Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1993

A Simulation of Rapid Evolution: Its Development in the Object-Oriented Paradigm

John Bloom
Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1993-001

A Simulation of Rapid Evolution: Its Development in the Object-Oriented Paradigm

John Bloom

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

A Simulation of Rapid Evolution:
Its Development in the Object-Oriented Paradigm

by

John Bloom
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #93-001 01/93

A SIMULATION OF RAPID EVOLUTION:

ITS DEVELOPMENT IN THE OBJECT-ORIENTED PARADIGM

Final Report

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Systems Analysis
in the
Graduate School of Miami University

By

John Bloom

Miami University

1992

Reading Committee:

Dr. Thomas G. Gregg, Department of Zoology

Dr. James D. Kiper, Advisor

Dr. Yaman Barlas

TABLE OF CONTENTS

INTRODUCTION

This project centered around an event in population genetics, the modeling necessary to explain it, and the way that the object-oriented paradigm influenced the implementation of the model. This paper is divided into three major sections which echo the three lines of investigation in the project. The first section is a summary of the concepts and terminology of genetics necessary for the understanding of what follows. The second is a description of the model and its results. The third is a report on the process of moving the software from a procedural language to an object-oriented one.

1. GENETICS

When plants and animals reproduce, the development of the offspring is guided by the information passed on to them by the parent or parents. The field of genetics (the study of this "passing on") is divided roughly into two areas: molecular genetics and population genetics.

1.1 MOLECULAR GENETICS

The physical structure of genes and the way in which they preserve, transmit, and alter genetic information is immensely complex. Only in the latter half of this century have molecular geneticists had the tools necessary for studying the mechanisms by which information is passed from generation to generation. The discovery of the structure of DNA in the early 1950's was the key to understanding this process. The basic units of DNA (the "bits" in the genetic code) are nucleotides. Nucleotides are molecules that include one of four kinds of nitrogen bases: adenine, cytosine, guanine, or thymine. The DNA molecule is a double stranded helix in which each strand is made of long sequences of nucleotides. The nucleotides are utilized in groups of three, called codons, making sixty four distinct characters in the

alphabet that defines individual genes. But there is some redundancy of chemical activity among the codons, so there are only twenty characters and an end-of-word signal available. The existence of this signal means that the words (genes) of the genetic language can be of arbitrary length and from this fact comes the nearly unlimited variety of genes. Genes act as instructions for the assembly of amino acids into proteins which determine the individual characteristics of cells and organisms. The incredible intricacy and diversity of organic evolution is the expression of the accumulated differences in these proteins.

1.2 POPULATION GENETICS

Population genetics is the study of the dynamics of genes in a population (changes in relative frequency of genes as they are passed down through successive generations of a group of individuals). Population genetics takes the complexities studied in molecular genetics as given, and deals with genes as atomic objects. Similarly, the impact of an individual's genetic inheritance on its physical growth and makeup is immensely complex. Here again population genetics abstracts a single value, fitness (the individual's ability to produce offspring), as its object of interest.

1.2.1 DEFINITIONS

These simplifications allows population genetics to be built up from a few basic definitions. The gene is the basic unit of inheritance. A locus is the position on a chromosome occupied by a gene. An allele is one of several alternative forms of a given gene which can appear at a locus. The entire collection of genes carried by an individual is its genotype; and the phenotype is the physical manifestation of genetic information in an individual genotype.

In populations of greatest interest, those in which diploid individuals reproduce sexually, there are two alleles at each locus, one inherited from each parent. The notation common in multiple locus studies uses a letter to denote the locus and a

subscript to denote the allele. So B_3 refers to the type three allele at the B locus and, in general, B_3 is not the same gene as A_3. Because the chemical action of each allele at a locus is independent of the other, the combination $[A_{12}]$ is equivalent to the combination $[A_{21}]$.

1.2.2 THE HARDY-WEINBERG EQUATION

The basic mathematics of the way alleles are distributed in a population is expressed in the Hardy-Weinberg equation, which was developed in 1908 and is the foundation of mathematical population genetics. The equation states that, barring mutations and evolutionary pressures (differences in fitness) and assuming random mating and distinct generations, the frequencies of given alleles in a population will not vary under the action of heredity, and that the frequencies of genotypes in the population is stable and determined by the allelic frequencies. For instance if we know the alleles A_1, A_2 and A_3 appears in the population with frequency f_1, f_2 and f_3 respectively (with $f_1 + f_2 + f_3 = 1$). Then the frequencies in the next generation will be same and the genotypes will have frequencies given by the square law:

$(f_1+f_2+f_3)^2$ ==>

frequency	f_1^2	f_2^2	f_3^2	$2f_1f_2$	$2f_1f_3$	$2f_2f_3$
for genotype	$[A_{11}]$	$[A_{22}]$	$[A_{33}]$	$[A_{12}]$	$[A_{13}]$	$[A_{23}]$

The analogy here would be dumping three colors of balls into a barrel and taking them out with replacement two at a time at without regard to order.

These equations yield more interesting results when the fitness factor (the frequency with which a genotype passes it alleles on to the next generation) is added. The notation used for the fitness of genotype $[A_{11}]$ is usually W_{11}. Now f_1 in the second generation of a three allele one locus genotype can be calculated as:

$$
\begin{aligned}
& W_{11} * f_{11} * 1 && \text{(each allele from } [A_{11}] \text{ is } A_1) \\
+\ & W_{12} * f_{12} * .5 && \text{(half the alleles from } [A_{12}] \text{ are } A_1) \\
+\ & W_{13} * f_{13} * .5 && \text{(half the alleles from } [A_{13}] \text{ are } A_1) \\
+\ & W_{22} * f_{22} * 0 && \text{(none of the alleles from } [A_{23}] \text{ are } A_1) \\
+\ & W_{23} * f_{23} * 0 && \text{(none of the alleles from } [A_{23}] \text{ are } A_1) \\
+\ & W_{33} * f_{33} * 0 && \text{(none of the alleles from } [A_{33}] \text{ are } A_1)
\end{aligned}
$$

Because of the highly abstract nature of the variables in these equations, they can be churned algebraically without interference from field data in most cases. An incredibly large (potentially unlimited) number of papers and dissertations have been written using the permutations of this algebraic formulation with a list of modifying assumptions such as mutation, migration, chance events, diversity, competition etc. Most of these studies suffer from two limitations based in the algebra. First, they use only one-locus genotypes because calculations becomes very awkward as soon as multiple loci are considered. This limitation is far from trivial. In practice many genes seem to contribute to multiple phenotypic traits in nonlinear combination with other genes. Thus a phenotypic change that affects fitness will usually caused by changes in the alleles at several loci. Second, while algebraic models in which one of the Hardy-Weinberg assumptions is altered can be built, changing several assumption leads to extreme difficulties in calculation.

