAR, *compositional pattern producing networks* (CPPNs), which are a variant of artificial neural network (ANNs), genetically encode and control particle system weapons. The CPPNs evolve and increase in complexity through cgNEAT, which tracks which weapons the player fires the most. That way, during the game, weapon behavior becomes increasingly sophisticated while consistently evolving to suit player tastes. Thus, it is the *player* rather than the designer who ultimately implicitly determines what kind of content will populate the game.

1.3 Contributions

This dissertation achieves several significant results: First, it shows how particle systems can be evolved, thereby creating a new evolvable class of content. Second, it validates the first algorithm (cgNEAT) for evolving content in games. The GAR application domain shows that it is indeed possible to create games in which players discover a wide variety of automatically generated content that is not only novel, but also based on and extended from previous content that they liked in the past. GAR, the first game in which content evolves as the game is played, is the third major contribution.

For developers, these contributions mean that it is possible to produce games and simulations that create their own content to satisfy users, impacting both the production cost and longevity of future such games. While the evolved content in GAR is the weapons, in principle cgNEAT can evolve any class of content in the same way, opening up an exciting new direction in video game design.

1.4 Dissertation Overview

The dissertation will proceed as follows. Chapter 2 covers background material related to the technologies developed in this dissertation. NEAT Particles and NEAT Projectiles and experiments in interactive particle system evolution are discussed in Chapters 3 and 4. Next, the cgNEAT algorithm is described in detail in Chapter 5. Then, chapter 6

demonstrates cgNEAT in practice through evolving weapons in the Galactic Arms Race video game, and the outcome of both single player and multiplayer weapon evolution experiments are described in Chapter 7. Finally, overall implications are discussed and the dissertation is concluded in Chapters 8 and 9.

CHAPTER 2

FOUNDATIONS

This chapter reviews background material that inspired and contributed to the main contributions of this dissertation (i.e. NEAT Particles, cgNEAT, and GAR). First, evolutionary computation (EC) and interactive evolutionary computation (IEC), which are the general approaches toward content evolution in this work, are explained. Next, artificial neural networks (ANNs) and compositional pattern-producing networks (CPPNs), which are the specific approach to content representation in this dissertation, are compared. Then, the NeuroEvolution of Augmenting Topologies (NEAT) method is described, which is the foundation upon which cgNEAT is built. Next, existing commercial games that implement machine learning and existing research on content evolution in games are reviewed. Finally, particle systems, which are the type of content evolved in NEAT Particles and in GAR, are reviewed.

2.1 Evolutionary Computation (EC)

Evolutionary computation (EC) is a set of methods for solving complex problems inspired by the mechanics of *natural evolution* [Jon06]. Several important classes of evolutionary algorithms are *classifier systems* [Hol86], *evolutionary programming* [FOW66], *evolution strategies* [BS02], *genetic algorithms* [Bar57], and *genetic programming* [Cra85].

Despite differences, all such classes of evolutionary algorithms share the common mechanics of simulating the evolution of candidate solutions through the processes of *selection*, *mutation*, and *reproduction*. Generally, the performance, or *fitness*, of individual structures is determined by a *fitness function*. One or more structures are then selected for reproduction based upon fitness, producing recombined and mutated offspring that are potentially more effective at solving the problem. In effect, the evolutionary algorithm process searches a solution space for optimal solutions.

Applied EC methods have effectively solved complex problems in such diverse domains as radio frequency antenna design and configuration [LLH04], robotics [Fon07], Internet search engine optimization [Gal08], and subatomic particle physics [Aal09]. However, there exist problems for which fitness functions that are intended to be automatically evaluated by a computer cannot be easily created. Such problems have inspired an *interactive* form of EC, which relies on *human* guidance to evaluate fitness, as described in the next section.

2.2 Interactive Evolutionary Computation (IEC)

IEC is an approach to evolutionary computation (EC) in which human evaluation replaces the fitness function [Tak01]. A typical IEC application presents to the user the current generation of content. The user then interactively determines which members of the population will reproduce and the IEC application automatically generates the next generation of content based on the user's input. Through repeated rounds of content generation and fitness assignment, IEC enables unique content to evolve that suits the user's preferences. In some cases such content cannot be discovered or created in any other way.

IEC aids especially in evolving content for which fitness functions would be difficult or impossible to formalize (e.g. for aesthetic appeal). Thus, graphical content generation is a common application of IEC [Daw86, TL99, Une94, NTC01, HGM96, Fag05].

IEC was first introduced in Biomorphs, which aims to illustrate theories about natural evolution [Daw86]. Biomorphs are patterns encoded as *Lindenmayer Systems* (L-systems) [Lin68], i.e. grammars that specify the order in which a set of replacement rules are carried out. Abstract figures that resemble animals or plants are interactively evolved in this manner.

Representations in *genetic art* (i.e. IEC applied to art) often vary, including linear or non-linear functions, fractals, and automata. Some notable examples include (1) Mutator [TL99], a cartoon and facial animation system, (2) SBART [Une94], a two-dimensional art exploration tool, (3) a tool that evolves implicit surface models such as fruits and pots

[NTC01, NTC00a, NTC00b], and (4) a system for evolving quadric models used as machine components [HGM96].

Figure 2.1 illustrates IEC’s content generation capabilities. The figure shows a progression of four user-selected parents in the evolution of a spaceship with a genetic art tool [Fag05, Sta06, Sta07]. In the example, the user starts by selecting a simple image that vaguely resembles what they wish to create and continues to evolve more complex images through selection until satisfied with the result. The sequence of images demonstrates the potential of IEC as an engine for content generation. These images, from Delphi NEAT Genetic Art (DNGA [Fag05]), are produced by ANNs evolved by NEAT, which is discussed in the next section.

IEC is demonstrably effective for evolving certain types of graphical content. An important prerequisite to evolving graphical content is that it must be represented by evolvable structures. The evolvable structures utilized in this dissertation are CPPNs, a specialized type of ANN; both are discussed in the following sections.

2.3 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational structures inspired by neurons in the human brain [Hay99]. Theoretically, ANNs can approximate any function [Cyb89] making them powerful tools for control and prediction.

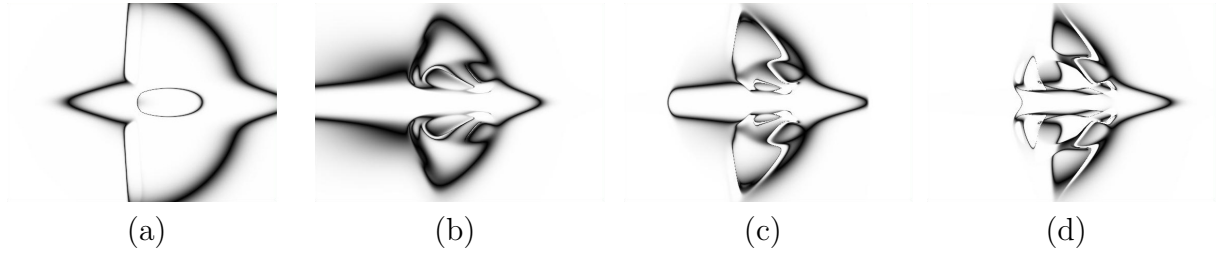


Figure 2.1: **IEC Evolution Example.** In this example a spaceship is interactively evolved with DelphiNEAT-based Genetic Art (DNGA) [Fag05, Sta06, Sta07]. The initial spaceship-like image (a) is evolved from an initial population of random images. An intermediate stage of evolution (b) suggests a tail section, wing section, and nose section. (c) As evolution proceeds the components become more defined and interesting details become apparent. By the final stage (d), a spaceship model evolves with elegant lines, a nose section, and tail stabilizers. This sequence illustrates how complex digital art can be evolved by user preference.

ANNs are composed of *nodes* joined by *weighted connections* that may be arranged into *layers* (figure 2.2a). Two special types of nodes are designated as *inputs* and *outputs*. All other nodes, besides inputs and outputs, are called *hidden nodes*. The arrangement of nodes and connections in an ANN may be *feedforward*, in which no loops exist in the ANN graph, or *recurrent*, in which nodes may have connections to themselves or to nodes from the same layer or lower in the ANN, forming a loop. While feedforward and recurrent network topologies offer distinct advantages in certain problem domains [Hay99], the ANNs utilized in the techniques in this dissertation are strictly feedforward.

Each node in the ANN is associated with an *activation function*, which is typically sigmoid or Gaussian. During *activation* data is processed through the ANN. First, inputs are loaded with data. Then, at each node, the weighted sum of all incoming connections is computed, and the node activation function is applied. Finally the output nodes are read, yielding a possible solution to the problem being solved. Note that, in multi-layer ANNs, it may be

necessary to activate a number of times equal to the *depth* of the ANN graph to ensure that data flows fully from the inputs to the outputs. Note also that successive activations can cause the activation levels of the outputs to fluctuate. If successive activations cease to cause output fluctuation, the ANN has reached a stable state.

When solving complex problems, hand-coding ANN topology can be infeasible. In such cases, an EC method known as *neuroevolution* can be employed in which ANN topologies and weights are evolved during the search of the solution space. Specifically, the ANN evolution techniques in this dissertation are based on the NeuroEvolution of Augmenting Topologies (NEAT) [SM02, SBM05] method discussed in detail in section 2.5.

ANNs are proven effective evolvable structures for evolutionary machine learning in a wide array of domains including financial market analysis [Ska96], agent control [SBM05], digital art [SBD08], and particle physics [Aal09]. Specialized types of ANNs called *compositional pattern-producing networks* are implemented in the approach in this dissertation, as discussed in the next section.

2.4 Compositional Pattern-Producing Networks (CPPNs)

Compositional pattern-producing networks (CPPNs) are a variation of ANNs that differ in their set of activation functions and how they are applied [Sta06, Sta07] (figure 2.2). While CPPNs are similar to ANNs, the different terminology originated because CPPNs

were introduced as pattern-generators rather than as controllers. This section explains the difference in implementation and application between CPPNs and ANNs.

While ANNs often contain only sigmoid or Gaussian activation functions, CPPNs can include both such functions and many others. The choice of CPPN functions can be biased toward specific patterns or regularities. For example, periodic functions such as sine produce segmented patterns with repetitions, while symmetric functions such as Gaussian produce symmetric patterns. Linear functions can be employed to produce patterns with straight lines. In this way, CPPN-based systems can be biased toward desired types of patterns by carefully selecting the set of available activation functions.

Additionally, unlike typical ANNs, CPPNs are usually applied across a broad space of possible inputs so that they can represent a complete image or pattern. Because they are compositions of functions, CPPNs in effect encode patterns at infinite resolution and can be sampled at whatever resolution is desired.

Successful CPPN-based applications such as Picbreeder [SBD08], in which users from around the Internet collaborate to evolve pictures, and NEAT Drummer [HS09], which evolves drum track patterns to accompany songs, demonstrate that CPPNs can evolve diverse content. The approach in this dissertation evolves particle systems encoded by CPPNs with the NEAT algorithm, which is discussed next.

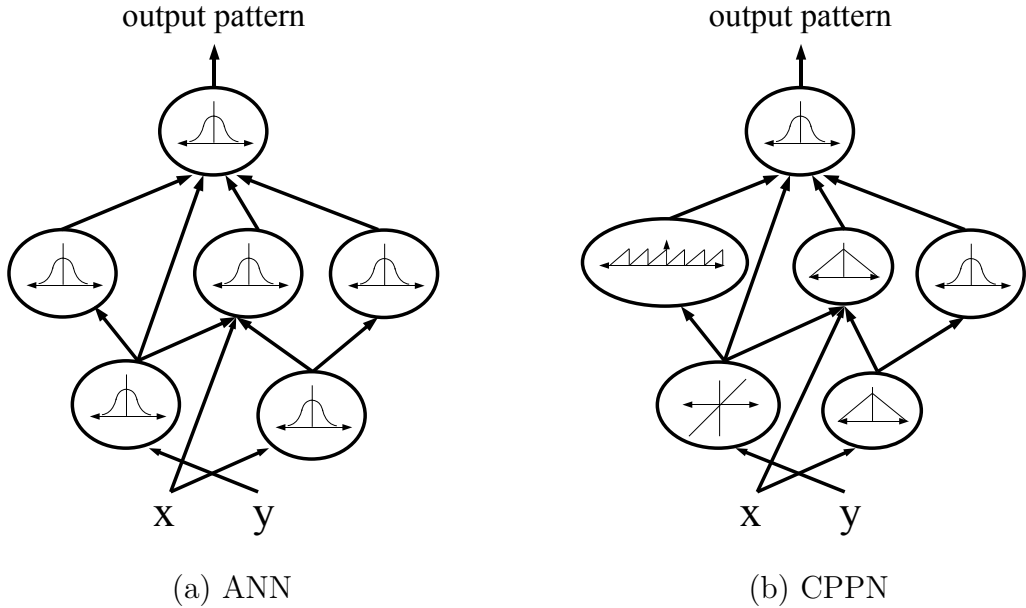


Figure 2.2: **ANNs and CPPNs.** Unlike traditional ANNs (a), which typically only have sigmoid or Gaussian activation functions, CPPNs (b) may have sigmoids, Gaussians, and many other activation functions in the same network. These additional activation functions makes CPPNs encode variety more quickly and help them discover certain patterns that are difficult to obtain with standard ANNs (e.g. straight lines). Additionally, CPPNs are activated across an entire input range to represent a complete pattern. The ability of CPPNs to encode patterns makes them ideal for representing graphical content.

2.5 NeuroEvolution of Augmenting Topologies (NEAT)

The NEAT method was originally developed to solve control and sequential decision tasks. The ANNs evolved with NEAT control agents that select actions based on their sensory inputs. While previous methods that evolved ANNs (i.e. neuroevolution methods) evolved either fixed topology networks [GM99, SF95], or arbitrary random-topology networks [GWP96, ZM93, Yao99], NEAT begins evolution with a population of small, simple networks and *complexifies* the network topology into diverse species over generations, leading to increasingly sophisticated behavior. A similar process of gradually adding new genes has been confirmed

in natural evolution [Mar99, WHR87] and shown to improve adaptation in a few prior evolutionary [Alt94] and neuroevolutionary [Har93] approaches. This section briefly reviews the NEAT method; Stanley and Miikkulainen [SM02, SBM05] provide complete introductions.

To keep track of which gene is which while new genes are added, a *historical marking* is uniquely assigned to each new structural component. During crossover, genes with the same historical markings are aligned, producing meaningful offspring efficiently. Traditionally, speciation in NEAT protects new structural innovations by reducing competition between differing structures and network complexities. However, in the work in this dissertation, because a human performs selection rather than an automated process, the usual speciation procedure in NEAT is unnecessary.

Most importantly, complexification, which resembles how genes are added over the course of natural evolution [Mar99], allows NEAT to establish high-level features early in evolution and then later elaborate on them. For evolving content, complexification means that content can become more elaborate and intricate over generations.

In this dissertation, particle system weapons are controlled by CPPNs evolved by NEAT. NEAT is chosen because (1) it is proven effective for evolving ANNs and CPPNs in a diversity of domains [SM02, TWS06, SBD08, Aal09], and (2) it is fast enough to run in real time (in the NERO video game [SBM05]), which is required for an interactive system. Because NEAT is a strong method for evolving controllers for dynamic physical systems, it can naturally be extended to evolve the motion of particle effects as well, such as those featured in GAR.

The next section explains the limited impact so far of machine learning in commercial games, and how a game called NERO shows that real-time evolutionary learning in a game is possible.

2.6 Machine Learning in Commercial Games

The impact of machine learning so far on the video game industry has been limited, although some games are beginning to incorporate learning techniques. However, content generation continues to be absent from applications of machine learning in commercial game. The most common application of machine learning is to optimize the policy that controls non-player characters (NPCs). For example, the ANN race car controllers Colin McRae Rally 2.0¹ and Forza Motorsport 2², and the creature brains in Creatures 3³ and Black and White 2⁴ are learned. Generally, the NPC behavior in such games is trained by developers before release. Recently, although it is not a commercial game, NeuroEvolving Robotic Operatives (NERO [SBM05]; <http://nerogame.org/>) enabled players to evolve the tactics for a squad of virtual soldiers in real-time, while the game is played, demonstrating the potential viability of evolution in commercial gaming.

The success of learning algorithms in these games suggests the potential to apply learning to create content beyond NPC behavior. In fact, automatically generating content could

¹Copyright 2001 Codemasters, <http://www.codemasters.com/>

²Copyright 2007 Microsoft Game Studios, <http://forzamotorsport.net/>

³Copyright 2004 Creature Labs, <http://www.gamewaredevelopment.co.uk/>

⁴Copyright 2005 Lionhead Studios, <http://www.lionhead.com/>

further open the video game industry to the possibilities created by machine learning. The next section discusses the few existing published methods that evolve game content.

2.7 Evolving Game Content

Evolving game content is an emerging research area with great potential to contribute to the mainstream gaming industry. Two of the few current examples of evolved game content include race tracks [TNL07] and even the rules of the game itself [TNL08].

To evolve the race track itself [TNL07], ANN drivers are evolved by comparing their performance to human controllers on the same track. Then new tracks are generated and the ANN controllers are evaluated on those tracks. The selected tracks are those upon which the ANN controllers perform similarly to how they perform on other tracks, under the assumption that they will provide an appropriate level of challenge to human players.

In a different example of evolving game content [TNL08], the rules of the game itself, rather than NPC controllers that play a specific game well, are evolved. Evolution begins with a grid-like environment containing random sets of walls, an ANN-controlled NPC representing the player, and various game objects that can alter state when the ANN-controlled NPC collides with them. State changes include death, teleportation, bonuses, goals, and other typical two-dimensional game effects. The fitness of the games created is based upon

the amount of learning the controller must undergo in order to beat the game. This idea was shown to successfully evolve *Pac-Man*-like games.

These investigations thus represent the cutting edge of an exciting new research direction. However, in these examples content is evolved *outside* the game; there currently exists no game that evolves novel content based on usage statistics as the game is played. The aim of cgNEAT is thus to evolve such content in real time, based on tracked player preferences by a process called collaborative content evolution (CCE), as illustrated in figure 2.3.

CCE is a new concept that has not, until this dissertation, been implemented in a game. In short, players begin the game with an initial set of content. If players use content often, it is inferred that they enjoy that content, and the game produces new content that extends from or elaborates on that preferred content. However, if players are unhappy with certain content, they will not use it (or may discard it); thus the game will not produce more content of that type. The aim is that the game creates a stream of evolving content based on the preferences of the players. The main type of content evolved in this dissertation is three-dimensional particle systems, which are described in the next section.

2.8 Particle Systems

The first computer-generated particle system in commercial computer graphics, called the *Genesis Effect*, appeared in Star Trek II: The Wrath of Khan¹ [Ree83]. Soon after, particle

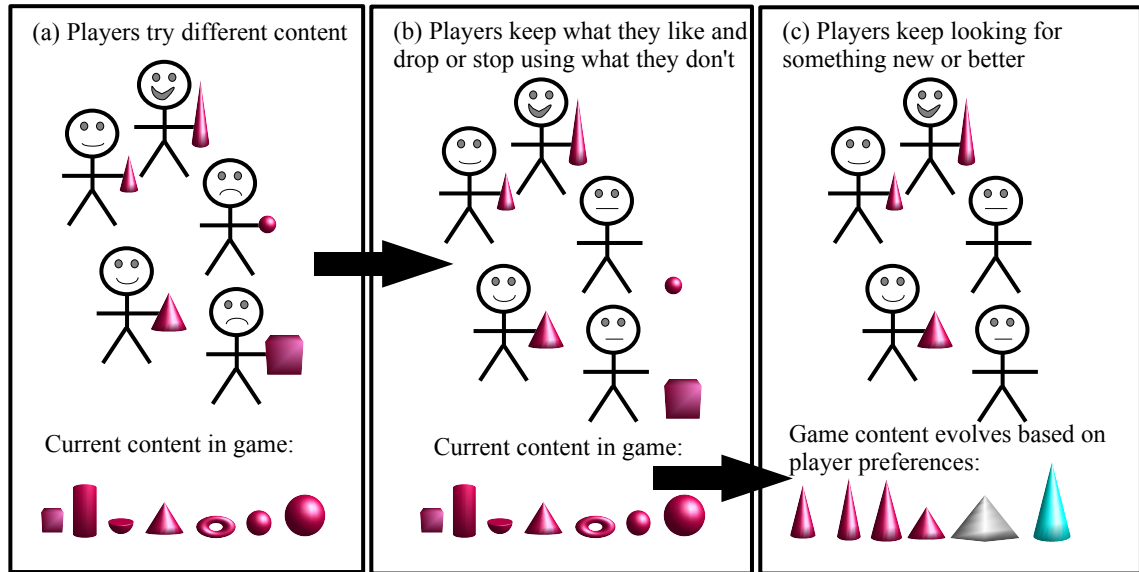


Figure 2.3: **Illustrating Collaborative Content Evolution (CCE) in Games.** The main idea in CCE is that content evolution in games begins with a diverse population of randomized content (left). As players explore the world and discover new content, they likely keep content with which they are satisfied and discard that with which they are not (center). As evolution continues, content that is widely disliked filters out of the game (right) and content that players enjoy becomes the parents of new generations of content. In this way, players continually explore a stream of changing content. Note that, at the time of writing, no published game has implemented CCE evolution except *Galactic Arms Race*, which is detailed in this dissertation.

systems effects became widespread on television as well. Nearly all modern video games include a particle system engine [Lan97, Ber00]; special effects in games such as magical spells and futuristic weapons are usually implemented with particle systems.

In addition to diffuse phenomena such as fire, smoke, and explosions, particle systems can also model concrete objects such as dense trees in a forest [Ree85], folded cloth and fabric [Bre94, EWS96], and simulated fluid motion [OFL01, MCG03]. Realistic particle movement is often achieved by simulating real-world physics [Rey99]. At a more abstract level, particle

systems can simulate animal and insect flocking as well as swarming behavior [Rey87]. The prevalence and diversity of particle system applications demonstrates their importance to computer graphics in modern media and games.

2.9 Conclusion

So far, research in automated content generation has been applied mainly to graphics and media outside of games. Aside from NPC controllers, little machine learning technology has been applied to intelligently create content in games based on player preference. The aim of this dissertation is to show that automated content generation is possible in mainstream games in real-time, as a game is played.

The remainder of this dissertation discusses three works that establish that automated content generation is feasible. First, NEAT Particles, a tool that enables users to interactively evolve complex three-dimensional particle systems, is presented. Next, content-generating NeuroEvolution of Augmenting Topologies, a method created explicitly for automated content generation based on user preference, is detailed. Finally, Galactic Arms Race, a multiplayer video game that applies cgNEAT to automatically generate unique based on past player preferences, is introduced. Galactic Arms Race is the first video game to feature CCE.

CHAPTER 3

NEAT PARTICLES AND NEAT PROJECTILES

IEC creates the intriguing possibility that a large variety of useful content can be produced quickly and easily for practical computer graphics and gaming applications. To show that IEC can produce such content, NEAT Particles and NEAT Projectiles apply IEC to particle system effects, which are the de facto method in computer graphics for generating fire, smoke, explosions, electricity, water, and many other special effects. The techniques developed in NEAT Particles and NEAT Projectiles are published in [HGS07] and [HGS09a] and serve as a stepping stone to the evolving weapons technology in Galactic Arms Race introduced in Chapter 6.

While particle systems are capable of producing a broad array of effects, they require substantial mathematical and programming knowledge to produce. Therefore, efficient particle system generation tools are required for content developers to produce special effects in a timely manner. This chapter details the design, representation, and animation of particle systems via two IEC tools called NEAT Particles and NEAT Projectiles. Both tools evolve CPPNs with the NeuroEvolution of Augmenting Topologies (NEAT) method to control the behavior of particles. NEAT Particles evolves general-purpose particle effects, whereas NEAT Projectiles specializes in evolving particle weapon effects for video games. The primary ad-

vantage of this NEAT-based IEC approach is to decouple the creation of new effects from mathematics and programming, enabling content developers without programming knowledge to produce complex effects.

3.1 Motivation

For media developers, content generation consumes significant time and money to produce today's complex graphics and game content [Sti06, Lom04]. In part to address this problem, in the video game industry, it is becoming increasingly popular to provide extensive character customization tools within games and to distribute tools that allow users to create their own content outside of the game as well [Sof07b, Gam07, Sof07a]. Furthermore, there is a new trend towards *content generation tools as games themselves*, that is, *sandbox games* such as The Sims¹, Second Life², and Spore³. These games feature creating houses, vehicles, clothing, and creatures as primary game play features [Ent07]. Thus, there is a growing need for powerful and user-friendly content generation tools both to reduce the content bottleneck and further empower users.

Particle systems are ubiquitous in computer graphics for producing animated effects such as fire, smoke, clouds, gunfire, water, cloth, explosions, magic, lighting, electricity, flocking, and many others [Lan97, Ber00]. They are defined by (1) a set of points in space and (2) a set

¹Copyright 2007 Electronic Arts, <http://thesims.ea.com/>

²Copyright 2003 Linden Research Inc., <http://secondlife.com/>

³Copyright 2007 Electronic Arts, <http://www.spore.com/>

of rules guiding their behavior and appearance, e.g. velocity, color, size, shape, transparency, rotation, etc.

Because such rule sets are often complex, creating each new effect requires considerable mathematics and programming knowledge. For example, consider designing a *a spherical flame shield of pulsing colors* effect for a futuristic video game or movie. Alternatively, consider designing a *particle weapon effect that fires multiple curving arcs toward the target*. In current practice, the precise mechanics for either scenario must be hand coded by a programmer. To simplify design, particle effect packages typically provide developers with a set of particle system classes, each suitable for a certain type of effect. Content developers manipulate the parameters of each particle system class by hand to produce the desired effect. The problem is that there is no way to efficiently explore the range of effects within each class.

To address this problem, this dissertation presents a new design, representation, and animation approach for particle systems in which (1) CPPNs control particle system behavior, (2) the *NeuroEvolution of Augmenting Topologies* (NEAT) method [SM02, SM04] produces sophisticated particle system behaviors by evolving increasingly complex CPPNs, and (3) evolution is guided by user preference through an IEC interface.

Two prototype systems are discussed, NEAT Particles, a general-purpose particle effect generator, and NEAT Projectiles, which is specialized to evolve particle weapon effects for video games. Both systems interactively evolve CPPNs with NEAT to control the motion

and appearance of particles. An IEC interface provides a user-friendly method to evolve unique content.

