University of Richmond
UR Scholarship Repository

Honors Theses | Student Research

4-22-2005

Ecological niching in an interactive simulation

Ryan T. Webb
University of Richmond

Follow this and additional works at: https://scholarship.richmond.edu/honors-theses

Part of the Computer Sciences Commons, and the Mathematics Commons

Recommended Citation
Webb, Ryan T., "Ecological niching in an interactive simulation" (2005). *Honors Theses*. 494.
https://scholarship.richmond.edu/honors-theses/494

This Thesis is brought to you for free and open access by the Student Research at UR Scholarship Repository. It has been accepted for inclusion in Honors Theses by an authorized administrator of UR Scholarship Repository. For more information, please contact scholarshiprepository@richmond.edu.

UNIVERSITY OF RICHMOND LIBRARIES
3 3082 00937 3894

Math
Web

Ecological Niching in an Interactive Simulation

Ryan T. Webb
Honors thesis[1]
Department of Mathematics & Computer Science
University of Richmond

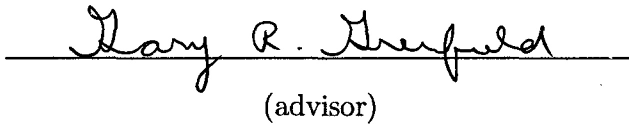
April 22, 2005

[1]Under the direction of Dr. Gary R. Greenfield

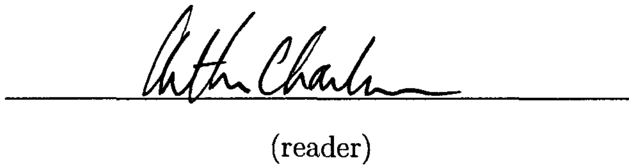
Abstract

Our goal is to create a simulation platform for the study of ecological niching that can be extended to suit the needs of biological research. Ecological niching and the accompanying evolutionary process of speciation are difficult to observe in situ, which makes them prime candidates for study via the methods of computer simulation. To this end, we have created an interactive, real-time ecosystem simulation based on the standard predator/prey interaction model, in which interacting populations of organisms exhibit swarming behavior. We hope to provide the basic simulation components necessary to bring about niching and speciation, that may be extended for the purposes of experimentation.

The signatures below, by the thesis advisor, a departmental reader, and the honors coordinator for computer science, certify that this thesis, prepared by Ryan Webb, has been approved, as to style and content.



(advisor)



(reader)

(honors committee representative)

1 Introduction

The phenomenon of speciation in ecological systems is among the most studied and debated topics in evolutionary biology. There is no complete agreement on the set of ecological conditions necessary for the emergence of a new species, though several modes of speciation have been proposed. The most traditionally accepted mode is allopatric speciation, whereby a new species may arise in an ecological system when a physical barrier is introduced into a population of a particular species. This physical barrier has the effect of splitting the population into multiple subpopulations. A second mode of speciation, parapatric speciation, may occur when a subpopulation migrates to a new ecological space. Implicit in both of these modes of speciation is the concept of an ecological niche. In the broadest terms, an ecological niche is the relational situation of a population within an ecosystem. To refine the idea, a niche includes how a population reacts to and affects the resources and competing organisms in its environment as well as the physical space that the population occupies.

The emergence of ecological niches has been a difficult phenomenon to observe in situ. As with all evolutionary processes, the development of niche species takes place over such a long period of time that it is impossible to observe it in a controlled environment. For this reason, the process of ecological niching is a prime candidate for study via the methods of computer simulation. A computer simulation that models an ecological space and the

phenotypic variations within would benefit greatly from real-time visualization. Furthermore, to extend the metaphor of a virtual laboratory, it is useful to include interactive controls to the simulation that allow real-time alteration of the environment. The objective of our current work is to develop a platform for the study of ecological niching that may be adapted and extended to suit the needs of biological research.

The proposed platform bears resemblance to other recent platforms for ecological simulation, such as *Echo* [3] and *Gecko* [1]. Like these ecology simulators, our work is intended to be a general-purpose simulator that will serve as a platform for experimentation. Unlike *Echo* and *Gecko*, however, our work focuses on swarm dynamics in a three-dimensional space and the evolution of niche species. We have developed a set of rules that bring about niching and speciation in a qualitatively observable way. The ability to generalize in terms of phenotypic traits and fitness function selection were built into our model from the ground up, and interactive controls provide the level of direct user interaction necessary to perform real-time experimentation.

2 Background

Simulated ecosystems in Artificial Life traditionally consist of multiple autonomous agents competing for resources in a shared space. One of the earliest examples of such an ecosystem is Thomas Ray's *Tierra* [4]. In Ray's model, artificial organisms are conceptualized as assembly language programs that compete for CPU time and space in the computer's physical memory.

Using a genetic algorithm, Ray was able to evolve several different "species" of artificial organisms over multiple executions of the *Tierra* simulator. Ray differentiated between these species on the basis of their behavior. The three most common emergent species were parasites, which preempt the CPU time of other creatures, hyper-parasites, which preempt the CPU time of parasites, and hyper hyper parasite cheaters, which preempt the CPU time of hyper parasites. Speciation in *Tierra* occurs through the random mutation of bits in an organisms machine language representation. Because this speciation occurs in a shared space, it is appropriately classified as sympatric speciation. Ray's simulation does not include a real-time visualization, as its execution is not intended to be observable in real time. A particular difficulty with *Tierra* is the fact that in order to obtain any useful data about the evolved organisms, they must be disassembled from their machine language instructions and then exhaustively analyzed. There is no sense of evolution based on transparent, observable phenotypic changes.

Another important example of a simulated ecosystem is Larry Yeager's *PolyWorld* [11]. Conceived as a "real-world" ecology simulator, *PolyWorld* models all of the principal components of a real, living ecosystem. The simulation features biologically motivated genetics, simple physiology and metabolism, artificial neural networks, a visual perception mechanism, and a suite of ecologically based primitive behaviors. The organisms of *PolyWorld* interact in a two-dimensional landscape, sharing and competing for resources. The landscape may be marked by impassable barriers, which can

create niches necessary for speciation. As in *Tierra*, "species" in PolyWorld are differentiated based on common group behaviors. Though each run of the simulator produced somewhat different results, certain species were recurrent across multiple runs. The organisms of the first such recurrent species were dubbed "frenetic joggers" by Yeager. These organisms constantly ran across the landscape at full speed, always wanting to eat and mate. Another recurrent species constantly ran around the edges of the landscape. Speciation in *PolyWorld* occurs by both allopatric and sympatric means. Unlike *Tierra*, *PolyWorld* features a real-time visualization system that represents organisms as solid-colored, two-dimensional polygons on a flat landscape. The overwhelming complexity of the *PolyWorld* simulation makes it an impractical platform for experimentation, however.

Visualization systems for artificial ecosystems that involve large numbers of simulated organisms typically follow the model of *PolyWorld*, representing the environment as a flat, two-dimensional surface. Other examples of this type of visualization system include *Gaïa* [2] and *Darwin Pond* [10]. When dealing with a three-dimensional simulated ecosystem, it is useful to have a cohesion mechanism of some sort that will allow organisms of a similar species to move about in tight clusters. This makes the phenotypic similarities within a species more visually apparent as well as accentuating unique group behavior.

