



Evolution
An Artificial Life Allegro Project

Jasen Moley

University of Advancing Technology

CSC404 – SU07386

Mr. Kris Jamsa

06/17/2007

Table of Contents

1. Introduction to Artificial Life	E
2. Introduction to Application	
2.1 Screenshots	
2.2 Players	
2.3 Food Chain	V
2.4 Code	
3. Life Defined	
3.1 Implementation of Life Expectancy	
3.2 Code	O
4. Hunger	
4.1 Represented	
4.2 Grow/Shrink	
4.3 Code	I
5. Movement	
5.1 Pursue/Flee	
5.2 Objects wondering	
5.3 Code	U
6. Vision	
6.1 Represented	
6.2 Code	
7. Reproduction	
7.1 Labito	
7.2 Inability to reproduce	t
7.3 Code	
8. Death	
8.1 Represented	
8.2 Code	i
9. Injury/Illness	
9.1 Represented	
10. Genetics	
10.1 Represented	
10.2 Impacted Attributes	O
10.3 Code	
11. Steady State	
11.1 Steady State Achievement	n

Introduction to Artificial Life

The definition of artificial life according to the Wikipedia article “is the study of life through the use of human-made analogs of living systems. Computer scientist Christopher Langton coined the term in the late 1980s when he held the first "International Conference on the Synthesis and Simulation of Living Systems" (otherwise known as Artificial Life I) at the Los Alamos National Laboratory in 1987.” (wikipedia.com) In simpler terms, artificial life is the ability to recreate real world living eco-systems and mimic them in a virtual world. Using this ability could teach a lot about how eco-systems work in real life. Recreating evolution virtually gives a scientist the ability to conduct experiments in a controlled manner as opposed to a study based off mice for example. Many different methods could be done in a computer simulation which just cannot be done in real life for many reasons such as mass evolution, quick evolution, and many other methods.

Artificial life has been argued repeatedly because there have been many claims that “life” created in a computer simulation will never be as realistic as actual experiments due to so many variables that occur in the real world. Although, according to Tom Ray, artificial life is not simulating life, but actually synthesizing it. Tom Ray was the creator of the computer simulation called Tierra. Tierra is cited quite frequently as an artificial life model. This program creates digital organisms which compete for energy (CPU time) and resources (main memory). It is actually a derivative of the computer programmer's game Core War. There are many other simulators such as Avida, Nanopond, Evolve 4.0, and many others but there are many different techniques of artificial life as well.

Many techniques that are used are actually originally created from artificial intelligence methods such as a Neural Network. Cellular automata is also another popular method due to its scalability and parallelization. Artificial life and cellular automata have shared a closely tied history. Programs have also been created which mimic artificial life but they are not strongly

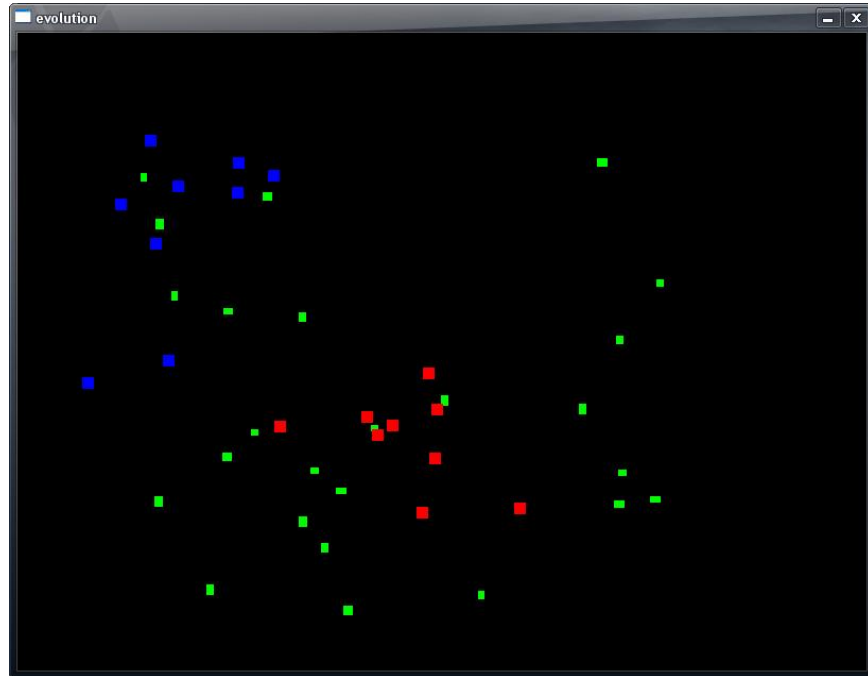
considered to be artificial life because the primary difference lies in explicitly defining the fitness of an agent by its ability to solve a problem, instead of its ability to find food, reproduce, or avoid death. There is an embedded structure as well which defines artificial life separate from artificial intelligence. Artificial life uses a bottom up approach to logic while artificial intelligence is top down.