2. THE MODEL

While the general framework of population genetics lends itself to algebraic analysis, there are specific problems that require a finer tool for modeling. In this section I discuss one such problem. I then describe the discrete simulation built to study it; first its inner workings, then its output.

2.1 THE PROBLEM

The problem our model is designed to investigate is the question of the speed at which major evolutionary changes can occur. Major change means a new phenotype replacing an existing one in a population or, in the language of population genetics, a shift in the frequencies of alleles at several loci that allows a new fitter genotype to become predominant. The problem of the time it takes for this to happen became apparent at the interface of population genetics and paleontology. Evolution as imagined by Darwin, although he worked without any knowledge of modern genetics, was essentially a single locus model. He thought that favorable mutations would be spread through a population gradually because of the higher fitness of the inheritors. This is a good model if we can allow enough time for the mutation rate and a slight change of fitness to establish a new allele and thus a new genotype; and many of the classic algebraic models were built to fit this paradigm. Times of 1,000,000 generations or more were typical.

But there are two distinct problems with Darwinian gradualism. First, the fossil record shows that the predominant phenotype in populations often changes very rapidly (rapidly, in evolutionary terms, is 5000 generations). Rates like these cannot be modeled with gradualism because calculated rates of gene substitution are to slow. Second, as stated above, many phenotypic changes require allelic changes at several loci. For these changes the gradualist model will not work. Take for example the following situation. A fitter genotype that requires allelic changes at each of 5 locus may be available, but if changes occur at only three or four loci a less fit genotype is produced. In a large population, and with enough time, the superior, but rare, 5 locus genotype will occasionally be formed by chance from the combination of two inferior parents. That individual will probably survive, but, when it mates, its genotype will be lost because the genetic contribution of its inferior mate. Its offspring will be of the weaker three or four locus type. Under these conditions, in which mating diffusion

overcomes an advantage in fitness, no amount of time would be enough for fixation of the new genotype.

2.2 THE SOLUTION

The model I propose as a solution for this problem allows a small colony to be isolated from the main population. In such a population the law of large numbers would be suspended and the superior genotype might be fixed. If this colony were then reintroduced to the population it would be immune to the effects of mating diffusion because there would be enough superior individuals in the local area for inbreeding. It would then be able to rapidly invade and replace the larger group. Such a scenario is supported by field data that show the geographical break up of many population at the edges of their range. Although this model deals with a significant problem and has a simple intuitive appeal, it has not been studied because it cannot be modeled algebraically. It requires that individuals mate with particular neighboring individuals and this action is too detailed for a stochastic approach. This limitation and the existence of easily recognizable individuals in the problem formulation argue for a discrete simulation as a research tool.

2.3 MODEL ORGANIZATION

A model sufficient to simulate this solution should at minimum have a population of discrete individuals each with values for genotype and position. Position is important because the isolation of the colony and the subsequent invasion would be geographic in nature, and because it is also an obvious way to control random mating. With random mating the tendency of the colony members to mate with each other after the reintroduction would be eliminated and the superior genotype would be lost due to mating diffusion. The concept for position in the model is that individuals exist on a two dimensional map and are given a search distance in which to find a mate. When an acceptable mate is found, the genes from the parents are mixed and the

offspring's genotype is established. The offspring then competes with existing individuals in the search area. For this competition the genotypic fitnesses of the offspring and the existing individual are compared to a random variable. If the offspring is successful, the position of the existing individual is taken over by the offspring and the existing individual dies. This arrangement has two advantages. First, the local search for mates and living space is a mode common in many species. Second, varying the search distance provides a simple way to control the degree of inbreeding.

Each particular run is controlled by 5 parameters:

LOCI	the number of loci; typically 2-4
ALLELES	the number of possible alleles that can appear at each locus; typically 2-3
SEARCH	the distance each individual can travel for mating and competition. A search distance of 1 covers nine squares, 2 covers twenty five; typically 1-10 (at 10 the action is essentially random.
FITNESS[]	the competitive strength of each genotype. For a two locus two allele run, the fitness values might be $W_{1111} = 1$, $W_{2222} = 1.2$ and all other genotype sets = 0.9. ; typically 0.8-3.0
INIT[]	the initial frequencies of the alleles in the population; typically 0.1-0.9

The model is designed with a large square region and a small square region that share one common side. If the large region is initialized with type 1 alleles at a frequency of 0.9 and type 2 at 0.1, the frequency of $[A_{11}B_{11}]$ genotype will be $0.9^4 = 0.656$. The smaller region has only type 2 alleles and thus is uniformly $[A_{22}B_{22}]$ genotype. Thus the model is started at the moment at which the smaller, fitter colony is reintroduced to the main population. A cycle of generations in which the individual at each position is put through the mating-reproduction cycle is

then begun. The progress of the run is tracked by keeping statistics on the frequencies of the genotypes and alleles in the population. The model produces two types of output, static and dynamic. The static output is a graph (see Figures 2-3 below) showing the changes in the frequencies of the phenotypes over time. The dynamic output is a display of the map with each square filled with a color that represents the genotype of the individual that occupies it. The map is updated every time an individual is replaced which provides an animated display. The time axis of the graph is replaced by real time and the user can observe the shape of an invasion or the clustering effects of inbreeding. I found that watching the animated display helped me to see the workings of the model while the graphs tended to confirm what I already suspected to be true. Figure 1 is an example of a map.

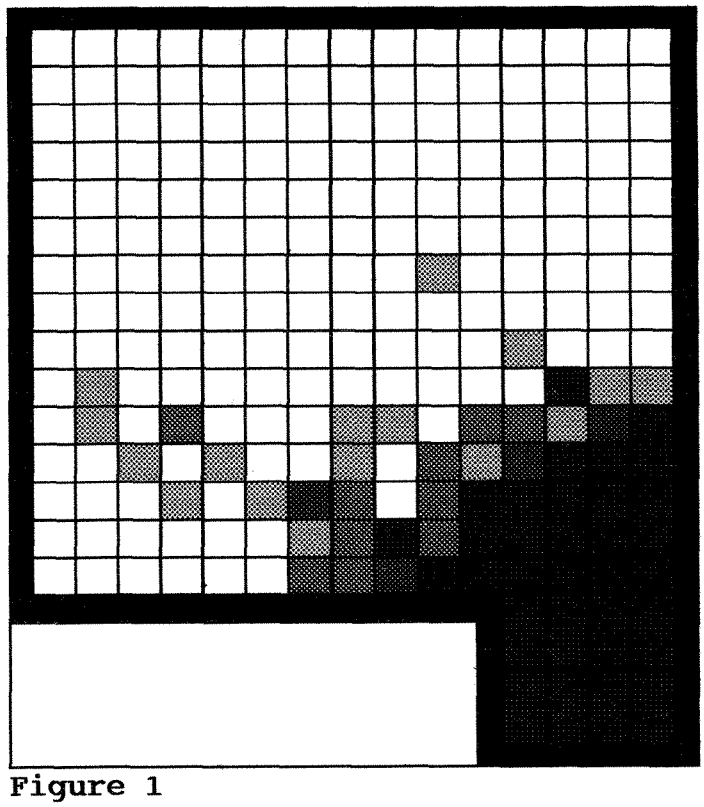

Figure 1

Figure 1 shows the 20th generation of an invasion of by $[A_{22}B_{22}]$, in black, of a population in which $[A_{11}B_{11}]$ had been fixed. The gray squares

symbolizing mixed genotypes with each level of grey denoting another representing a higher total of type 2 alleles. The search distance is 1 and the fitness of $[A_{22}B_{22}]$ is 3.0 which accounts for the uniformity of the original colony. The main population was initialized with all $[A_{11}B_{11}]$ for the sake of visual clarity. In most runs their frequency was 0.9.

