Evolving Virtual Creatures

Kenrick E. Drew
Andrew S. Forsberg
Paton J. Lewis

cs224 Final Project Proposal
Brown University
April 13, 1995

Abstract

We propose to implement a two-dimensional version of the work described in Karl Sims's 1994 SIGGRAPH Proceedings paper "Evolving Virtual Creatures." [SIMSa] We will investigate goal-directed genetic algorithms and the dynamics of complex rigid bodies. Our application will generate animations of virtual creatures exhibiting evolved behaviors. The user will specify the characteristics of the environment in which two-dimensional creatures will evolve. An initial random set of creatures will be created and simulated, and those that are successful at specified behaviors will be selected for breeding. Goal behaviors will include jumping, walking, and swimming. Once the program has bred several generations of creatures, survivors may be saved for future examination.

Overview

Motivation. One of the great attractions of software development is the outlet for creativity. Genetic algorithms provide an especially effective outlet by adding unpredictable elements that make the creative process interactive and the results surprising. In addition, articulated rigid body dynamics are used throughout computer animation, making it a useful topic of study for students interested in interactive graphics and animation. Finally, by combining genetic algorithms, articulated rigid body dynamics, and computer graphics it is possible to model and simulate creations that would be difficult to invent and impossible to encounter in everyday life.

Functional Description. The user will begin by specifying system parameters such as initial population size, number of generations to evolve, mutation rates, and breeding methods. The application will generate an initial random set of creatures and then simulate each creature in the specified environment. After simulation, creatures will be selected for breeding based on performance. The parent and offspring populations are then mixed, and the cycle repeats, starting with the simulation stage.

We plan to have several hundred creatures in each simulation cycle. To handle computational requirements, the ability to perform creature simulations in parallel will be built in at the start. Initially these parallel simulations will be executed by multiple child processes running on a single multi-processor machine. One possible extension of this project (see below) is to extend this concept further to allow the simulation to be distributed across multiple machines.

Extended Functional Description

Creature Morphology. Creature genotypes will be represented by directed graphs, where nodes represent body parts or neurons, and edges represent connections. A node may be connected to itself either directly or through a cycle. Each node will contain a sensor, effector, and the type of joint connecting it to its parent.

When a genotype is expressed to form a phenotype, the directed graph will be traversed and neurons and body parts will be created accordingly. Figure one illustrates this relationship. To prevent infinite cycles, a node will contain a limit for the number of times that it can be expressed. A Boolean value will indicate whether multiple connections to that node should be expressed in a symmetric fashion.

A creature phenotype will consist of one or more connected body parts represented by 2D rectangles, and a brain modeled by a collection of interconnected function nodes.

Body Parts. Body parts will be connected by joints, which may be of type *rigid* or *rotary.* A rigid joint connects two body parts but does not allow relative movement. A rotary joint can rotate around the axis perpendicular to the plane in which the body parts exist. A rotary joint includes motion constraints as indicated by its associated gene. When an effector attempts to move a joint, the joint constrains the motion before it applies forces to its associated body parts. Note that joints do not have a physical representation—they merely describe the point of contact and motion constraints between two rigid bodies.

Sensors. Creature body parts may have sensors associated with them. We will implement *joint* sensors and *contact* sensors. Joint sensors will indicate the current scalar value (whether radians or length) associated with a joint. A contact sensor is associated with any one face of a body part. A contact sensor signals any form of contact, including creature self-contact. The brain will query sensors for information about the world.

Neurons. Neurons are functional units with one or more scalar inputs that produce one scalar output. Inputs can come from sensors or neurons. Outputs may be connected to neurons or effectors. A single output may be connected to multiple inputs, and feedback is allowed. Inputs are weighted by values derived from asso-