The bottom up approach is more of a learning approach because an object learns how to “live” and survive. It must learn this as it evolves rather than having pre-set definitions. The top down approach is the exact opposite. It gives an object an extensive array of instances that could occur depending on the situation. But have their appropriate uses but they can be mixed together for a program. Neural Networks for example is a algorithm that could learn although it is still given a list of options that it must follow.

I believe despite the criticism that artificial life has received, it has many particular uses that could prove very useful to biology or to understanding how living organisms work. These abilities also give programmers the ability to recreate living eco-systems within programs or whatever they desire such as a game. An example of this is the game Spore which is a module-based artificial life program. Combined with artificial life, a very real living eco-system could be replicated on a computer which could closely (but not perfectly) copy an eco-system in the real world. My version of evolution is a crude but working model of just how artificial life could be used.

Introduction to Application

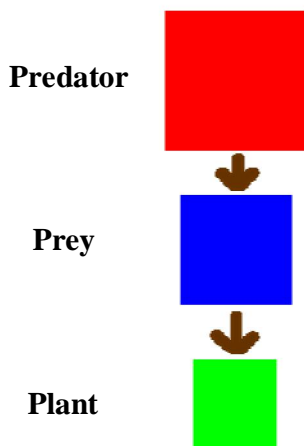
This project was a simple rendition of recreating artificial life. It was done using the allegro libraries in C++. Absolutely no sprites were in this project. All of the renderings are drawn using primitives which are vector graphics although the source code does have the capability to implement sprites. It



pretty much implements the basics as what any AL program would have.

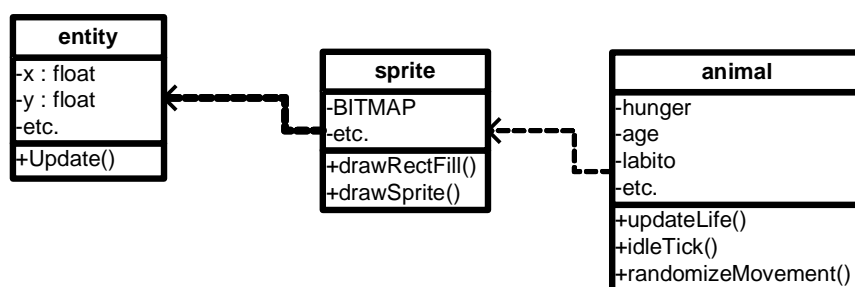
There are your typical predators (red), your prey (blue), and finally the plants (green). This is the food chain in our case. Plants are typically an overabundance which is why there are

Food Chain



so many of them. There are the same amount of predators to prey because they both contain the same in intelligence level. This means that each is given the same abilities so they both on an equal “playing field”. The code on this project is not extremely complicated. The plants, predators, and prey are classes. They are considered an animal class which is derived from a sprite class which is also derived from the entity class. Everything begins with the entity class which contains items such as the x and y position of each object as well as color, speed, and other things. The sprite class is where the objects are drawn. This class gives the ability for

objects created to be drawn to the sprite either by an image or by vector coordinates. In this prototype, vectors are used to demonstrate functionality. The next class is the animal class which stores all the variables which can be used for intelligence. Variables such as hunger, libido, age, and other things that represent an animal is included in the animal class. An example of this hierarchy is shown below. This is the basic layout of the object on the project. The structure of the different abilities of the animals are explained in much greater detail below.