In this way, NEAT Particles shows how IEC can enable practical content generation that provides an easy alternative to current, potentially cumbersome practice. In particular, NEAT Particles and NEAT Projectiles (1) enable users without programming or artistic skill to evolve unique particle system effects through a simple interface, (2) allow developers to evolve a broad range of effects within each particle class, and (3) serve as concept generators, enabling novel effect types to be easily discovered. By allowing users to evolve particle behavior without knowledge of physics or programming, NEAT Particles and NEAT Projectiles are a step toward the larger goal of automated content generation for games, which is realized in the Galactic Arms Race video game, introduced later.

3.2 Approach - NEAT Particles

NEAT Particles combines IEC and NEAT to enable users to evolve complex particle systems. CPPNs control particle system behavior, NEAT evolves the CPPNs, and an IEC interface gives the user control over evolution. NEAT Particles consists of five major components: 1) particle systems, 2) CPPNs, 3) physics, 4) rendering, and 5) evolution.

3.3 Particle System Representation

A particle system is specified by an absolute *system position* in three-dimensional space and a set of particles. Each individual particle is defined by its position, velocity, color, and size. Particle lifespan unfolds in three phases.

1. At birth particles are introduced into space relative to system position and according to a *generation shape* (figure 3.1) that defines the volume within which new particles may spawn.
2. During its lifetime, each particle changes and moves according to a set of rules, i.e. an *update function*.
3. Each particle dies, and is removed from the system, when its *time to live* has expired.

NEAT Particles effects are divided into *classes* for two primary reasons: (1) user convenience and (2) performance. First, to evolve effects in a reasonable time frame, it is helpful to divide the search space for the user. Second, effects may be highly dependent upon certain variables, and unaffected by other variables. For performance reasons, it is not feasible to evolve all possible particle variables simultaneously. A better approach is implemented in NEAT Particles, in which only key variables are evolved in each particle effect class. Five particle system classes are implemented in NEAT Particles to facilitate evolving a variety of common types of effects.

- The *generic system* (figure 3.2a) models effects such as fire, smoke, and explosions. Each particle has a position, velocity, color, and size.
- The *plane system* (figure 3.2b) warps individual particles into different shapes for bright flashes, lens flares, and engine exhaust effects. A single particle in the plane system is represented by four points, each of which has position, velocity, and color.
- The *beam system* (figure 3.2c) models beam, laser, or electricity effects using Bezier curves. Each particle in the beam system is a control point for the Bezier curve, including its position, velocity, and color attributes.
- The *rotator system* (figure 3.2d) models effects whose primary behavior is orbital rotation, common in many applications. Each particle in a rotator system has rotation, position, and color attributes.
- The *trail system* (figure 3.2e) behaves similarly to the generic system, but additionally drops a trail of static particles behind each moving particle.

By providing an array of particle system classes, NEAT Particles allows designers to evolve a substantial variety of effects while conveniently constraining the search space during any particular run.

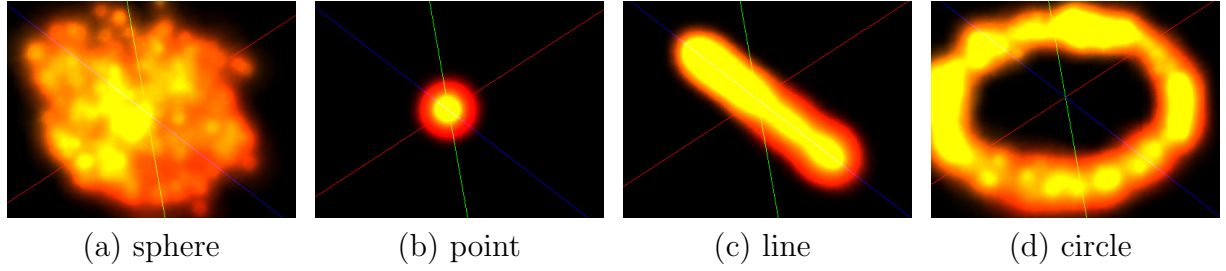


Figure 3.1: **Generation Shapes.** A particle system’s generation shape defines the region in which new particles spawn. (a) Spherical generation produces area effects such as smoke and explosions. (b) Point generation facilitates effects that are attached to specific points on objects, such as vehicle thrust and muzzle flash. (c) Line generation commonly produces effects attached to characters or melee weapons, such as glowing swords. (d) Circular generation enables effects that surround objects, such as energy fields.

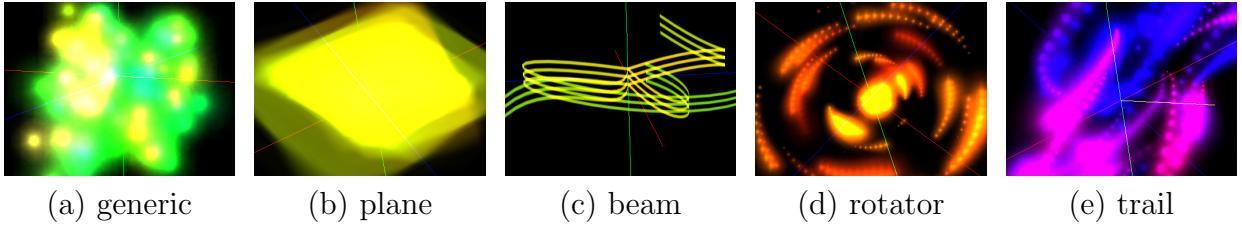


Figure 3.2: **Particle System Classes.** Predefined classes constrain the search space for designers. (a) The generic particle system models effects such as fire, smoke, and explosions. (b) The plane system warps and stretches individual particles for flashes, lens flares, and other effects. (c) The beam system simulates beam, laser, or electricity effects. (d) The rotator system models effects based on orbital rotation common in explosions, energy, and magic. (e) The trail system is similar to the generic system; however each particle drops a trail of smaller particles. Trail systems commonly implement magic, energy, weapon, and exhaust effects.

3.4 CPPN Implementation

CPPNs (a kind of ANN) control particle behavior in NEAT Particles for two primary reasons.

First, CPPNs are a proven method for autonomous control. Second, NEAT is a powerful method for evolving CPPNs for control and sequential decision tasks.

An important question is why evolving CPPNs is preferable to directly evolving the variables of a traditional particle system implementation. While feasible, such an approach still ultimately relies on hand-coded rules (which constitute such systems), which thus depend on programmers to make the search possible. For example, in a traditional particle system implementation, when a new effect class is needed it requires programmers to define the effect parameters (e.g. color change, motion pattern physics, etc.). In contrast, in NEAT Particles the effects of any class are represented by the same structure: CPPNs.

The CPPN for each particle effect dictates the characteristics and behavior of the system. Therefore, each particle effect class includes its own CPPN input and output configuration. In NEAT Particles, the CPPN replaces the math and physics rules that must be programmed in traditional particle systems. Because special effects in most movie and game graphics need to be visually appealing yet not necessarily physically plausible, CPPNs do not need to equate to physically realistic models. However, evolved CPPNN-controlled particle behaviors (e.g. *spin in a spiral while changing color from green to orange*) are still compatible with rules in physically accurate particle simulations such as gravity, friction, or collision.

Every particle in a system is guided by the same CPPN. However, the CPPN is activated separately for each particle. During every frame of animation in NEAT Particles an update function (figure 3.3) is executed that (1) loads inputs, (2) activates the CPPN, and (3) reads outputs. The CPPN outputs determine particle behavior for the current frame of animation. An appropriate set of inputs and outputs is associated with each effect class as follows.

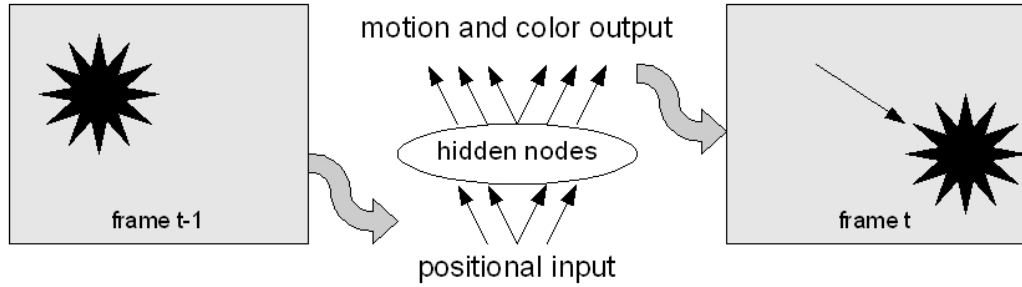


Figure 3.3: **Update Function.** Every frame of animation, each particle passes through an update function to compute velocity and color for that frame. Suppose animation for a particle is being computed at frame t (on the right). The particle's position and distance from center in the previous frame $t - 1$ (on the left) are input into the particle system CPPN. After the CPPN is activated, its outputs are interpreted as velocity and color at frame t . The high frame rate of real-time animation produces small position changes; thus animation and color change is fluid. Over the long term; however, position changes are large, producing a variety of patterns and behaviors.

The primary inputs in NEAT Particles are position and distance from center of the system. The main outputs are velocity and color. These are good inputs and outputs because they can encode significant variety over the long term. However, because animation happens in real-time, the change in position and distance from center are small from one frame to the next, producing incremental changes that look smooth.

The generic particle system CPPN (figure 3.4a) takes the current position of the particle (p_x, p_y, p_z) and distance from the center of the system (d_c) as inputs. Distance from center introduces the potential for symmetry by allowing particles to move in relation to the system center. The outputs are the velocity (v_x, v_y, v_z) and color (R, G, B) of the particle for the next frame of animation. The generic particle system produces behaviors suitable for explosions, fire, and smoke effects.

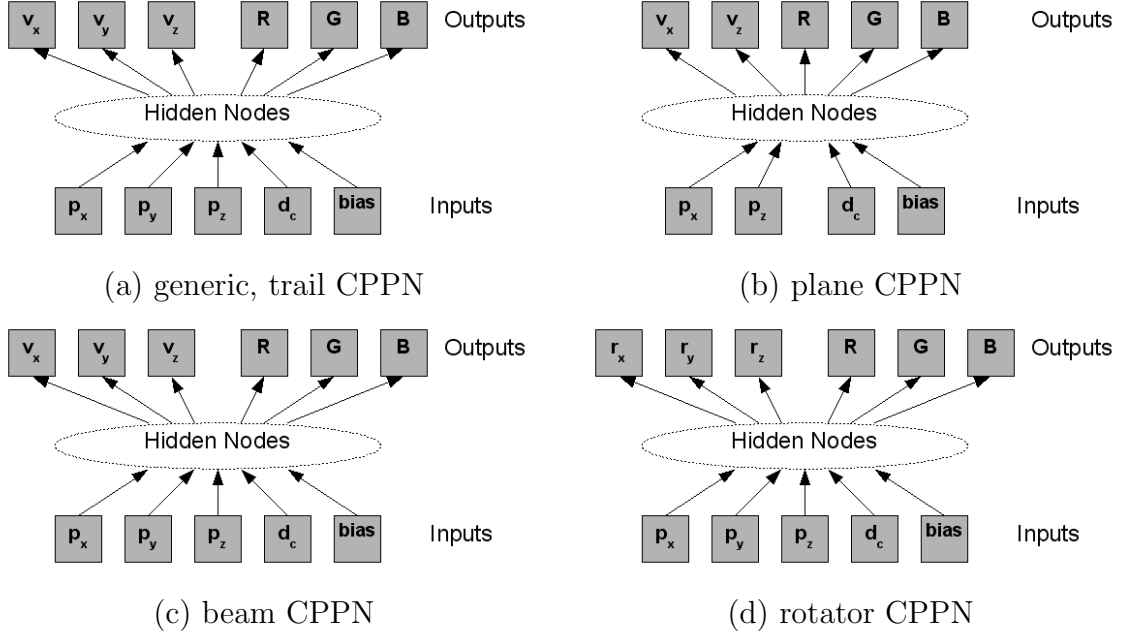


Figure 3.4: **Particle System CPPNs.** To produce a specific range of effects, each particle class CPPN uses different inputs (i.e. position and distance from center) and outputs (i.e. velocity, color, and rotation), which are shown for (a) the generic and trail particle system, (b) the beam particle system, (c) the plane particle system, and (d) the rotator particle system. The beam system CPPN appears similar to the generic and trail system CPPNs; however a generic system CPPN controls individual particles, whereas a beam system CPPN controls Bezier curve control points. The plane system CPPN controls four corners of a warped quad, and the rotator system CPPN controls individual particle rotation. Each CPPN is evolved in NEAT Particles to connect the inputs to the outputs of each class.

Each particle in the plane system consists four co-planar points that may be warped into different shapes. Because the corners must be coplanar for rendering purposes, the y component of velocity for each corner is fixed. Thus, the inputs to the plane system CPPN (figure 3.4b) are the position of each corner (p_x, p_z) and the distance from the center of the plane (d_c). The warped quads of plane systems are commonly found in explosions, engine thrust, and glow effects.

The beam system CPPN (figure 3.4c) controls directed beam effects. To produce twisting beams, a Bezier curve is implemented with mobile control points directed by the CPPN. The inputs are the position of each Bezier control point (p_x, p_y, p_z) and distance of the control point from the center of the system (d_c) . The outputs are the velocity (v_x, v_y, v_z) and color (R, G, B) of the control point for the next frame of animation. Beam systems produce curving, multi-colored beams typically found in futuristic weapons, magic spells, lightning, and energy effects.

The rotator system (figure 3.4d) enables evolving rotation-based effects. The inputs to the CPPN are particle position (p_x, p_y, p_z) and distance from the center of the system (d_c) . The outputs are rotation around the x , y , and z axes (r_x, r_y, r_z) and color (R, G, B) . Rotation-based particle systems are common in explosions, halos, and energy effects.

The trail system behaves similarly to the generic system yet provides a more complex visual effect by periodically dropping stationary particles that shrink and fade out. Therefore, the trail system CPPN takes the same inputs and emits the same outputs as the generic CPPN. Trail systems are convenient because they provide a computationally inexpensive form of motion blur or visual trail behind moving objects.

CPPNs control particle behavior and CPPN input/outputs divide effects into classes, which shrinks the search space for users. While CPPN topology and weights significantly contribute to particle behavior, activation functions within each node play an important role as well; they are detailed in the next section.

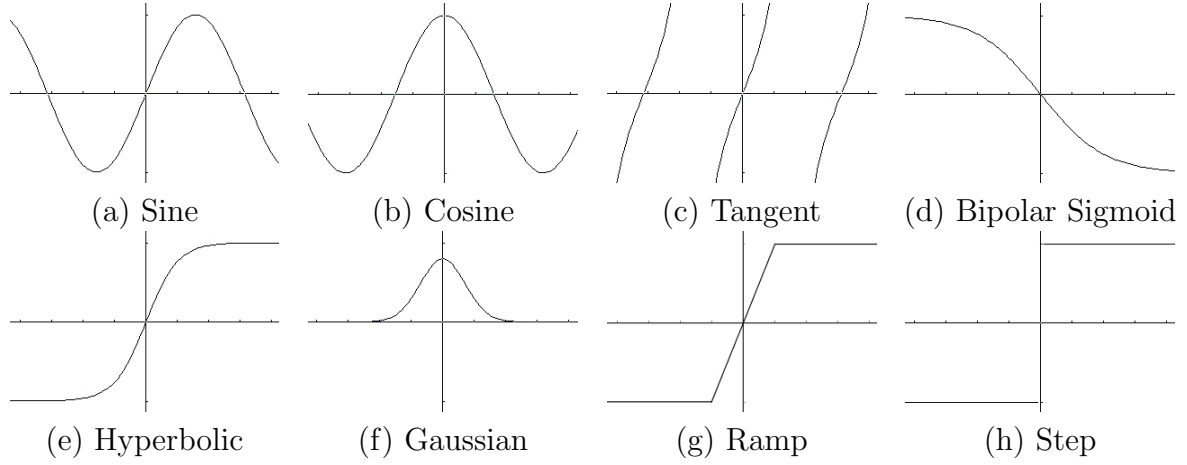


Figure 3.5: **CPPN Activation Functions.** In NEAT Particles all CPPN hidden nodes and output nodes are randomly assigned one of eight activation functions: (a) sine, (b) cosine, (c) tangent, (d) bipolar sigmoid ($\frac{[1-\exp(-x)]}{[1+\exp(-x)]}$), (e) hyperbolic ($\frac{[e(x)-e(-x)]}{[e(x)+e(-x)]}$), (f) Gaussian ($[\frac{1}{\sqrt{0.5*PI}}] * e^{(-x^2)}$), (g) ramp ($x = \begin{cases} -1 & \text{if } (x < -1) \\ 1 & \text{if } (x > 1) \end{cases}$), or (h) step ($x = \begin{cases} -1 & \text{if } (x < 0) \\ 1 & \text{if } (x \geq 0) \end{cases}$).

3.5 Activation Functions

Unlike traditional ANNs, NEAT Particles CPPN hidden nodes and output nodes contain an activation function selected from a set of eight possibilities (figure 3.5). Theoretically, ANNs with a single activation function can evolve any behavior [Cyb89]; however, multiple activation functions are preferable in NEAT Particles because the user can obtain variety more quickly and thereby evolve toward the intended effect sooner.

3.6 Physics

Each frame of animation, after the CPPN is activated, the velocity for each particle is determined by the outputs. To animate a particle each frame (i.e. move the particle through

space) a linear motion model calculates the position of the particle at time t based on *time elapsed* τ since the last frame of animation:

$$P_t = P_{t-1} + V\tau s, \quad (3.1)$$

where P_t is the particle's new position vector, P_{t-1} is the particle's position vector in the previous animation frame, V is the particle's velocity vector, and s is a scaling value to adjust the speed of animation.

3.7 Rendering

NEAT Particles renders particles to the screen with *billboarding* [Fer06], a technique in which two-dimensional bitmap textures are mapped onto a plane (i.e. a *quad*) that faces perpendicular to the camera. The corners of the quad are offsets from the particle position. By facing the quad toward the camera the billboarding method convincingly conveys the illusion of translucent three-dimensional particles in space.

The billboarding technique is implemented in NEAT Particles because it is the most common and versatile method to render particles. An alternative particle rendering method is point sprites [Lun03]; however, they do not allow arbitrary warping of particle shape required for the beam and plane systems.

There are several ways to optimize particle system rendering including level of detail (LOD) [OFL01], batch rendering [Lun03], and GPU acceleration [Lat04]. NEAT Particles is compatible with all such methods; however they are not explored in this implementation.

The next section explains how particle classes, CPPNs, physics, and rendering combine to enable particle effect evolution.

3.8 Evolution

Evolution in NEAT Particles follows a similar procedure to other IEC applications. The user is initially presented a population of nine randomized particle systems represented by simple CPPNs (figure 3.6a). Each individual system and its CPPN can be inspected by *zooming in* on the system (figure 3.6b). If the initial population of nine systems is unsatisfactory, a new random batch of effects can be generated by restarting evolution.

The user begins evolution by selecting a single system from the population to spawn a new generation. A population of eight new systems (i.e. offspring) is then generated from the CPPN of the selected system (i.e. parent) by mutating its connection weights and possibly adding new nodes and connections. That is, offspring complexify following the NEAT method. Evolution proceeds with repeated rounds of selection and offspring production until the user is satisfied with the results. If the user is unsatisfied with an entire new generation, an *undo* function recalls the previous generation.

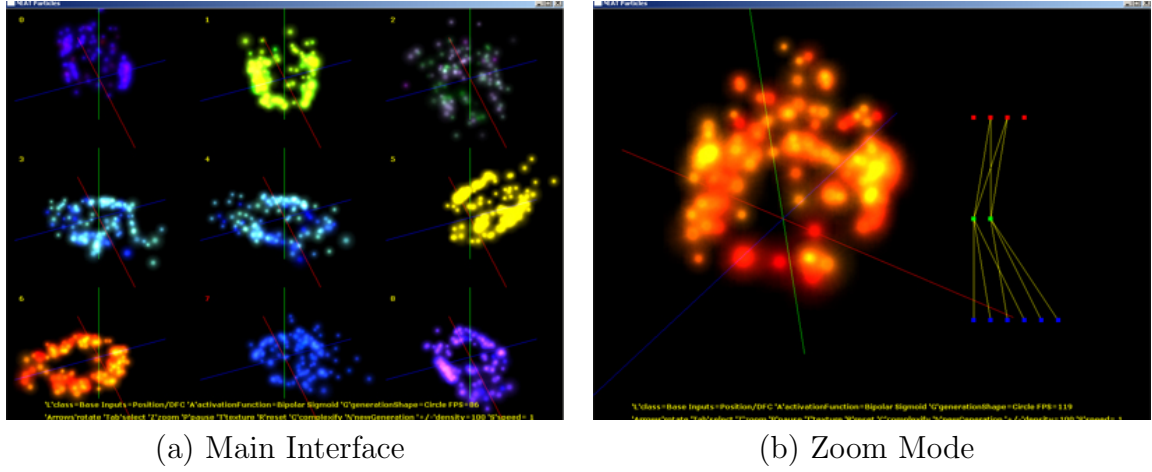


Figure 3.6: **NEAT Particles Interface**. In the main interface (a), the user is presented with nine particle systems. System parameters such as generation shape and inputs are displayed on the bottom of the screen. In zoom mode (b), a single particle system and its CPPN can be inspected.

Specifically, each new generation preserves the parent exactly and the other eight members of the population are mutated from the parent. For each offspring, a uniformly random number of connections (between one and the number of connections in the network) are mutated by a uniformly random value between -0.5 and 0.5 . Adding new nodes and connections is controlled by separate mutation rates. The probability of adding a new connection is 0.3 and the probability of adding a new node is 0.2 . New nodes are assigned a random activation function and connected into the existing CPPN [SM02]. These parameters were found to be effective for IEC in preliminary experimentation.

Through complexification, particle system effects become increasingly sophisticated as evolution progresses. Thus, complex and unique effects are discovered that follow user preferences. The next section explains evolving particle system content for a more specialized purpose, weapons effects for video games.

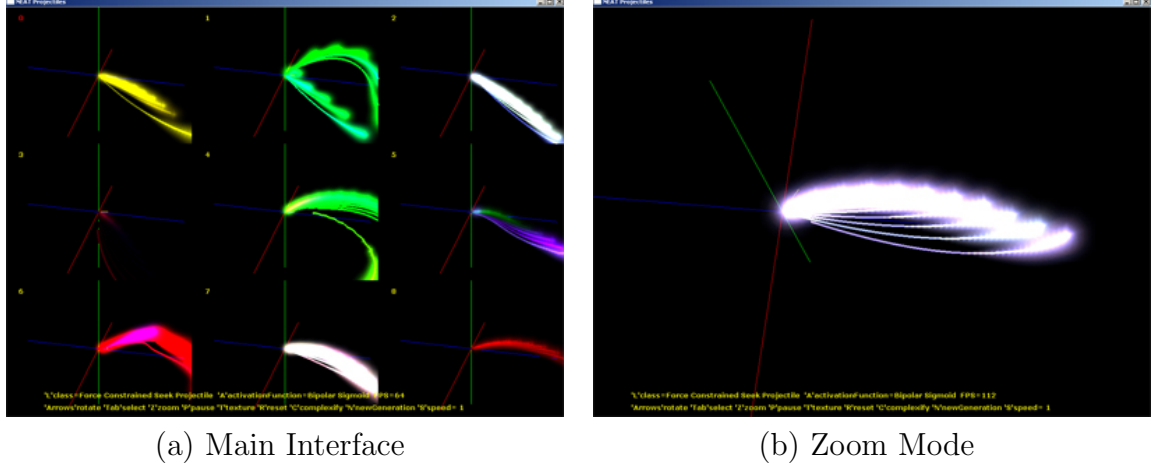


Figure 3.7: **NEAT Projectiles Interface**. In the main interface (a), the user chooses among nine projectile systems of any weapon class. The weapons can be rotated (as in this figure) and the refire rate adjusted to display their full behavior. In zoom mode (b), a single projectile system and its CPPN can be inspected.

3.9 NEAT Projectiles

NEAT Projectiles is an extension of NEAT Particles designed to evolve particle weapon effects for video games, and served as the initial experiments on evolving weapon behavior for later integrating into Galactic Arms Race. The aim is to exhibit a concrete, practical application of NEAT Particles that can potentially enhance content generation in existing real-world products. NEAT Projectiles uses similar rendering, physics, and activation functions as NEAT Particles. Furthermore, the same IEC interface (figure 3.7) drives evolution. The major differences are (1) the projectile classes, (2) the projectile constraints, and (3) the CPPN inputs and outputs.

3.10 Projectile Classes

Three classes of weapon-like systems are implemented in NEAT Projectiles to mirror common weapon models in video games: (1) *dumb weapons*, (2) *directed weapons*, and (3) *smart weapons*. Dumb weapons fire simple, non-target-aware projectiles and exhibit a fixed behavior in flight. Directed weapons fire projectiles that may be steered by the user during flight. Smart weapons see the target; like a heat-seeking missile, the in-flight behavior of smart projectiles is influenced by target motion.

3.11 Projectile Constraint

Particle weapons provide two new significant constraints on particle motion beyond generic particle effects. First, to avoid weapons firing backward, projectile velocity is limited to overall forward motion (this constraint is later relaxed in Galactic Arms Race). Second, evolved projectile weapons fire in the same pattern regardless of what direction the weapon is facing. It would not make sense for projectiles emitted from a weapon to behave differently when a user points the weapon in different directions. Therefore, projectile coordinates are defined relative to the heading of the gun when it is fired.

The new projectile classes and constraint mechanisms also influence the interpretation of NEAT Projectiles CPPNs, as explained next.

3.12 Projectile CPPNs

Because there is more than one way to make particles act as projectiles, two approaches are implemented and tested in NEAT Particles: (1) the *offset-constrained model* and (2) the *force constrained model*.

In the offset-constrained model (figure 3.8a), a 90° *offset cone* in front of each particle is computed in each frame. The outputs from each particle's CPPN represent a vector within the offset cone, which becomes the particle's new velocity. Offset angles are computed differently for each weapon type. A particle fired from the *dumb weapon* has a fixed offset in the direction the gun was facing on discharge (figure 3.9a). The *directed weapon* allows the user to influence projectiles while in flight; therefore particle offset is constrained to a 90° cone around the vector the weapon is currently facing (figure 3.9b). Particles fired from the *smart weapon* seek their target. Therefore, the smart particle's offset is constrained to the 90° cone around a vector from the projectile to the target (figure 3.9c).