It is to this end that we chose to base our models of organism movement and interaction on the three standard flocking rules described by Craig

Reynolds in his 1987 SIGGRAPH paper "Flocks, Herds, and Schools: A Distributed Behavioral Model." These rules include flock centering (cohesion), velocity matching (alignment), and collision avoidance (separation). The first of these rules, cohesion, is intended to maintain a certain level of clustering between organisms in a flock. The second rule, alignment, models the tendencies of flocking organisms to travel in the same direction. The third rule, separation, attempts to maintain a certain amount of distance between neighboring flock mates within a cluster. When all of these rules are working simultaneously, they have been shown to produce results that qualitatively resemble impromptu flocking behavior [5]. Reynolds' flocking rules are the most effective mechanisms for managing large populations of artificial organisms in the artificial life literature, and the ability to implement them in a real-time context makes them ideal for our work.

Along with visualization systems, artificial ecosystem simulators often include interactive controls to allow for real-time manipulation of the simulated environment. One example of such a simulator is *Darwin Pond* [10]. Following Sims' work on evolving artificial organisms interactively [6], the artificial ecosystem modeled by *Darwin Pond* reacts to user interaction and intervention in the evolution of segmented organisms called "swimmers," that move about in a virtual primordial soup competing for food and reproducing. Interactive controls are provided that allow a user to modify the ecosystem in real time by adding food, erasing food, adding random swimmers, killing swimmers, and altering the genome of a particular swimmer. The various

controls provided alter the environment in such a way that their effects are immediately observable. For instance, if the user adds a large cluster of food to a somewhat remote area of the primordial soup, many swimmers will be drawn to the area. The swimmers will consume the food and proceed to reproduce. Interactive controls of this sort promote the metaphor of a virtual laboratory that supports interactions that are impractical in a real-world laboratory setting.

The line between the virtual laboratory and the real world becomes somewhat blurred in the *A-Volve* installation by Sommerer and Mignonneau [8]. The artificial ecosystem of *A-Volve* evolves in response to user interactions with the real world, as users design virtual organisms on a touch-pad and introduce them into a virtual environment that is projected onto a pool of water. These organisms mate and fight with each other and also respond to user interactions with the water. By moving his or her hand through the water, a user may attempt to guide the movement of a creature in order to protect it from other creatures or to bring it into direct contact with other creatures. Creatures survive contingent upon their ability to reach a target in a certain amount of time, which is contingent upon the form of their user designed body. The most fit creatures will consistently behave as predators, feeding on the energy of less fit creatures.

3 Design Goals

In order to design a general-purpose simulation for the study of ecological niching, it is important to make simplicity the top priority. Within the context of our simulation, this simplicity has been realized at multiple levels of design. It is important to build simplicity into the system from the ground up, beginning with the most fundamental unit of simulation. In this case, the fundamental unit in question is the simulated organism. For the purposes of experimentation, it is necessary to create an organism with a relatively simple genome. Unlike the organisms of *Tierra*, whose genomes are composed of machine language code that must be disassembled and exhaustively analyzed, we propose a design for organisms with a small number of biologically-motivated genes that are expressed as easily-observed, phenotypic traits. In our design, speciation is expressed through variations in these phenotypic traits. It is desirable to enable the easy selection of a phenotypic trait on which to base simulated evolution; that is, the trait by which speciation will be expressed.

The next fundamental unit of simulation is the swarm itself. The inherent simplicity and elegance of Reynolds' flocking rules provide a platform for swarm behavior that is both easy to implement and highly extensible. The basic swarm model can be expanded upon to provide custom behaviors by specifying new rules that can be implemented in much the same way that the cohesion, alignment, and separation rules are implemented. In the interests of

flexibility, it is also important to allow the alteration of certain parameters of the swarm, including swarm size, swarm light orientation, and the weighting of various swarm behaviors and flocking rules. Each of these parameters should be easily accessible and mutable at run time.

Another important simulation unit is the realization of environmental resources. In the context of our work, environmental resources are realized as spotlights from which organisms draw energy by flying within the projected cone of light. For the interest of experimentation and flexibility, it is desirable to be able to modify many of the parameters of the spotlights as well. This may include, but is not limited to, spotlight orientation, intensity, and opening angle. The desire to receive real-time feedback from changes in each of these parameters provides one motivation for including interactive controls.

Another motivation for the inclusion of interactive controls is the desire to create the illusion of working in a virtual laboratory. While it is impossible to observe the entire process of speciation in a controlled environment, such as a laboratory, it is possible to simulate an ecosystem with specific boundaries. This enables a user of the simulation to view anything and everything that occurs within the borders of the artificial ecosystem. The inclusion of interactive controls for real-time user interaction allows for the transparent manipulation of underlying data structures and frees the user from unnecessary concern for the implementation. This does, however, require the inclusion of a real-time visualization system. As with any real-time

visualization system, the primary concerns for our work are clarity of image and speed of visual update. Both goals may be addressed by maintaining a strict adherance to simplicity. The physical structure of organisms must be both distinctive and simple, allowing them to be easily distinguished and drawn quickly. The same rules must apply to the physical structure of the landscape and to the visual representation of spotlights.

As a final concern for the creation of a virtual laboratory metaphor, it must be easy to obtain quantitative data about the behavioral, genotypic, and phenotypic traits exhibited by the organisms. This quantitative data should take the form of output statistics that are dumped to a data file when the simulation terminates. This quantitative data will be an invaluable supplement to the qualitative data provided by the real-time visualization.

4 Implementation

4.1 The Visualization System

The first step in the implementation of our design was the development of a framework for real-time visualization. In keeping with the flexible nature of the platform approach, we chose graphics and windowing libraries that are both widely-supported and compatible across many platforms: OpenGL and GLUT. There are native implementations of OpenGL for all major platforms and it is supported in hardware by many video card vendors. GLUT hides the internals of complex windowing APIs like Windows, X-Windows, and MacOS X, making it easier to initialize a window frame for viewing and

basic GUI components for interactivity. We found GLUT somewhat limiting in its selection of interactive controls, a problem that was addressed much later in development.

The basic components of our visualization system include an orthographic viewing volume, a stationary camera, and a grid of polygons representing a "stage" area. All graphics are drawn in double-buffered mode, and the majority of drawn objects are lit. The main simulation loop is implemented as a procedure that is passed as a function pointer to the GLUT idle callback, which invokes the procedure whenever windowing events are not being received. The result is a constant, real-time update of the all of the ecosystem components, that are then drawn onscreen by a separate routine.

Another important component of the visualization system is the `mesh` class, which encapsulates stored vertices, normals, and faces, as well as the methods necessary to draw them onscreen. This class is primarily used to store geometric descriptions of organism models, which are represented as vertext lists. By allowing models to be stored as vertex lists, we maintain a level of flexibility that enables the geometric structure of organisms to be altered quickly and simply in code.