Life Defined

In my project, each animal is given a lifespan. These animals are the prey and predators. This does not include plants at the current time because due to the way the project was structured, it would have been better to keep plants able to outlive the animals. A lifeTick function (lifeTick()) was implemented which allows the world to “tick” once every second. This now give the ability for thresholds for age or hunger be structured via time rather than guesses and trial and error. Each animal is given a minimum value which it will live but it also includes a random number which will give each animal a different lifespan. Code on how this was implemented is shown below.

```
int lifeTicks = 0;
int lifeTickThreshold = 60;
```

```
//continued...
```

```

void checkThoughts(allegroInit &allegro, animal prey[], animal predator[],
animal plant[]) {
//...
    if (lifeTicks++ > lifeTickThreshold) {
        for (int n=0; n <currentPrey; n++)
            prey[n].updateLife();
        for (int n=0; n <currentPredators; n++)
            predator[n].updateLife();
    }
//...
}
/**
 * Function: updateLife
 * Purpose: will update the "life" variables in class
 */
void animal::updateLife(void) {
    this->age++;
    this->hunger += this->incHungerAmt;
    this->labito += this->incLabitoAmt;

    if (this->hunger > this->hungryThreshold)
        this->pursuingFood = true;
    if (this->labito > this->labitoThreshold)
        this->pursuingSex = true;
}

```

This is basically a controlled way to update the “life” of each object because with out any structure, it would create a difficult time to QA the settings for each animal.

Hunger

Hunger is currently handled in a very crude fashion. The original idea was to have hunger as a factor in determining whether the animal would get more desperate and in the end die due to starvation. Unfortunately since this is currently a prototype, hunger only affects the animals desire to find food if it can “see” it and go after it. This utilizes multiple functions with each called through the checkThoughts function. As lifetick() occurs, updateLife() will eventually throw a flag that the animal is hungry. When checkThoughts() sees it, it calls findClosestTarget() to search for plants in the animals sight range. If anything is found, then it will target it and head to that position. Here is a coded example. The predators are handled pretty much in the same fashion as the prey only they require searching through prey instead.

```

for (int n=0; n < currentPrey; n++) {
//...
    if (prey[n].getPursuingFood()) { //After food
        for (int i=0; i < MAXPLANTS; i++) {
            if (plant[i].getAlive()) {
                if (onTarget(preyn)) {
                    findClosestTarget(preyn,plant[i]);
                }
                else
                    followTarget(preyn);
            }
        }
        checkForFood(plant, prey[n]);
    }
    if (predator[n].getPursuingFood()) { //After food
        for (int i=0; i < currentPrey; i++) {
            if (prey[n].getAlive()) {
                if (onTarget(predator[n])) {
                    if (findClosestTarget(predator[n], prey[i]))
                        prey[i].setFleeing(true);
                }
                else
                    followTarget(predator[n]);
                if(checkForFood(predator[n], prey[i]))
                    currentPrey--;
            }
        }
    }
//...
}
bool checkForFood(Animal &attacker, Animal &defender) {
    if (defender.getAlive()) {
        if (boxCollision(attacker, defender)) {
            defender.setAlive(false);
            attacker.setPursuingFood(false);
            return true;
        }
    }
    return false;
}

```

Movement

Movement was handled in an ingenious way. Due to the way the class was structured, each entity can move in one direction via a direction. This allowed the object to move up, down, right, or left depending on the direction it is heading. So when an object see either a predator or food, the direction will change accordingly.