2.4 RESULTS

The parameters that were read in for creating runs were set follows:

population size was set to 10,000. This size is common in the literature and could be produced with a 100 X 100 map size.

colony size was always 36. It had to be small enough to allow the unlikely event of fixing the rare superior genotype.

the number of generations was always 500. This is a short time in evolution and the largest time that can be comfortably graphed on the screen.

number of possible alleles was 2. More alleles would have made the division of genotypes into sets more complex without adding to the generality of the experiment.

the number of loci in the genotype varied from 2 to 4

the number of genotype sets was always twice the number of loci plus one. I used an algorithm that counted the number of type 2 alleles, so that with two loci, the set number would always be between 0 and 4.

the fitness of the genotype sets varied from 0.9 to 2.5. Traditionally fitness of the dominant genotype is taken to be 1.0

the search areas were 1, 2, 3 and 5.

Testing for each one of these combinations led to 48 runs. Analyzing the results confirmed the strength of the colony invasion hypothesis but showed that it is not perfectly general. Runs with relatively low superior fitness and a large search

factor allowed the large population to defuse the small population. The graph in Figure 2 shows fitness 1.15 and search 3.

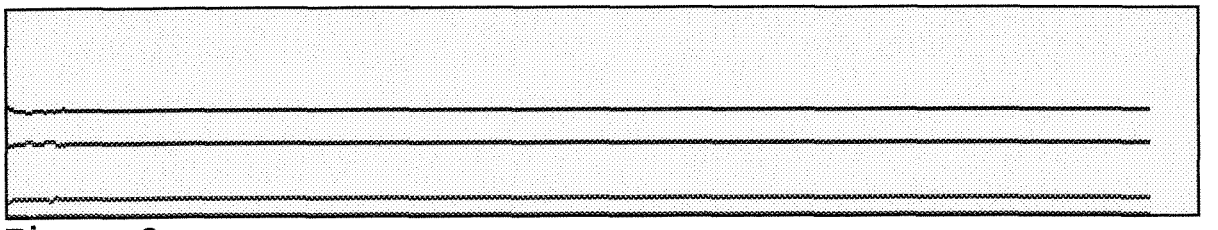

Figure 2

The straight lines in this figure signify the elimination of the superior genotype after only 50 generations. With a smaller search area, which promotes inbreeding, even a relatively weak superior genotype will succeed as Figure 3 shows. Here I used fitness 1.20 and search 1.

Figure 3

It is important to note that the absolute values of the search and fitness parameters are not significant. The ranges I constrained them to seem to be reasonable but the detailed working of the model does not represent any particular species and there is very little empirical data of fitness. The statements that can be made about the data concern the differences in run results as values are changed.

In population genetics, and in this model, there is no attempt to explain the effect of genotype on fitness with, for instance, an algorithm that develops fitnesses using the details of the genotypes. Rather the fitness values are defined

externally to suit the needs of a particular run. This restriction means that this is not a model in which unimagined behavior is likely to emerge. With so many parameters set from the outside and the limited output, the model does not have the latitude to be creative.

However, there are general lessons that can be learned from reviewing its output. One is that multi locus change should be fast. When the superior genotype is successful we see a typical logistic curve as it overcomes the other types. Why do we not see the flat curve predicted by gradualism. Calculations with the algebraic model, predict a fitness of 1.001 in a one locus genotype would take some 9300 generations to fix a ten thousand member population. However such a low fitness advantage in a multi locus model would lead to the superior genotype being lost through diffusion. In this case as we saw in Figures 2 & 3 the fitness must be 1.20 before invasion is possible and with this fitness the curve will be steep and the evolution will be fast.

Another lesson is, given that the colony has been established, the pace of evolution is not greatly effected by the number of loci. In my model there are 10,000 non-superior individuals, and it makes no difference how many genotypes they are divided into, since the superior genotype reproduces itself through inbreeding.

A final lesson is that under some circumstances the separate colony is not needed at all. A small search area can produce enough inbreeding for the rare superior genotype to become fixed in a local area inside the population, and spread from there.

3. SOFTWARE

Software development was a major focus of this project. As an exercise, after the model was running the C code was rewritten in C++. In this part of the paper I focus on design issues that emerged during the change over from functional to object-oriented mode.

3.1 THE FUNCTIONAL DESIGN

Designing a program to implement our model using the functional paradigm, was quite straight forward. In order to make the contrast with object orientation clear, I looked for what actions the model required and designed to them.

3.1.1 PROGRAM STRUCTURE

A top down analysis of the model (as shown in Figure 4) yields a structure with initialization routines and drivers on the highest level.