4.2 Environmental Resources

4.2.1 The `light` Class

Environmental resources in our simulation are realized as spotlights, and each spotlight is inhabited by a swarm of organisms. The spotlight inhab-

ited by the predator population is red, and the spotlight inhabited by the prey population is green. Spotlights are implemented as a special case of the `light` class. An object of type `light` is instantiated by passing a value corresponding to one of the enumerated light sources provided by OpenGL to the constructor. The newly-instantiated spotlight object may then be modified to exhibit a subset of the properties of a standard OpenGL light. The mutable properties of a light include its position, color, orientation, intensity, its status as a positional or directional source, its status as a spotlight or normal light, and its spot cutoff angle. When a change is made to one or more of these properties, a call must be made to the `setup()` method of the `light` class in order for the changes to take effect.

The visualization of spotlights is divided into two components. The first visualization component is based on calls to the native OpenGL lighting system. OpenGL provides calls to initialize a positional light source that behaves like a spotlight, but since light itself is invisible, the existence of a spotlight is only evident when an object passes beneath it. For the purposes of our simulation, it is necessary to visualize the "cone" of the spotlight, so another visualization component is required.

The cone of the spotlight is generated dynamically as a triangular mesh, based on the orientation of the light. Beginning with the vector $\mathbf{o} = (0, -1, 0)$, representing the default orientation vector, the vector is rotated about the z-axis by the spot cutoff angle α using a standard rotation matrix. It is then rotated counter-clockwise about the y-axis in increments of ten degrees until

```
class light
{
  public:
      light(GLenum gll);

      void setup();
      void on();
      void off();
      void draw();

      GLfloat x, y, z;
      GLfloat pitch, yaw, roll;
      GLfloat color[3];
      GLfloat intensity;
      bool positional;
      bool spot;
      GLfloat spot_cutoff;

      .
      .
      .

      vector get_direction();
      bool light_is_on();

  private:

      vector direction;
      bool is_on;
      GLenum gl_light;

      GLfloat fan[LIGHT_CONE_SEGS][3];

};
```

Figure 1: `light` class declaration

reaching 360 degrees. After each rotation, the vector is rotated once again to account for the orientation of the light. The orientation is represented by pitch (ρ), yaw (γ), and roll (ν) angles, which are stored in Euler angle form. Standard rotation matrices are used to compute an orientation vector from these angles, as demonstrated below, where $R = R_\rho \cdot R_\gamma \cdot R_\nu$ is the matrix composed from the three rotation matrices:

$$R_\rho = \begin{bmatrix} \cos\rho & -\sin\rho & 0 \\ \sin\rho & \cos\rho & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R_\gamma = \begin{bmatrix} \cos(\gamma) & 0 & \sin(\gamma) \\ 0 & 1 & 0 \\ -\sin(\gamma) & 0 & \cos(\gamma) \end{bmatrix}$$

$$R_\nu = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\nu) & -\sin(\nu) \\ 0 & \sin(\nu) & \cos(\nu) \end{bmatrix}.$$

After the vector is rotated by R, producing the vector $\mathbf{c}$, the intersection of the stage and the line emanating from the light source in the direction of $\mathbf{c}$ is calculated. The calculated intersection points and the position of the light source form the set of vertices that define the cone mesh.

This mesh is generated when the call to `setup()` is made after the orientation is altered. The mesh is stored as a private data member of the `light` instance and is drawn by the `draw()` method of the `light` class. The cone is rendered as a solid-colored, semi-transparent mesh with the OpenGL lighting system disabled. The transparency of the light is directly related to the light's intensity value; the lower the light's intensity, the more transparent it will be rendered.

4.2.2 Calculating the Intersection of Spotlights

In order to support competition between predator and prey swarms, both swarms must seek to inhabit a common space. In our simulation, this common space is the intersection of the two spotlights. In order for the predator and prey swarms to seek this intersection, it must first be calculated. The initial step in this calculation is to find the "shortest distance line" between the two center lines of the spotlights. Given the equation of the center line of the red spotlight (L_r) and the equation of the center line of the green spotlight (L_g), we must find a point along each line such that the distance between the two points is minimal. Let $Q(s)$ be a point along the parametric representation of the line L_r, and let $R(t)$ be a point along the parametric representation of the line L_g. Also, let l_g and $\mathbf{o}_g$ be the position and orientation vector of the green spotlight, and let l_r and $\mathbf{o}_r$ be the position and orientation vector of the red spotlight. Finally, let $\mathbf{w}(s,t)$ be a vector between points on the two lines. Then,

$$L_r \text{ is } Q(s) = l_r + s\mathbf{o}_r, \quad L_g \text{ is } R(t) = l_g + t\mathbf{o}_g, \text{ and}$$

$$\mathbf{w}(s,t) = Q(s) - R(t) \ .$$

Unless the two lines are parallel, they are closest at unique points $Q(s_c)$ and $R(t_c)$ for which $\mathbf{w}$ attains its minimum length; also, the vector $\mathbf{w}_c = \mathbf{w}(s_c, t_c)$ is uniquely perpendicular to both $\mathbf{o}_r$ and $\mathbf{o}_g$, which is equivalent to satisfying the equations $\mathbf{o}_r \cdot \mathbf{w}_c = 0$ and $\mathbf{o}_g \cdot \mathbf{w}_c = 0$. These equations may

be solved by substituting $\mathbf{w}_c = Q(s_c) - R(s_c) = \mathbf{w}_0 + s_c\mathbf{o}_r - t_c\mathbf{o}_g$, where $\mathbf{w}_0 = Q_0 - R_0$ into each equation to obtain the simultaneous linear equations

$$\begin{cases} (\mathbf{o}_r \cdot \mathbf{o}_r)s_c - (\mathbf{o}_r \cdot \mathbf{o}_g)t_c = -\mathbf{o}_r \cdot \mathbf{w}_0 \\ (\mathbf{o}_g \cdot \mathbf{o}_r)s_c - (\mathbf{o}_g \cdot \mathbf{o}_g)t_c = -\mathbf{o}_g \cdot \mathbf{w}_0 \end{cases} .$$

Letting $a = \mathbf{o}_r \cdot \mathbf{o}_r$, $b = \mathbf{o}_r \cdot \mathbf{o}_g$, $c = \mathbf{o}_g \cdot \mathbf{o}_g$, $d = \mathbf{o}_r \cdot \mathbf{w}_0$, and $e = \mathbf{o}_g \cdot \mathbf{w}_0$, we solve for s_c and t_c as

$$s_c = \frac{be - cd}{ac - b^2} \quad t_c = \frac{ae - bd}{ac - b^2} .$$

The line segment connecting $Q(s_c)$ and $R(t_c)$ is the unique shortest distance segment between L_r and L_g. If the length of this segment is less than the sum of the radii of the spotlight cones about the center points $Q(s_c)$ and $R(t_c)$, then the cones interesect, and the midpoint of the segment is the center point C of the intersection volume.