An object wondering is also handled pretty well as well. A timer is setup within the animal class which calls the randomizeMovement function after a desired time limit. randomizeMovement() will pick a number at random and depending on the number, will set the

direction. This gives the impression that the animal is wondering around waiting until he desires food, sex, or when he encounters a predator. In order to keep the animal within bounds of the screen, a function is called which loops until the direction is not heading towards the end of the screen.

```

/**
 * Function: idleTick
 * Purpose: Will increment idleCount in order to give 'animal' time to "think"
 before moving
 */
void animal::idleTick(void) {
    if (idleCount++ > idleDelay) {
        idleCount = 0;
        this->randomizeMovement();
    }
}
/**
 * Function: randomizeMovement
 * Purpose: will change the direction of the 'animal' based on a random number
 */
void animal::randomizeMovement(void) {

    int randomNum = rand() % 100;

    if (randomNum < 25)
        this->setDirection('u');
    else if (randomNum < 50)
        this->setDirection('r');
    else if (randomNum < 75)
        this->setDirection('d');
    else
        this->setDirection('l');
}
void checkBoundary(animal &thing) {
    if (thing.getX() < 5) {
        do {
            thing.randomizeMovement();
        } while(thing.getDirection() == 'l');
    }
    if (thing.getY2() > SCREENHEIGHT - 5) {
        do {
            thing.randomizeMovement();
        } while(thing.getDirection() == 'd');
    }
    if (thing.getX2() > SCREENWIDTH - 5) {
        do {
            thing.randomizeMovement();
        } while(thing.getDirection() == 'r');
    }
}

```

```

    if (thing.getY() < 5) {
        do {
            thing.randomizeMovement();
        } while(thing.getDirection() == 'u');
    }
}

```

Vision

Vision was a very tough thing for me to implement. I had many different methods for handling vision such as finding the color of an object, incrementing the sight from the origin, box collision, and pixel collision but I wanted to end up with a function that is versatile and only requires the animal class. It has also been optimized as much as possible to prevent slowdown because much of my slowdown was coming from this function. Basically, the function determines the max sight of the animal and performs a box collision check to see if it is within the given target entity. If so, give the animal the coordinates of that position and head towards that position. Else the function would set the animal's targets to NULL. This can be used for anything that is an entity so it is possible to use this function to search for plants, prey, or predators.

```

bool findClosestTarget(Animal &attacker, Entity &defender) {

int n = attacker.getSightRange()+1; //sets the sight. Was originally a loop

if (boxCollision(defender, attacker.getX(), attacker.getY()-n,
attacker.getX2(), attacker.getY2())) { //up
    attacker.targetY = attacker.getY()-n;
    attacker.targetX = attacker.getX();
    if(attacker.targetY < attacker.getHeight())//prevents out of bounds
searching
        attacker.targetY = attacker.getHeight();
    return true;
}
else if (boxCollision(defender, attacker.getX()+n, attacker.getY(),
attacker.getX2(), attacker.getY2())) { //right
    attacker.targetX = attacker.getX()+n;
    attacker.targetY = attacker.getY();
    if(attacker.targetX > SCREENWIDTH -
attacker.getWidth())//prevents out of bounds searching
        attacker.targetX = SCREENWIDTH - attacker.getWidth();
    return true;
}
}

```



```

void reproduce(Animal targetArray[], Animal &animal1, Animal &animal2, char
type[10]) {
    if (type == "prey" ) {
        if (currentPrey < MAXPREY) {
            for (int n = 0; n < MAXPREY; n++) {
                targetArray[n] = animal1 + animal2;
                //if (n > currentPrey)
                    currentPrey++;
                break;
            }
        }
    }

    if (type == "predator" ) {
        if (currentPredators < MAXPREDATOR) {
            for (int n = 0; n < MAXPREDATOR; n++) {
                targetArray[n] = animal1 + animal2;
                //if (n > currentPredators)
                    currentPredators++;
                break;
            }
        }
    }

    animal1.setPursuingSex(false);
}