In the force-constrained model (figure 3.8b), the CPPN is similar to that used in the generic system of NEAT Particles; however a push force is applied to constrain particle movement to a general direction. The direction of the push force depends on the weapon type. The dumb weapon projectile is pushed in the direction of the gun when it discharges. The directed projectile pushes in the direction the gun is currently facing. The smart weapon pushes projectiles in the direction of the target.

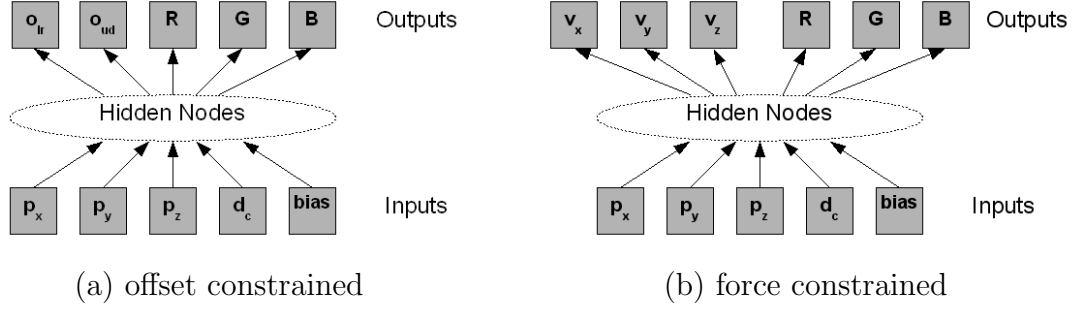


Figure 3.8: **Projectile System CPPNs.** Projectile models are designed to minimize undesirable behaviors (e.g. firing backward). (a) In the offset constrained model, projectile movement is constrained by an offset cone in front of the projectile. The offset cone is computed by adding two vectors *up-down* (o_{ud}) and *left-right* (o_{lr}). CPPN inputs include the current position of the particle, distance from center of the weapon, and a bias. The outputs are left-right offset, up-down offset, and color. (b) In the force constrained model, projectile motion is perturbed by an additional push force applied to the projectile after CPPN processing. Inputs are the current position of the particle, the distance from the system center, and a bias. The outputs are the particle velocity and color. Both models constrain projectile behavior while allowing sufficient evolutionary variety.

The combination of constraint model, classes, and correct CPPN design minimizes defective offspring while allowing a sufficiently large variety of unique weapons to evolve, which is integral to efficiently producing useful content through IEC.

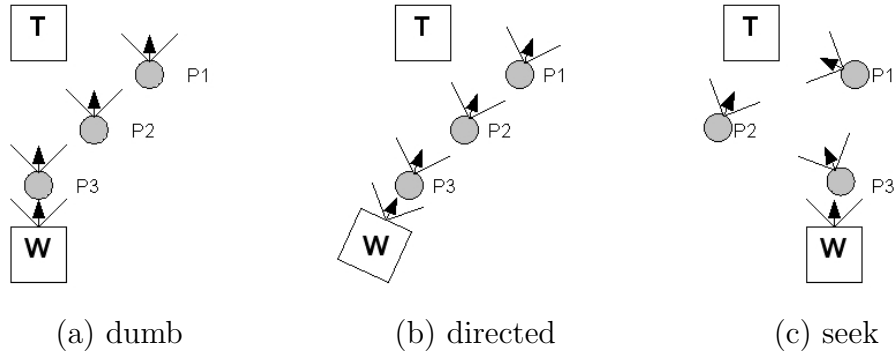


Figure 3.9: **Projectile Constraint Mechanics.** To ensure that weapons behave as projectiles, particle velocities are constrained to the 90° cones shown above. In the figure, W is the weapon and T is the target. (a) Dumb weapon particles are not target-aware so they are constrained in velocity to a fixed 90° cone in front of the weapon at the moment of discharge. (b) Directed weapon particles are not target-aware but may be influenced while in flight by the weapon. Thus directed particle velocity is constrained to the 90° cone in which the weapon is currently facing. (c) Smart particles are target-aware; therefore they are constrained to a 90° cone around a vector from the particle to the target.

CHAPTER 4

EXPERIMENTS IN EVOLVING PARTICLE SYSTEMS

This chapter shows how NEAT Particles and NEAT Projectiles work in practice to produce useful particle system content. All particle systems reported were evolved in between five and ten minutes and between 20 and 30 user-guided generations. The NEAT particles executable, source code, and examples effects in this dissertation can be downloaded at <http://eplex.cs.ucf.edu>.

Figure 4.1 illustrates evolving a *flame shield* effect with NEAT Particles. The goal of the effect is a halo of flaming red particles around the user. Evolution begins with a red ring (figure 4.1a). As evolution proceeds, particles begin breaking away from the ring (figure 4.1b). A full orbital sphere of particles develops eventually (figure 4.1c) and the final effect (figure 4.1d) exhibits brighter colors.

Figure 4.2 depicts the evolution of an *arc gun* effect with NEAT Projectiles. The aim is to create a multi-beam effect that tracks a target. The starting point is a single curving beam to the target, which is marked with a cross (figure 4.2a). During evolution the beam splits (figures 4.2b, 4.2c). Finally, the desired effect is achieved with two stylized, parallel arcs that track the target (figure 4.2d).

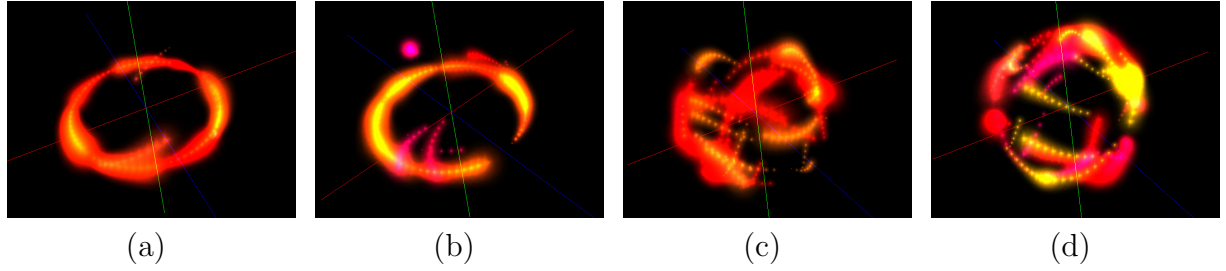


Figure 4.1: **Particle System Evolution.** This figure illustrates evolution of a *flame shield* effect with NEAT Particles, in which red and yellow particles are evolved to orbit a player at the center. (a) Evolution begins with a ring shaped rotator system with an appropriate red and yellow color scheme. (b) After a few generations particles begin to detach from the ring. (c) Several generations later a prominent orbital behavior becomes apparent. (d) Evolution concludes with a full orbital pattern and a brighter color scheme, producing a convincing flame shield effect suitable for use in a video game.

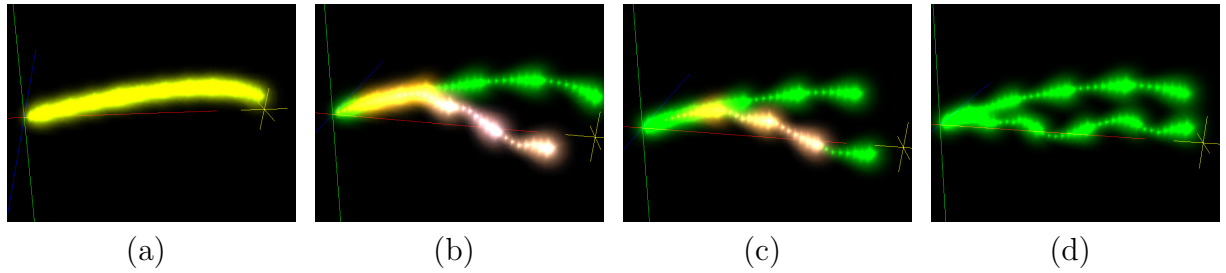


Figure 4.2: **NEAT Projectiles Evolution.** A double-arc beam weapon that seeks a target is evolved with NEAT Projectiles. The weapon emits particles from the left side of each frame and the target is marked with a cross on the right side. (a) Evolution begins with a single arc that connects to the target. (b) After several generations, the beam begins to split in the middle. (c) Continuing evolution, the double arc becomes more pronounced. (d) Eventually the arcs become fully disjoint and the intended projectile behavior is achieved.

Preliminary testing of both NEAT Projectiles constraint models suggests that, compared to the force-constrained model, the offset-constrained model over-constrains evolution. It generates less variety in evolved weapon effects. However, unlike the force-constrained model, it also produces no offspring that fire back at the user. Thus, both models have their pros and cons.

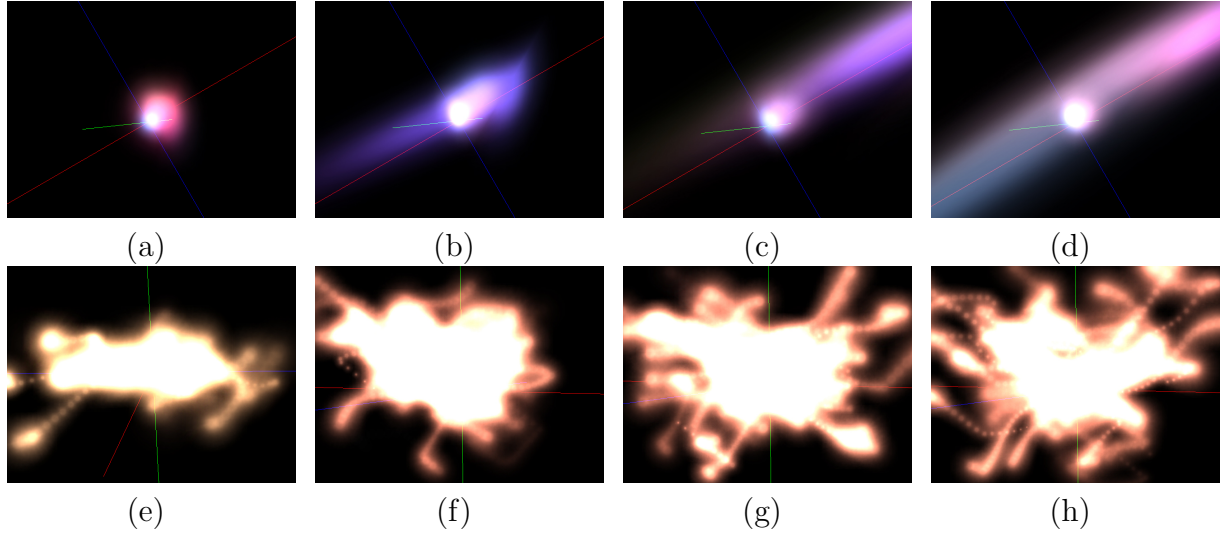


Figure 4.3: **Evolved Particle Animations.** Key frames from animations of two evolved particle systems are presented in this example. Figures (a) through (d) depict an expanding vortex or explosion-like effect of an evolved plane system. Figures (e) through (h) depict a realistic billowing smoke cloud or explosion effect produced by an evolved trail system. Such effects illustrate that NEAT Particles can evolve effects appropriate for graphics and games.

Keyframe animations for evolved particle and projectile systems are depicted in figures 4.3 and 4.4. Additional evolved systems are presented in figures 4.5 and 4.6.

4.1 Comparison to Hand-Coded Particle Systems

To compare the quality of IEC particle effects to those generated by traditional methods, two hand-coded particle emitters were implemented with the same rendering method as NEAT Particles (figure 4.7). The resulting effects exhibit similar visual quality; however, they are limited to simple behaviors because the behavioral complexity of hand-coded particle sys-

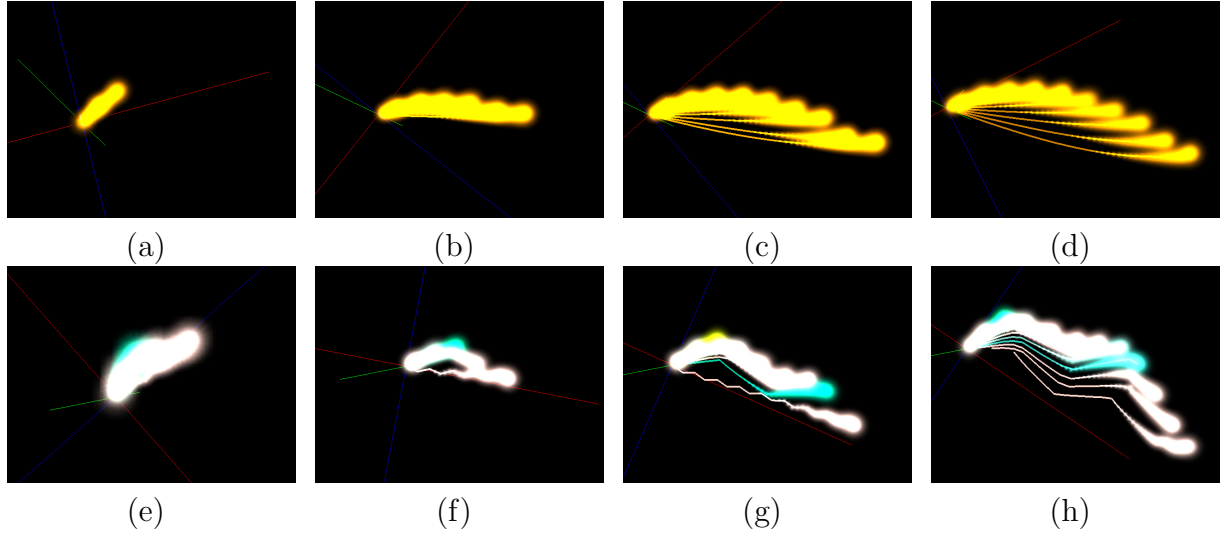


Figure 4.4: **Evolved Projectile Animations.** Key frames of evolved single projectiles are displayed above. The projectiles are fired from the left side of the screen towards the right side. The trailing lines mark motion over time. Figures (a) through (d) display a solid yellow projectile with a smooth curving behavior. Figures (e) through (h) depict a spiraling projectile that changes colors. Multiple projectiles evolved within the constraints combine to form complex weapon effects in NEAT Projectiles.

tems is dependent upon mathematics, physics, and programming, which become increasingly difficult to coordinate through hand-coded policies as more is added.

Another interesting comparison can be drawn with the IEC fireworks application by Tsuneto [Tsu02], which produces a specialized class of particle effects. In this system, fireworks are defined by real-world attributes such as powder type, explosive payload, number of stages, stage configuration, etc. A rule-based physics system defines the behavior of fireworks based on these attributes. Through repeated selection in an IEC interface, users can evolve fireworks to suit their preferences. Thus, unlike NEAT Particles, this system demonstrates evolving the variables of a rule system. In contrast, NEAT Particles evolves the behavior rules themselves. Both approaches offer unique advantages. The special rule set of

the fireworks application allows it to focus on a specific class of effects. NEAT Particles in contrast can evolve effects in a large variety of classes because of its generality and lack of domain-specific parameters.

4.2 User Study

An informal user study was conducted to test the viability of the NEAT Particles method. In this study, eight users were introduced to the system and encouraged to explore the search space to evolve effects that please them. Samples of the user-evolved effects are presented in figure 4.8. In general, users found the generic and trail particle system classes to produce the most variety, while the rotator, plane, and beam systems were acknowledged to be more constrained. The examples presented in figure 4.8, all evolved within 15 generations, demonstrate that the IEC approach enables users to quickly and easily generate complex and unique particle effects.

In summary, the totality of results demonstrate the ability of NEAT Particles and NEAT Projectiles to evolve particle systems of similar complexity to those in mainstream games.

4.3 Performance

NEAT Particles’ computational requirements scale at $O(n)$, where n is the number of particles. The position of each particle is input to the CPPN once per frame. Similarly, in traditional particle systems each particle passes through an update function once per frame. While the complexity of the CPPN increases with the complexity of the effect, the same is likely true for traditional formulations. Thus NEAT Particles is expected to perform comparably to traditional particle systems.

4.4 Discussion and Future Work

The complexity and cost of producing content for modern graphics and games creates the need for tools that quickly and efficiently generate novel content. IEC can alleviate this problem by enabling automated content generation guided by user preferences. NEAT Particles demonstrates the promise of this approach by constraining the search space for the user, thereby defining a content space large enough to evolve many interesting and useful results, yet not so large that producing useful output is too time-consuming. The separate classes implemented in NEAT Particles provide this constraint.

Besides intentionally evolving specific particle systems that they have in mind, users can also employ the IEC approach of NEAT Particles as a concept generation tool. While evolving a specific effect, the user often generates novel, compelling effects that were not

initially planned. Thus an additional advantage of NEAT Particles over traditional particle system implementations is that it may act as an idea or concept generator.

4.5 Conclusion

There is an increasing need for powerful graphics and game content generation tools. Content developers require such tools to augment and assist the slow and expensive content pipeline. End-users benefit from such tools because today's games are distributed with content generation tools for users to customize or build their own content. Additionally, there is the emerging trend of content generation as a major part of game play itself. IEC can potentially solve this problem by providing an intuitive way to easily generate complex and unique content by user preference.

NEAT Particles and NEAT Projectiles demonstrate how particle system effects for graphics and video games can be interactively evolved through user preference. In this approach, particle systems are represented by CPPNs, the CPPNs are evolved by NEAT, and an IEC interface enables the user to guide evolution. By replacing the complex, hand-coded rules of traditional particle systems with CPPNs, the dependence on programmers to create new effects is reduced. The IEC interface provides easy, interactive exploration of the search space and a way to discover novel effects useful to both developers of graphics and gaming media, and to end users of the content generation tools provided with such software.

While the focus of this work is on evolving particle system effects, the techniques are applicable to generating other types of graphical and gaming content. Thus, automated content generation is a promising research direction in which evolutionary computation can significantly contribute to popular media and games.

The cgNEAT algorithm, described in the next section, evolves content in the game itself. It is later combined with the concepts from NEAT Particles and NEAT Projectiles in Galactic Arms Race.

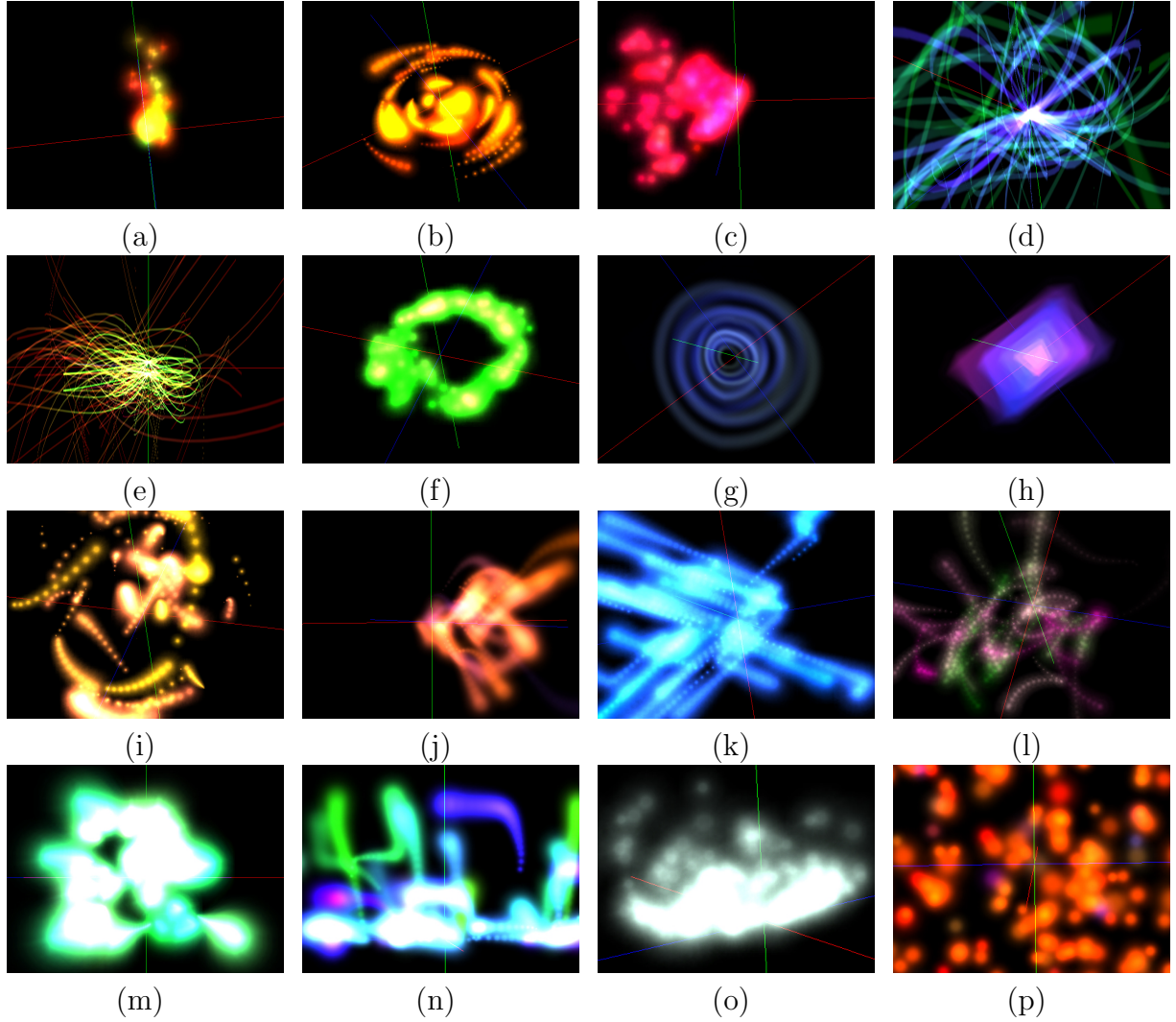


Figure 4.5: **Sample Evolved Particle Systems.** The images in this figure are single animation frames from effects evolved with NEAT Particles. These images demonstrate the variety of effects evolved through IEC.

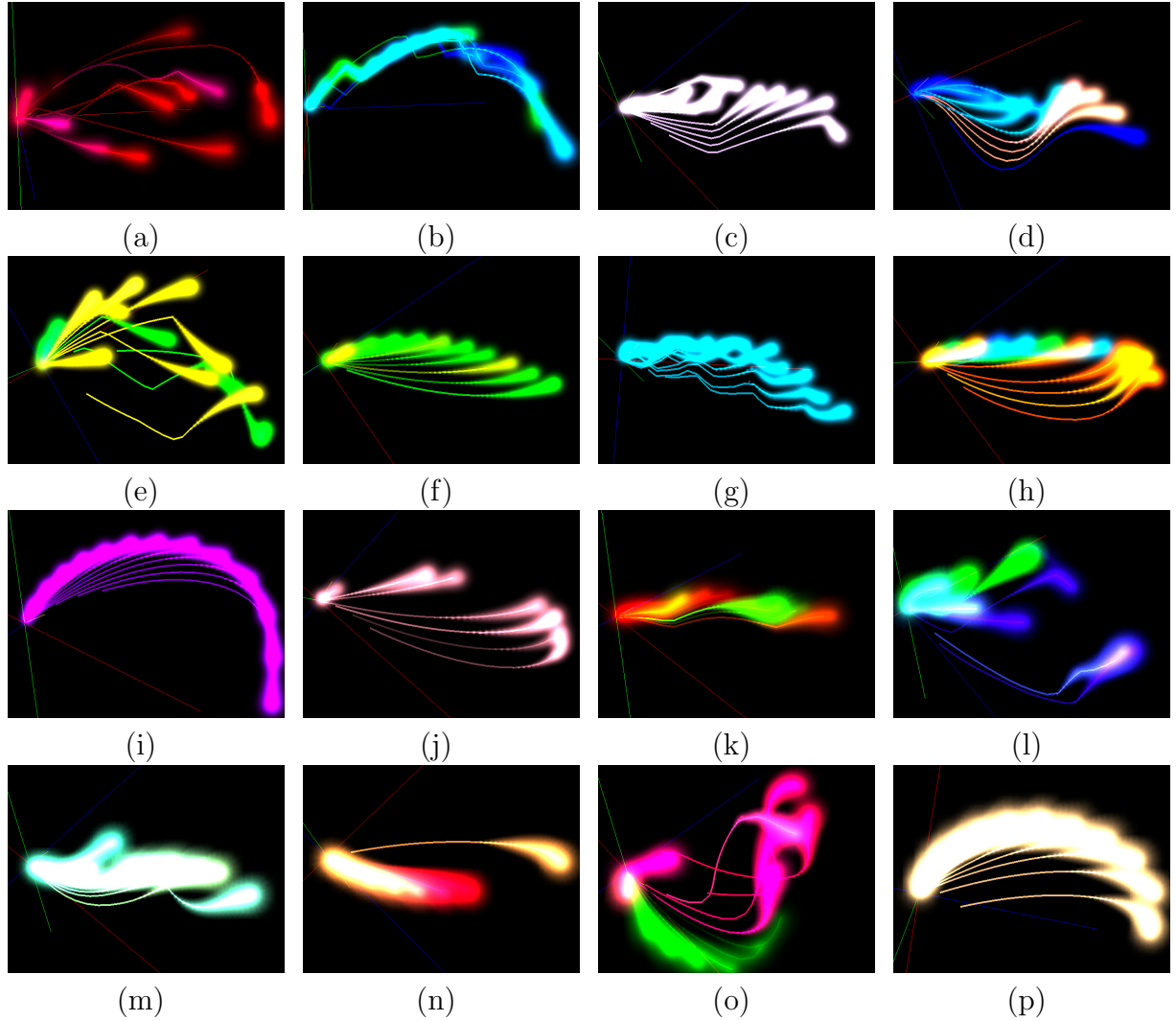


Figure 4.6: **Sample Evolved Projectile Systems.** Single animation frames are shown of evolved NEAT Projectile systems fired from the left side of the screen toward targets on the right. The trailing lines plot motion over time. These images demonstrate the variety of weapon behaviors evolved by NEAT Projectiles.