4.3 Swarms and Artificial Organisms

4.3.1 The `organism` Class

The `organism` class encapsulates all of the functionality of individual artificial organisms. This includes the organism's genome, which consists of its size (a scaling variable) and its field of view. The class also contains members for storing the pitch, yaw, and roll of individual organisms, as well as their position, velocity vector, and the maximum distance within which they will seek a viable mate. The genome consists of only two genes, and only the size gene is variable. The size gene is expressed phenotypically by scaling

```
class organism
{
  public:
     organism(GLfloat ox, GLfloat oy, GLfloat oz, mesh *om);

     void draw();

     vector pos;
     vector vel;
     GLfloat fov;
     GLfloat mate_range;
     GLfloat pitch, yaw, roll;
     GLfloat size;

     organism *next;
     organism *prev;

     GLfloat fitness;

  private:
     void compute_py();
     mesh *org_mesh;
};
```

Figure 2: `organism` class declaration

the geometric representation of an organism by its `size` member. The `size` member is also used in the calculation of the fitness of an organism, which is described in section 4.4.

4.3.2 Implementation of Flocking Rules

The `flock`, `prey_flock`, and `pred_flock` classes encapsulate the functionality for the basic flock, the prey flock F_y, and the predator flock F_r, respectively. Most importantly, these classes provide an implementation of Reynolds' flocking rules. The `flock` class functions as a base class for the `prey_flock` and `pred_flock` classes and implements the basic flocking rules, cohesion, alignment, and separation. These rules are invoked every time the

```
class flock
{
  public:
     flock(int s, int cx, int cy, int cz, mesh *om);

     void update();
     void draw();
     void set_max_velocity(GLfloat mv);

  protected:

     void avoid_world(organism *o);

     organism *first;
     mesh *flock_mesh;
     int size;
     GLfloat max_vel;

};
```

Figure 3: `flock` class declaration

`update()` method of the `flock` class is called. The `flock` class exhibits the impromptu flocking behavior typical of standard swarms.

The derived classes `prey_flock` and `pred_flock` extend the basic Reynolds' rules to provide more specific behavior. The expanded rule set consists of the following:

- Cohesion
- Alignment
- Separation
- Tendency toward native light source
- Tendency to stay within native light cone
- Tendency toward the intersection of two lights

The contribution of each rule to an individual organism's behavior is realized as a vector of a specified magnitude, pointing in an appropriate direction. The final "velocity" vector is a linear combination of the vectors produced by each of the rules listed above. The term "velocity vector" is used in this context because "direction vector" is an inadequate term. The vector produced by the linear combination contains both a heading and a magnitude for the organism's next movement. It is not merely a unit vector, but a linear combination of several scaled vectors.

The cohesion rule is implemented by first calculating the center of mass $M(F_j)$ of the flock F_j, which is the average position of individual flock organisms. A difference vector is then calculated between the flock's center of mass and the position $P(O_i)$ of the organism O_i, where O_i is a member of the flock F_j. This difference vector is normalized and scaled by a positive constant k_1, whose value is dependent on the type of flock performing the update operation. This produces the cohesion vector $\mathbf{c}$:

$$\mathbf{c}(O_i) = k_1 \frac{M(F_j) - P(O_i)}{|M(F_j) - P(O_i)|} .$$

The alignment rule is implemented in a similar fashion. First, an average velocity vector $\mathbf{v}(F_j)$ for the flock is calculated. This velocity vector is then normalized and scaled by a positive constant k_2, producing the alignment vector $\mathbf{a}$:

$$\mathbf{a}(O_i) = k_2 \frac{\mathbf{v}(F_j)}{|\mathbf{v}(F_j)|} .$$

The separation rule is implemented by first seeking the nearest flockmate $N(O_i)$ of O_i. A difference vector is then calculated between the position of O_i and the position of $N(O_i)$. This vector is normalized and scaled by a negative constant k_3 in order to influence the movement of the organism in a direction away from its closest flockmate. This produces the separation vector **s**:

$$\mathbf{s}(O_i) = k_3 \frac{P(N(O_i)) - P(O_i)}{|P(N(O_i)) - P(O_i)|} .$$

The tendency of an organism to move toward its native light source is based on the idea that energy levels near a light source are higher than energy levels far away. Thus, an organism seeks the light source in order to draw more energy, thus gaining greater "fitness." We will return to the idea of fitness in a later section. The vector toward the native light source is computed by finding the difference vector between the position $P(L_n)$ of the native light source L_n and the position of O_i. This vector is normalized and scaled by a positive constant k_4, producing the vector **n**:

$$\mathbf{n}(O_i) = k_4 \frac{P(L_n) - P(O_i)}{|P(L_n) - P(O_i)|} .$$

The tendency of an organism to stay within its native light cone models the desire of an artificial organism to remain under the spotlight in order to

acquire fitness. This rule is only invoked when an organism strays outside of its native light cone. In order to determine whether an organism is within its native light cone or not, the `in_cone()` function (expressed symbolically as $in(O_i, L_n)$) is invoked. This function projects the current position of an organism onto two planes that contain the origin point $T(L_n)$ of the native spotlight and are parallel to the yz-plane and to the xy-plane, respectively, producing points P_1 and P_2. Angles β and δ are then computed. These are the angles between each of the projected points and the line L_I created by the intersection of the two planes, using the origin of the native light as a reference point. The computed angles are then tested to determine whether or not they lie within the spot cutoff range with respect to the pitch and roll angles of the native spot light. If the computed angles do lie within the spot cutoff range, then the organism is inside the spotlight cone. Otherwise, the organism is not within the spotlight cone, and the function returns `false`. If the organism is not within the cone, a difference vector is calculated between the organism's current position and the closest point $Q(L_n)$ on the center line of the native spotlight. This difference vector is normalized and scaled by a positive constant k_5, producing the vector $\mathbf{l}$:

$$\mathbf{l}(O_i) = \begin{cases} \mathbf{0} & \text{if } in(O_i, L_n) = 1 \\ k_5 \frac{Q(L_n) - P(O_i)}{|Q(L_n) - P(O_i)|} & \text{if } in(O_i, L_n) = 0 \end{cases} .$$

When the two spotlights intersect and the center of intersection is calculated, the organisms of both the predator swarm and the prey swarm have a tendency to move toward the intersection. This models the desire of the

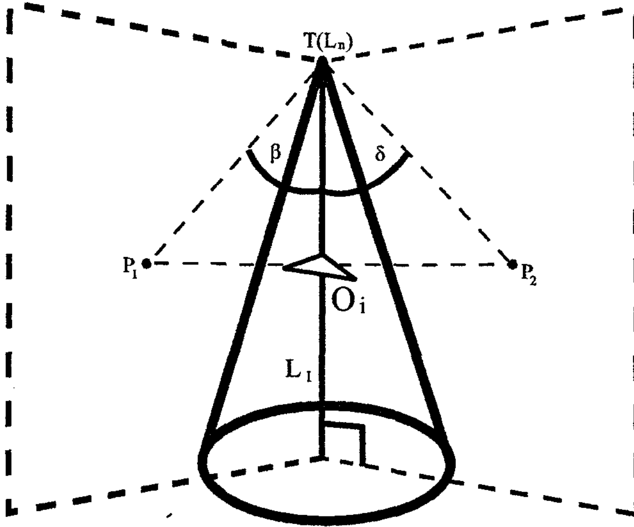


Figure 4: Diagram of an organism well in front of the spotlight cone

predator organisms to inhabit common space with the prey organisms and models the desire of the prey organisms to maximize their fitness by inhabiting two spotlights at once. A difference vector is calculated between the center of the intersection C and the position of the organism O_i. This difference vector is normalized and scaled by a positive constant k_6, thus producing the vector $\mathbf{i}$:

$$\mathbf{i}(O_i) = k_6 \frac{C - P(O_i)}{|C - P(O_i)|} .$$

Prey organisms follow another rule in addition to the six rules common to both predator and prey populations: prey organisms must avoid predator organisms at all costs. This rule is implemented by finding the closest predator organism $R(O_i)$ to the prey organism O_i. If the predator is within

the prey organism's field of view, a difference vector is calculated between predator organism and the prey organism. This vector is normalized and scaled by a negative constant k_7 in order to influence the movement of the prey away from the predator. This produces the vector f:

$$\mathbf{f}(O_i) = \begin{cases} \mathbf{0} & \text{if } O_i \in F_r \\ k_7 \frac{P(R(O_i))) - P(O_i)}{|P(R(O_i))) - P(O_i)|} & \text{if } O_i \in F_y \end{cases} .$$