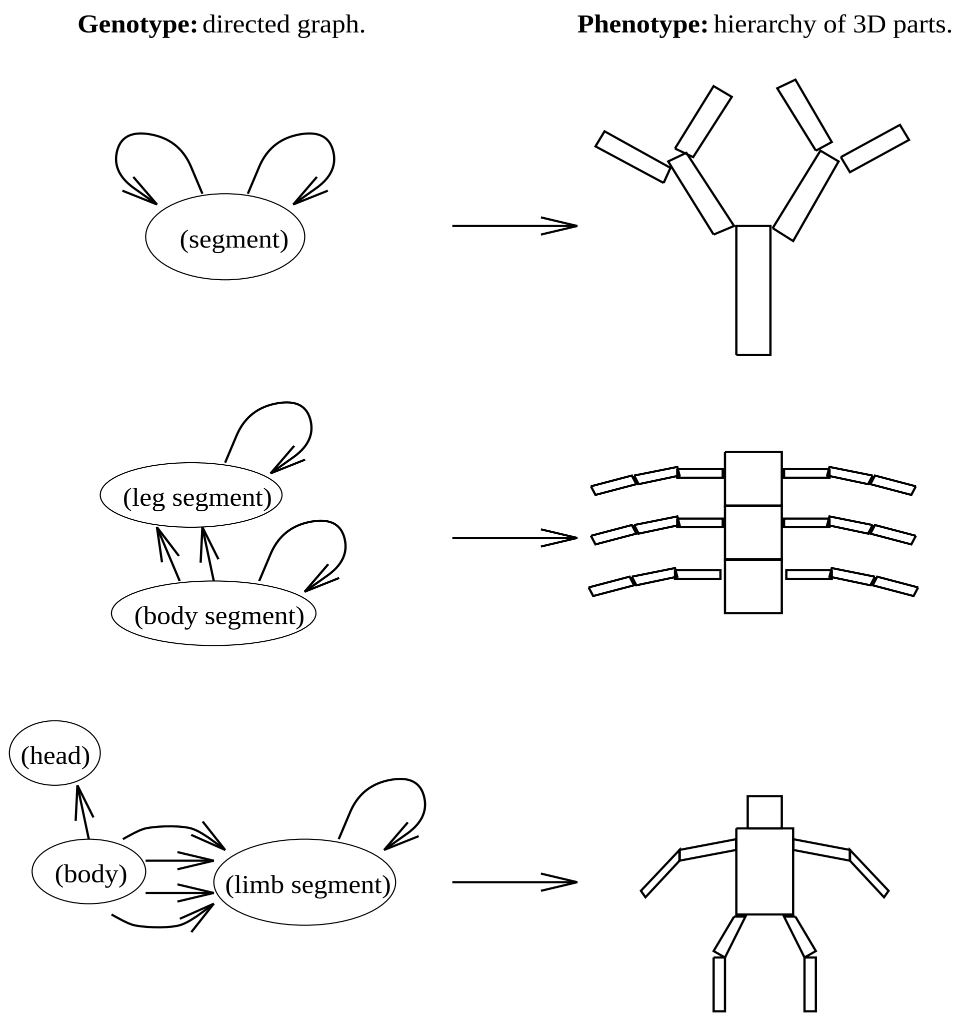

Genotype: directed graph.
Phenotype: hierarchy of 3D parts.
(segment)
(leg segment)
(body segment)
(head)
(body)
(limb segment)

Figure 1: Designed examples of genotype graphs and corresponding creature morphologies.

ciated genes. Neurons may represent any one of the following functions: *sum, product, divide, sum-threshold, greater-than, sign-of, min, max, abs, if, interpolate, sin, cos, atan, log, expt, sigmoid, integrate, differentiate, smooth, memory, oscillate-wave,* and *oscillate-saw.* Some of these functions act on values collected over time, and in those cases the neuron will retain a short history of inputs.

Effectors. An effector receives a scalar value from the brain and multiplies that value by the weight specified by an associated gene. We will initially implement only one effector type, the *joint effector.* A joint effector will drive an associated joint by passing a scalar value on to that joint. The joint will interpret that value in units appropriate for the joint—for example, radians or length.

Simulation. Creature phenotypes will be simulated in a 2D environment with user-specified characteristics. These parameters include *gravity, viscosity,* whether *ground* exists, and how much *friction* and *elasticity* the ground has. If present, the ground will be a flat plane of infinite dimensions.

Creatures will be simulated in batch to determine which ones meet the selection criteria. The simulator will process a creature phenotype and return information describing the creature's performance. For the behaviors jumping, walking and swimming, this will just be the final and maximum distances that the creature moved from the origin. Creatures may be simulated in an associated viewport so that the user may examine the creature in its environment.

Physical elements simulated will include *articulated body dynamics*, *collision detection*, *collision response*, *friction*, and *viscosity.* Accelerations will be integrated with the Adaptive Runge-Kutta technique to determine velocities and positions.

Selection. After simulation, creatures will be selected for breeding based on how well they perform specified tasks. The user may choose to breed creatures so that they are selected for success at performing one of the following behaviors: *jumping*, *walking*, or *swimming.* Jumping creatures will be judged by how high they jump. Walking creatures will be judged by how far they move from the start position in the allotted time. This distance will be scaled by creature size so that large creatures that fall over are not selected for that ability. Swimming creatures will be judged in a manner similar to walking creatures, except that consistent forward motion will be rewarded over drifting motion resulting from a single initial quick movement.

Breeding. Once creatures have been selected for breeding they produce a number of offspring proportional to their degree of success at the desired behavior. Creatures are bred using one of three methods: *asexual*, sexual by *crossover*, and sexual by *grafting*. The breeding method will be specified by the user as an attribute of the creature, or as a percentage of each generation to be bred with each method.

Crossover breeding is performed by aligning the gene graphs for the parents, traversing one parent's gene graph, and at some randomly selected point crossing over to the other parent's gene graph. Crossovers can occur more than once.

Grafting breeding is performed by randomly pruning one section of one parent's gene graph and grafting on a random portion of the other parent's graph at that point. These methods are illustrated in figure two.

Internal node parameters can be mutated, after which a new random node may be added, followed by possible connection mutations. Each type of mutation has a probability associated with it. These probabilities are scaled so that each creature experiences exactly one mutation 50% of the time, zero mutations 25% of the time, and more than one mutation 25% of the time.

Parallel Implementation. By far the most compute-intensive portion of this project is the creature simulation. Because of this, we intend to exploit multi-processor workstations by spawning multiple processes to perform simulations. The architecture that supports this will be designed so that it may be easily extended to use a distributed, multi-workstation package (such as quahog) if there is sufficient time remaining at the end of the project.

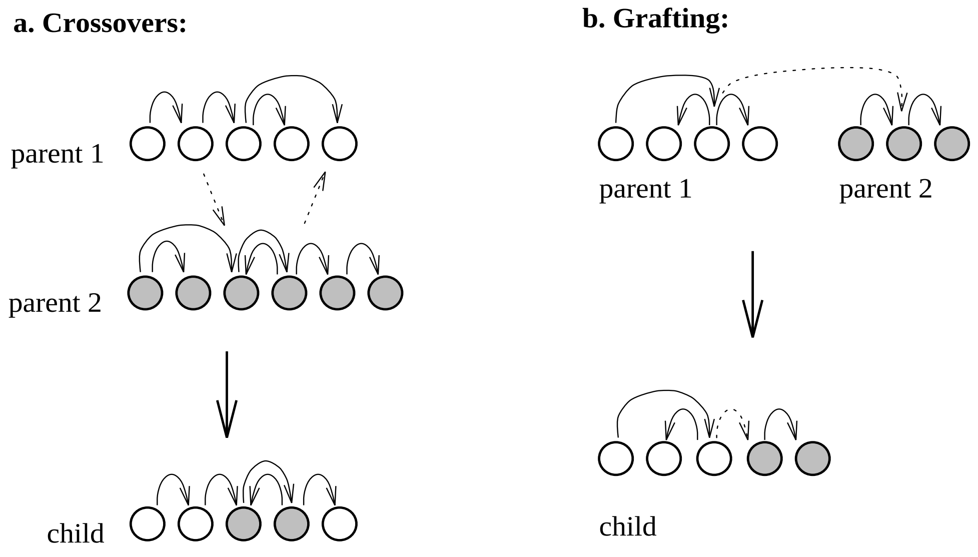


Figure 2: Two methods for mating directed graphs.