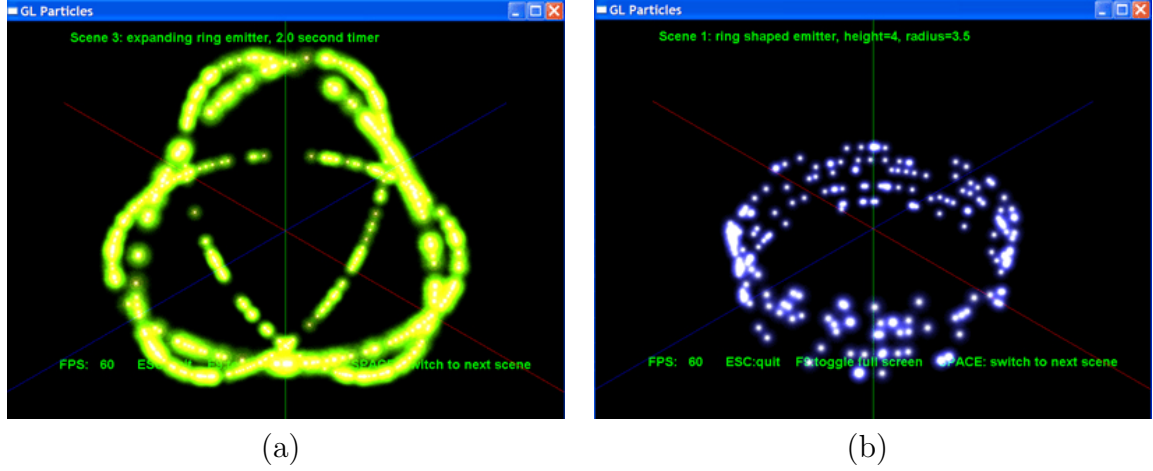


Figure 4.7: **Traditional Particle System Comparison.** To compare IEC-generated particle systems to traditional ones, two hand-coded particle emitters were implemented within the same renderer as NEAT Particles. The expanding ring emitter (a) supports explosion effects and the simple ring emitter (b) can convey a variety of magical and force effects. Both systems display similar visual quality to NEAT Particles. However, they are capable of comparatively much less behavioral complexity. This example demonstrates the dependence on math and programming of traditional particle system implementations.

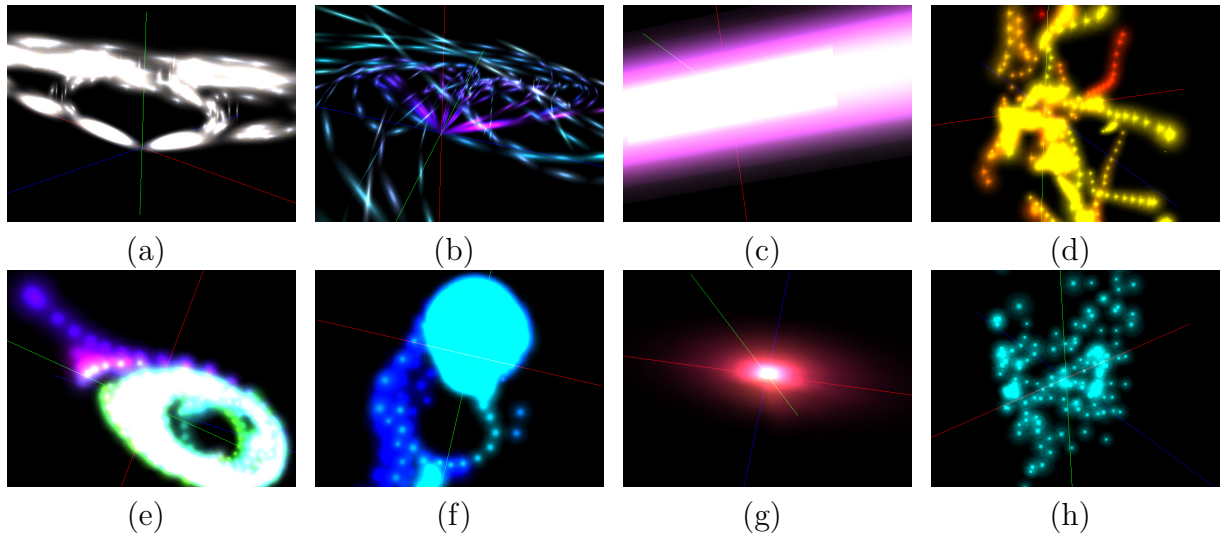


Figure 4.8: **User-Evolved Particle Effects.** The images in this figure depict effects evolved with NEAT Particles by participants in the user study. The variety and complexity of these examples demonstrate that the IEC approach of NEAT Particles enables users to quickly evolve compelling particle system effects.

CHAPTER 5

CONTENT GENERATING NEUROEVOLUTION OF AUGMENTING TOPOLOGIES (CGNEAT)

The aim of the cgNEAT algorithm is to automatically generate computer graphics and video game content based on user behavior as the game is played. It represents a step beyond generating content based only on randomization. While there are technologies for evolving content like pictures [SBD08] or three-dimensional models [Sim94], these technologies are not designed to work in real-time during a game; rather they require users to explicitly designate which items are the best, which is something that a user playing a game does not want to do. That is, constantly answering questions about what they like and what should be produced in the future would disrupt players' experience. In contrast, the cgNEAT method makes these decisions automatically based on implicit information within usage statistics.

This chapter describes the cgNEAT algorithm in detail. First, an overview of the algorithm is presented. Then, unique features of cgNEAT are described, illustrating the substantial differences between cgNEAT and standard evolutionary algorithms. Finally, all steps of the cgNEAT algorithm are explicitly defined, the special features of cgNEAT are discussed.

5.1 Algorithm Overview

The main principles of cgNEAT follow:

1. Each content item is represented by a CPPN. Different types of content can be represented by different CPPN input/output configurations (the specific representation for particle weapons is described later). In principle, a different representation than CPPNs can also be evolved.
2. During the game, each content item is assigned a fitness that is computed based on how often players actually *use* the content. That way, the system knows the relative popularity of each content item currently in the game.
3. Players begin the game with either (1) random content or (2) content from the *starter pool*, which is a special pool of content appropriate to beginners in the game. Starter pool content *does not* contribute to evolution and cannot be selected for reproduction.
4. Content is spawned in the game world, which means that it is placed in parts of the world where users can obtain it. However, unlike in most evolutionary systems, spawned content is not eligible for reproduction until players pick it up.
5. Content is reproduced in cgNEAT as follows: The algorithm selects content items from among content that players in the world *already possess* as parents that reproduce to form new content, which is spawned as described in step 3. The content items that are chosen as parents are selected probabilistically based on a roulette wheel scheme

in which the chance of being chosen as a parent is proportional to the popularity (i.e. fitness) of the item. Reproduction, including mutation and crossover, is performed in accordance with the NEAT algorithm. Thus, there is a chance that CPPNs may become more complex than their parents.

6. For any new content that is spawned, there is a probability (selected by the designer) that it will be chosen from a *spawning pool*, which is a collection of pre-evolved content, instead of being reproduced from parents. This pool ensures that diversity is not lost and that good types of content from the past (i.e. those that users liked) might reappear. Additionally, it ensures an initial seed of good content when the game first starts and players' preferences are unknown. The game designers initially select content, which may be pre-evolved before the game is released, to put into the spawning pool.
7. Content that obtains a very high fitness (i.e. is popular with players) may optionally be saved to the *content archive*. There are several options for content archive material including (1) data analysis, (2) cycling it back into the game by adding it to the spawning pool, or (3) giving it to NPCs for use against players.

5.2 Unique Features

The cgNEAT algorithm incorporates some mechanics of NEAT and standard evolutionary computation (EC), yet exhibits several major differences. Unlike in normal EC, the population size (i.e. those items that are eligible at any given time to reproduce) is variable and depends entirely on the number of users in the system. Furthermore, when an offspring is produced, unlike in normal evolutionary computation, it is not immediately placed into the population eligible to reproduce. Instead, it is in a special temporary state (placed somewhere in game world) in which it may join the population only if a user chooses to acquire it. Also unlike normal evolutionary computation, instead of fitness determining which items are eliminated from the population, users entirely determine which items leave the population simply by discarding them.

Unlike standard interactive evolutionary computation (IEC [Tak01]), users never explicitly communicate to the system which content they like. Instead, the preferred content is deduced by the system implicitly from natural human behavior. That is, users do not need to know that they are interacting with an evolutionary algorithm yet evolution still works anyway.

Unlike regular NEAT, speciation is not necessary because users determine what is popular and the diversity of the population reflects the diversity of user preferences. Every step of the cgNEAT algorithm is asynchronous. At any time players may cause content to join the population or be eliminated.

5.3 Starter Pool, Spawning Pool, and Archive Pool

At the beginning of the game or simulation, players must often be provided content to start out with. One approach is to give players randomized content to begin with; however, if the content search space is sufficiently large, new players might then begin with ineffective content, thereby creating a poor first impression of the game.

The cgNEAT approach is to define a starter pool of known effective content items for players to begin the game with. Because all players begin the game with them, starter pool content does not contribute to evolution and cannot be selected for reproduction. Starter pool content does gain fitness, and if during evolution a starter pool item is selected for reproduction, a spawning pool content item is randomly selected to spawn into the game world.

The spawning pool is a pre-evolved set of content that is known to be effective and can be distributed with the game or application. The primary reason for including the spawning pool is that, at the beginning of the game, there is no past data on player preferences. Thus the spawning pool content effectively jump starts evolution on a promising course. Like the starter pool, the spawning pool helps guarantee a good initial impression of the game to new players, which could be difficult to accomplish with purely random starting content. Additionally, if NPCs in the game are equipped with evolved content, arming them with spawning pool content can ensure an appropriate level of NPC difficulty.

Finally, the archive pool consists consists of all content in the game that achieves a high level of fitness. When such content reaches a certain level of fitness (which by necessity varies among application) it is automatically saved to the archive pool. There are several options for the archive content, including: (1) recycling it into the spawning pool, for later appearing in the game, (2) analysis for data collection purposes (it provides insight into what content players like), and (3) giving archive content to NPCs in the game.

5.4 Conclusion

The general approach of cgNEAT described in this dissertation can be applied to evolving many forms of content (e.g. images, models, shader effects, etc.) by other forms of content representation (i.e. evolvable structures other than CPPNs). Thus cgNEAT might be more generically called cgEA, or *content-generating evolutionary algorithm*. Note that the starter pool, spawning pool, archive pool, and roulette parameters can give game designers considerable control of over the course of evolution, should they so desire it. The next section details the first application of cgNEAT in practice, which is to evolve weapons in the Galactic Arms Race video game.

CHAPTER 6

GALACTIC ARMS RACE (GAR)

This chapter introduces the Galactic Arms Race multiplayer video game, which applies cgNEAT to evolve an endless array of unique particle system weapons, with which players can fight space enemies and up to 32 other players online. GAR is the first game to enable cooperative evolution of novel game content. A major goal of GAR is to show that CCE can work in an actual near-commercial quality game. Thus it is an important vehicle to demonstrating the potential of cgNEAT to impact how game content is created in the future.

Due to the large scope of the GAR project, an overview of development is presented first. Then a complete description of the game is given; content generation is integrated into the game’s story and mechanics. After that, an overview of some of the technology integrated into GAR is presented, including spatial hashing and soft body model simulation. Next details on how GAR represents, evolves, and renders particle system weapons is discussed, based on previous work in NEAT Projectiles. Then, cgNEAT-specific issues are discussed, such as the starter pool, spawning pool, and archive pool. Finally, differences between GAR single player and multiplayer modes are described.

6.1 Development

GAR is intentionally designed to look and feel like a near-commercial quality video game so that it can convincingly demonstrate the promise of automatic content generation for mainstream games. To reach that level of quality, it took over a year to build by a nine-member mostly student team. GAR version 1.0 was released on June 2, 2009 and is available online at <http://gar.eecs.ucf.edu>.

The *GAR Client 1.0* (figure 6.1) contains 36,820 source lines of code (SLOC) and over 90MB of three-dimensional models, two-dimensional texture art, music, and sound effects. Additionally, for multiplayer modes, the *GAR Server 1.0* (figure 6.2) contains 6,796 SLOC and the *GAR Master Server* (figure 6.3) contains 873 SLOC.

GAR single and multiplayer game play are described next.

6.2 Game Mechanics Overview

In GAR (figure 6.1), the goal is to pilot a space ship to defeat enemies, gain experience, earn money, and most importantly, to find advantageous new weapons that are automatically generated by cgNEAT.

GAR contains both a single player game and a full multiplayer game (figure 6.4) in which weapons evolve based on the aggregate usage of all players online. In GAR's single-player mode, evolution is directed by the actions of a single player battling NPC aliens in the

game, which are controlled by scripted steering behaviors [Rey99] and the BOIDS algorithm [Rey87]. The GAR multi-player game enables up to 32 players to fight cooperative online battles against NPCs, or competitive battles against other each other. GAR multiplayer evolution is substantially more diverse because the evolutionary population consists of the weapons currently possessed by all players in the game (i.e. it is CCE).

Every new weapon found in GAR is unique and players can continually find novel weapons with characteristics evolved from those weapons players favored in the past. It is important to note that weapons evolved in GAR all fire particle bursts with the same strength and number. Thus it is not sheer power that is evolving, but rather the pattern in which particles spray from the gun, which has complex tactical implications. Therefore, the space of weapons is not a total order from worst to best, but rather a complex multi-objective coevolutionary landscape.

Players are limited to three *weapon slots*, each of which holds a single weapon. Destroyed enemies and enemy bases may drop a *weapon pickup* that contains a novel weapon evolved by cgNEAT. Players choose in which weapon slot to place the new weapon, but doing so drops the existing weapon in that slot. Thus players must be selective about which weapons to keep. In this context, an important goal for any game that generates unpredictable content is to indicate what that content will be like before it is taken. To give players an idea of how a weapon functions before picking it up, weapon pickups emit a miniature particle system preview that behaves exactly as the actual weapon does. In the game this preview is called a *neuralium isotope* (figure 6.1, left side).

6.3 Game Interfaces

As with most video games, GAR is heavily GUI-driven. The following in-game screens describe the majority of GAR functionality.

1. **Main Menu.** The main menu screen (figure 6.5) greets players when the game starts. The following options are available: (1) view an animated scrolling text introduction that describes the game's background story and provides some helpful tips for beginners, (2) start a single player game, (3) start a multiplayer game, (4) set options such as character name, screen resolution, and shader effects, (5) view the animated scrolling text credits, and (6) exit the application.
2. **Multiplayer Menu.** The multiplayer menu screen (figure 6.6) enables players to participate in Internet games with up to 32 other players at once. To connect to a GAR Internet game, the player proceeds in the following manner: (1) select a character name that will be the player's handle in games, (2) set a password with which the player can retrieve persistent statistics and progress from a GAR server, (3) refresh the Internet server, which sends a query for a list of available Internet servers, and (4) choose the server to join. There is also a button to connect to local area network (LAN) or LOCALHOST games (a GAR Server hosted on the same physical machine as the GAR Client).

3. **HUD.** The GAR in-game HUD (bottom of figure 6.7) is always displayed while the player is in space. The HUD displays (1) current player ship armor, shield, and hull strengths, (2) teleport cooldown status, (3) current target and its status, (4) the weapons bank (which enables selecting an active CPPN weapon), and (5) a radar that displays objects in the local star system. The HUD also offers buttons to access to the following screens: (1) the ship screen for detailed ship statistics, (2) the weapons screen with the player’s currently equipped CPPN weapons, (3) the galactic map screen, (4) the logoff screen, which exits the current game, (5) the help screen, (6) a music button that toggles the background music on or off, and finally, (7) the mission menu, which displays the player’s current mission.
4. **Weapons Screen.** The in-game weapons screen (figure 6.8) displays statistics and CPPNs for the three weapons currently held by the player. In the game’s lore, CPPNs are called “neuralium isotopes”, each representing a unique isotope of the strange neuralium element that powers weapons and ships. The CPPNs are displayed in three dimensions and slowly rotate. Taking advantage of the three-dimensional rendering view, CPPN inner nodes are aligned in a circle, making viewing them easy and aesthetically pleasing.
5. **Mods Screen.** On the mods (i.e. ship modifications) menu (figure 6.9), players can upgrade their ship with a variety of modifications including armor, shields, hull, teleportation, and weapons. The mods screen is accessed at any friendly space station.

6. **Buy Ships.** On the buy ships menu (figure 6.10), players can purchase faster and more formidable ships. This menu is accessed at any friendly space station. Higher level and more difficult systems have better ships for sale.
7. **Galactic Map Screen.** The extensive network of star systems in GAR is depicted on the galaxy map screen (figure 6.11). Players utilize jump gates to travel between systems. Systems are designated with a recommended difficulty range for players. Some systems are designated as *PVP*; in such systems players may fight each other. In multiplayer mode, the number of players in each system is displayed.
8. **Missions.** Players progress through a series of increasingly difficult missions accessed through the mission screen (figure 6.12). Missions grant rewards such as experience and credits. The early GAR missions function as an in-game tutorial by instructing players how to perform basic game tasks such as picking up new weapons, selling weapons, upgrading their ship, and traveling through jump gates.
9. **Logoff.** The logoff screen (figure 6.13) enables players to exit the current game. In multiplayer mode, the logoff screen also displays the *leader board* with the top 10 players currently online.
10. **Help.** The help menu (figure 6.14) displays the keyboard configuration and helpful hints on playing GAR.

The next section details the spatial hashing methods utilized in GAR to enable hundreds of ships, asteroids, and particle systems to update and render quickly.



Figure 6.1: **GAR Client 1.0**. Players in GAR pilot their space ship (screen center) from a third-person perspective. This picture demonstrates a player destroying enemies with an evolved corkscrew-shaped weapon. Left of the player ship is a weapon pickup dropped from a destroyed enemy base. A particle system preview emits from the weapon pickup (i.e. “neuralium isotope,” left of player) to visually indicate how the weapon will function before the player picks it up. GAR is designed to look and feel like a near-commercial quality video game to effectively demonstrate the potential of automatic content generation in mainstream games. The GAR Client software is available online at <http://gar.eecs.ucf.edu> and runs on any Windows PC.

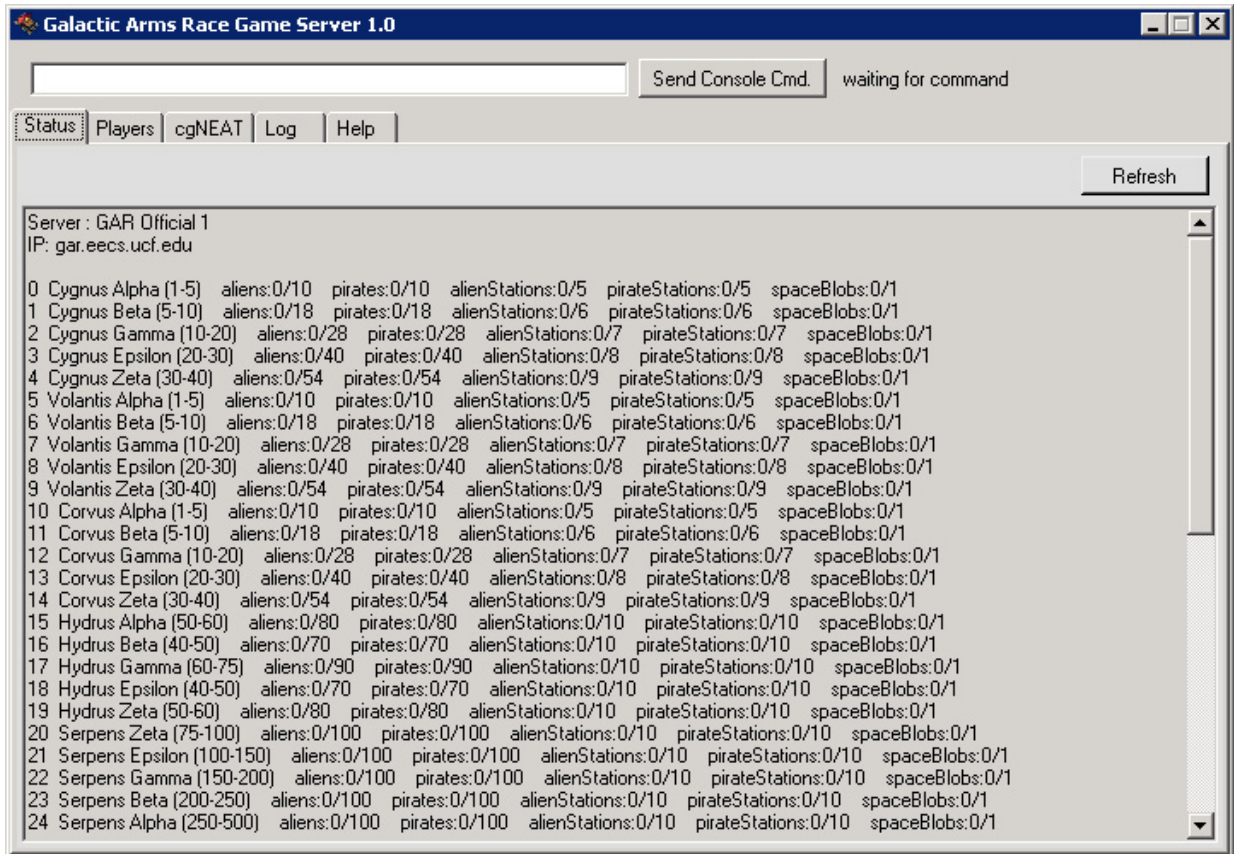


Figure 6.2: **GAR Server 1.0.** The GAR Servers 1.0 software provides a simple GUI through which users can host and monitor their own GAR game world with up to 32 players. At startup and periodically while running, GAR servers contact the GAR Master server to advertise the server name and IP. Players from around the world may then join the game server after retrieving the name and IP from the GAR Master Server. Each GAR Server has a separate persistent database for players that have joined that server; thus cgNEAT-evolved player weapons and progress are saved between online sessions. The GAR Server software is freely available online at <http://gar.eecs.ucf.edu> for those who wish to run their own game world on any Windows or Windows Server PC.

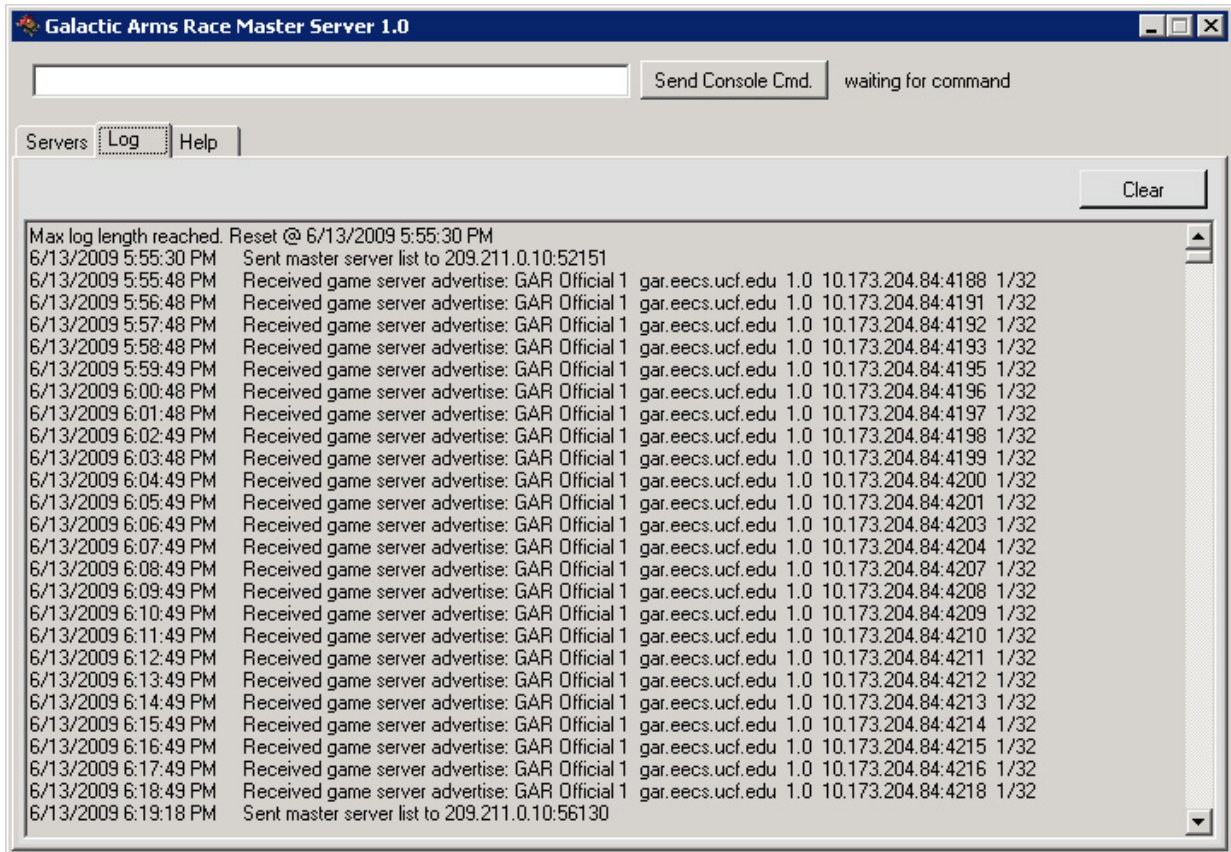


Figure 6.3: **GAR Master Server 1.0**. The GAR multiplayer mode enables players engage with up to 32 other players online. When joining multiplayer mode, GAR Clients first query the GAR Master Server for eligible GAR Servers. Players then select from a list of available servers to join a game. There is only a single GAR Master Server that indexes all GAR Servers, which is hosted by the Evolutionary Complexity Research Group at UCF.

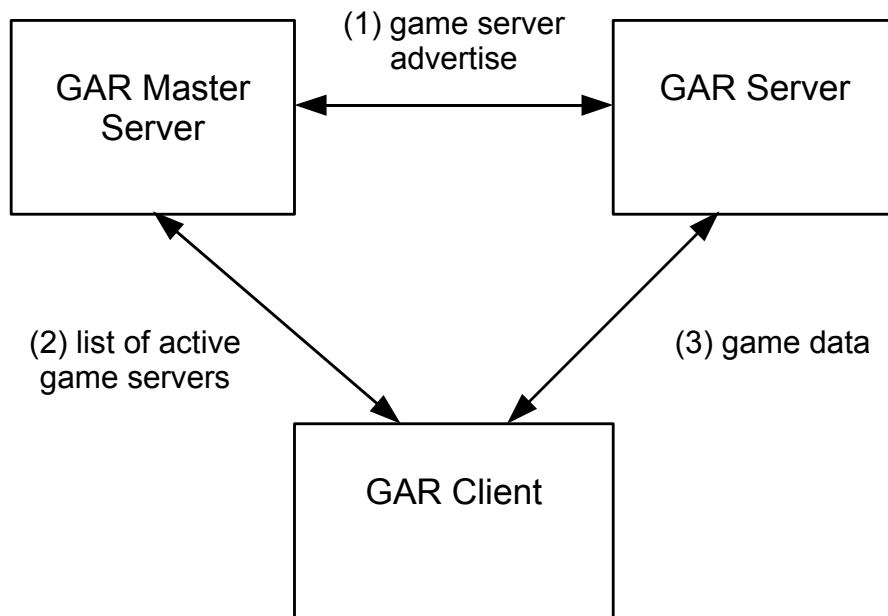


Figure 6.4: **GAR Multiplayer Architecture.** This diagram illustrates the networked communication between the GAR Client, Server, and Master Server. (1) When GAR Servers are online, they periodically advertise to the Master Server which indexes all currently active Game Servers. (2) When a player wishes to join an Internet game, a list of all active game servers is retrieved from the Master Server. (3) The player then chooses from the list of active Internet games and connects to the desired Game Server.