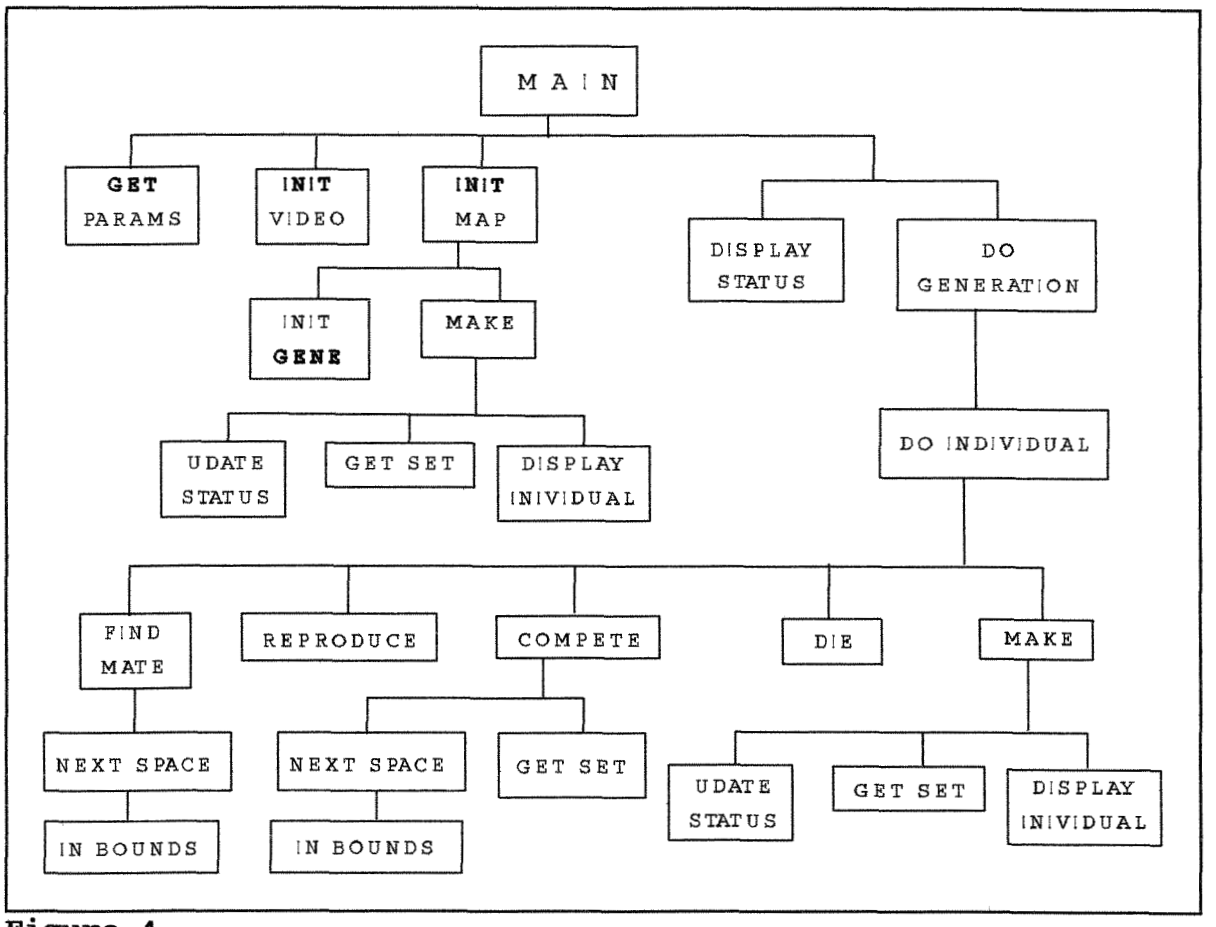

Figure 4

The first function in the initialization routine, get_params(), brings in the run parameters from an external text file. These parameters include values for the fitness, allele

initialization and a number of constants that define the size of the genotype. As the initialization proceeds init_vid() is called. Here video mode is established and the screen is formatted. Finally the map itself is supplied with the first generation of individual by init_map(). Init_map() iterates through each square creating individuals. It calls init_gene() to established the genotype of each new individual. This function chooses alleles with a Monte Carlo function using the initial frequencies given for the run. These genotypes are then sent to make(). Make() calls get_set() with the newly created genotype as a parameter and uses the returned set to find the fitness of the new individual. With generation set to zero, the individual's structure is complete; and make() updates the model's status and assigns the structure to the active position in the ind[] array. Make() (and the die() function) are the only places in the program where the values in the main two dimensional array are actually changed.

The other branch in the top level hierarchy, the driver, puts each generation of individuals through the annual cycle and keeps track of the statistics. The main loop calls display_stats() and do_generation() so that the model's display is updated each generation. The loop continues until a preset number of generations brought in from the external file is reached. Do_generation is a loop nested inside the main loop that indexes through each of the locations on the map putting the individual there through its annual cycle. The first step in the cycle, implemented in find_mate(), is for the active individual to search for a mate. Here next_space() is called repeatedly until an acceptable mate is found. For random mating, next_space() would simply return a random (x,y) position. But when there is a local search area in use, next_space() indexes through the spaces that are within the search distance of the active individual. Since an individual on the edge of the map will sometimes search beyond the edge, a function that defines the map, in_bounds(), is called by next space to filter out undefined positions. When a mate has been found, the genotypes of the active individual and the mate are mixed by the reproduce()

function with each locus in the offspring's genotype receiving one allele from the analogous locus in the active individual and one from the mate. Competition for space proceeds by calling next_space() to find an existing individual in the search range. To parallel the action of the Hardy-Wineberg equation, the offspring's fitness is divided by the fitness of the existing individual plus the offspring's fitness and the resulting fraction is compared to a random variable. If the fraction is higher, the offspring replaces the existing individual. When this happens, the die(existing) is called. This function zeros out the existing individual's structure and updates the model's status. Then make() is called and the offspring's data is entered at the position it has just won.

3.1.2 DATA TYPES

The main data structures are suggested by an inspection of this design. There are two main actors: the map and the population. Early in the project I assumed that there would have to be a structure for each one. But this would have led to a duplication of data since the individuals have to know their position and each position has to be able to return the individual who is there. This dilemma was solved when it was realized that squares that make up the map are always occupied by one individual (the death of an individual can only happen when it is replaced) and that individuals never move from space to space; i.e., in the context of this program, an individual is a space. This one-to-one mapping allowed me to combine the two structures into one. Since the model simulates the actions of individuals on an X Y map, a two dimensional array of structures representing individuals and indexed on their positions was made the central data structure.

An accommodation that was made for the sake of prototyping was making the array of individuals, ind[], global. On its face this decision runs counter to the tenets of modular programming. But closer examination supports the move. Ind[] is used in more than half of the functions in the program and would have to be passed through many others to be visible where it is needed. Thus

making it local would be of no value since, in C, an array cannot be passed by value and any of these functions would then be capable of making the dreaded inadvertent change.

The variables carried by the elements of this array of individual structures are its genotype, fitness and generation. The genotype is used in reproduction to determine the genotype of the offspring and in competition to assign a fitness. Since there can be a very large number of possible genotypes, it was decided that they would be grouped into sets for the purpose of assigning these values. The second important data structure then, is an array of fitnesses indexed on set number. The sets can be defined in various ways for various runs and do not all have to include the same number of genotypes. To obtain a fitness for a given individual its genotype must be mapped to a set number by an algorithm and that number is used as the index to the fitness array. To avoid having to do this repeatedly for a given individual, fitness was added to the individual structure.

3.1.4 MODULARIZATION

These examples show how the functions and data were tailored to the needs of the model. But there was another parameter in the design process. A major feature in the project was its exploratory nature. While the broad outlines of the event I was modeling were understood from the beginning, the details were not. It was expected that by experimenting with the model, new avenues for research into fast paced evolution would be found. From the software development standpoint this meant that the design would have to be seen as a prototype and that the program structure would have to accommodate change. Examples of two changes I made were: a different algorithm for defining the sets of genotypes; and additions to the data structure of the individual. Changing the set mapping algorithm did not present problems from the view point of design. Standard modular design methods made this change seamless. The new function was called with the same parameters, so no changes had to be made anywhere in the rest of the code.