User Interface

The project will in fact include three applications—a master control application, an application that performs simulations, and an application that displays creature simulations in real time. The control application will collect initial conditions from the user, generate the initial creature population, spawn instances of the simulator, evaluate simulator results to determine creature fitness, and perform the mutations. Once the final generation of creatures has been created, the control application can spawn instances of the viewer so that the user can view creature simulations in real time.

Before an initial random population is created, the user will outline constraints on the domain of the structure of the genomes in the initial population. He will also specify the selection criteria, as well as other gross parameters such as environment characteristics, initial population size, and the number of generations desired. After this, the controller program will run without intervention until completion.

Because GP consumes a great deal of processing time when handling X events, we may decide to implement our controlling application without it so that all parameters are specified on the command line or in a parameter file.

Key Algorithms

Main Loop.

I. *Collect system parameters from user*

II. *Generate initial random genome population*

III. *Express genotypes*

IV. *Simulate phenotypes in environment*

V. *Evaluate phenotypes for fitness*

VI. *Breed*

VII. *Allow user to review and save creatures*

Articulated Body Dynamics.

I. *Collect explicit forces and torques*
These external forces include gravity, impulse, contact and effector forces.

II. *Construct constraint-force equation*
Construct the constraint equation for the current time step. The constraint equation is a set of linear constraint equations expressed in matrix form. When solved, this equation will determine the set of internal forces required to satisfy the constraints of the system once those internal forces are combined with the external forces. The constraint-force equation for a single constraint is:

$$\sum_j \left(\sum_i \left(\Gamma^i G_j{}^i + \Lambda^i H_j{}^i \right) \vec{F}_{c_j} \right) + \sum_i \left(\Gamma^i \overrightarrow{F_E}^i + \Lambda^i \overrightarrow{T_E}^i \right) + \vec{\beta} + \frac{2}{\tau} \vec{D}^{(1)} + \frac{1}{\tau^2} \vec{D} = 0$$

where the *i* represent the bodies, the *j* represent constraints, Γ^i is the net force on body *i*, G^i_j is a matrix which in conjunction with $\vec{F}_{c_j}$ describes the constraint forces on body *i*, Λ_i is the net torque on body *i*, H^i_j is a matrix which in conjunction with $\vec{F}_{c_j}$ describes the constraint torques on body *i*, $\overrightarrow{F_E}^i$ and $\overrightarrow{T_E}^i$ are the external forces and torques, τ is the timestep (Δt), $\vec{D}$ is measure of the deviation of the constraints, $\vec{D}^{(1)}$ is the rate of change of $\vec{D}$, and $\vec{\beta}$ is the part of the acceleration of $\vec{D}$ that is independent of net force and net torque.

III. *Solve constraint-force equation*
We will use a least-squares solution solver in the Matrix package to solve the system of constraint-force equations. This method is required because the system is over-determined. The system is

over-determined in part due to the fact that we are using dependent coordinates. We are using dependent coordinates because they simplify the task of determining absolute cartesian coordinates of each component of the system.

IV. *Compute net forces and torques*
Combine the forces collected in part I and computed in part III, compute the resulting accelerations, and pass those accelerations on to the Collision Detection algorithm.

Collision Detection. In the following, `I` is the set of pairs of interpenetrating objects. `C` is the set of pairs of contacting objects, where contact is indicated by low-velocity objects within some distance ε of each other. If a collision is detected, the algorithm performs a binary search to find the time step at which the interpenetrating bodies are within ε of each other. `world->evalDynamics(t)` integrates accelerations and velocities to determine velocities and positions at the specified time.

```
t ← t + $\Delta$t
world->evalDynamics( t )
computeCollisions( C, $I_0$ )
if ( $I_0 \neq \emptyset$ )
  I ← $I_0$
  s ← 2
  do
    if ( $I \neq \emptyset$ )
      t ← t - $\Delta$t/s
    else
      t ← t + $\Delta$t/s
    s ← s $\times$ 2
    world->evalDynamics( t )
    computeCollisions( C, I )
  while ( $( I \neq \emptyset ) \vee ( \neg\exists e.\ e \in I_0 \wedge e \in C )$ )

computeResponse( C )
```

Modules

See the class hierarchy graphs for a pictorial representation of class hierarchies and containment relationships. The modules (also known as “packages”) listed below are distinguished by a common functional nature, and will often contain more than one class. Modules have well-defined APIs which will not change much during application development.

Genome. The Genome module will contain the Genome class and the Gene class and subclasses. The Genome class will contain methods for expressing a genome as a phenotype, mutating, and reading and writing creatures to files. The Gene subclass constructors will create initially randomized genes.

BodyPart. The BodyPart module will contain the BodyPart class. A BodyPart object will contain a joint, a joint sensor, an effector, a list of additional sensors, and a integration object. Note that body part scale, rotation and translation information are encapsulated within the integration object.

NeuralLink. The NeuralLink module will contain the NeuralLink class. The NeuralLink class is a superclass common to effectors, neurons, and sensors. A NeuralLink subclass inherits the ability to have multiple inputs and one output. This allows us to exploit polymorphism—for example, a "connect-from" list of pointers to NerualLink objects could have both sensors and neurons in it, while a similar "connect-to" list could contain both neurons and effectors. This greatly simplifies the tasks of connecting components in the brain and propagating signals through it during simulations.

Sensor. The Sensor module will contain the Sensor class and subclasses. A contact sensor contains a pointer to the BodyPart that it is associated with, and a joint sensor contains a pointer to the joint with which it is associated. A sensor subclass may interact with the world class to detect other objects in the environment.

Joint. The Joint module will contain the Joint class and subclasses. A joint contains pointers to the two body parts that it connects. An associated joint effector will update the joint angle or position, and the joint in turn updates the forces on the body parts, after applying its motion constraints.