Figure 6.5: **GAR Main Menu.** The main menu is displayed when GAR starts. It has an animated space scene and background music meant to convey the atmosphere of the game.



Figure 6.6: **GAR Multiplayer Menu.** The GAR multiplayer menu enables players to connect to Internet or LAN games. Name and password are set so that game servers can retrieve online progress at a later date. A list of available multiplayer servers retrieved from the GAR Master Server is displayed. There is an additional button at the bottom that connects to a local host (i.e. when the GAR server is on the same machine as the GAR Client) or LAN game.



Figure 6.7: **GAR Head-Up Display (HUD)**. The GAR HUD is always displayed while the player is in game. The lower left panel displays the player's current status. The numbered vertical panels at center-left display which of the player's three weapons is currently active. The center-right panel displays a portrait and information on the player's current target, and the lower-right panel is a radar showing objects in the local system. The seven small buttons above the lower panels give access to the other in-game screens described in this chapter.

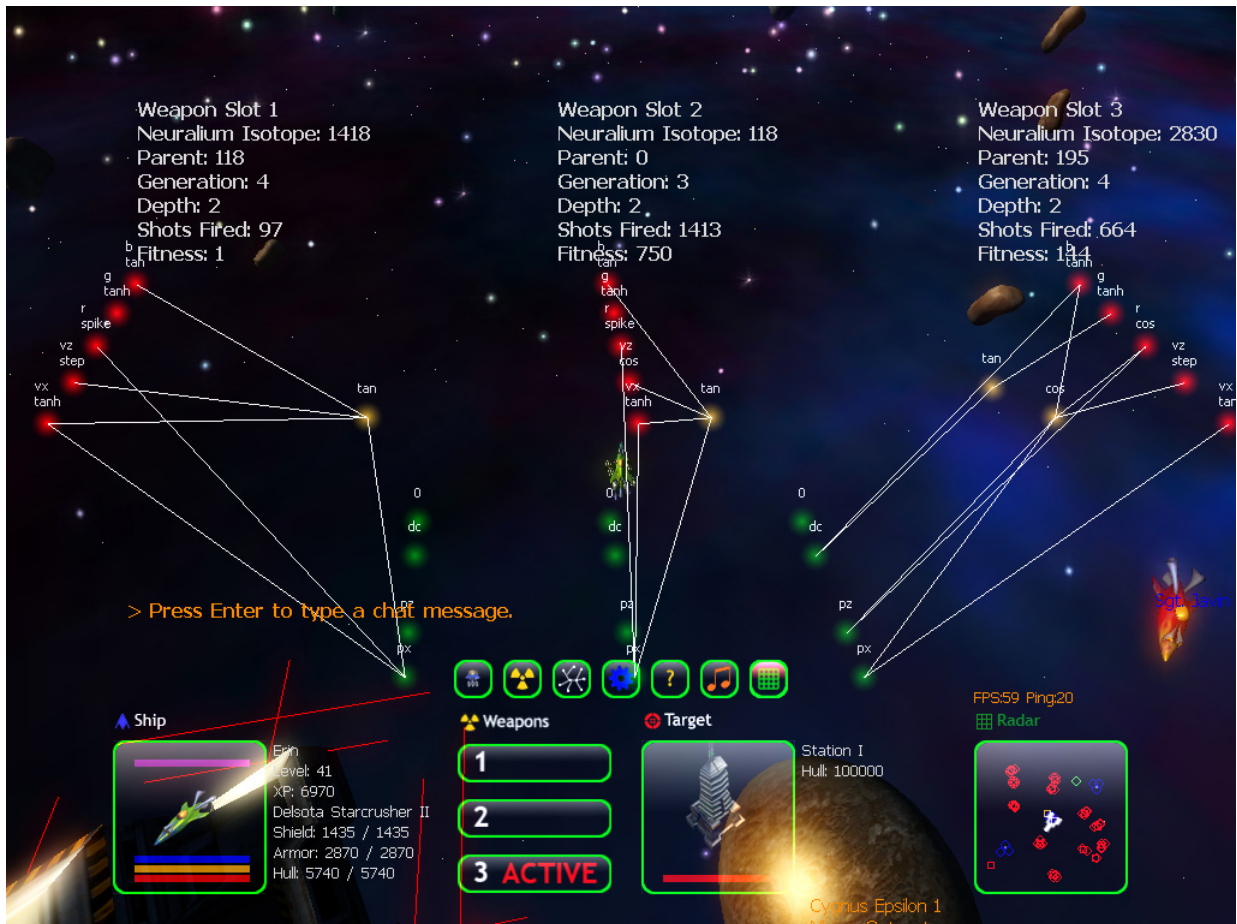


Figure 6.8: **GAR Weapons Screen.** The GAR weapons screen displays information on the player's three current weapons, including fitness and number of shots fired. The actual CPPNs for the weapons are displayed (known in the game as "neuralium isotopes"). A three-dimensional rendering scheme for the CPPNs displays all inner nodes in a circle, which eliminates the need for special methods to clearly display CPPN graphs in two dimensions.



Figure 6.9: **GAR Ship Mods Screen.** GAR offers ships modifications to players for upgrading their ships. Ship mods include upgrades to armor, shields, hull, weapons, and teleport systems. Mods can be installed at any friendly station. Players earn one point for additional mod installs for each level they gain by defeating enemies and completing missions.



Figure 6.10: **GAR Buy Ships Screen.** Players can access the buy ships screen by right-clicking on a friendly station. Stations in each system offer different ships, which are purchased with credits that players earn by defeating enemies and completing missions.

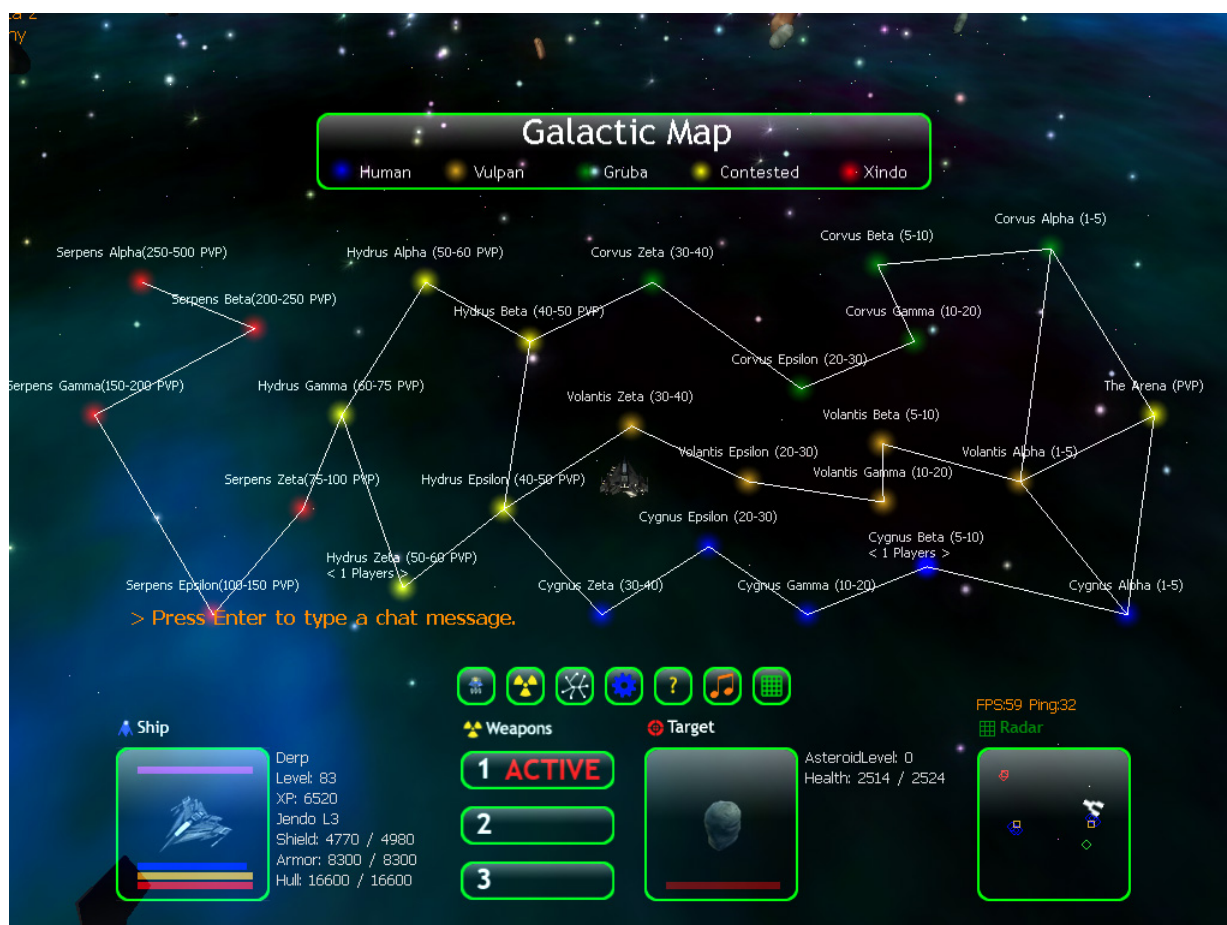


Figure 6.11: **GAR Galaxy Map.** The map screen displays a three-dimensional graph of the GAR galaxy. Colors denote the faction that owns the system, and connections designate routes between systems. Players can travel between systems through a network of jump gates. Additional information displayed on the map screen includes system level, which indicates the relative challenge of the system, and the number of players currently in the system in multiplayer mode.



Figure 6.12: **GAR Mission Screen.** The player's current mission objectives are shown on this screen. Missions award both experience towards level advancement and credits with which ship upgrades, ship mods, and new ships may be purchased. The early missions in GAR function as an in-game tutorial by directing the player to perform objectives integral to the game such as picking up new weapons, installing mods, buying ships, and using jump gates.



Figure 6.13: **GAR Logoff Screen.** The option is given on this screen to exit the current game. In multiplayer mode, the logoff screen displays a leaderboard with statistics on the highest ranked players currently online.



Figure 6.14: **GAR Help Screen.** The GAR help screen displays all keyboard and mouse commands along with helpful hints on game play.

6.4 Spatial Hashing

Spatial hashing is an optimization method that indexes objects in continuous space into discrete *hash buckets* by a *hash function*. Indexing objects in such a manner vastly reduces the computation required to find adjacent objects, which is a common problem in games and simulations. The specific spatial hashing methods in GAR are documented in two papers I co-authored [MG05, HMG04].

Spatial hashing proceeds in GAR as follows.

1. Space is divided into grid cells, or *sectors* (figure 6.15). Although it has no effect on gameplay, players can toggle the grid on or off for aesthetic reasons.
2. A two-dimensional hash table is created, which has an equivalent number of buckets to the number of sectors in the grid.
3. Every frame of animation, a hash function is computed for each mobile object in the game, which places the index of that object into the appropriate hash bucket. Stationary game objects are only hashed once when they spawn into the game. Mobile objects are cleared from the hash table at the beginning of each update.
4. Since GAR game play takes place on the $y = 0$ plane, only two dimensions, x and z , are considered for hashing.
5. The hash function for a single point, (p_x, p_z) , returns a unique sector (s_x, s_z) .

6. The hash function for p_x is $(p_x + max) * \frac{1}{w}$ and similarly for p_z is $(p_z + max) * \frac{1}{w}$, where max is the maximum point (the farthest point to which players and NPCs can travel) in the sector grid that subdivides space and w is the width of a single sector.
7. Game objects such as space ships are significantly larger than a single point, therefore they may hash to multiple cells. However, game objects are strictly smaller than a sector. Therefore, the hash phase will place any game object in up to four hash buckets.
8. All game objects are encased in an axis-aligned bounding box (AABB) defined by two points, min and max , which are the minimum and maximum points of the bounded object.
9. To hash a bounded game object, both min and max of the object's AABB are passed through the hash function, resulting in the object being placed in between one and four hash buckets.

The hash phase is $O(n)$ time complexity, where n is the number of objects hashed. Spatial hashing provides the substantial benefit that *adjacency queries* (e.g. “what objects are near object X?”) become $O(1)$ computational complexity, as opposed to $O(n^2)$ without optimization.

Spatial hashing optimizes GAR in several ways:

1. **Rendering.** Only objects in sectors near the player are rendered.

2. **Collision Detection.** Only objects occupying the same sectors are collided.
3. **Picking.** The sector of the mouse pointer is computed, then the pointer is collided only with objects in that sector.
4. **Sound Effects.** Only sound effects originated at locations in nearby sectors to the player are played.
5. **Visual Effects.** Special visual effects such as explosions, impacts, and floating text are only spawned if they occur in nearby sectors to the player.
6. **Non-Player Character (NPC) Decisions.** All NPCs in GAR are assigned a *sensor radius* defined in sectors. When making decisions, only players or NPCs in sensor range are considered. The hash table makes finding which NPCs are within sensor range an $O(1)$ process.

In this manner spatial hashing optimizes nearly all aspects of the GAR game engine, making it possible for players to explore systems with hundreds of enemies, stations, and asteroids, and engage in firefights with hundreds of particle system projectiles, all of which collide realistically with each other. The next section details real-time soft body model simulation, an additional technology integrated into GAR.

6.5 Soft Body Simulation

GAR contains an enemy called a “Space Blob” (figure 6.16), which is a green blob-like creature animated and rendered based on real-time soft body modeling techniques outlined in a paper I co-authored ([MHG06]). The space blob is a textured polygonal sphere. A *mass-spring system* connects adjacent vertices of the sphere, creating elastic forces between the nodes. Each vertex of the mass spring system has physical properties such as mass, acceleration, and velocity. To simulate volume, the soft body model is assigned an *internal pressure force*, without which it would simply fall flat due to gravity. To keep the space blob vertices from passing through each other, internal collision detection optimized by spatial hashing is performed every frame of animation. When set in motion, the physics, mass-spring system, and internal pressure force combine to convincingly convey the illusion of an undulating blob creature.

Now that all GAR game mechanics and technology have been discussed, the remainder of this chapter details the integration of cgNEAT in GAR, including (1) CPPN representation, (2) calculating weapon fitness, (3) evolving new weapons, and (4) the starter weapons, spawning pool, and archive pool.

6.6 Particle System Weapon CPPNs

Particle system CPPNs in GAR are based on the techniques developed in NEAT Particles and NEAT Projectiles [HGS09b] detailed in chapter 3. Each player weapon contains a single evolved CPPN (figure 6.17). Every frame of animation, each particle issued from the weapon inputs its current position relative to the ship (p_x, p_z) and distance from the ship (d_c) into the CPPN. There are two, rather than three, spatial inputs because the game is entirely situated on the $y = 0$ plane. The CPPN is activated and outputs the particle’s velocity (v_x, v_z) and color (r, g, b) for that animation frame. Representing particle velocity and color in this manner produces a wide of variety of vivid patterns [HGS09b].

When GAR was first released, it implemented force-controlled projectiles (as in NEAT Projectiles) to ensure that all weapons shoot only forwards. However, after further experimentation, it was determined that weapons with particles that move backwards can create compelling patterns (e.g. hurricanes), so the constraint was later lifted.

The next section describes the physics of moving particles in GAR.

6.7 Physics

In GAR, particles are animated in the same way as in NEAT Particles and NEAT Projectiles (Section 3.6). Their position is updated through a linear model based on elapsed time since

the last frame of animation. Thus even players with differing frame rates (e.g. if one computer is slower) should see similar particle weapon patterns for the same weapon.

A similar motion model is applied to asteroids and ships in GAR. The next describes how CPPN projectiles are rendered to the screen.

6.8 Particle Weapon Rendering

GAR renders particles to the screen with point sprites [Lun03], a technique in which two-dimensional bitmap textures are mapped from three-dimensional space to the two-dimensional video screen. Transparency and additive texture blending convincingly conveys the illusion of translucent three-dimensional particles in space.

The point sprite technique is implemented in GAR because it is a common and versatile method to render particles. An alternative particle rendering method called *billboarding* [Fer06], which allows arbitrary warping of particle shapes, is utilized for rendering in NEAT Particles.

There are several ways to optimize particle system rendering including level of detail (LOD) [OFL01], batch rendering [Lun03], and GPU acceleration [Lat04]. The particle representation and rendering system in GAR is compatible with all such methods; however they are not explored in this implementation.

The following sections describe how unique weapons are evolved in GAR by the cgNEAT algorithm.

6.9 Calculating Weapon Fitness

Because it would disrupt the gameplay experience to query the player's opinion of every piece of content, weapon fitness is automatically calculated based on usage statistics. Two potential challenges to calculating fitness based on usage are that (1) certain weapons, by their nature, require more shots to be effective (e.g. wall guns for blocking incoming projectiles) and (2) players that participate in the game more often might disproportionately influence evolution (i.e. by firing their weapons more often). To address these two issues fitness is calculated in GAR in the following manner.

Players possess up to three weapons at one time. When a player fires a weapon, that weapon (which is a unique member of the population) gains fitness at a constant rate and the other weapons in that player's arsenal lose fitness at the same rate. This *fitness decay* mechanism for unused weapons emphasizes emerging new weapon trends and ensures that weapons that require more firing do not come to dominate. Furthermore, the *minimum fitness* is 1 and the *maximum fitness* is 1,000, which means that older players do not create a disproportionate effect. Some of the popular weapons that achieved over 800 fitness in multiplayer games are presented in the next section.

Thus, fitness decay and maximum fitness address the challenges of disparate play times and weapons that require firing more often to be effective. Without these constraints, both scenarios could skew evolution towards weapons that fired many times in the distant past, but are currently unpopular.

The next section describes how new weapons are evolved based on the fitness of the weapons currently possessed by players in the game.

6.10 Evolving New Weapons

When players destroy an enemy station or blob, a new weapon is spawned in one of the following ways: (1) reproduction within the current weapon population selected by roulette [Jon06], (2) from the spawning pool, or (3) random generation. The probabilities for each to occur in GAR 1.0 are:

1. 80%. A roulette die roll based on weapon fitness determines which specific weapon will reproduce. If a starter weapon is selected by the roulette die roll, then a spawning pool weapon is reproduced instead.
2. 10%. The spawning pool is a set of pre-evolved weapons chosen by the game designers.
3. 10%. Random weapons have between one and four hidden nodes and random weights.

Novel weapons created by cgNEAT are evolved from the current *weapon population*. In single-player GAR, the weapon population is only the three weapons the player currently holds. In multi-player GAR, the weapon population includes the weapons currently held by all players. Thus single-player evolution is to some extent greedy; however, it is not equivalent to a normal evolutionary algorithm with a population of three because the player encounters a significant number of weapon previews *in addition* to the weapons in the ship's current arsenal. Therefore, the player is in effect judging such previews by taking them or not. Furthermore, the spawning pool ensures a diverse set of jumping-off points are injected at regular intervals.

As results in this dissertation show, the net effect is that a single player can genuinely discover a diverse array of highly specialized and effective weapons. Because in multiplayer mode the population includes every weapon held by every player in the game, multiplayer mode in GAR is genuine CCE. Figure 6.18 illustrates weapon evolution in GAR with two genealogies of related weapons.

6.11 Starter Weapons, Spawning Pool, and Archive Pool

When the game begins players have no history of weapon preference. One possible policy is to initially give players three random weapons. However, such randomization could cause new players to receive three undesirable weapons. A better solution is for players to begin the game with a predefined set of *starter weapons*. The starter weapons in GAR (1) shoot

only in a straight line, and (2) are not eligible to reproduce during evolution. Thus, new players are guaranteed to begin with viable weapons.

Because starter weapons cannot reproduce and players begin the game with only starter weapons, a method is needed to start evolution. For this purpose, the *spawning pool* is a diverse collection of good weapons evolved by the game developers. If cgNEAT selects a starter weapon to reproduce because it is fired often, a random spawning pool weapon is spawned instead. The advantages of the spawning pool are (1) it jump starts evolution at the beginning of the game and (2) it enables developers to influence what weapons players will see early on, which is a critical time to make a good first impression on players.

Finally, the *archive pool* is where popular weapons are saved. In GAR weapons are saved to the archive if they have been fired over 800 times. In multiplayer mode they are saved to the host server, creating a history of popular weapons in the game. The archive makes it possible to track which weapons players find effective, and it can also serve as a *hall of fame*, to which popular weapons are retired, possibly reappearing later in game.

The next chapter presents results from weapon evolution from real gameplay and analyzes statistics collected to characterize how players interact with GAR.



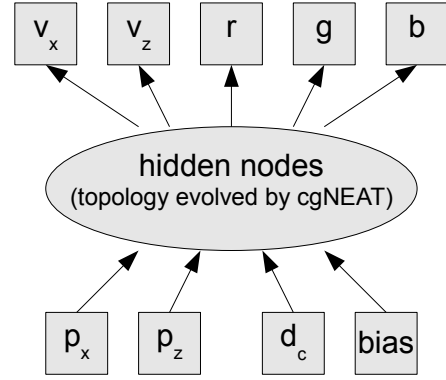
Figure 6.15: **GAR Spatial Hashing.** The spatial hashing methods in GAR are based on two papers I co-authored [MG05, HMG04]. To optimize rendering, collision, picking, sound effects, particle special effects, and AI routines in GAR, each system in GAR is divided into grid cells called sectors (shown above). Every frame, all objects in the game are placed into grid cells by a hash function. This spatial hashing routine optimizes nearly all aspects of the GAR game engine, including rendering, collision, NPC decision routines, picking, sound effects, and special visual effects.



Figure 6.16: **Space Blob**. The “Space Blob” in GAR is a large green blob-like enemy creature that is animated and rendered based on a real-time soft body modeling techniques outlined in a paper I co-authored ([MHG06]). The space blob functions as a powerful “boss” enemy like those commonly seen in other games.



(a)



(b)

Figure 6.17: **How CPPNs Represent Particle Weapons.** (a) Each frame of animation, each particle separately inputs the position (p_x, p_z) and distance (d_c) from where it was *initially* fired into the CPPN (p_y is ignored because the game is situated entirely on the $y = 0$ plane). (b) The CPPN is activated and particle velocity (v_x, v_z) and color (r, g, b) are obtained from CPPN outputs. This method provides GAR with smooth particle animations and a wide variety of possible evolved weapons.

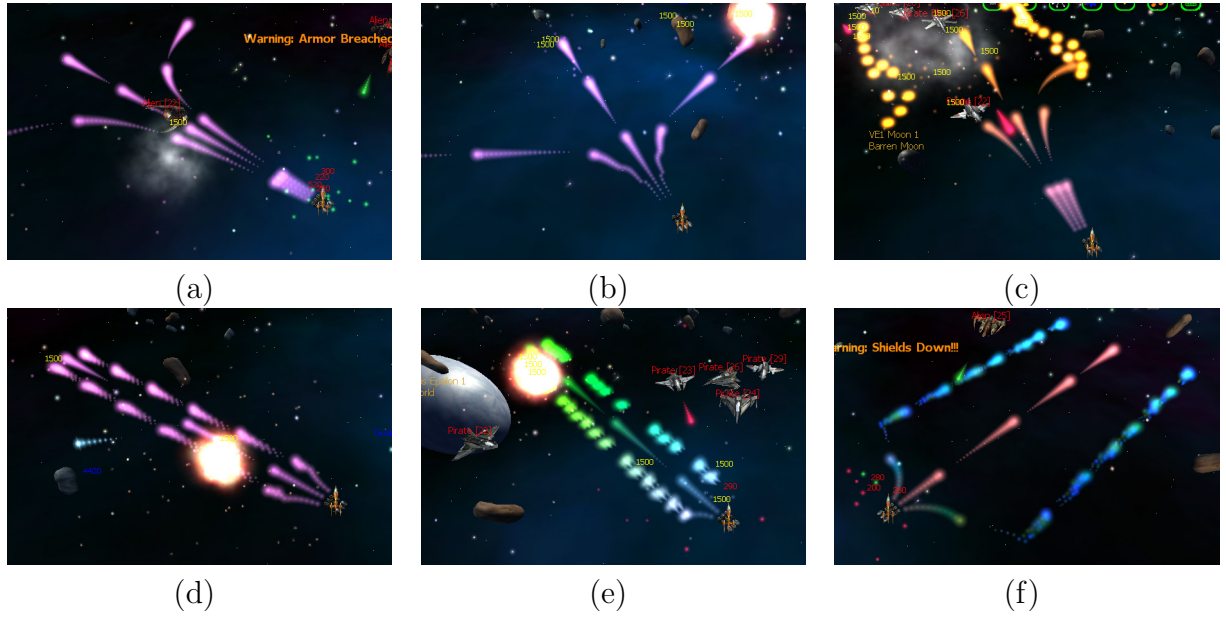


Figure 6.18: **Weapon Evolution Examples.** As weapons evolve over the course of the game, players are likely to find weapons with qualities similar to those they favored in the past. In this example from actual single-player gameplay, the player often fired a spread weapon (a). Later in the game, new spread gun variations (b,c) evolved. Another interesting spread gun (d) fires two slower-firing outer projectiles and a fast inner projectile. Later descendants of this weapon (e,f) exaggerated the speed difference between the inner and outer projectiles, diversified the color pattern, and modified the spread width. These examples illustrate how cgNEAT evolves novel content based on past user preferences.



Figure 6.19: **GAR Starter Weapons.** When the game begins, players are equipped with straight-shooting starter weapons that do not contribute to evolution. Starter weapons ensure that players begin the game with effective weapons (which would not be guaranteed by randomization) and additionally act as a control to which the effectiveness of evolved weapons is compared. Starter weapons do earn fitness by being fired; however, if a starter weapon is selected by a roulette roll during evolutionary selection, a spawning pool weapon is created instead.

CHAPTER 7

AUTOMATIC CONTENT GENERATION RESULTS IN GAR

The results presented in this chapter demonstrate the variety of weapons that are evolved by players in GAR. All weapons displayed are created by the cgNEAT algorithm itself, i.e. not by the game developers. The results of both single player and multiplayer weapon evolution are presented.