But adding to the main data structure which is accessed from

many places in the code was quite disruptive. The second change involved adding an origin value to each allele. The value is set at initialization giving a unique number to each allele in the population. As the run proceeds, data is kept on the number of loci in which the two alleles are identical by decent i.e. which have the same origin number. To do this the allele was redefined as a structure with two elements, the allele and the origin number. In retrospect this arrangement was seen to be a mistake. The change reverberated throughout the program because everywhere an allele value was used the variable had to be called as ind[].geno.al instead of ind[].al. This required altering lines of code in many places in the program. The error, however, made sense from a functional standpoint. The allele structure neatly instantiated the semantic requirements of the new value, and made the code for copying and transferring alleles more straight forward.

The exploratory nature of the project was also expressed in the structure of the program which was dividing into 10 different files. The modularization of large programs into programmer-sized files is a standard method for software developers. But this was a small program and, if it had been developed with the aid of a complete set of requirements, it could have been written in the one-programmer, one-module mode. In fact it was started that way and as each feature was added the entire program would be saved. Blind alleys made this procedure awkward. New code was being written in several places in the module at any given time. So when it became necessary to backtrack to a previously saved module because one of the concepts did not work, improvements made at other places were lost. The grouping of functions into files was arranged with an eye toward future changes. Functions that would be altered by an imaginable change were put in the same file. The outcome was as follows.

Module	Functions	
U T I L I T I E S		
MAIN	main()	This module was created for sentimental reasons.
UTIL	getnum() getfract()	These random number functions should never be changed
SETUP	setup() get next()	These two functions bring constants in from the external file and set the values of the global variables.
DRIVE	do_individual() do_generation()	These two functions define the flow of control in the program.
A C T O R S		
INIT	init()	This function defines the genotypes that fill the initial map
COMP	compete()	This function was changed often to make our system comparable to algebraic systems.
NSPAC	nextspace()	This function controls the degree of inbreeding.
INBOU	inbounds()	This function defines the shape of the map.
SET	get_set()	This function defines the genotype sets
D A T A		
BKEEP	update_stat() display_stat()	The output statistics were local to this module.
GENE	reproduce() copygene() order gene() equal() initgene()	These functions manipulate the gene structure.
VIDEO	initvid() setpallete() cllr() format() do_ledger()	These functions all have to do with the screen. They include <graphics.h> and share several constants.

INDIV	die() make()	These two functions have access to the array of individuals

The functions were grouped into modules for three different reasons delineated by the headings. The modules in the UTILITY section were formed because their functions perform related services for the program. Changes in, for instance, the format of the external data file would be reflected in the Setup module only. The ACTOR modules are groups of functions that define some part of the individual's behavior and were changed often as I simulated different situations. The modules in the DATA section were formed because they would have to be changed if one of the data structures were changed. The different reasons for forming modules and the fact that modules do not fit unambiguously into one section, hint that the system did not make changes perfectly local. For instance, get_set() takes a genotype as a parameter and returns the number of the set to which the genotype belongs. It was put in its own module because the genotype-to-set mapping changed with the needs of the simulation. However, it uses the genotype structure and, if data were the driving concern, it could have been put in GENE. This sort of overlap made the modulerization less than perfect, but on the whole it worked well.

3.2 OBJECT ORIENTED DESIGN

After the functional code was completed and the model was running, the entire project was rewritten in C++. What follows are some observations about the change over.

3.2.1 CLASSES

In Object Oriented Design (OOD) data and functions are tied together in classes that express the functionality of some part of the problem space. The question the designer asks is "what really exists in the problem space?" or "What are the entities that act in the problem space" The goal is to find a dog and a

cat instead of a pile of legs and a pair of bodies. Answering these questions divides the problem into chunks that have two interrelated qualities. They tend to model real world entities and thus our fashion of thinking; and they have low coupling. Within the context of the module formation above, these decisions combine the same parameters as DATA and ACTIVITY for real entities or DATA and UTILITY of programming entities. Rewriting the simulation using OOD allowed me to study the way function grouping decisions are made in a different context. The first difference was that the grouping of functions into classes is forced on the designer early in the process. With the functional concept it was an afterthought - done to accommodate ongoing changes. With OOD, the designer's first approach to a problem is deciding what the objects are; and these decisions will be freed from the details of implementation. It is also true that the search for objects is a more general guide to program development than designing for modifications, because it does not require the designer to look into the future.

3.2.1.1 **BEINGS**

The most obvious class in this simulation is the beings (the individuals in the functional design). They combine data: their genetic make up, age and position; actions: searching, mating and competing; and a clear is-ness. This last quality is easy to see in the thing being simulated (what laymen like to call reality). There will be actual fruit flies, or whatever, swapping genetic material. Is-ness is also easy to see at the modelling concept level. Discrete simulations are defined by their technique of keeping track of individuals. And at the programming level, using the same analysis as the functional method, the action functions are going to share the data associated with a being.

At the other end of the scale of objectness is Pos, the structure that holds X,Y position values. It has data, and functions act on it; and the fact that it is abstract and passive should not rule it out (see map below). Why not make Pos a class? Inspection of the way Pos is used shows that it would be a

nuisance class. There are no special operations on positions that combine or change them. The only function that deals with pos exclusively is the extractor: getPos(). So its only effect of a position class would be to create awkward code like:

```
pos2 = being.getPosition().getPos();
```

The value of thinking about Pos as a class is that it teaches the designer to ask another question:"Why should this not be a class?"

3.2.1.2 GENES

A second class is clearly the gene. It combines data, the arrangement of particular alleles; actions, reproducing, copying and initializing; and they clearly exist in the problem space. The interesting question raised by the gene class is its relationship to beings. In the real world, genes are contained in individuals. In the model, genes are part of the data of the individual. This inclusion is expressed in C as a nested data structure. C++ offers an additional mechanism: inheritance. The classic use for inheritance is for a number of related classes to inherit functions and data from an abstract parent class that expresses their similarities. This implies a spreading tree structure with functionality and diversity being added at each level. The basic relationship between child and parent in this sort of tree is "is a". For example, horse is a mammal. This relationship does not obtain between beings and genes. There is no spreading in the move from the gene level to the being level; and a being is not a less general gene. With this in mind, the original implementation had the gene object, declared as gnt, included in the being object along with position and age. But this created two subtle problems. The first was basically aesthetic. Including the gene created ugly code like:

```
newgene = gnt->reproduce( *gnt, *(mate->getGene()) );
```

in which the included gene is used to locate its reproduce() function for which the gene itself is a parameter. A more important problem involves the calling of constructors. Gene was coded with three constructors: default, copy, initialize. The copy constructor for gene is:

```
Gene( Gene& gn ) { *this= gnt; }
```

It takes an existing gene as its parameter and copies its genotype and set onto the new gene. In a being's life cycle, its gene is created, competes for space, and then, if it is successful, is copied into the newly created being along with age and position. This copying is done by the being make constructor. The problem is that this constructor knows that the gene is included and it automatically calls the default constructor for gene:

```
Gene(){}
```

so the copying must be done in the body of the code. However, if Gene is a base class inherited by Being, a syntax is provided for specifying the Gene copy constructor from the Being make constructor:

```
Being( Pos pos, int gen, Gene  gn ): Gene(gn)
{
  setUp( pos );
  generation = gen;
}
```

This code explicitly calls the Gene copy constructor with gn and then creates a being with position pos and age gen. This solution was chosen for its cleanliness even though is bends the rules of inheritance.