Once these vectors have been calculated, they are combined with a scaled version of the current velocity vector $\mathbf{v}'(O_i)$, to account for inertia. The resulting linear combination is normalized and scaled by a maximum magnitude m if the magnitude of the combination is greater than or equal to m. This allows the speed of individual organisms to be effectively capped. The resulting velocity vector $\mathbf{v}(O_i)$ is computed by letting:

$$\mathbf{t}(O_i) = \mathbf{v}'(O_i) + \mathbf{c}(O_i) + \mathbf{a}(O_i) + \mathbf{s}(O_i) + \mathbf{n}(O_i) + \mathbf{l}(O_i) + \mathbf{i}(O_i) + \mathbf{f}(O_i) \ ,$$

and setting

$$\mathbf{v}(O_i) = \begin{cases} \mathbf{t}(O_i) & \text{if } |\mathbf{t}(O_i)| < m \\ m \frac{\mathbf{t}(O_i)}{|\mathbf{t}(O_i)|} & \text{if } |\mathbf{t}(O_i)| \geq m \end{cases} .$$

4.3.3 Visual Representation of Swarms

Swarms are visualized by drawing the current position and heading of each of the swarm organisms onscreen. The heading (or direction) of an organism is computed as pitch, $\rho(O_i)$, and yaw, $\gamma(O_i)$, angles for each organism O_i. These angles are extracted from the velocity vector of O_i using the following

formulas,

$$\rho(O_i) = -\tan^{-1}\left(\frac{\mathbf{v}_y(O_i)}{\sqrt{\mathbf{v}_z(O_i)^2 + \mathbf{v}_x(O_i)^2}}\right) ,$$

$$\gamma(O_i) = \tan_2^{-1}(\mathbf{v}_x(O_i), \mathbf{v}_z(O_i))$$

where $\mathbf{v}_x(O_i)$, $\mathbf{v}_y(O_i)$, and $\mathbf{v}_z(O_i)$ are the x, y, and z components of the velocity vector of O_i, respectively.

When the `draw()` method of the `organism` class is invoked, the organism is first translated to its correct position within the three-dimensional space using `glTranslatef()`. It is then rotated by its yaw and pitch angles using `glRotatef()`. Lastly, it is scaled by its `size` attribute using `glScalef()` and drawn onscreen.

4.4 Reproduction and Evolution

4.4.1 Calculating Fitness

As previously mentioned, prey organisms primarily gain "fitness" by drawing energy from the spotlights. An organism's fitness is calculated by adding this energy component to two reward components: a reward based on proximity to the intersection and a reward based on the size of an organism and its proximity to the native light source. An organism O_i may gain fitness only when it resides within the cone of its native spotlight. Energy gained from the "native" spotlight is based on the organisms size, its distance from the light, and the intensity of the light. The intensity of light passing through a particular point is attenuated by the distance of the point from the light

source. This energy factor, $e_n(O_i)$, is calculated as

$$e_n(O_i) = s(O_i)\frac{I_n}{d(O_i)} ,$$

where $s(O_i)$ is the size of organism O_i, I_n is the intensity of its native spotlight, and $d(O_i)$ is the distance of organism O_i from its native spotlight.

An organism gains an additional reward based on its proximity to the intersection of the spotlights. This reward is calculated by first finding the distance of the organism from the edge of a sphere of radius R about the center C of the intersection volume. The radius of the sphere is calculated by testing sample points within the intersection volume to determine if they lie within both the red cone and the green cone. These points are passed to the `in_cone()` function, and the point P_m of greatest distance from C that lies within both cones is used to determine the value of R. The radius R is computed as

$$R = k|C - P_m| ,$$

where k is a constant such that $0 < k \leq 1$.

In order to support a gradual fall-off of reward values as an organism moves farther away from C, the reward factor $r_c(O_i)$ is calculated as a decaying exponential expression, factoring in the distance of an organism O_i from C. The reward factor $r_c(O_i)$ is calculated as follows: let M be the maximum reward value, $\epsilon(O_i)$ the distance of organism O_i from C, and k_1 and k_2 constants. Then

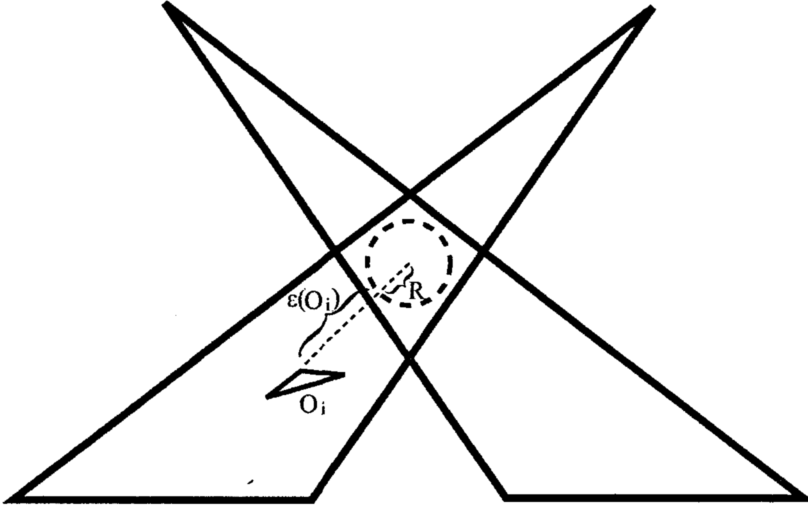


Figure 5: Fitness calculation

$$r_c(O_i) = \frac{M}{[1 + k_1(R + \epsilon(O_i))]^{1+k_2R}} \ .$$

An organism is also rewarded based on its size and proximity to the native light source. This factor $r_n(O_i)$ is calculated as follows: let $s(O_i)$ denote the size of organism O_i, and $d(O_i)$ the distance of organism O_i from the native light source:

$$r_n(O_i) = \frac{d(O_i)}{s(O_i)} \ .$$

Once all of the energy factors have been calculated, they are added together and the sum is added to the old fitness value $f'(O_i)$ of O_i, generating the new fitness value $f(O_i)$.

$$f(O_i) = f'(O_i) + e_n(O_i) + r_c(O_i) + r_n(O_i)$$

4.4.2 Reproduction

Once fitness values have been calculated for all of the organisms in the prey population, the population may be culled and repopulated. The is done by the `cull_and_repopulate()` method of the `prey_flock` class. When this method is invoked, a mating pool is generated that includes all of the prey organisms in the current population except the two least fit. This effectively excludes the two least fit organisms from the reproduction process. Each organism in the mating pool is then given an opportunity to mate with another nearby organism by searching for viable mates within a predetermined radius. If the organism is unable to locate a viable mate, its search radius is increased, and the organism is allowed to search again. This continues until the organism locates at least one viable mate. Once the organism has located viable mates, it selects one at random and produces a single offspring.