Since before GAR was released, the `gar.eecs.ucf.edu` website has attracted over 30,000 visitors from over 100 different countries. On July 8th, 2009, GAR appeared on the popular Internet news site Slashdot (<http://games.slashdot.org/story/09/07/08/1419242/Experimental-Video-Game-Evolves-Its-Own-Content>), highlighting the public's general interest in automated content generation and attracting many players to GAR from around the world.

Two versions of the GAR client were released, GAR 1.0 on June 2, 2009 and GAR 1.1, which addressed many feature requests by players, on July 6th 2009. GAR (mostly version 1.1) has been downloaded over 8,500 times. The results presented in this section are based on evolution data generated by actual players who downloaded and played the game.

Before discussing results, the experimental hypotheses, experimental setup, and hypothesis validation methods are detailed in the next section.

7.1 Hypotheses, Experimental Setup, and Hypothesis Validation

The primary goal of this dissertation is to create an algorithm that automatically generates game content similar to what actual game designers might create, and to evaluate the results in a game. If the algorithm is successful, the weapons generated in GAR will not only be varied, but also effective in the game environment. In addition, they should contribute to player enjoyment, thereby adding *value* to the game. Thus the main dissertation hypotheses are as follows.

1. Hypothesis 1: Because weapons are automatically evolved and not hand-coded, a wide variety of weapons will be generated in GAR, at least on par with, or greater than, the variety of weapons seen in typical shooter games.
2. Hypothesis 2: Because weapons evolve based on the preferences of players, weapon differences will not be simply cosmetic, but will evolve to provide unique tactical effects within the game mechanics and game environment.
3. Hypothesis 3: The search for evolving weapons itself creates an engaging game mechanic, thus providing increased replay value to the game.

The two major experiments that test these hypotheses are: (1) a user study of the GAR single player game presented in Section 7.2, and (2) an analysis of multiplayer results obtained from two months of data from the GAR 32-player server hosted on the UCF campus, presented in Section 7.3.

Because the enjoyability and effectiveness of game mechanics are inherently subjective, validation of game-based hypotheses is by its nature inexact. Nevertheless, the following methods of analysis are employed to validate the three hypotheses as objectively as possible:

1. Validation Method 1: Popular weapons evolved by players are archived and screenshots presented, validating Hypothesis 1 by demonstrating the variety of weapons evolved.
2. Validation Method 2: The most popular weapons evolved are stored and then, if possible, categorized into trends. Each trend is analyzed for tactical implications in the game, thereby validating Hypothesis 2 by demonstrating that weapons evolve to become effective in the game world in different ways.
3. Validation Method 3: Because scripted NPCs are inherently easier to defeat than actual players, an additional archive of popular *player versus player* (PVP) weapons is stored (i.e. only those weapons with which players score PVP kills). The PVP archive is analyzed to provide additional validation for Hypothesis 2.
4. Validation Method 4: The GAR starter weapons provide a control with which to compare evolved weapons. Because players can obtain starter weapons at any time (that is, they can scrap evolved weapons to trade in for starter weapons), their relative popularity compared to evolved weapons can validate Hypotheses 2 and 3.
5. Validation Method 5: Detailed statistics on player behavior will be archived, including shots fired, weapons picked up, levels attained, and number of kills. Such statistics can support validating Hypotheses 1, 2, and 3.

The remainder of the chapter proceeds as follows. First, single-player and multiplayer results are given in Sections 7.2 and 7.3, respectively. Section 7.4 then analyzes these results through the five validation methods, yielding support for the three hypotheses.

7.2 Automatic Content Generation Results in GAR Single Player Mode

To investigate the creativity of GAR in single-player mode before public release of the game, a group of over twenty test players piloted space ships in separate games for at least one hour each. The results in this section (including figures 7.1 and 7.2) are from these test sessions, and are accepted for publication in a paper I co-authored [HGS09c]. Of course, single-player evolution is inherently more limited than multiplayer. However, the main result is that players indeed discovered a variety of genuinely unique weapons with compelling tactical implications and aesthetics, suggesting that even the behavior of a single player is sufficient to produce compelling content evolution.

As the weapons showcased in this chapter will show, the gameplay implications of evolved content sometimes seem intentional, as if designed purposely to create a specific capability. Thus it is important to keep in mind that *all* the weapons are entirely invented by the game itself with no forethought by the game developers. In many cases powerful guns were invented that were unlike anything the developers had seen or imagined before. They often

exhibit both appealing tactical and aesthetic (through changing color patterns) qualities. Yet of course these guns are not the result of random luck either; just as in other evolutionary algorithms, they result from selection pressure, which is wrought by the preferences of the player in GAR. In this way, GAR is a credible demonstration of the potential of this approach.

In GAR it is possible for player projectiles to intercept enemy projectiles. Therefore, several key tactical trade-offs are explored by evolution. Slow projectiles make it easier to block incoming fire whereas fast projectiles are easier to aim at distant enemies. Weapons with a wide spread are more effective at blocking incoming projectiles; however, concentrated patterns more effectively destroy distant targets quickly. Hybrid weapons with variable spread pattern and speed over time evolve as well. Yet these tactical principles are only the beginning. In fact, figures 7.1 and 7.2 present samples of the wide range of generated single-player weapons and describe some of their tactical implications. To highlight the creativity of cgNEAT, we have assigned descriptive names to each such gun to help to more easily appreciate their concept. Two especially interesting evolved weapon types are *wallmakers* (figure 7.2d,e), which literally create a wall of particles in front of the player, and *tunnelmakers* (figure 7.2f), which create a line of particles on either side of the player. Both weapon types are defense-oriented, enabling players to switch between them and more offense-oriented weapons, as the tactical situation dictates. Most importantly, the authors had never conceived of such guns, yet cgNEAT invented them. These examples demonstrate that cgNEAT evolves unique and tactically diverse weapons as the game is played.

It is important to point out that it does not take long for players to begin to find effective weapons. As figures 7.1 and 7.2 show, compelling weapons often arise within the first ten generations (e.g. the *tunnelmaker* in figure 7.2f is from generation two). Furthermore, weapons continue to evolve into novel forms over dozens of generations, such as the *blue ladder* (figure 7.1f) from generation 42.

Finally note that, although the population eligible for reproduction is only three in the single player game, cgNEAT is still capable of generating a large variety of novel content for two reasons. First, players are exposed to many weapons (aside from those they currently possess) by previews of weapons dropped in the game world. Second, for every new weapon generated, there is always a chance of either a random weapon or a weapon from the spawning pool. Thus, a variety of content is possible even with a limited population.

The next section presents results of multiplayer evolution from after GAR’s public release.

7.3 Automatic Content Generation Results in GAR Multiplayer Mode

In total, the GAR client was downloaded over 8,500 times. The experimental data in this section is from the “GAR Official” 32-player public game server hosted by the Evolutionary Complexity Research Group (Eplex) on the UCF campus. The server began collecting

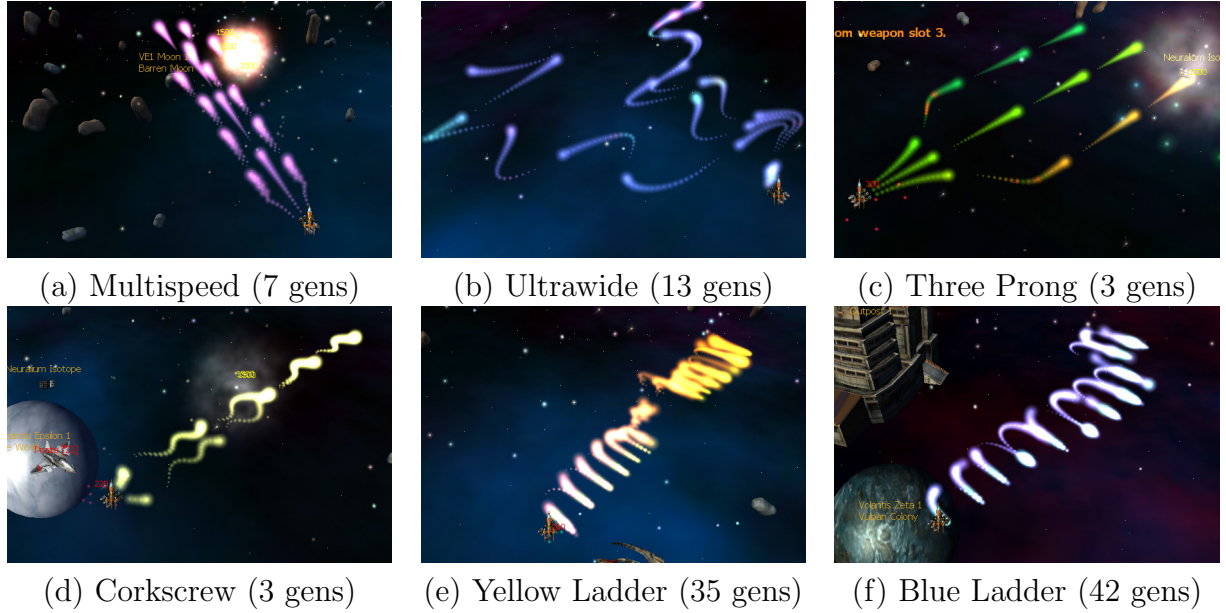


Figure 7.1: **Weapons Evolved During Single Player Gameplay.** GAR players discovered many useful and aesthetically pleasing weapons. The number of generations of reproduction taken to evolve each weapon is shown next to its name. The *multispeed* (a) fires two slow outer projectiles, which are useful for blocking incoming enemy fire, and a fast center projectile for quickly striking distant targets. The *ultrawide* (b) and *three prong* (c) emit wide particle patterns that are effective for fighting many enemies at once. The *corkscrew* (d) emits a pattern that is initially wide, for blocking, but later converges for concentrated damage at a distance. Two version of the *ladder gun* (e,f) fire a wide wave-like pattern that can swivel around obstacles like asteroids. Additionally results are presented in figure 7.2.

multiplayer data from players across the world on June 2nd, 2009. The data presented in tables 7.1, 7.2, and 7.3 is a snapshot in time on July 30th 2009.

In total, 1,007 unique player accounts (table 7.1) were created on the server in approximately two months. Excluded from this data are an additional 236 invalid accounts that were registered but that essentially remained inactive, that is, the players owning these accounts played so little that they never saw a weapon drop and therefore their account data cannot validate or invalidate the hypotheses about the value of weapon drops. In many cases, these

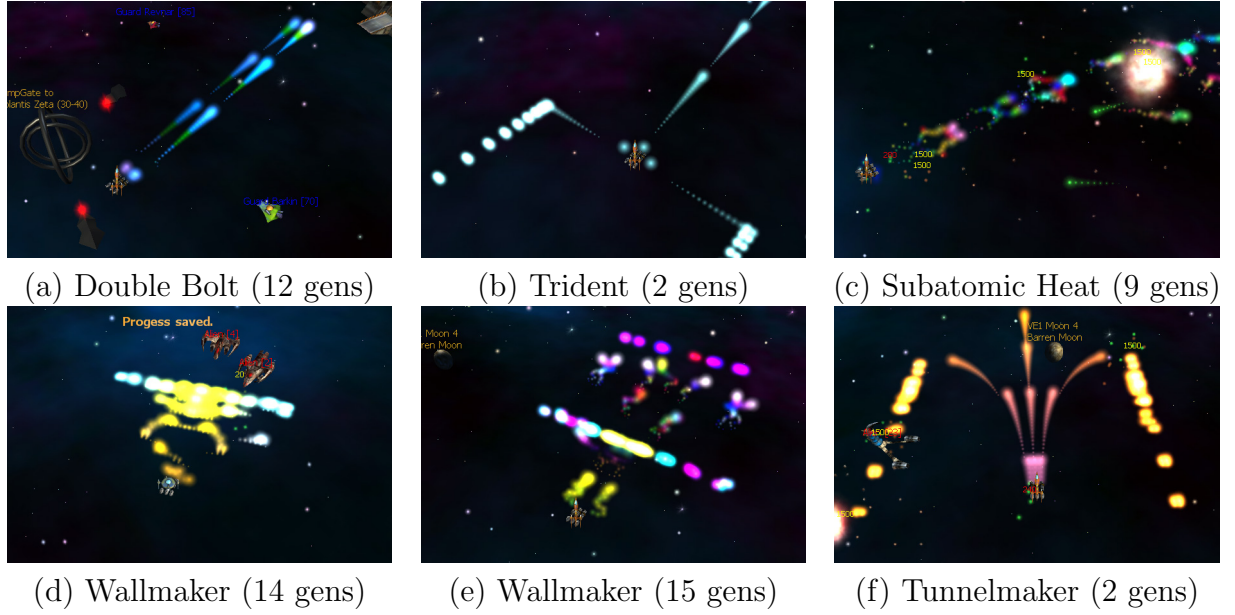


Figure 7.2: Additional Weapons Evolved During Single Player Gameplay. This figure displays additional results to those presented in figure 7.1. The *double bolt* (a) demonstrates that weapons similar to those in typical space shooters can evolve. The *trident* (b) launches a single projectile forward and two perpendicular projectiles that can block enemy fire from the sides. *Subatomic heat* (c) fires a chaotic multi-colored stream resembling bouncing subatomic particles. Two types of *wallmaker* (d,e) literally create defensive walls of particles in front of the player. The *tunnelmaker* (f) creates a defensive line of particles as well, but on both sides of the player, yielding a defensive sheath. These results demonstrate the ability of cgNEAT to generate a tactically and aesthetically diverse and genuinely useful array of weapons. Furthermore, useful weapons appear in early generations and continue to elaborate over successive generations.

unused accounts result from connection problems or players’ computers not meeting the minimum system specifications. At the time of the sample, over 73% of valid player accounts progressed past level 20, indicating significant time invested. A significant proportion of players progressed to much higher levels, which takes several days in-game, suggesting that the evolving content mechanic demonstrated in GAR can enhance game replay value.

The primary method of obtaining new weapons in GAR is to defeat enemies. The aggregated player kill counts for the snapshot are displayed in figure 7.2. The 1,007 players on

the server scored 9,038 PVP kills, 721,456 Alien kills, 714,274 Pirate kills, and 22,670 Space Blob kills. Such large kill totals (over 1.46 million) hint at the attraction of the weapons automatically evolved in GAR for fighting both NPCs and other players.

Snapshot data for weapon evolution on the GAR official server is presented in table 7.3. In total, 379,081 weapons were evolved (note that about 10% of these are from the spawning pool) by players destroying enemies, their bases, and other players. This number is remarkably high for an IEC system. On average, each player encountered over 375 weapon drops. Additionally, players fired over *23.6 million* shots with the evolved weapons they discovered. These results indicate that cgNEAT is capable of exposing players to a large variety of content quickly and, as can be inferred through sheer volume, that players found searching for novel weapons engaging.

Of the 379,081 weapons dropped, 132,722 were picked up by players, which is roughly 35%. This proportion suggests that the GAR weapon previews are effective for giving players a good estimation of what weapons are like, without actually picking them up. That is, players do not need to pick up every weapon they see. At the same time it suggests that a significant proportion of weapons evolved (about one third) are attractive enough to pick up.

The starter weapons in GAR (which shoot in a straight line) act as an experimental control to compare with the weapons evolved by cgNEAT. During the snapshot, the number of starter weapons possessed by players above level 50 was 70, which is only 4% of the total weapons held by those players. Note that in GAR players are able (and even given incentive)

to obtain starter weapons at any point during the game by selling unwanted isotopes. Of the over 1.46 million kills of players and NPCs on the GAR official server, only 22,935 were by starter weapons (roughly 1.5%). From these results it can be inferred through player behavior that they considered the evolved weapons superior to starter weapons.

The total number of combined *generations* of all weapon lineages in the snapshot is 50,646, and the highest generation weapon is 98, suggesting that weapons continue to be effective even into later generations. Additionally, the average number of generations per weapon in the population sample is 16, indicating that it does not take many generations for players to find effective weapons.

Overall, players in the GAR multiplayer experiments discovered a wide variety of both aesthetically and tactically diverse weaponry evolved through their implicit preferences. Discussion of these results and verification of the hypotheses is presented in the next section.

Additional server data that was stored includes the *weapon archive*, where all weapons that were fired at least 800 times (i.e. highly fit weapons) are saved, and the *PVP archive*, where all weapons that score PVP kills are stored. By the time of the snapshot, these archives contained 5,209 and 1,662 weapons, respectively. These numbers suggest a significant number of weapons were highly effective at killing NPCs and other players. The weapons presented in this section from those archives were evolved by players on the official server from around the world.

Figures 7.3 through 7.19 present the visual results of weapon evolution in the form of *weapon trends* on the server. That is, they present general styles of weapons that proved

Player Accounts Data	
Total Player Accounts	1,007
Level 1-20	274
Level 21-40	186
Level 41-60	175
Level 61-80	81
Level 81-100	87
Level 101-120	36
Level 121-140	44
Level 141-160	11
Level 161-180	12
Level 181-200	89

Table 7.1: **Official Server Player Accounts.** A summary of a snapshot of player accounts on the GAR 1.1 Official 32-player server taken on July 30th, 2009 is shown. Over 73% of players progressed past level 20, indicating significant time invested. A large number of players progressed to higher levels, which could take several days in-game, suggesting that the evolving content mechanic enhances replay value.

popular with many players on the server. Note that all weapons in the same trend are not necessarily related to each other, but could be members of separate lineages that followed a similar evolutionary path. All weapons displayed are from the GAR server archive (i.e. fired by their owners at least 800 times) or the PVP archive (i.e. scored kills in PVP). Therefore, it is likely that players found all of these weapons either effective or worth keeping for their novelty. The following trends were discovered:

1. **Corkscrew Guns** (figure 7.3): The corkscrew trend fires fast twisting patterns that are effective for blocking and hitting distant targets.
2. **Double Shot Guns** (figure 7.4): The double shot trend fires two parallel shots that are effective at hitting distant targets.

Kill Data	
Total Kills	1,467,438
Total PVP Kills	9,038
Total Alien Kills	721,456
Total Pirate Kills	714,274
Total Blob Kills	22,670
Total Player Deaths	38,409
Max PVP Kills by a Player	1,147
Max Alien Kills by a Player	10,325
Max Pirate Kills by a Player	10,478
Max Blob Kills by a Player	402

Table 7.2: **Official Server Kill Counts.** Aggregate kill counts and the maximum kill counts for a single player, for the 1,007 player accounts on the GAR 1.1 Official 32-player server snapshot taken on July 30th, 2009, are shown. Such significant totals demonstrate the playability of the weapons evolved by cgNEAT in GAR.

3. **Fork Guns** (figure 7.5): The fork trend produces extremely wide triple shots that are good for firing into crowds of enemies or for hitting fast moving enemies.
4. **Goop Guns** (figure 7.6): The goop trend drops masses of slowly moving particle clouds. Goop guns create effective “space mines” that block incoming bullets and can be dropped while fleeing.
5. **Multi-speed Guns** (figure 7.7): The multi-speed trend fires fast inner shots and slower outer shots. The fast inner shot can easily hit distant targets, while the slower outer shots act as a shield.
6. **Plasma Guns** (figure 7.8): The plasma trend fires random bursts of particles that resemble plasma. Plasma guns are good for blocking and their chaotic patterns make them hard to dodge in PVP combat.

Weapon Evolution Data	
Total Evolved CPPNs	379,081
Total Looted Weapons	132,722
Total Shots by All Players	23,657,178
Total Evolved Gun Kills	1,444,503
Total Starter Gun Kills	22,935
Total Guns Currently Owned by Players Above Level 50:	1,641
Total Starter Guns Currently Owned by Players Above Level 50:	70
Max Generation	98
Total Generations of All Lineages Combined	50,646
Average Shots Fired Per Gun	178
Average Generations Per Gun	16
Guns Shot Over 800 Times	5,209
Guns That Scored PVP kills	1,662

Table 7.3: **Official Server Weapon Evolution.** Aggregate weapon evolution data is shown for the GAR 1.1 Official 32-player server snapshot taken on July 30th, 2009. Given that players fired over 23.6 million shots and looted over 130,000 weapons (about one third of weapons evolved), by sheer volume (and the relatively low number of starter gun kills) it can be inferred that players found the evolving weapon mechanic engaging, and the weapons useful against NPCs and other players. The average weapon generation of 16 suggests that effective weapons can be obtained quickly, and the maximum generation of 98 indicates that weapons continue to be effective many generations later.

7. **Shield Guns** (figure 7.11): Shield guns create a particle shield that completely encases the player ship.
8. **Spread Guns** (figure 7.12): The spread gun trend fires tight streams that widen as they travel away from the player. Spread guns deliver concentrated fire at close range but spread later to make distant targets easier to hit.
9. **Vortex Guns** (figure 7.16): The vortex trends creates patterns that resemble cyclones. Their wide pattern makes it easy to hit enemies and block projectiles.

10. **Wall Guns** (figure 7.17): The wall gun trend creates a literal wall of particles in front of the player that can be used as a defensive shield.
11. **Zig Zag Guns** (figure 7.18): The zig zag trend fires patterns with sharp angles that move quickly and are hard to dodge.
12. **Miscellaneous Guns** (figure 7.19): The weapons in this trend are popular, but do not closely resemble any other weapon trends.

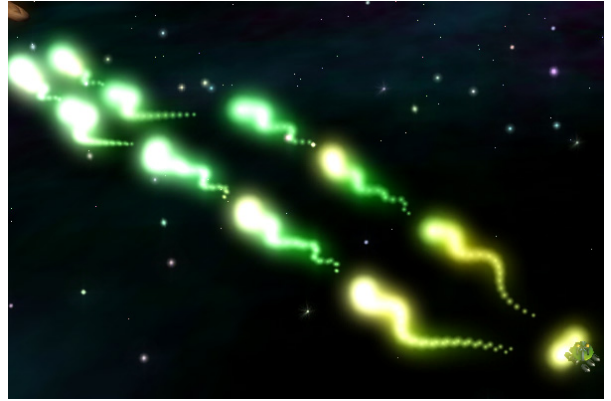
To assess viability of evolved weapons against other players, all weapons that scored PVP kills were saved in the PVP archive, examples of which are presented in figure 7.20. The popular PVP weapons displayed as much variety as the non-PVP weapons; however, one major additional trend that occurred among the PVP weapons is that players significantly more often favored weapons that shoot a particle to a fixed distance at which it stops and remains stationary, leaving a hazard for other players. These weapons differ from the *mine layer* weapons (figure 7.19d) that drop a particle at the same position as the player, which were more prevalent in non-PVP combat.

GAR does also produce weapons that are not picked up or fired often (e.g. weapons that fire very slowly or in extremely erratic patterns). Of course, because players did not use such guns, they did not develop into their own evolutionary trends.

The weapon trends demonstrate the variety of tactically diverse weapons automatically generated by cgNEAT. Like trends in the real world, certain weapon trends were popular for a time before their popularity waned and new weapon trends replaced them. The fact



(a) 2 gens



(b) 4 gens



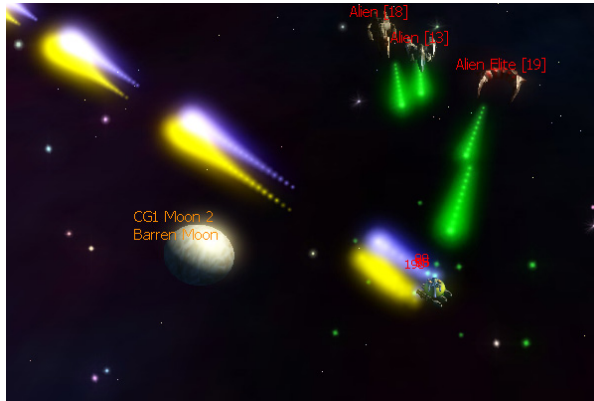
(c) 5 gens



(d) 8 gens

Figure 7.3: **Corkscrew Weapon Trend.** Corkscrew guns fire twisting corkscrew patterns that are effective for blocking and travel quickly for hitting distant targets.

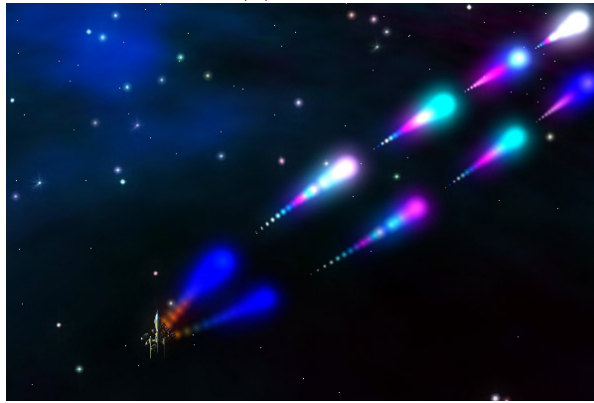
that all weapons shown are from the server archive means that players found them either effective in the game, or novel enough to be worth keeping for a long time. The next section matches the results so far to the five validation methods for the three hypotheses in Section 7.1.



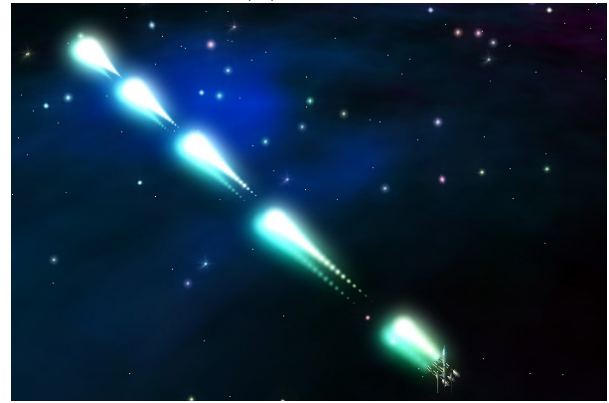
(a) 6 gens



(b) 8 gens



(c) 11 gens



(d) 13 gens

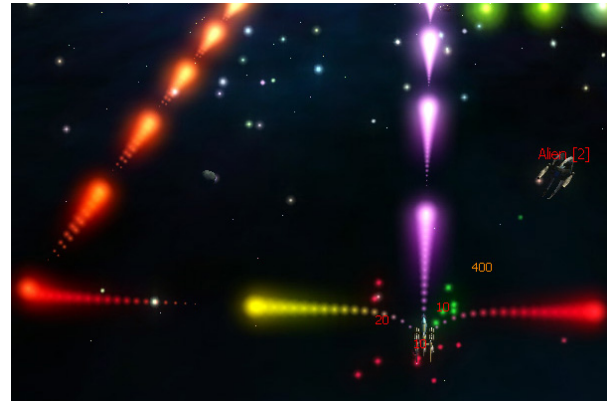
Figure 7.4: **Double Shot Weapon Trend.** Double shot weapons fire two parallel shots, demonstrating that weapons similar to those in typical space shooters can evolve in cgNEAT. Double shot weapons travel quickly and are effective for hitting distant targets.