Effector. The Effector module will contain the effector class and subclasses. A joint effector contains a pointer to the joint with which it is associated.

Brain. The Brain module will contain the Brain class and the Neuron class and subclasses. The brain contains a list of neurons, sensors, and effectors associated with it, and the connections among them. It also contains a "think" method that iterates over effectors and determines inputs values to those effectors and their connected neurons by performing lazy evaluation.

A neuron can have one or more inputs as required by its function, and one output (which may be connected to multiple inputs). The inputs are weighted by gene-dictated values. When evaluated, a neuron will compare its localState data member against the static globalState data member in order to prevent multiple neuron evaluations during a single call to Brain::think(), which would otherwise occur in neural graphs that contain cycles.

Creature. The Creature module will contain the Creature class. A creature will contain a list of BodyPart objects and a Brain object.

World. The World module will contain the World class, which will contain a list of creatures and objects (such as light sources and the ground) in the environment being simulated. Sensor classes and the collision class will query the world object.

Simulator. The Simulator module will contain the code that performs creature simulations. It will provide facilities for both batch simulation and interactive simulation. The manner in which batch simulations are done in parallel (whether multi-process or distributed) will be encapsulated within this module.

Dynamics. The Dynamics module will contain the Dynamics class, the Integration class and subclasses, and the Collision class. The Dynamics class will implement the articulated rigid body dynamics. IntegrationARK (for Adaptive-Runge-Kutta integration) is the only initially planned Integration subclass, but this architecture allows us to easily try other integration techniques if we so desire. The Collision class will implement collision detection and collision responses.

Application. The Application module will contain the mainline code for the three applications, including the user interface components, the fitness evaluator, the breeder code, and the viewer.

Implementation Strategy

Responsibilities. We will all work together to design the critical algorithms, including articulated dynamics, collision detection and response, and mutation. Once these algorithms are specified to a level of detail that allows individuals to write code, we will each take responsibility for developing a particular subset of the project. Andy will develop the dynamics code that takes collision response forces and joint effector forces and solves the articulated rigid body system to determine the resulting accelerations of the components. Ken will develop the collision detection, collision response and dynamics code. Pat will implement everything else, including genotype and phenotype classes, the user interface code, the distributed processing code, and the fitness evaluation code.

Third-Party Software. We may use GP for our user interface when collecting the parameters for the initial condition, and we will use it for the viewport when viewing 2D simulations. However, we will structure the code so that we can use a separate Open/GL application as our 3D viewport should we find time to implement that extension.

We will use the Adaptive-Runge-Kutta code provided to us for the cs224 dynamics assignment. We will also use the Matrix and Vector packages provided then.

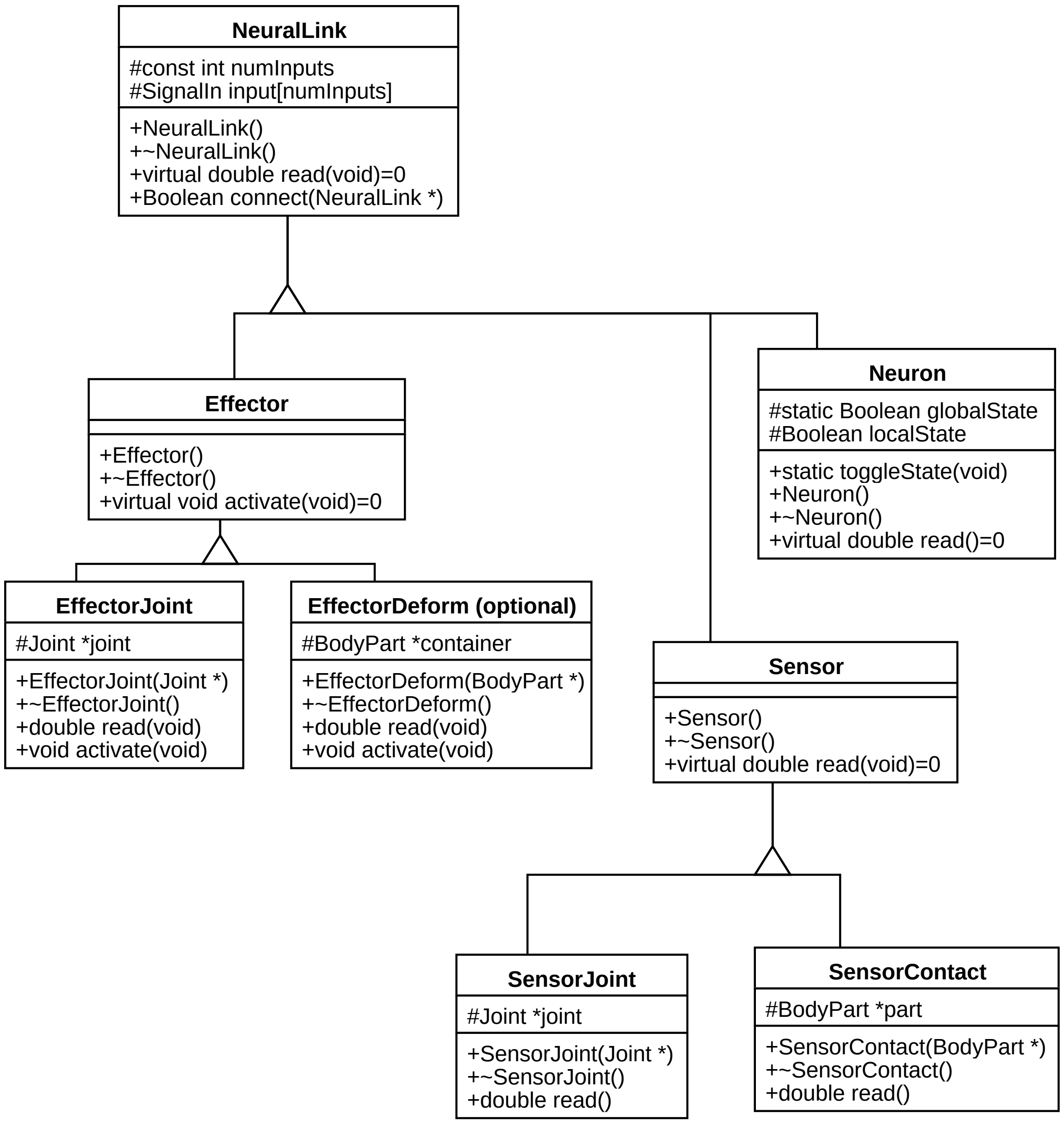
Development Policies. We will develop modules in such a fashion as to hide the implementation details from the client users of those packages. Modules will be developed and unit-tested individually. When a module passes its unit test suite it will be promoted for use in the main application. Once a module's API has settled, a simple, robust, and minimally functional version will be implemented and published to allow complete system testing in order to help verify that the module's interface design is sufficient.