```

Death

Death is just a part of life for any animal. “Every beginning has an End, Neo” as Agent Smith so eloquently puts it. An animal can die due to many reasons. In our case, an animal can die due to getting old or being eaten by a predator. Also, the objects are currently being held by arrays which must be managed when an animal must be removed. Currently this is a very buggy situation so this is one key item that must be changed in order for this project to take off. One possibility it to create a vector class which I can create or use from the standard template library.

```

//check if dead
if (prey[n].getAge() > prey[n].getMaxAge()) {
    currentPrey--;
    prey[n].setAlive(false);
}
if(checkForFood(predator[n], prey[i]))
    currentPrey--;

```

Injury/Illness

Currently Injury and Illness is not being implemented at this time. The idea is for prey or predators to get injured and decrease their abilities the more they are hurt. Plants could also benefit because an animal might eat only half which will not erase the plant visually. The only

real implementation is within the + operator. There is a probability that the baby created could turn out to be “mentally retarded” due to the way it is structured.

Genetics

Genetics is the key item in Artificial Life. This is where the magic begins which replicates life. When two animals reproduce, a pho-genetic combination occurs. The code below can show in its entirety the genetic combination. The way the genetic material is combined is using a variable which adds the stored variables from both animals. What then happens is a random number will be chosen with the range of 0 to the variable with the added animals. This gives an equal opportunity for baby to come out better then their parents, or worse.

```

animal animal::operator+(animal &animal2) {
    int randomNum = rand()%11;
    int combinedVal = 0;

    animal baby;
    baby.setX(this->getX());
    baby.setY(this->getY());
    baby.setWidth(this->getWidth());
    baby.setHeight(this->getHeight());
    baby.setSpeed(this->getSpeed());
    baby.setColor(this->getColor());
    baby.setDirection('u');
    baby.setAge(0);
    combinedVal = (this->getMaxAge()/2) + (animal2.getMaxAge()/2);
    baby.setMaxAge(combinedVal+100);
    baby.setHunger(0);
    combinedVal = (this->getHungryThreshold()/2) +
(animal2.getHungryThreshold()/2);
    baby.setHungryThreshold(combinedVal+50);
    combinedVal = (this->getIncHungerAmt()/2) +
(animal2.getIncHungerAmt()/2);
    baby.setIncHungerAmt(combinedVal);
    baby.setHealth(100);
    combinedVal = this->getLabitoThreshold()+animal2.getLabitoThreshold();
    combinedVal = rand()%combinedVal;
    if (combinedVal == 0)
        combinedVal++;
    baby.setLabitoThreshold(combinedVal);
    combinedVal = this->getIncLabitoAmt() + animal2.getIncLabitoAmt();
    combinedVal = rand()%combinedVal;
    if (combinedVal == 0)
        combinedVal++;
    baby.setIncLabitoAmt(combinedVal); //needs work
    baby.setAlive(true);
    baby.setFleeing(false);
    baby.setPursuingFood(false);
    baby.setPursuingSex(false);
    baby.randomizeMovement();
}

```

```

combinedVal = this->getIdleDelay() + animal2.getIdleDelay();
combinedVal = rand()%combinedVal;
if (combinedVal == 0)
    combinedVal++;
baby.setIdleDelay(combinedVal);
combinedVal = this->getSightRange() + animal2.getSightRange();
combinedVal = rand()%combinedVal;
if (combinedVal == 0)
    combinedVal++;
baby.setSightRange(combinedVal); //needs work

this->setPursuingSex(false);
animal2.setPursuingSex(false);

baby.targetX = 0;
baby.targetY = 0;

return baby;
}

```

Steady State

Currently it is near impossible to have a steady state in the current project for a couple of reasons. First, the list of prey, animals, or plants should not be arrays, but linked lists. This frees up much needed CPU power to be used on other things. The next item on the list is not having the variables optimized for the project to work on. After each of these problems, a steady state could seem to be a real possibility. Much of the work will be in determining what initial numbers work well. Also, I need to look into the timing because I believe that the timing is incorrect for everything on the project.