7.4 Validating Hypotheses

From the results of the experiments, it is now possible to assess the validity of the three hypotheses stated at the beginning of the chapter: (1) the variety of weapons generated will be on par with, or greater than those in typical shooter games, (2) weapons will evolve to produce unique tactical effects within the game mechanics and the environment, and (3) the



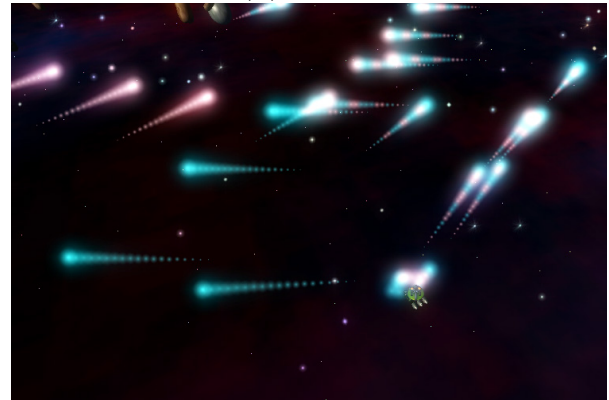
(a) 5 gens



(b) 3 gens



(c) 2 gens



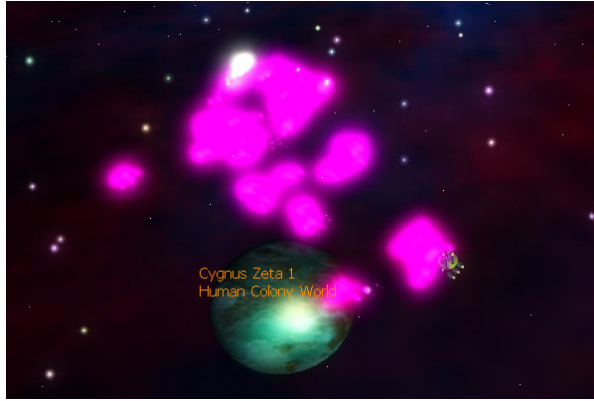
(d) 16 gens

Figure 7.5: **Fork Weapon Trend.** Fork guns fire a wide triple-shot that is effective at firing into crowds of enemies or for hitting distant fast-moving enemies. The projectiles issues from fork gun (d) alternately spread widely and converge toward the center.

search for evolving weapons itself can create an engaging game mechanic, providing greater replayability.

First, the variety of weapons presented (validation methods 1 and 2) in the single player and multiplayer experiments satisfy Hypothesis 1. The variety of weapons types discovered in GAR so far significantly exceeds the typical 8-10 weapons in most space shooter games.

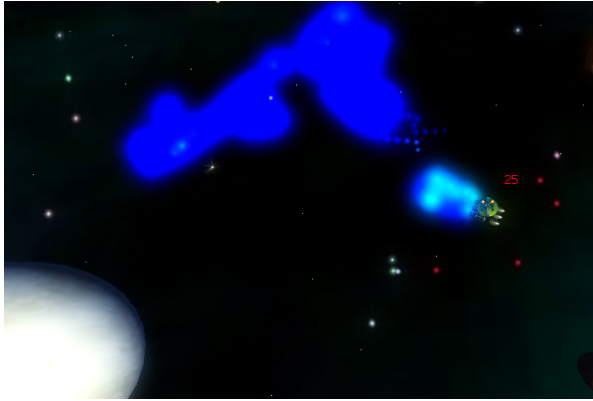
Second, the tactical variance of the weapons players found most effective (i.e. those used most often and stored in the archives for validation methods 2 and 3) supports Hypothesis



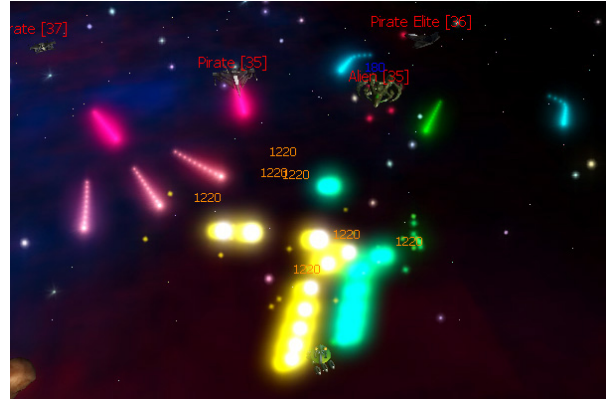
(a) 7 gens



(b) 6 gens



(c) 13 gens



(d) 3 gens

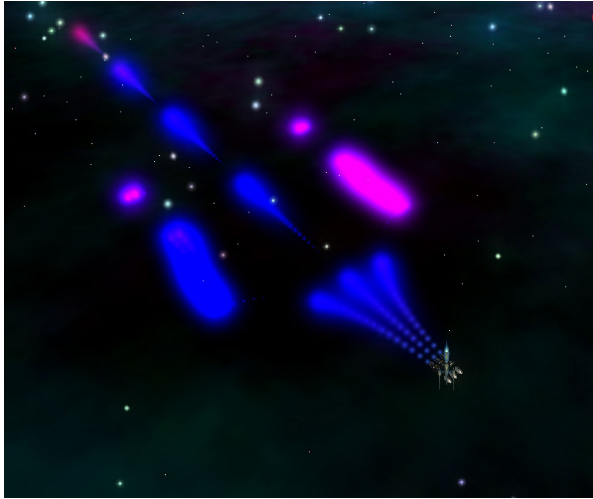
Figure 7.6: **Goop Weapon Trend.** Goop guns drop animated clouds of particles resembling liquids. They create effective “space mines” that can block incoming bullets and that can be dropped as obstacles while fleeing.

2. Only 1.5% of the over 1.46 million kills on the server were by starter weapons (validation method 4). Thus players clearly favored evolved weapons, lending additional support to Hypothesis 2. Players effectively employed a variety of weapon tactics against both NPCs and other players.

Third, the amount of time players willingly participated in GAR helps to validate Hypothesis 3. Just over 1,000 players on the GAR Official multiplayer server fired nearly 23.6 million shots, defeated over 1.4 million NPCs, and engaged in over 8,000 PVP skirmishes

(validation method 5). Thus, the evolving weapon mechanic generated weapons that provided value.

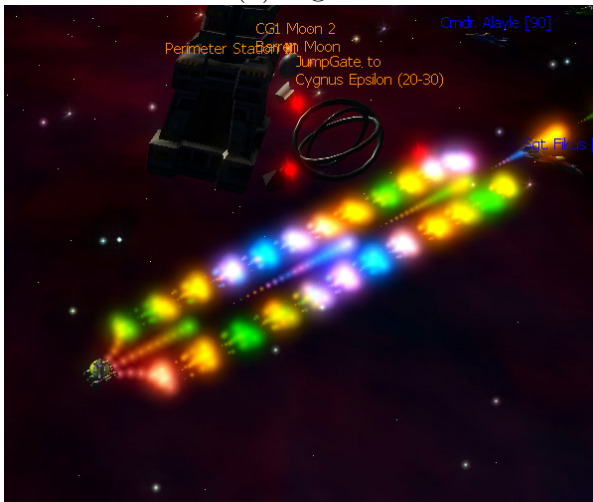
These results indicate that the initial application of cgNEAT to evolving weapons is successful and, because many players found GAR enjoyable enough to play to high levels, that CCE as a game mechanic may yield commercial applications. The future of GAR and the possibilities for evolving content beyond particle weapons is outlined next.



(a) 4 gens



(b) 12 gens

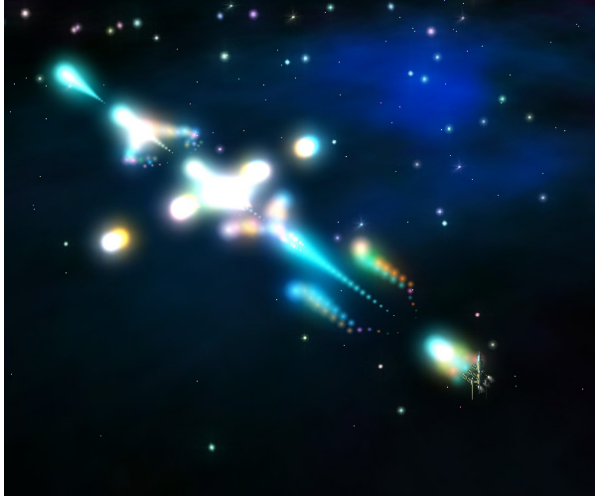


(c) 7 gens

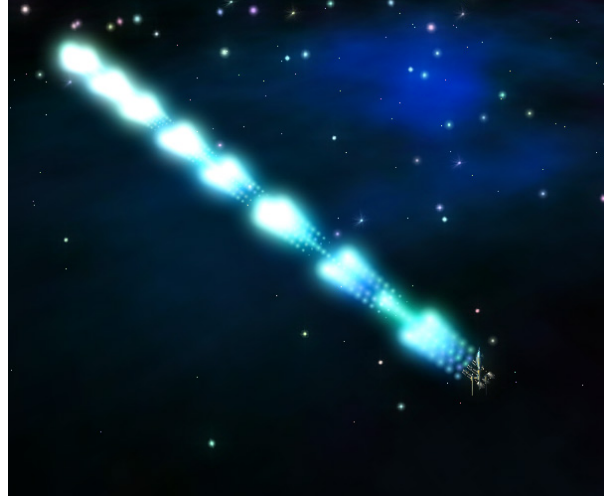


(d) 2 gens

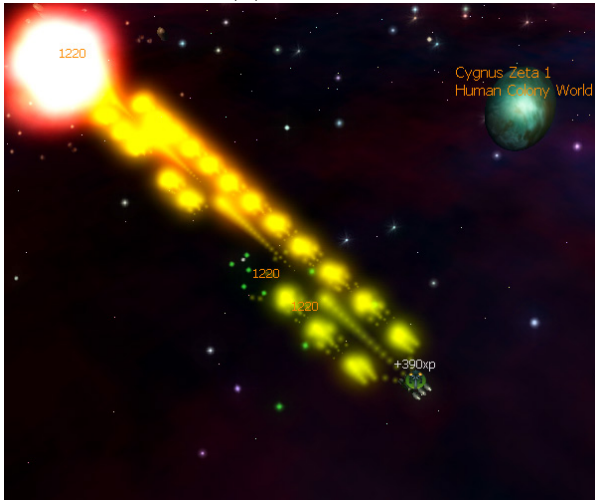
Figure 7.7: **Multispeed Weapon Trend 1.** Multispeed guns, which proved highly popular on the test server, have a fast center projectile and slower outer projectiles that move in a variety of patterns. Examples (a) and (d) are also called “tunnelmakers” because they create a defensive tunnel of near-stationary particles.



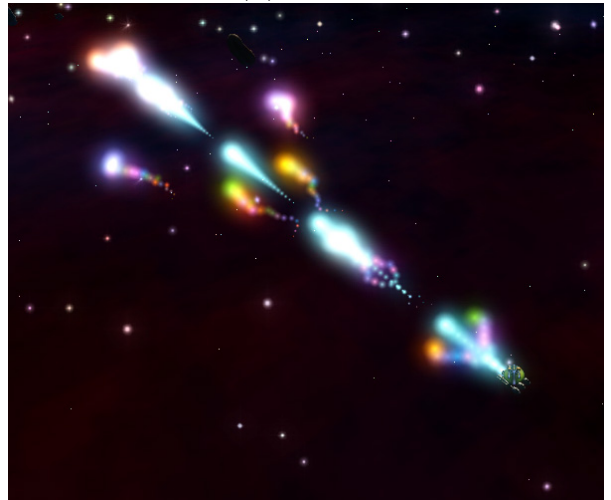
(a) 21 gens



(b) 9 gens



(c) 5 gens

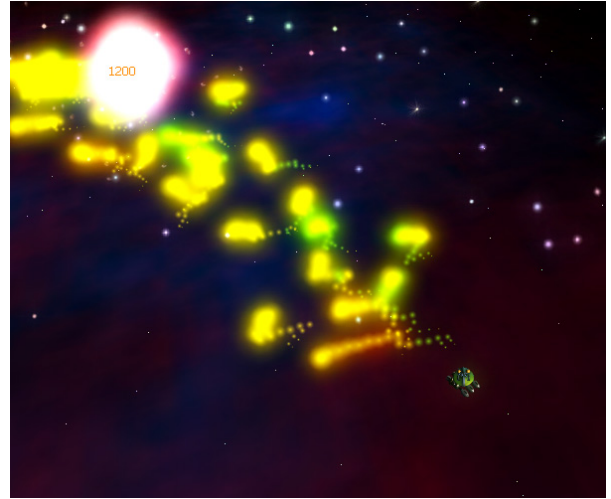


(d) 15 gens

Figure 7.8: **Multispeed Weapon Trend 2.** This figure presents additional Multispeed guns. Example (a) has traits of both multispeed and “wallmakers”; since it acts generally like a multispeed, but creates an intermittent wall of particles at a fixed distance from the player.



(a) 18 gens



(b) 11 gens

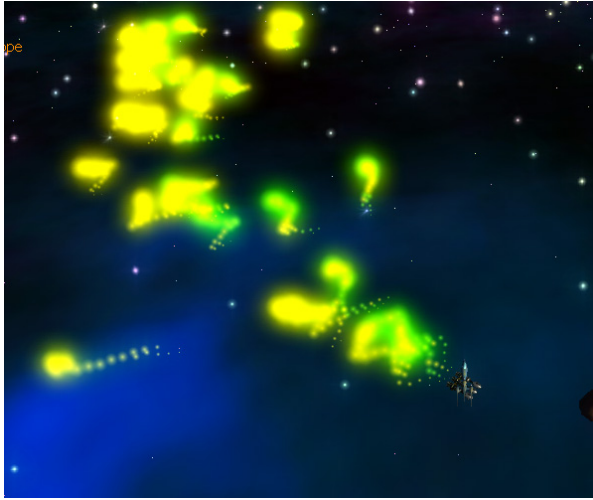


(c) 12 gens



(d) 11 gens

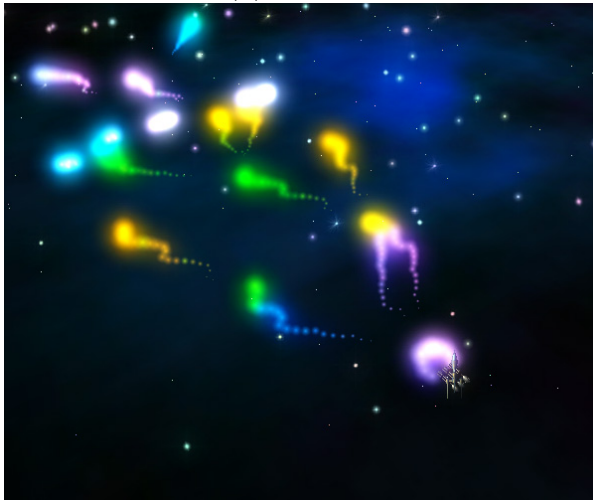
Figure 7.9: **Plasma Weapon Trend 1.** Plasma guns, which also were popular on the server, fire chaotic streams of colorful particles resembling plasma. Plasma guns fire fast and erratic particles that are excellent for blocking incoming projectiles and are difficult for other players to dodge in PVP mode.



(a) 12 gens



(b) 6 gens



(c) 9 gens



(d) 12 gens

Figure 7.10: **Plasma Weapon Trend 2.** This figure presents additional Plasma weapons.



(a) 38 gens



(b) 17 gens

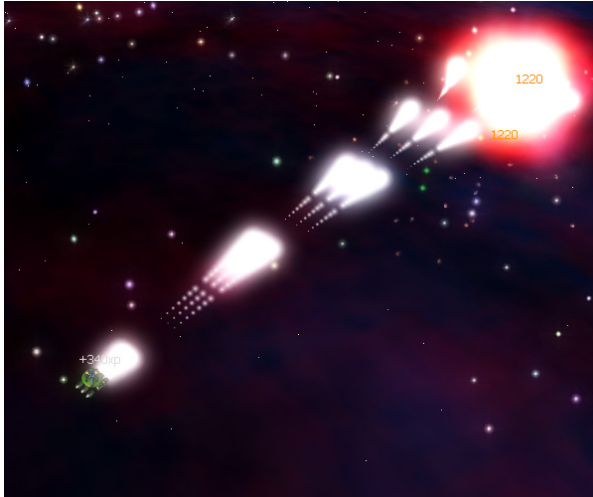


(c) 9 gens

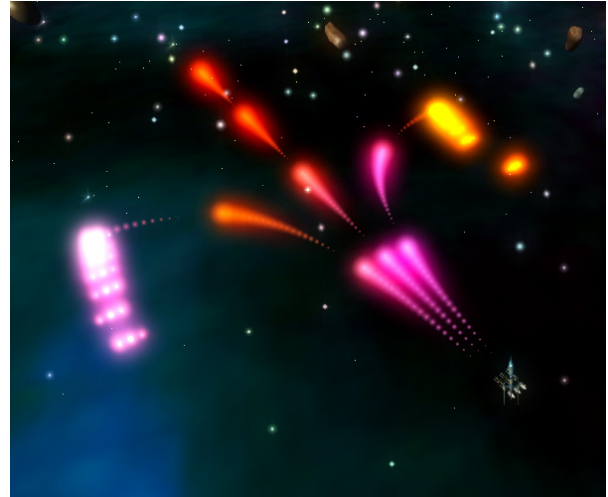


(d) 5 gens

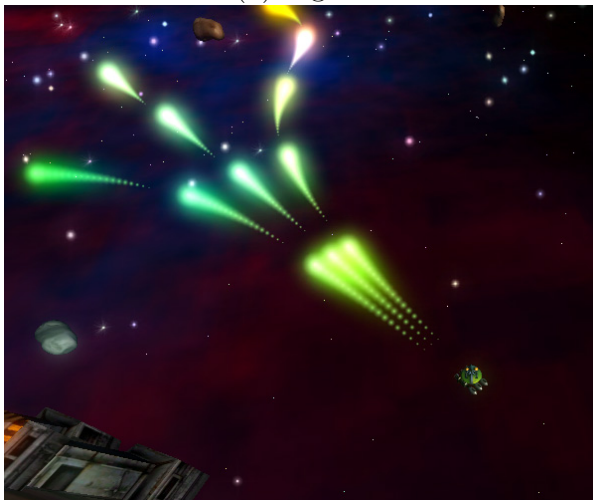
Figure 7.11: **Shield Weapon Trend.** Shield guns are excellent defensive weapons that create a particle shield completely encasing the player ship.



(a) 4 gens



(b) 3 gens

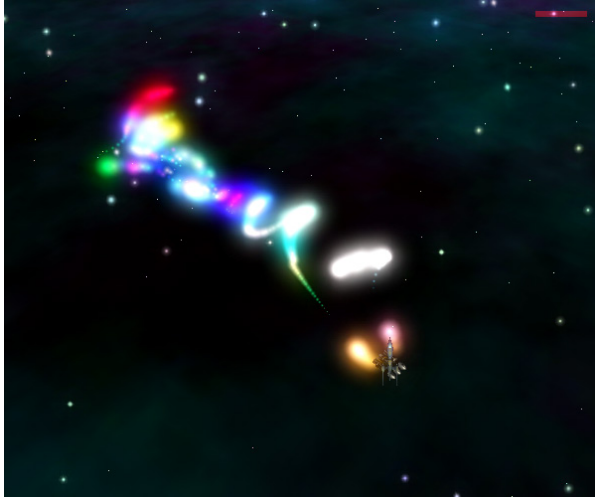


(c) 3 gens

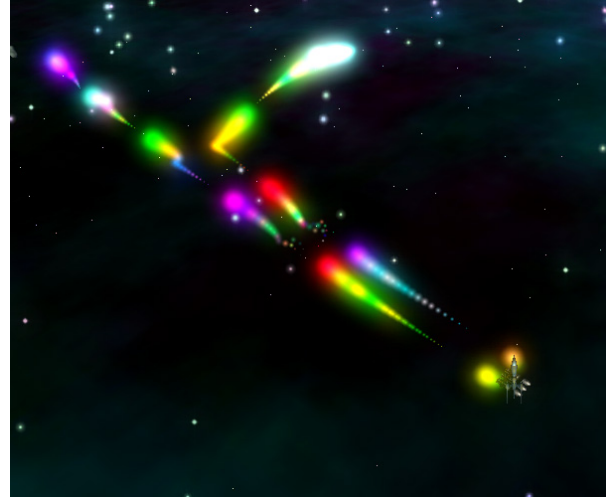


(d) 5 gens

Figure 7.12: **Spread Weapon Trend.** Spread guns fire a tight stream that widens as it travels; thus these weapons deliver concentrated fire at close range but spread later to make distant targets easier to hit. Some spread guns (b) are also called *tunnelmakers* because of the lines of stationary particles created on either side of the player ship.



(a) 22 gens



(b) 21 gens

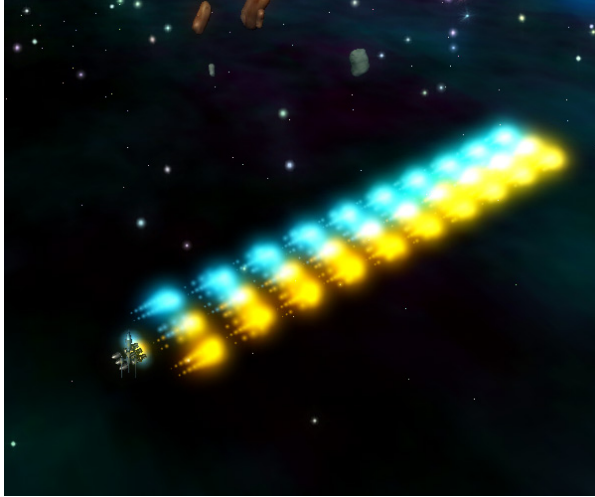


(c) 36 gens

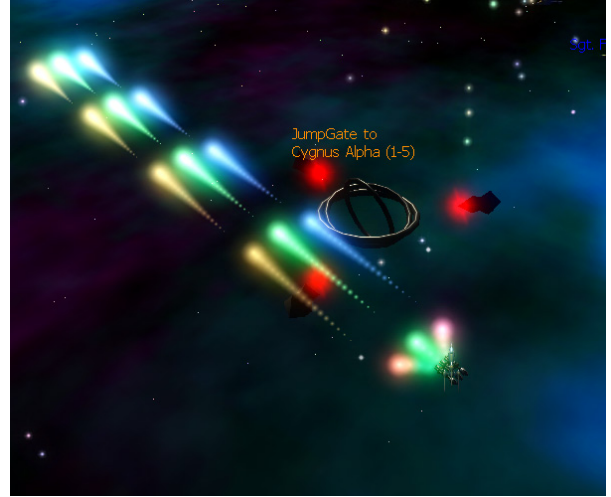


(d) 5 gens

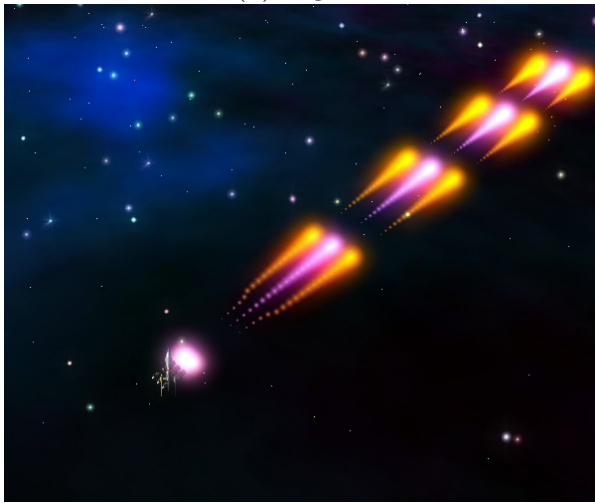
Figure 7.13: **Squiggle Weapon Trend.** Squiggle guns create diverse curved patterns that resemble hand-written script. Squiggle guns display some of the unique color and shape patterns possible through cgNEAT weapon evolution.



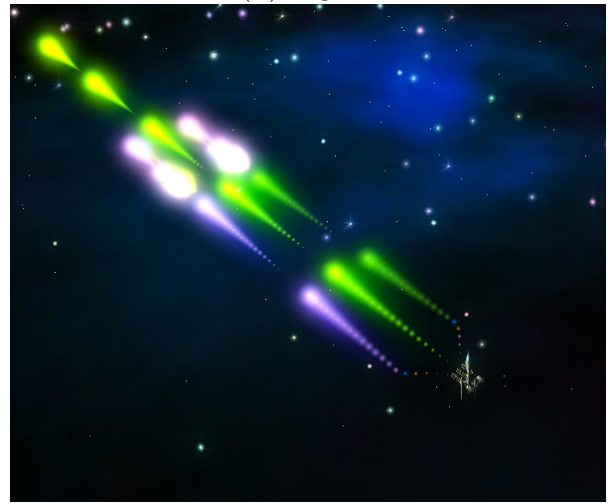
(a) 4 gens



(b) 9 gens



(c) 6 gens



(d) 11 gens

Figure 7.14: **Triple Weapon Trend 1.** Triple shot weapons fire straight patterns with various widths and colors. These weapons demonstrate that GAR, in addition to producing exotic weapons, is also capable of creating weapons similar to those found in typical space shooters.