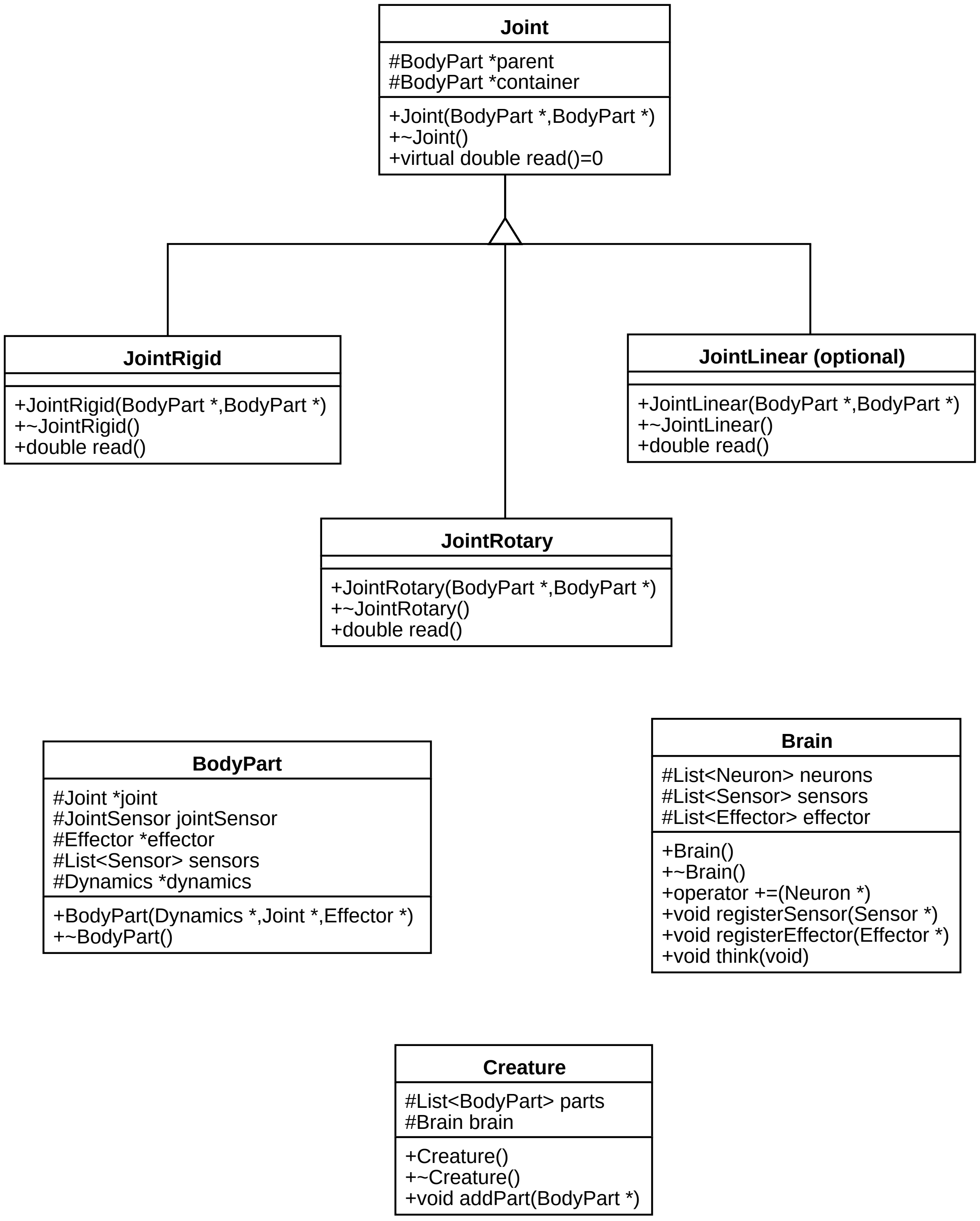
Order of Development. We will implement the dynamics module first, because we believe it is the hardest part of the application and is the most likely source of significant problems. By implementing it first, we will have as much time as possible available to deal with any problems associated with that package, should they arise. We will also develop the viewer and phenotype code in parallel with the dynamics module code so that the dynamics module has a test base and some hard-wired test case creatures. Next we will develop the genome code, the user interface code, and the rest of the controller application. Finally, if we have time, we will modify our code to support 3D creatures and simulations.