3.2.1.3 **MAP**

A more difficult question for the designer is "does the map

exist?". At first glance it does not seem to be a good candidate for classhood. The map structure could actually be eliminated from the program all together. Searching could be performed by polling random individuals until one with a position in the search range is found. With this in mind, it could be argued that the map is just a convenient index that returns a nearby individual. The statement is true with the exception of the word 'just'. An index is more efficient computationally, but also parallels more closely the way that searching should be done. In the field, searching a local area is done by the individual sensing other individuals are in the range, not by contacting others and asking if they are nearby. Here there is no agent necessary to specify which individuals are in the range; they are simply there. But in the code, specifying is an action that must be performed by something and that thing is clearly the map. The map-as-object has another responsibility: defining its own shape. In the development of the model, when the colony was added to the square map, another function, in_bounds() was written to mark the inaccessible squares in the two dimensional array. This function became a private member of the Map class.

Now we have an action and an actor, the basic requirements for building a class, but there are two more arguments for making the Map a class: coupling and control. In the functional program searching is expressed as a two dimensional array and a function. Finding the next square is done by nextspace(). Specifying the individual is done by using the position from nextspace() as an index to the map array with code like:

```
pos = nextspace();
mate( ind[pos.x, pos.y] );
```

This code has the functionality required for the model but mate() is tightly coupled with the implementation of the map data structure; in this case ind[]. To loosen the coupling we need a function that takes position as a parameter and returns an individual, allowing code like:

```
pos = nextspace();
mate( get_ind(pos) );
```

which might have been used if changing maps had been required by the model. In its effort to reduce coupling, this design would create a primitive object with ind[] as its data and get_ind() its function. This is another argument for making Map a class.

The other OOD tenet that influenced the decision to make Map a class is decentralization of control. As much as possible, the flow of a program should be driven by the interaction of objects, as opposed to the hierarchical system in functional design. In the functional program initializing is implemented as a module that indexes through the map calling init_gene() and make(). With OOD the indexing function forEachSpace() becomes a member of the Map object. It takes a pointer to the being initialization function as a parameter and sends each position on the map to it. With a different parameter, a pointer to a function analogous to do_individual, it can be called from the main() module instead of building a control loop there. The ultimate arrangement for control by object interaction would be each being telling the next one to go through its cycle, but this would require concurrency and is thus beyond the scope of C++ when it is run on a DOS system.

3.2.1.5 **UTILITIES**

In addition to objects present in the thing being modeled, classhood can be bestowed on objects present in the program. The output section has data and functions local to it and was actually implemented as a primitive class in the functional program. The output module, which holds update_status() and display_status(), was formed because it allowed the variables that they both use (set_count[] and allele_count[]) to have file scope. The values of these variables are the numbers of individuals in each genotype set and the total numbers of various alleles in the population. These values are never used by the

individuals; they are called only for output. Update_status() is called by make() and die() every time there is a change in the population. At the end of each generation the driver calls display_status(). It formats those values and prints them to the screen. In order for update_status() to have access to these variables, in a standard modular program, they would have to be passed all the way down the control structure to make() and die() and used as parameters. To avoid this awkward arrangement in our code, the functions were placed in a separate file and the variables were declared static within that file. This file, then, operated in much the same way as a class in C++. In fact, classes that preserve most object oriented functionality (with the major exception of inheritance) can be written in a procedural language. Object oriented languages like C++ provide syntax that makes classes cleaner, but a large part of the power of OOD, the guidance it gives to the designer for building in low coupling, is available without these enhancements. In saying this I am in disagreement with the idea [Ames 91] that inheritance is as central to OOD as data abstraction.

It should be noted that this class did not survive the recoding to C++. A class can have variables that have the same values for each of its objects. This classification, static, is a good place to put numSet[] and numAl[]. It meant that a separate file to make the output variables local was no longer necessary. The functionality of update_status was moved to the Being class; and output_status() which deals directly with the screen was put in the video class where it belonged in the first place.

The first C++ video class written, treated the entire screen as its entity. It included a number of constants for positioning the map, the legend and the graph, and functions for displaying them. The class declaration was as follows:

```
class Video
{
     private:

          int box ;                    // side length of individual
                                            square

          int Tvrt, Thor;              // text to pixel conversions
          int border ;                 // width of outline
          int graphtop    ;            // graph position parameters
          int graphbottom ;
          int graphleft   ;

          int left ;                   // text position parameters
          int top  ;

          char* format(float, int );     // format numeric text
          void  do_legend (void);        // initialize screen
          void  format_display(void);    //    areas

     public:

          void initVid(void);            // initialize video
                                         //  system & screen areas
          void display_ind( Pos, char); // display an individual
          void exit_vid();           // remove video system
};
```

This first module worked, but the numerous examples of video classes in the literature led me to rethink the arrangement. The second video module was divided into finer classes held together in a network of inheritance. In fact, GraphicElements classes are so finely divided that calls to them look just like calls to their underlying C++ functions. In this situation the criteria used in developing classes is somewhat different than it was with

the independent class discussed above. The second arrangement is shown in Figure 5.