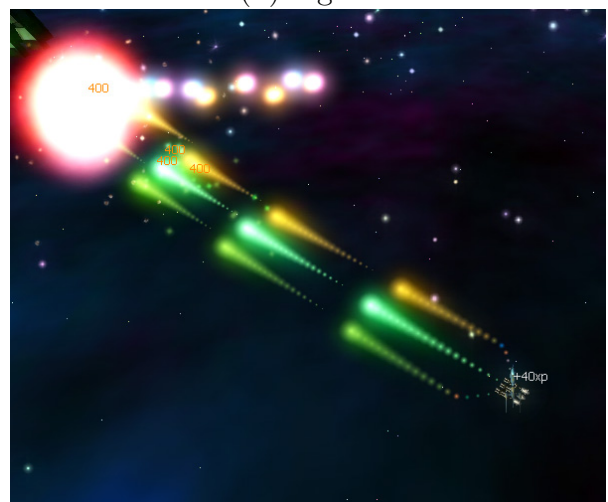
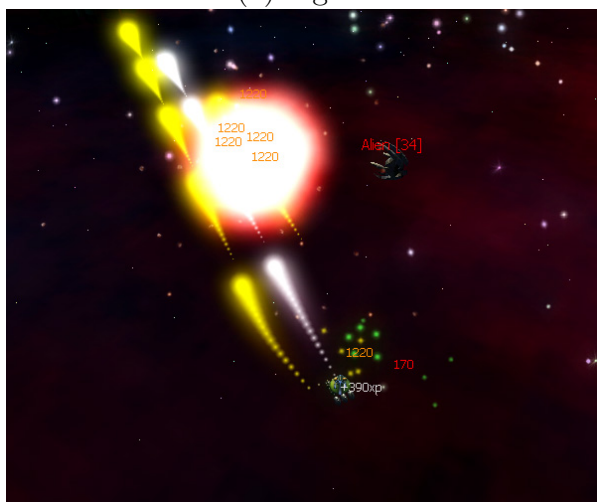
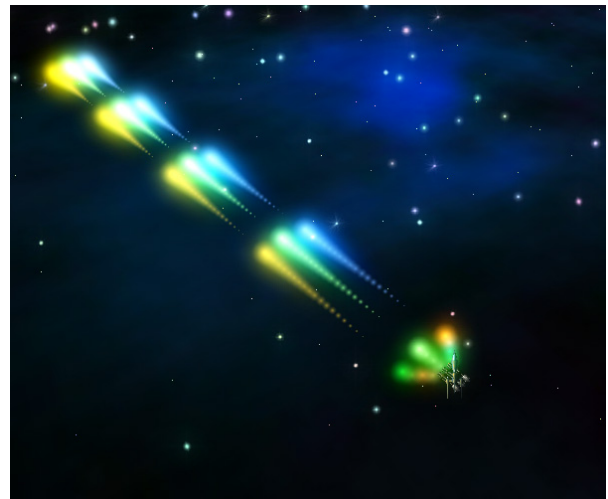


Figure 7.15: **Triple Weapon Trend 2.** This figure presents additional triple shot weapons.



(a) 51 gens



(b) 50 gens

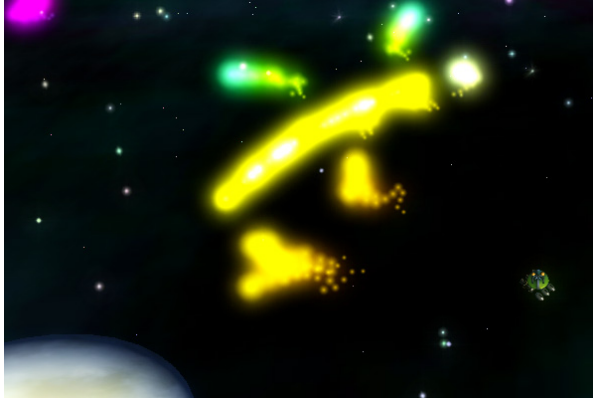


(c) 3 gens



(d) 5 gens

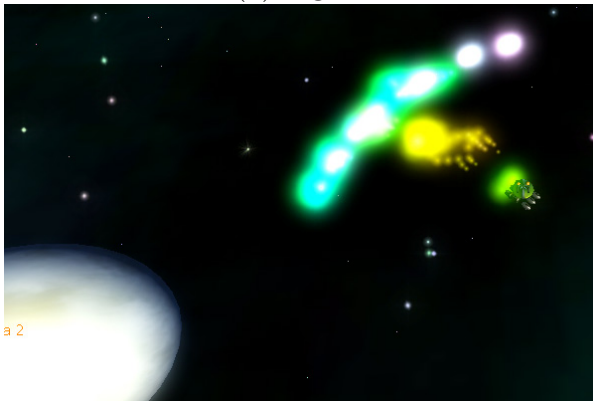
Figure 7.16: **Vortex Weapon Trend.** Vortex weapons produce spinning patterns similar to wind tornadoes. The seemingly random wide patterns are effective for blocking incoming projectiles and make it easy to hit targets.



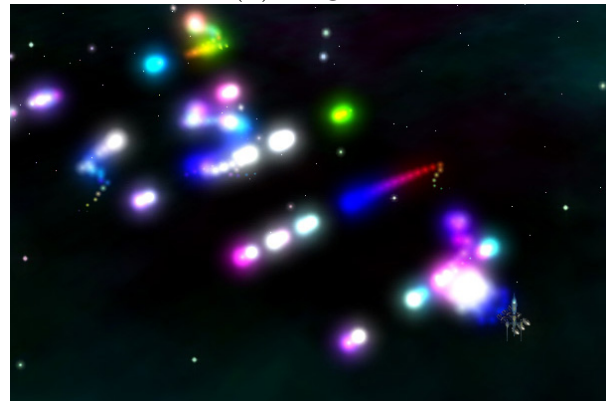
(a) 9 gens



(b) 27 gens

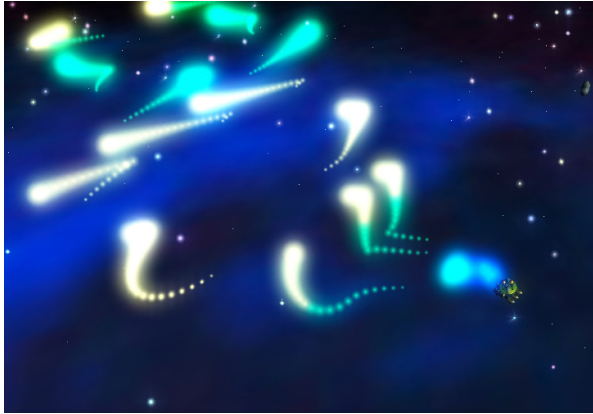


(c) 13 gens



(d) 37 gens

Figure 7.17: **Wall Gun Weapon Trend.** Wall guns create a literal wall of defensive particles in front of the player, useful for blocking or dropping behind while fleeing. Some wall guns such as (b) and (d) create multiple lines of particles.



(a) 8 gens



(b) 11 gens

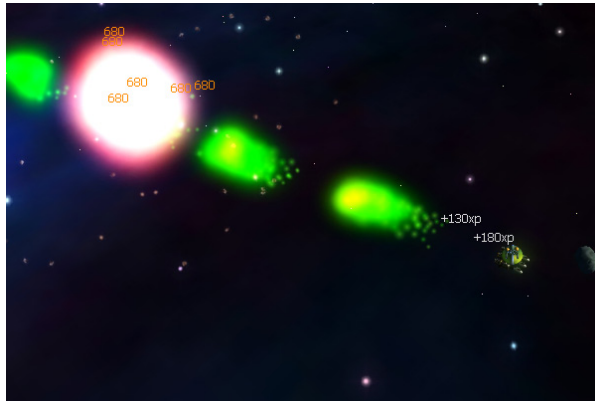


(c) 5 gens



(d) 17 gens

Figure 7.18: **Zig Zag Weapon Trend.** Zigzag guns produce jagged linear patterns, often in very wide formations, making them effective for both blocking and hitting targets.



(a) 16 gens



(b) 13 gens

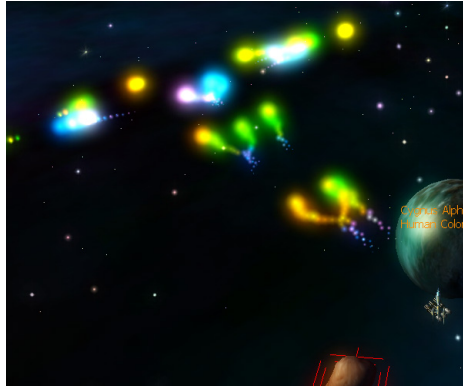


(c) 15 gens

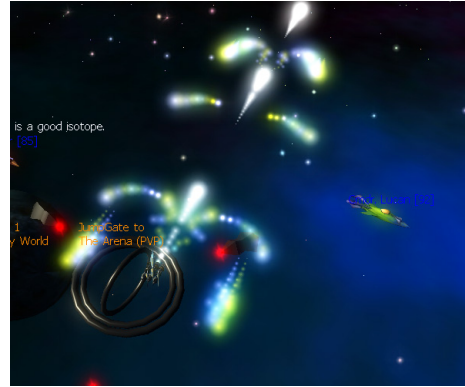


(d) 16 gens

Figure 7.19: **Miscellaneous Evolved Weapons.** These weapons from the GAR server archive are not easily categorized. (a) The *slime gun* fires a large green mass resembling slime. (b) The *rainbow gun* issues colorful curving arcs. (c) The *stealth gun* produces a very faint projectiles. (Note that, due to the rendering methods in GAR, it is impossible to produce completely invisible projectiles.) (d) Several *mine layer* weapons proved popular to drop as traps for enemies pursuing the player.



(a) 15 gens



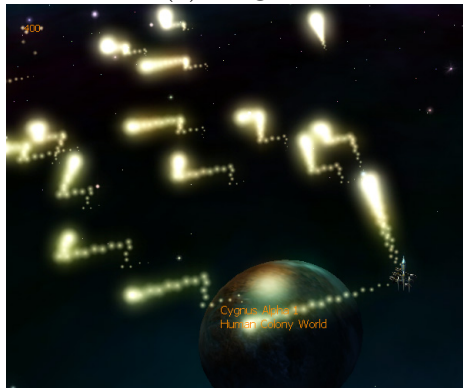
(b) 27 gens



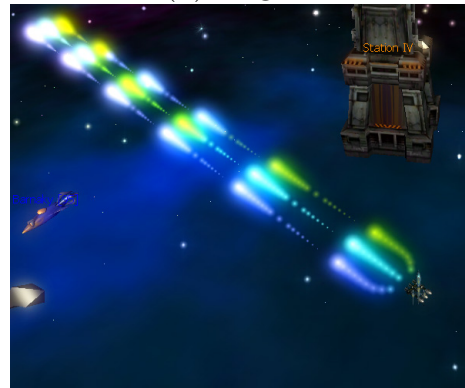
(c) 14 gens



(d) 21 gens



(e) 27 gens



(f) 16 gens

Figure 7.20: **PVP Archive Weapons.** This example displays PVP archive weapons with which players scored PVP kills on the GAR Official 32-player server. Overall, the popular PVP weapons display as much variety as those used solely against NPCs; thus players employed many evolved weapon tactics to defeat each other.

CHAPTER 8

DISCUSSION AND FUTURE OPPORTUNITIES

The cgNEAT weapons in GAR demonstrate that automatic content generation is a viable new technology. The main application is in simulations and games wherein the designers want users to be able to discover and experience a continual stream of new content beyond what the original artists and programmers are able to provide. For players, the main implication is a new kind of experience in which not only is novelty a constant, but the pursuit of novelty itself is an integral part of the game. In fact, players informally indicated enjoying the consistent satisfaction of novel discovery. For some game designers, this loss of control will be viewed as a risky sacrifice; yet others will see it for its potential, just as any new frontier opens up an unknown world of possibilities. In fact, the interactive evolutionary dynamic automatically creates a kind of implicit game balance because, as soon as a player acquires a weapon that tips the equilibrium, variants of that weapon become available to other players in proportion to its use, thereby continually balancing the game.

In addition to weapons, a wide variety of other game content could potentially be generated by cgNEAT including two-dimensional textures, three-dimensional models, many other types of particle effects, and programmable shader effects. Video games that automatically generate their own content (e.g. characters, clothing, weapons, houses, vehicles, music, spe-

cial effects, etc.) could keep players engaged much longer in such a constantly evolving game world than in a static one. Thus the potential future applications of cgNEAT for automated content generation are broad, especially in virtual worlds or massively multiplayer games (MMOGs) in which unique content creates value in the virtual economy.

An interesting issue for future research is the role of pure aesthetics in bringing value and entertainment to game content. To what extent do players value aesthetics above tactical value? While many players and game designers may *think* that players ultimately care about the effectiveness of usable content, it is possible that players sometimes *behave* otherwise. That is, players may sometimes choose content that looks nice over content that is more powerful. One appeal of CCE is that it addresses this issue implicitly without requiring designers to ascertain the answer. If players do prefer aesthetic qualities to utility, that will be reflected in their behavior and thereby amplified by cgNEAT. If they do not, the same is true. Thus, by its implicit nature, cgNEAT can potentially transcend tricky philosophical design questions that are difficult to answer explicitly.

Future work on NEAT Particles will focus on user-designation of inputs and outputs, decoupling particle system creation from programming even further. Essentially, users would be enabled to create their own particle classes suited to the task. Another promising area for future research is applying similar techniques to evolving three dimensional models and programmable shader effects by IEC.

The main goal for the near-term future for GAR is to continue developing the game by adding new levels, ship modifications, star systems, and other types of evolvable content.

A wider release of future GAR versions could yield a significantly broader explosion of content. Potentially, GAR may be open-sourced, enabling others to utilize GAR for research or game development. In the long term, GAR will be hosted permanently by the Evolutionary Complexity Research Group at UCF, and continue to function as a platform for experimental game development.

CHAPTER 9

CONCLUSIONS

This dissertation presented three novel works that together establish that automated content generation in mainstream games is possible. First, NEAT Particles demonstrates evolving three-dimensional particle system effects by IEC. Second, cgNEAT is an algorithm created explicitly to automatically generate game content based on perceived user preferences in real time, as games are played. The cgNEAT algorithm, unlike standard evolutionary algorithms, selects content for reproduction implicitly through player behavior within the game. That is, content players utilize often is more likely to reproduce. The result is a constant stream of novel content suited to players' tastes. Third, cgNEAT is implemented in Galactic Arms Race, a 32-player persistent online game in which particle system weapons evolve based on the preferences of players.

The GAR client was downloaded by thousands of players and over 1,000 players from around the world participated on the gar.eecs.edu server hosted at UCF. Experimental results from the official multiplayer server suggest that cgNEAT is not only capable of automatically creating effective content based on player preferences, but also that the search for novel content can be enjoyable for players as a primary game mechanic.

The initial success of GAR hints at the potential of cgNEAT, and automatic content generation in general, to generate a myriad of other types of content. For players, such a novel content stream can potentially significantly increase game replay value, keeping players engaged in the evolving world. For the game industry, it means that it is possible to build games that automatically create their own content to satisfy users, possibly impacting the way games are made.

LIST OF REFERENCES

- [Aal09] T. Aaltonen et al. “Measurement of the Top Quark Mass with Dilepton Events Selected using Neuroevolution at CDF.” *Physical Review Letters*, 2009. To appear.
- [Alt94] Lee Altenberg. “Evolving Better Representations through Selective Genome Growth.” In *Proceedings of the IEEE World Congress on Computational Intelligence*, pp. 182–187, Piscataway, NJ, 1994. IEEE Press.
- [Bar57] N. A. Barricelli. “Symbiogenetic evolution processes realized by artificial methods.” *Methodos*, 1957.
- [Ber00] J. Van der Berg. “Building an Advanced Particle System.” *Game Developer Magazine*, pp. 44–50, March 2000.
- [Bre94] D. Breen. “A Particle Based Model for Simulating Draping Behavior of Woven Cloth.” *Textile Research Journal*, **64**(11):663–685, 1994.
- [BS02] H. G. Beyer and H. P. Schwefel. “Evolution Strategies: A Comprehensive Introduction.” *Journal of Natural Computing*, **1**(1), 2002.
- [Cra85] N. L. Cramer. “A representation for the Adaptive Generation of Simple Sequential Programs.” *Proceedings of the International Conference on Genetic Algorithms and the Applications*, 1985.
- [Cyb89] G. Cybenko. “Approximation by Superpositions of a Sigmoidal Function.” *Mathematics of Control, Signals, and Systems*, **2**(4):303–314, 1989.
- [Daw86] R. Dawkins. *The Blind Watchmaker*. Longman, Essex, U.K., 1986.
- [Edw06] R. Edwards. “The economics of game publishing.” *IGN Entertainment*, 2006.
- [Ent07] G. Entis. “Recent Accomplishments and Upcoming Challenges for Interactive Graphics in Videogames.”, 2007.
- [EWS96] B. Eberhardt, A. Weber, and W. Strasser. “A Fast, Flexible, Particle-System Model for Cloth Draping.” *IEEE Transactions on Computer Graphics and Applications*, **16**(5), 1996.

- [Fag05] Mattias Fagerlund. “DelphiNEAT-based Genetic Art homepage.” <http://www.cambrianlabs.com/mattias/GeneticArt/>, 2005.
- [Fer06] A. Fernandes. “Lighthouse 3D Billboarding Tutorial.”, 2006.
- [Fon07] A. Fontana. “Genetic Arm 2.0.”, 2007.
- [FOW66] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, 1966.
- [Gal08] S. Galazzo. “Genetic algorithms applied to the theme SEO (Search Engine Optimization).”, 2008.
- [Gam07] Epic Games. “Unreal Engine SDK.”, 2007.
- [GM99] Faustino Gomez and Risto Miikkulainen. “Solving Non-Markovian Control Tasks with Neuroevolution.” In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pp. 1356–1361, San Francisco, 1999. Kaufmann.
- [GWP96] Frederic Gruau, Darrell Whitley, and Larry Pyeatt. “A Comparison Between Cellular Encoding and Direct Encoding for Genetic Neural Networks.” In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pp. 81–89, Cambridge, MA, 1996. MIT Press.
- [Har93] Inman Harvey. *The Artificial Evolution of Adaptive Behavior*. PhD thesis, School of Cognitive and Computing Sciences, University of Sussex, Sussex, 1993.
- [Hay99] Simon Haykin. *Neural Networks, A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, New Jersey, 1999.
- [HGM96] P. Husbands, G. Gerny, M. McIlhagga, and R. Ives. “Two applications of genetic algorithms to component design.” *Evolutionary Computing. LNCS 1143*, pp. 50–61, 1996.
- [HGS07] E. Hastings, R. Guha, and K. O. Stanley. “NEAT Particles: Design, Representation, and Animation of Particle System Effects.” In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.
- [HGS09a] E. Hastings, R. Guha, and K. Stanley. “Interactive Evolution of Particle Systems for Computer Graphics and Animation.” *IEEE Transactions on Evolutionary Computation*, **13**(2), 2009.
- [HGS09b] E. Hastings, R. Guha, and K. O. Stanley. “Interactive Evolution of Particle Systems for Computer Graphics and Animation.” *IEEE Transactions on Evolutionary Computation*, 2009.

- [HGS09c] E. J. Hastings, R. K. Guha, and K. O. Stanley. “Evolving Content in the Galactic Arms Race Video Game.” *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2009.
- [HMG04] E. Hastings, J. Mesit, and R. Guha. “T-Collide: A Temporal, Real-Time Collision Detection Technique for Bounded Objects.” In *5th International Conference on Computer Games: Artificial Intelligence, Design, and Education*, 2004.
- [Hol86] J. H. Holland. “A mathematical framework for studying learning in classifier systems.” *Physica D*, **2**(1-3), 1986.
- [HS09] A. Hoover and K. O. Stanley. “Exploiting Functional Relationships in Musical Composition.” *Connection Science Special Issue on Music, Brain, and Cognition*, 2009. To appear.
- [Irw08] M. J. Irwin. “Game developers’ trade off.” *Forbes.com*, 2008.
- [Jon06] K. A. De Jong. *Evolutionary Computation: A Unified Approach*. The MIT Press, 2006.
- [Lan97] J. Lander. “The Ocean Spray in Your Face.” *Game Developer Magazine*, pp. 13–20, July 1997.
- [Lat04] Lutz Latta. “Building a Million Particle System.” In *Proceedings of Game Developers Conference*, 2004.
- [Lin68] A. Lindenmayer. “Mathematical Models for Cellular Interaction in Development Parts I and II.” *Journal of Theoretical Biology*, **18**:280–299 and 300–315, 1968.
- [LLH04] J. D. Lohn, D. S. Linden, G. S. Hornby, W. F. Kraus, A. Rodriguez, and S. Seufert. “Evolutionary Design of an X-Band Antenna for NASA’s Space Technology 5 Mission.” *IEEE Antenna and Propagation Society International Symposium and USNC/URSI National Radio Science Meeting*, 2004.
- [Lom04] B. Lomborg. “These Hollywood special effects may cost the world 15 trillion.” *Telegraph.co.uk*, 2004.
- [Lun03] Frank D. Luna. *3D Game Programming with Direct X 9.0*. Wordware, 2003.
- [Mar99] Andrew P. Martin. “Increasing Genomic Complexity by Gene Duplication and the Origin of Vertebrates.” *The American Naturalist*, **154**(2):111–128, 1999.
- [MCG03] M. Muller, D. Charypar, and M. Gross. “Particle-based fluid simulation for interactive applications.” In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 154–159, 2003.

- [MG05] E. Hastings J. Mesit and R. Guha. “Optimization of Rendering, Collision and AI Routines in Large-Scale, Real-Time Simulations by Spatial Hashing.” In *Proceedings of the Summer Simulation Conference*, 2005.
- [MHG06] J. Mesit, E. Hastings, and R. Guha. “Multi-Level SB Collide: Collision and Self-Collision in Soft Body Objects.” In *Proceedings of the 8th International Conference on Computer Games: AI and Mobile Systems*, 2006.
- [NTC00a] H. Nishino, H. Takagi, S. Cho, and K. Utsumiya. “A 3D Modeling System for Creative Design.” *Technical Report of the Institute of Electronics, Information, and Communication Engineers*, **100**(461):1–8, 2000.
- [NTC00b] H. Nishino, H. Takagi, S. Cho, and K. Utsumiya. “A Digital Prototyping Sytem for Designing Novel 3d Geometries.” *Proceedings of the 6th International Conference on Virtual Systems and Multimedia*, pp. 473–482, October 2000.
- [NTC01] H. Nishino, H. Takagi, S. Cho, and K. Utsumiya. “A 3D Modeling System for Creative Design.” In *Proceedings of the 15th International Conference on Information Networking*, pp. 479–487, 2001.
- [OFL01] D. Obrien, S. Fisher, and M. Lin. “Automatic Simplification of Particle System Dynamics.” In *Proceedings of the 14th Conference on Computer Animation*, pp. 210–257, 2001.
- [Ree83] W. Reeves. “Particle Systems: A Technique for Modeling a Class of Fuzzy Objects.” *ACM Transactions on Computer Graphics*, **17**(3):91 – 108, 1983.
- [Ree85] W. Reeves. “Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems.” *ACM Transactions on Computer Graphics*, **19**(3):313 – 322, 1985.
- [Rey87] C. Reynolds. “Flocks, Herds, and Schools: A Distributed Behavioral Model.” In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 25 – 34, 1987.
- [Rey99] C. Reynolds. “Steering Behaviors of Autonomous Characters.” In *Proceedings of the Game Developers Conference*, pp. 763–782, 1999.
- [SBD08] J. Secretan, N. Beato, D. B. D’Ambrosio, A. Rodriguez, A. Campbell, and K. O. Stanley. “Picbreeder: Evolving Pictures Collaboratively Online.” In *Proceedings of the Computer Human Interaction Conference*, 2008.
- [SBM05] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. “Real-Time Neuroevolution in the NERO Video Game.” *IEEE Transactions on Evolutionary Computation Special Issue on Evolutionary Computation and Games*, **9**(6):653–668, 2005.

- [SF95] N. Saravanan and David B. Fogel. “Evolving Neural Control Systems.” *IEEE Expert*, pp. 23–27, June 1995.
- [Sim94] K. Sims. “Evolving Virtual Creatures.” In *Proceedings of the ACM Special Interest Group on Graphics and Interactive Techniques*, pp. 50–62, 1994.
- [Ska96] D. M. Skapura. *Building Neural Networks*. ACM Press, 1996.
- [SM02] Kenneth O. Stanley and Risto Miikkulainen. “Evolving Neural Networks Through Augmenting Topologies.” *Evolutionary Computation*, **10**:99–127, 2002.
- [SM04] Kenneth O. Stanley and Risto Miikkulainen. “Competitive Coevolution Through Evolutionary Complexification.” *Journal of Artificial Intelligence Research*, **21**:63–100, 2004.
- [Sof07a] Id Software. “Quake Wars SDK.”, 2007.
- [Sof07b] Valve Software. “Source Engine SDK.”, 2007.
- [Sta06] Kenneth O. Stanley. “Exploiting Regularity Without Development.” In *Proceedings of the AAAI Fall Symposium on Developmental Systems*, Menlo Park, CA, 2006. AAAI Press.
- [Sta07] Kenneth O. Stanley. “Compositional Pattern Producing Networks: A Novel Abstraction of Development.” *Genetic Programming and Evolvable Machines Special Issue on Developmental Systems*, 2007. To appear.
- [Sti06] P. Stiff. “Special effects cost studios big bucks.” *Digital Spy*, 2006.
- [Tak01] H. Takagi. “Interactive Evolutionary Computation: Fusion of the Capacities of EC Optimization and Human Evaluation.” *Proceedings of the IEEE*, **89**(9):1275–1296, 2001.
- [TL99] S. Todd and W. Latham. *Evolutionary Design by Computers*. Morgan Kaufman, 1999.
- [TNL07] J. Togelius, R. De Nardi, and S. M. Lucas. “Towards automatic personalised content creation for racing games.” In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.
- [TNL08] J. Togelius, R. De Nardi, and S. M. Lucas. “An Experiment in Automatic Game Design.” In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.
- [Tsu02] Chieko Tsuneto. “A Fireworks Animation Support System Using Interactive Evolutionary Computation.”. Master’s thesis, Kyushu Institute of Design, 2002.

- [TWS06] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. “Comparing Evolutionary and Temporal Difference Methods in a Reinforcement Learning Domain.” In *GECCO 2006: Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1321–1328, July 2006.
- [Une94] T. Unemi. “Genetic Algorithms and Computer Graphic Arts.” *Journal of Japan Society for Artificial Intelligence*, **9**(4):518–523, 1994.
- [WHR87] James D. Watson, Nancy H. Hopkins, Jeffrey W. Roberts, Joan A. Steitz, and Alan M. Weiner. *Molecular Biology of the Gene Fourth Edition*. The Benjamin Cummings Publishing Company, Inc., Menlo Park, CA, 1987.
- [Yao99] Xin Yao. “Evolving Artificial Neural Networks.” *Proceedings of the IEEE*, **87**(9):1423–1447, 1999.
- [ZM93] Byoung-Tak Zhang and Heinz Muhlenbein. “Evolving Optimal Neural Networks Using Genetic Algorithms with Occam’s Razor.” *CplxSys*, **7**:199–220, 1993.