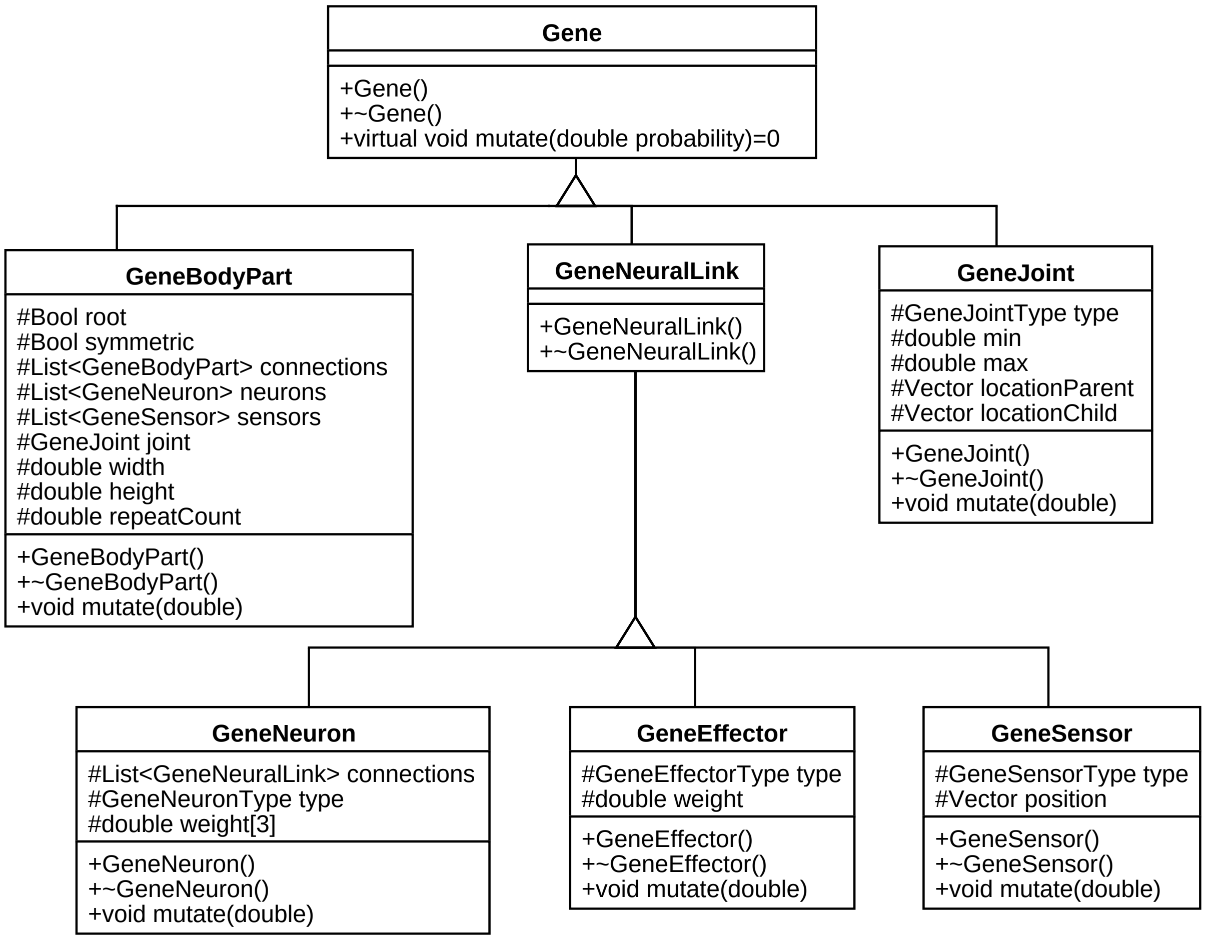
Milestones

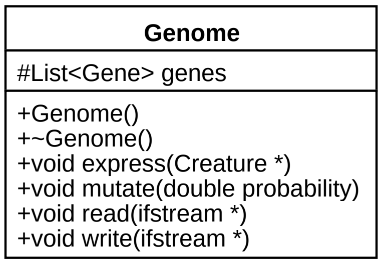
04/13/95	***Final Project Proposal due.***
04/21/95	***Articulated rigid body dynamics algorithms complete.*** Including integration, multi-body collision detection, force distribution, collision response, friction, and viscosity.
04/25/95	***Mid-project in-class demonstration.*** Everything complete except for creature file read/write operations.
05/01/95	***Optional 3D extensions completed***. If possible, the optional 3D extensions will be completed by this date.
05/05/95	***Final Project and User Guide due.***
05/07/95	***Demo materials completed.*** Video and materials completed in preparation for final project demo.
05/08/95	***Final Project demo.***