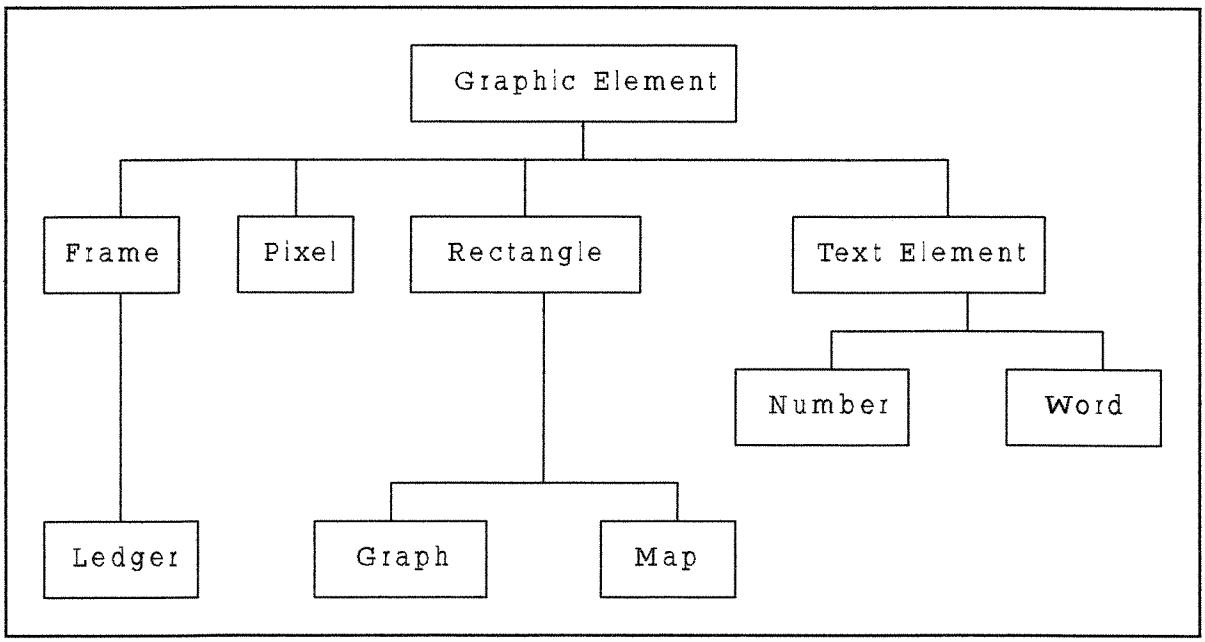

Figure 5

Here classes are sometimes established because they combine elements from the hierarchical layer below them rather than because they represent some active entity. An example of this is GraphicElement, the ultimate base class in the video module, which holds location and color, but has no methods of its own. It is an abstract class, one that will never be instantiated. The other abstract class, TextElements, holds no data but has the text-to-graphics conversion function needed by both Number and Caption. In a functional system this routine would have simply been a separate function called by Number and Caption. But inheritance provides a more structural way of letting these parts of the system work together. In the same way Pixel inherits its location data from GraphicElement and provides the plotting method. Together they display a single dot on the screen.

The facilities offered by these classes combined with the TextElement classes, are used by Ledger, Graph and Map to produce the three output windows. These classes have more claim to

modeling part of the problem space than the above do. The Windows classes coordinate the position of their elements and have both initialize and update functions. However, as Figure 5 shows there was no seminal Window class in the second design. They each inherited from a GraphicElement. This was not satisfying because nothing in the hierarchy expressed windowness. But it was an implementation issue that demonstrated the need for third design shown in Figure 6.

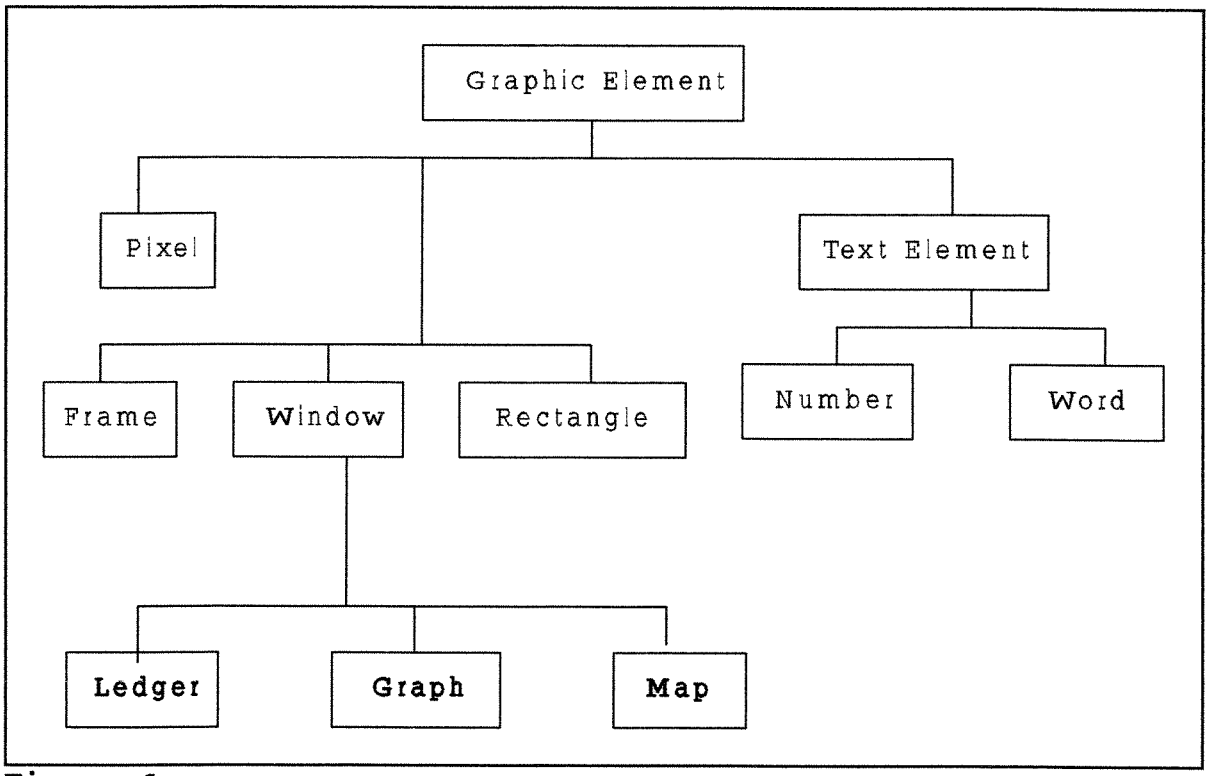

Figure 6

The Window class was needed to hold the virtual function update(). In C++ objects from different classes often perform similar actions. It is sometimes necessary to call these actions from an array. Having a common ancestor with a virtual function which is instantiated differently in each of the classes allows code such the following:

```
Window* clss[3];
Graph* gprt = new Graph( x, y, z );
Ledger* lptr = new Ledger( x, y, z );
clss[0] = gptr;
clss[1] = lptr;
for ( int i = 0; i < 1; i++ )
     window[i]->update();
```

This would not compiles if Graph and Ledger did not inherit from Window.

The third design tries to combine the concept of the class as an entity modeler with the organizational power of inheritance. It shows that the process of defining classes There is a satisfaction and a clarity in way that inheritance provides a syntax for the interdependence of functions and the way it builds then into a hierarchy. But it does these things at the expense of the basic role of classes: that of gathering together related parts of a program into a cohesive whole.

3.2.1 CLASS LIBRARIES

It was decided that letting the Map hold the Being objects coupled the two classes too tightly. A more object-oriented approach would be putting the beings in their own data structure. This way the order of indexing through each generation would not be tied to position. To implement the list of beings, I used code from a library.