The offspring houses the genome of its randomly selected dominant parent, and the size gene may be mutated randomly so that the offspring is slightly larger or slightly smaller than its dominant parent. The offspring organism is given an initial position that is slightly displaced from the position of its mother.

Once all of the organisms in the mating pool have been given an opportunity to mate, two additional organisms from the pool are selected at random,

and both are allowed to select a mate and to produce an additional offspring. This is done to offset the discarding of the two least fit organisms. Every round of mating completely replaces the existing prey population.

4.5 System Parameters

The majority of system parameters in our simulation are stored in the file `global.h`. This provides an accessible interface for important parameters that control the behavior of the various system components, like flocking rules and spotlight behavior. This consolidation of system parameters facilitates easy system manipulation for the purposes of experimentation and helps to achieve the design goal of flexibility.

4.6 Interface Controls

As mentioned in section 4.1, the GLUT library provides a very limited selection of user interface components. In order to accomplish our goals of flexibility and transparency, it is necessary to provide interface controls that are easy to manipulate and correspond in some intuitive way to the simulation components that they modify. To this end, we chose to incorporate the GLUI library of user interface components into our simulation. The GLUI library provides several useful interface controls, including buttons, check boxes, radio buttons, spinners, and arcballs.

The spotlight intensity and spotlight cutoff angles can be adjusted in real time by clicking on the appropriate spinner component. The values for each

```
#define FLOCK_ALIGNMENT_WEIGHT    0.03f
#define FLOCK_COHESION_WEIGHT     0.03f
#define FLOCK_SEPARATION_WEIGHT   -0.08f
#define FLOCK_INERTIA_WEIGHT      0.4f

#define FLOCK_LIGHT_BOUNCE        0.2f

#define FLOCK_MIN_DISTRANGE       0.0f
#define FLOCK_MAX_DISTRANGE       3.0f

#define FLOCK_MIN_VELRANGE        1.0f
#define FLOCK_MAX_VELRANGE        2.0f

#define FLOCK_MIN_ORGSIZE         0.5f
#define FLOCK_MAX_ORGSIZE         1.0f

#define FLOCK_MIN_FLOCKSIZE       1
#define FLOCK_MAX_FLOCKSIZE       200

#define PREY_ALIGNMENT_WEIGHT     0.03f
#define PREY_COHESION_WEIGHT      0.03f
#define PREY_SEPARATION_WEIGHT    -0.08f
#define PREY_INERTIA_WEIGHT       0.45f
#define PREY_INTERSECTION_WEIGHT 0.05f
#define PREY_TOLIGHT_WEIGHT       0.05f
#define PREY_TOCONE_WEIGHT        0.35f
#define PREY_PREDATOR_AVOID       -0.25f
#define PREY_MAX_MUTATION         0.1f
#define PREY_MAX_DISPLACEMENT     0.1f
#define PREY_MATE_RANGE_INC       0.1f

#define PRED_ALIGNMENT_WEIGHT     0.00f
#define PRED_COHESION_WEIGHT      0.00f
#define PRED_SEPARATION_WEIGHT    -0.08f
#define PRED_INERTIA_WEIGHT       0.4f
#define PRED_INTERSECTION_WEIGHT 0.07f
#define PRED_TOLIGHT_WEIGHT       0.05f
#define PRED_TOCONE_WEIGHT        0.35f
```

Figure 6: A sampling of system parameters related to flocking rules

are bounded, and these bounds may be altered by modifying the appropriate system parameters in `global.h`. Each iteration of the main simulation loop checks the floating point values associated with each spinner and updates the spotlight objects accordingly.

The orientation of spotlights may be altered by adjusting the three-dimensional arcball associated with each. It was necessary to reverse engineer the arcball component somewhat for use in our simulation, as the output generated by an arcball is a three-dimensional rotation matrix, but the orientation of our spotlights is based on Euler angles. The desired behavior can be achieved by using the following formulas

$$\rho = \tan_2^{-1}(-R_{31}, R_{23}) \quad \text{and}$$

$$\nu = -\tan_2^{-1}(R_{33}, R_{11}) \ ,$$

where ρ is the pitch angle and ν is the yaw angle. These angles are then passed to the appropriate spotlight as orientation angles, and the spotlight cone is generated when `setup()` is called.

5 Development and Testing

The first stage of simulation development consisted of the creation of a visualization framework, as the simulation itself is inextricably tied to visualization. The initial visualization framework consisted of a single window with an orthographic viewing volume, a single wash light, and a "stage" area composed of two triangles, which were stored in an early version of the `mesh`

class. An orthographic viewing volume was selected in order to allow each visual component to have equal "weight", as a perspective-projected volume would tend to obscure size differentiation among organism. The early `mesh` class consisted of no more than a vertex, normal, and triangle list and a single method for drawing the mesh onscreen.

The next step of development was the addition of OpenGL calls to initialize the lighting system. It was at this point, however, that we noticed a problem with the previous "stage" configuration: an OpenGL spotlight does not project well onto a rectangle composed of two triangles as OpenGL lighting calculations are performed per vertex. The realization of this slight oversight led to a complete revamping of the stage component. Instead of representing the stage as a rectangle composed of two triangles, we chose to represent it as a mesh of hundreds of smaller quadrangles. This led to a substantial increase in the number of rendered vertices, but increased the definition of the spotlight projection. The resolution of the stage had to be adjusted later in development to compensate for slower performance on lower-end machines than the development machine.

The next simulation component to be added was an initial attempt at a swarming species. An initial "draft" of the `organism` class was developed, which consisted of member variables to store the organism's position, and heading, as well as a pointer to an instance of the `mesh` class storing the physical structure description. A `draw()` method was included to translate the geometric representation of an organism to its current position and then

to draw it on screen. The swarm rules were based on a somewhat naive implementation of the basic Reynolds' rules. While cohesion and alignment vectors were calculated similarly to current simulation, the separation rule proved to be problematic. In the initial implementation, organisms exhibited a "springing" behavior when attempting to avoid neighboring organisms. Essentially, an organism would backtrack along its current heading vector until well out of the way of any other organisms. Since this behavior is not evident in any natural swarms, we had to reconsider the swarming implementation. This reconsideration was left on the cutting room floor, however, for some number of weeks. An additional problem with the initial swarm implementation was the physical structure of the organisms themselves. Organisms resembled small, flying, oblong structures, which seemed somewhat inappropriate for our purposes. The problem of appropriate physical representation of organisms recurred throughout many phases of simulation development.

The initial attempt at creating a swarming species brought to our attention the necessity of creating a bounding volume for our physical space, as some organisms within the swarm tended to stray outside of the viewing volume. This problem was quickly remedied by adding bounds checking to the swarm routine.