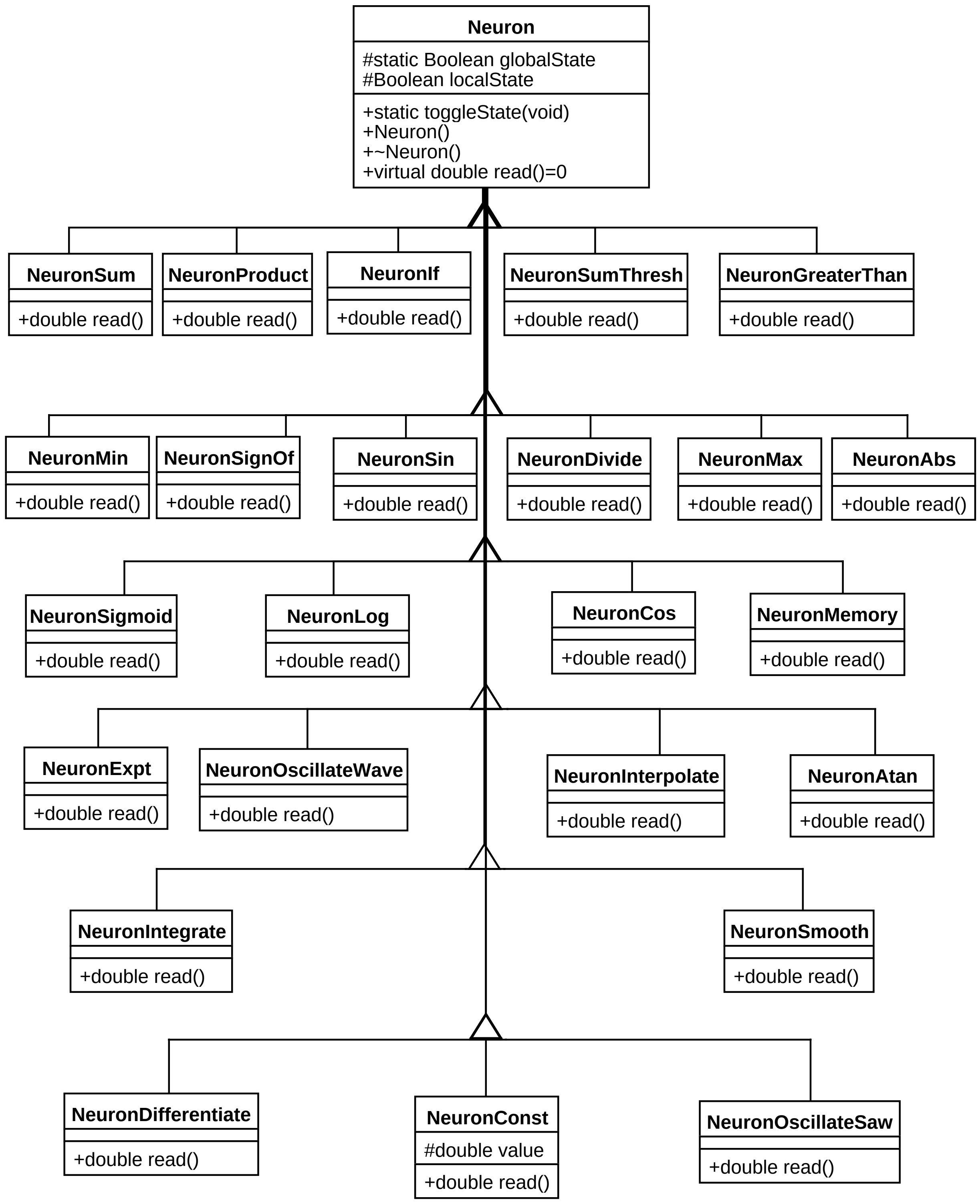
Class Hierarchy and Containment

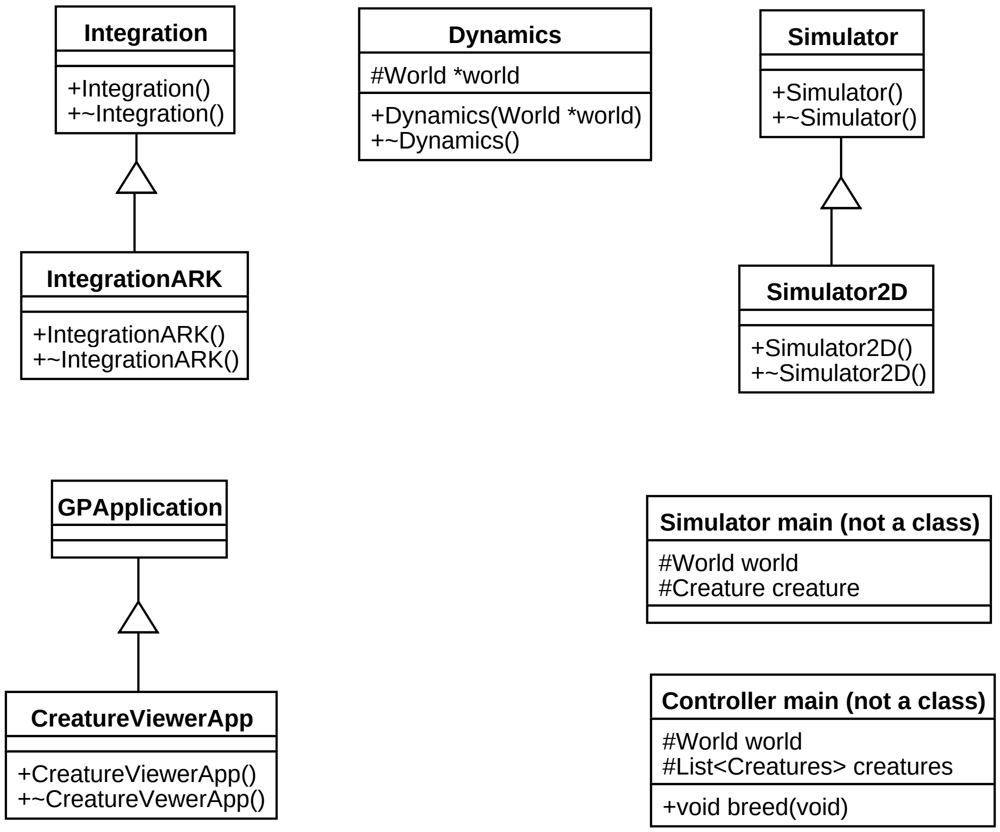



Joint
#BodyPart *parent
#BodyPart *container
+Joint(BodyPart *,BodyPart *)
+~Joint()
+virtual double read()=0
JointRigid
+JointRigid(BodyPart *,BodyPart *)
+~JointRigid()
+double read()
JointLinear (optional)
+JointLinear(BodyPart *,BodyPart *)
+~JointLinear()
+double read()
JointRotary
+JointRotary(BodyPart *,BodyPart *)
+~JointRotary()
+double read()
BodyPart
#Joint *joint
#JointSensor jointSensor
#Effector *effector
#List<Sensor> sensors
#Dynamics *dynamics
+BodyPart(Dynamics *,Joint *,Effector *)
+~BodyPart()
Brain
#List<Neuron> neurons
#List<Sensor> sensors
#List<Effector> effector
+Brain()
+~Brain()
+operator +=(Neuron *)
+void registerSensor(Sensor *)
+void registerEffector(Effector *)
+void think(void)
Creature
#List<BodyPart> parts
#Brain brain
+Creature()
+~Creature()
+void addPart(BodyPart *)


Gene
+Gene()
+~Gene()
+virtual void mutate(double probability)=0
GeneBodyPart
#Bool root
#Bool symmetric
#List<GeneBodyPart> connections
#List<GeneNeuron> neurons
#List<GeneSensor> sensors
#GeneJoint joint
#double width
#double height
#double repeatCount
+GeneBodyPart()
+~GeneBodyPart()
+void mutate(double)
GeneNeuralLink
+GeneNeuralLink()
+~GeneNeuralLink()
GeneJoint
#GeneJointType type
#double min
#double max
#Vector locationParent
#Vector locationChild
+GeneJoint()
+~GeneJoint()
+void mutate(double)
GeneNeuron
#List<GeneNeuralLink> connections
#GeneNeuronType type
#double weight[3]
+GeneNeuron()
+~GeneNeuron()
+void mutate(double)
GeneEffector
#GeneEffectorType type
#double weight
+GeneEffector()
+~GeneEffector()
+void mutate(double)
GeneSensor
#GeneSensorType type
#Vector position
+GeneSensor()
+~GeneSensor()
+void mutate(double)


Genome
#List<Gene> genes
+Genome()
+~Genome()
+void express(Creature *)
+void mutate(double probability)
+void read(ifstream *)
+void write(ifstream *)


Neuron
#static Boolean globalState
#Boolean localState
+static toggleState(void)
+Neuron()
+~Neuron()
+virtual double read()=0
NeuronSum
+double read()
NeuronProduct
+double read()
NeuronIf
+double read()
NeuronSumThresh
+double read()
NeuronGreaterThan
+double read()
NeuronMin
+double read()
NeuronSignOf
+double read()
NeuronSin
+double read()
NeuronDivide
+double read()
NeuronMax
+double read()
NeuronAbs
+double read()
NeuronSigmoid
+double read()
NeuronLog
+double read()
NeuronCos
+double read()
NeuronMemory
+double read()
NeuronExpt
+double read()
NeuronOscillateWave
+double read()
NeuronInterpolate
+double read()
NeuronAtan
+double read()
NeuronIntegrate
+double read()
NeuronSmooth
+double read()
NeuronDifferentiate
+double read()
NeuronConst
#double value
+double read()
NeuronOscillateSaw
+double read()



Possible Project Extensions

3-D Creatures and Simulations. We have designed our code from the start with the knowledge that we will extend our implementation to three dimensions if we have enough time at the end of the initial implementation effort.

Interactive Selection. A user could perform interactive creature selection in lieu of automated goal-directed selection.

Distributed Processing. The parallel simulation interface could be extended to make use of a distributed processing package such as quahog, which would allow each generation of simulations to be distributed across multiple machines.

The Preserve. Multiple creatures could be simulated simultaneously in a closed world, allowing interaction and possibly even predation. Collision detection prediction techniques outlined in [HUBB] may help optimize simulation of multiple creatures.

Additional Joint Types. Our initial simulation domain is two-dimensional, so we will be unable to implement many of the joint types that Karl Sims discusses. One way to introduce more variety in creature phenotypes would be to add a *linear* joint type. This joint would constrain one of its body parts to move in and out of the other in a manner inspired by cat claws and turtles.

Should we implement 3-D creatures and simulations, we will want to add some of the other joint types that the additional spacial dimension allows, including *twist*, *universal*, *bend-twist*, *twist-bend*, and *spherical*.

Additional Effectors. Some possible additional effectors include *sound* and *scent* emitters, and the *deformation* effector. Deformation effectors are encountered in natural creatures in the form of tentacles and worms. A deformer would deform an associated body part by scaling that body part in a nonuniform fashion. The body part would be constrained to conserve its area. Deformers would contain additional genetically-dictated constraints on the limits of deformation to prevent unrealistic movements. Techniques for implementing deformation dynamics may be found in [MILL].

Additional Senses. Some possible additional sensors include *accelerometers*, additional *proprioceptors*, *light* sensors, *sound* sensors and *smell* sensors. Sound sensors could introduce interesting behaviors if we also implemented the preserve and allowed collisions to emit sounds. If we implemented light sensors we would also add additional parameters that the user can specify at the outset, such as whether a *light source* exists, and light source characteristics such as brightness and location.

Additional Behaviors. Some additional possible behaviors include *seeking* and *flying*. Seeking creatures will be judged by how closely they follow a randomly moving light source. Flying creatures are judged in a manner similar to that for swimming creatures, except for the fact that viscosity is much lower.

References

[BARA] Baraff, David, “Fast Contact Force Computation for Nonpenetrating Rigid Bodies,” *Proceedings of SIGGRAPH ‘94,* ACM Press.

[BARZ] Barzel, Ronen, et al, “A Modeling System Based on Dynamic Constraints,” *Computer Graphics, Volume 22, Number 4, August 1988,* ACM Press.

[DEJA] de JalÓn, Javier GarcÍa, et al, *Kinematic and Dynamic Simulation of Multibody Systems, The Real-Time Challenge,* Springer-Verlag, 1994.

[FEAT] Featherstone, Roy, *Robot Dynamics Algorithms*, Kluwer Academic Publishers, 1987.

[HUBB] Hubbard, Phillip, “Space-Time Bounds for Collision Detection,” Unpublished, Department of Computer Science, Brown University, 1993.

[MILL] Miller, Gavin S. P., “The Motion Dynamics of Snakes and Worms,” *Computer Graphics, Volume 22, Number 4, August 1988*, ACM Press.

[RAIB] Raibert, Marc H., et al, “Animation of Dynamic Legged Locomotion,” *Computer Grahpics, Volume 25, Number 4, July 1991*, ACM Press.

[SHOE] Shoemake, Ken, “Animating Rotation with Quaternion Curves,” *Proceedings of SIGGRAPH ‘85*, ACM Press.

[SIMSa] Sims, Karl, “Evolving Virtual Creatures,” *Proceedings of SIGGRAPH ‘94*, ACM Press.

[SIMSb] Sims, Karl, “Evolving 3D Morphology and Behavior by Competition,” *Artificial Life IV Proceedings*, MIT Press, 1994.

[WITK] Witkin, Andrew, “Particle System Dynamics,” *Course Proceedings of SIGGRAPH ‘94*, ACM Press.