The next phase of simulation development focused on creating a more robust lighting component. The previous attempt at lighting relied exclusively on calls to the OpenGL lighting system, which is somewhat limiting in regard to the visualization of light sources themselves. Though light it-

self is invisible, it was necessary, for our purposes, to visualize the cone of a spotlight. Furthermore, it was desired, in the interests of flexibility, to encapsulate the abstract concept of a "light" within a C++ class, of which many instances may be created. It was at this stage of development that the first version of the `light` class was written, adding support for spotlight cone visualization and the easy instantiation and initialization of light sources. The implementation of this early version of the `light` class progressed with very few problems, and the current version resembles the early version in most respects.

Once an early version of the `light` class was completed, we resumed the development of a more robust swarming algorithm. In a lesson learned from the development of the light class, we decided to encapsulate the abstract idea of a "swarm" within its own C++ class. This contributed to the overall flexibility of our platform by allowing swarms to be instantiated and initialized easily, while hiding the complexity of the internal implementation. A new and somewhat different approach to the swarming algorithm was implemented and is presented in section 4.3.2. This algorithm produces the impromptu flocking behavior necessary to qualify a population of organisms as a swarm. Some adjustment of scaling constants was required to insure that organisms flock in tight formations. This implementation of the standard swarm rules remains relatively unchanged in our current simulation.

The next phase of the implementation was to satisfy the requirement that organisms remain within their native spotlight. The initial attempt to

implement this rule resulted in a clustering of organisms along the center line of the native spotlight. This was the result of a constant force applied to an organism in the direction of this center line. As this is in no way representative of "flocking" behavior, a new approach was needed. The next approach that was attempted proved to be mostly successful, but with one exception: organisms tended to fly out of the "top" of the cone. That is, upon reaching the apex of the spotlight cone, organisms did not turn around in order to stay inside the cone but instead flew through the apex. This was remedied by applying a positive force in the direction of the spotlight orientation vector once an organism reaches the cone apex. The resulting implementation proved to be successful and is documented in section 4.3.2. Initially, however, the `in_cone()` function was included as a method of the `flock` class. As development progressed, we realized that the simulation would be better served by defining the `in_cone()` function at the global scope, as it is also used by the intersection test for spotlights.

Once a more robust flocking algorithm was developed, we set about adding interactive controls to the simulation. Initial investigations into the controls provided by GLUT found the library lacking, so a search for an alternative library was begun. We came across GLUI rather quickly, but no pre-compiled libraries existed. Fortunately, a Visual C++ workspace was included in the source code package for the library, which we managed to compile and add to our project after only a few unsuccessful attempts. One benefit of the GLUI package is that it is compiled as a static library, which

eliminates additional software-requirement overhead in the form of a Windows DLL, which proved to be most convenient for demonstration purposes. An unfortunate disadvantage of the GLUI system is that it does not integrate well within the existing OpenGL framework. GLUI is an object oriented GUI toolkit, whereas OpenGL is implemented as a procedural state machine. This posed no practical problem for implementation, but does make the code a little more difficult to read and less intuitively grasped. We decided to make use of the spinner components and arcballs as described in section 4.6. It proved to be quite simple to integrate these interface components into the current GLUT-based framework, after only a bit of initial confusion about how to register the GLUI components with the GLUT windowing callbacks.

The problem of determining the intersection of spotlights was considered next and proved to be substantially more difficult. We researched various methods, but our primary goal was to find a closed-form mathematical solution for calculating the center of the intersection, as opposed to relying on computer approximation. It might have been very simple to use the existing `in_cone()` function to test sample points throughout the three-dimensional space and to take an average of the sample points that lie within both cones. However, the solution we chose provides an exact intersection point with relatively little computational overhead; in fact, the complexity of the algorithm itself is $O(1)$.

Once the solution was chosen, implementation was a relatively straightforward process. The only degenerate cases we had to consider were those

instances when the intersection of the spotlights is below the "stage" area and when the shortest distance line between the two spotlight center line lies above the apex of the two cones. After testing for these cases and excluding them, implementation proceeded with little difficulty. The problem of finding the intersection of two cones is twofold, however: finding the center of the intersection and determining the bounds of the intersection volume. Due to time constraints, we chose to simply approximate this volume with a sphere using the method described in section 4.4.1.

The next step of simulation development was the implementation of mechanisms for reproduction and simulated evolution. This included both the calculation of fitness for individual organisms and the reproduction process. We worked with a few different fitness functions, one based only on the energy component contributed by the native light, one based on the energy component and the reward term for proximity to the intersection, and one based on all three of the terms described in section 4.4.1. In conjunction with a simple, though naive, reproduction mechanism, each function gave us varying degrees of success. The most successful of the three, however, was the calculation based on three terms.

The first attempt at an implementation of the reproduction process resulted in permitting organisms in the mating pool to reproduce with any other organism in the mating pool, regardless of proximity. This effectively allowed organisms to mate with other organisms on the opposite side of the spotlight intersection. While logically this is inappropriate, it nonetheless

resulted in successful niche creation and speciation. It was preferable, however, to only allow organisms to mate with other organisms in close proximity. This proved to be an error-prone process, however, as "close proximity" in our simulation is a relative term. Organisms may be separated by a spotlight cone intersection or not, or the cone may disappear completely, causing the organisms to disperse throughout the three-dimensional space. Choosing a proximity value too small could result in an infinite loop, as an organism is repeatedly allowed to search for mates in a range where there are none. Choosing a proximity value too large may result in mating across the intersection of spotlights. Thus, a more flexible mating routine was developed, allowing the organism to search for a mate within a variable range.

6 Results

Though the lengthy development of the simulation left us with little time to develop a full suite of experiments with which to test our platform, early results show that the simulation is performing as expected. When the spotlights are crossed and an intersection is created, the prey swarm consistently splits into two tightly-clustered sub-swarms on either side of the intersection. After many repopulation steps, the two sub-swarms begin to diverge in terms of organism sizes. After many generations, the two populations exhibit markedly different organism sizes. Organisms above the intersection tend to be very small in relation to organisms below the intersection. Organisms below the intersection tended to grow larger by comparison as well.

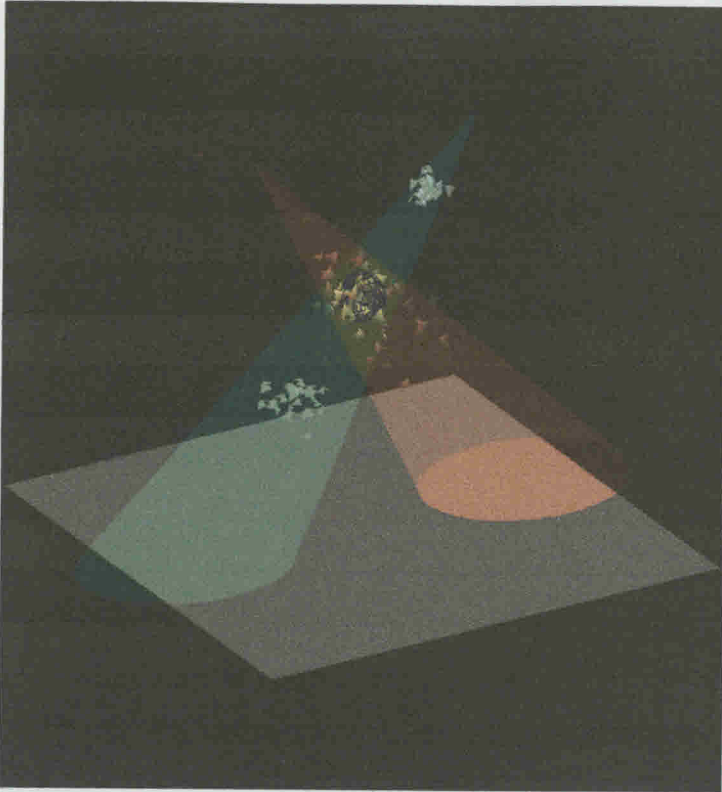


Figure 7: A screenshot of the simulation when spotlights intersect

It is worth noting that the simulation produces the most convincing results when the population above the intersection is initially much larger than the population below. This is due to the fact that organisms above the intersection tend to die off more quickly than organisms below. Organisms above the intersection have less room to move about and tend to stray outside of the spotlight cone more frequently. An organism accumulates no additional fitness when it strays outside of the spotlight cone and is thus more likely

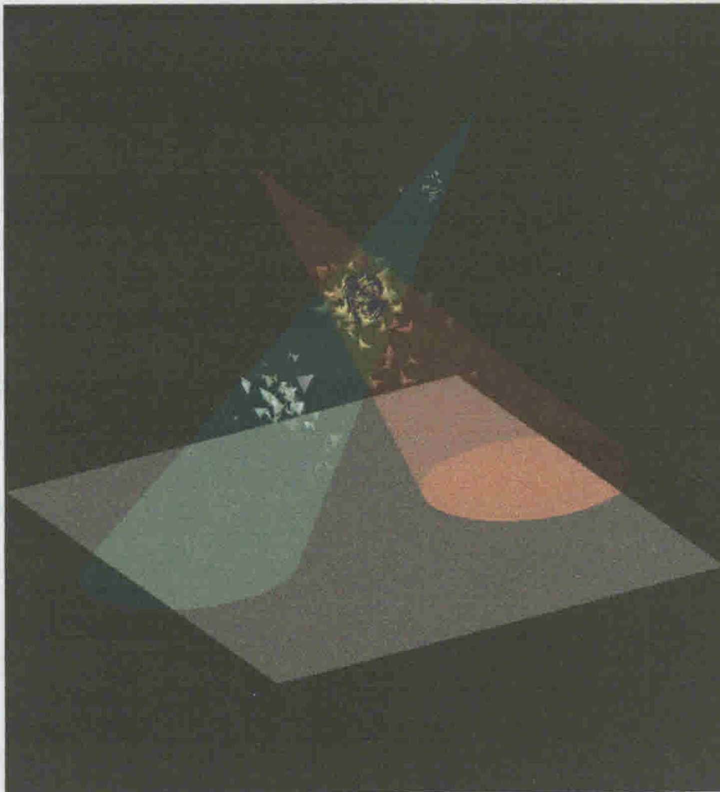


Figure 8: A screenshot of the intersection with divergent organism sizes

to be excluded from the reproduction process. After many generations, this results in a greater distribution of prey organisms in the lower half of the spotlight cone, as more organisms from the lower half are selected at random during the reproduction process to compensate for the loss of the two least fit.

Occasionally, an organism from the subswarm below the intersection will migrate across the intersection, despite the abundance of predators, to the top of the cone. This is largely due to the tendency of organisms to move toward the source of their native spotlight. While this migration poses little problem for the speciation process (larger organisms tend to die off quickly above the intersection), one can easily slow the flow of migration down by enlarging the spotlight cutoff angle of the red spotlight, thus allowing the predators to cover a larger volume. This, in turn, gives a prey organism less freedom to move around the prey swarm.

7 Conclusions and Future Work

There are many opportunities for expansion within our simulation framework, some small, others quite substantial. First on this list of expansions, because we were unable to implement due to time constraints, is an extension of an organism's genome to include a gene that influences its speed. That is, as the organism grows smaller, it is enabled to move about more quickly, and as it grows larger, it is forced to move more slowly.

Another potential modification would be to allow the user to specify a

distribution curve of organism sizes. Presently organism sizes are determined based on a uniform distribution. However, interactive controls could be added to allow the user to adjust the distribution curve in real-time and adjust organism sizes accordingly. For practical purposes it would only be useful to adjust the distribution curve at the beginning of the simulation, but adding a control of this sort would vastly increase user control and yield a wider variety of results.

It may be useful in certain experimentation scenarios to introduce multiple prey populations. In such a scenario, allowing the populations to interbreed within a shared niche, adjusting the visual representation of offspring to demonstrate this interbreeding, could be indicative of a sort of sympatric speciation. However, the interbreeding mechanism that this requires does not presently exist. Currently, our model supports only the allopatric model of speciation, and breeding within a single population. It would be useful to expand the platform to incorporate multiple speciation models.

We conclude, based on early indications, that our platform provides the basic rule set and components necessary for the development of new niches and, by extension, new niche species. As a platform for experimentation as extension, we believe it shows great promise. We believe we have achieved our goals of simplicity, flexibility, and transparency, and hope that future work will find the platform robust and highly extensible.

References

[1] G. Booth, Gecko: a continuous 2D world for ecological modeling, *Artificial Life 3*, 1997, 147–163.

[2] N. Gracias, H. Pereira, J. Lima, and A. Rosa, An artificial life environment for ecological systems simulation, *Artificial Life V*, 1996, 124–133.

[3] P. T. Hraber, T. Jones, and S. Forrest, The ecology of Echo, *Artificial Life 3*, 1997, 165–190

[4] T. S. Ray, An approach to the synthesis of life, *Alife II Conference Proceedings*, 1990, 371–408.

[5] C. W. Reynolds, Flocks, herds, and schools: a distributed behavioral model, *Computer Graphics, Annual Conference Proceedings, 1987*, ACM SIGGRAPH, 1987, 25–34.

[6] K. Sims, Virtual creatures, *Computer Graphics, Annual Conference Proceedings, 1994*, ACM SIGGRAPH, 1994, 15–23.

[7] C. Sommerer and L. Mignonneau (eds.), *Art @ Science*, Springer Verlag, 1998.

[8] C. Sommerer and L. Mignonneau, A-Volve — an evolutionary artificial life environment, *Artificial Life V Conference Proceedings*, 1998, 167–175.

[9] C. Sommerer and L. Mignonneau, Life spacies, *Siggraph '99 Conference Abstracts and Applications*, 1999, 170.

[10] J. Ventrella, Attractiveness vs. efficiency - how mate preference affects locomotion in the evolution of artificial swimming organisms, *Artificial Life VI*, C. Adami et al (ed.), MIT Press, Cambridge, MA, 1998, 178–186.

[11] L. Yeager, Computational genetics, physiology, metabolism, neural systems, learning, vision, and behavior or PolyWorld: life in a new context, *Alife III Conference Proceedings*, 1992, 263–298.