One of the goals of OOD is to increase the reuse of code. The Borland Turbo C++ 3.0 used for this project came with a set of class libraries that are designed to be plugged into user code. Because they are designed for general applications, the libraries are mainly data structures that do not make great use of the power of classes: they have member functions but not much private data. Most of them are container classes such as stacks and queues: classes that can contain objects of other classes. At this point, early in the development of reusable code, there is a

great gap between the general purpose data processing classes that are supplied with object-oriented languages and application specific classes written by users. It will be interesting to see how and whether the gap can be filled.

For this project I used the linked list implementation from the class library. In order to maintain generality the List clearly cannot hold data of a particular type. Borland avoids this by deriving both the containers-as-objects and the objects they store from the Object class. Containers can then store references or pointers to Object. The Object class has a set of pure virtual functions: functions that must be instantiate by the user in the inheriting class. One of these functions, isEqual() was written in Being as:

```
int isEqual( const Object& testBeing ) const
{
   return (serialNumber==((Being&)testBeing).serialNumber);
}
```

List can use this function to find a particular being, or other object, without ever knowing the details of its structure or even what it means to be equal.

4. CONCLUSIONS

This project offers an interesting example of one mechanism by which the evidence of rapid evolution found in the fossil record can be explained. The use of a discreet simulation makes it possible to analyze the conditions under which this mechanism, invasion by a small cohesive colony, can succeed. The simulation also provides an excellent subject for an investigation of object oriented design in general and the difference between classes shaped by inheritance and classes shaped by the problem space in specific.

B I B L O G R P H Y

Ayala, Francisco, Population and Evolutionary Genetics - A Primer, Benjamine/Cummings Publishing, Menlo Park, CA, 1982

Crookes, J, "Simulations Using C" in Computer Modeling for Discrete Simulation, John Wiley & Sons, New York, 1989

Kimura, Motoo, Theoretical Aspects of Population Genetics, Princeton University Press, New Jersey, 1971

Ladd, Scott, Turbo C++ Techniques and Applications, M&T Books, Redwood CA, 1990

Lafore, Robert, Object-Oriented Programming in Turbo C++, Waite Group Press, Emoryville, CA 1991

Lounamaa, Pertti "An Incremental Object-oriented Language for Continuous Simulation Models" in Artificial Intelligence, Simulation & Modeling Wiley Interscience, New York, 1989

Pidd, Michael "Developments in Discreet Simulation" in Computer Modeling for Discrete Simulation, John Wiley & Sons, New York, 1989

Spiess, Eliot B. Genes in Population, John Wiley & Sons, New York, 1977

Templeton, Alan R. "Adaptation and the Integration of Evolutionary Forces", in Perspectives on Evolution, Sinauer Associates, Sunderland MA, 1982

Wallace, Bruce, Basic Population Genetics, Columbia University Press, New York, 1981.

Winblad, Ann, Object-Oriented Software, Adison-Weley, Reading MA. 1990

APPENDIX A
Functional code

```
/*************************************************************
*                                                            *
*  FILE:            G S _ M A I N . C                        *
*                                                            *
*        main module for the genetic simulation project.     *
*                                                            *
*        Input:    argv[1] the name of the setup file        *
*                                                            *
*************************************************************/


include <stdio.h>
include <stdlib.h>
include <malloc.h>

include "gs_const.h"      /* global constants and types
include "gs_prots.h"      /* module for prototypes */

    /*************************************************
                          M A I N
            entry point for genetic simulation

       Input: argv[1]      name of the external parameter file
    *************************************************/
int main( int argc, char * argv[] )
{
 int gen;         /* the current generation */
 int numgen;      /* the last generation to be run */

    setup( &numgen, argv[1]); /* bring in run parameters from the text file */
      /* create dynamic array of individuals */
    if ((ind = (Ind *) malloc( (long) SIDE * SIDE * sizeof( Ind )))==NULL)
    {
        printf(" insufficient memory ");
        exit(100);
    }
    init_vid();          /* set video mode and palette */
    srand(RND);          /* start the pseudo random series at RND */
    format_display();    /* setup the display screen */
    initialize( );        /* create initial individuals */
    display_status( 0 ); /* print the frequencies of initial individuals */
      /* main loop */
    for ( gen = 1; gen < numgen; gen++ )
    {
        do_generation( gen ); /* put each individual through the annual cycle
*/
        display_status( gen ); /* print the frequencies in this generation  */
    }
    getchar();              /* leave display on screen */
    exit_vid();             /* reset video mode */
}
```

Code for John Bloom's report

```
/*************************************************************
*                                                            *
*  FILE:            G S _ B K E E P . C                      *
*                                                            *
*                                                            *
*************************************************************/

#include <stdio.h>
#include <graph.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "gs_const.h"
#include "gs_prots.h"

struct videoconfig vc;

static int set_count [MAX_SETS];          /* bookkeeping variables */
static int allele_count [MAX_SETS];
static int locus [MAX_SETS] [8];
static int size = 0;
static int f_stat = 0;

extern int graphbottom;              /* graph position parameters */
extern int graphleft;

extern int left ;                      /* text position parameters  */
extern int top  ;

FILE*    outfile;              /* external parameter file */


    /*************************************************************
                    U P D A T E    S T A T U S
        change the status to account for a birth or death

            Input:  pos     position of changed individual
                    count   count =  1 for birth, -1 for death
    *************************************************************/
void update_status( Pos pos, char count )
{
 int i, j;             /* loop control */
 Genotype gn;              /* space saver  */

      /* keep track of the total number of individuals */
    size += count;
      /* keep track of the number of individuals in each set */
    set_count[ind[D2(pos)].set] += count;
      /* put the genotype in the space holder */
    genecpy( gn, ind[D2(pos)].genotype );
      /* keep track of the number of each allele in the population */
    for (j = 0; j < GN_LEN; j++)
        allele_count[ gn[j].al-1 ] += count;

    for (i = 0; i < LOCI; i++)  */ count the alleles at each locus
    {                                that have a common origin */
        if (gn[i*2].origin == gn[i*2+1].origin)  f_stat += count;
          /* update the number of alleles at each locus  */
        for (j = 0; j < STRANDS; j++)
```

```
        {
            k = gn[i*2+j].al-1;
            locus[i][k] += count;
}     }     }


     /*************************************************************
                        D I S P L A Y   S T A T U S
            print the frequencies of each of the sets and each of
            the allele types to the screen

            Input:   gen      the current generation
     *************************************************************/
void display_status( int gen )
{
 int     i, j;                  /* loop counter */
 char    buffer[20];            /* text for output */
 static  int column = 0;        /* graphics column */
 int     decimal;               /* number of decimal points to display */
 int     point;                 /* graphic position holder */
 char*   gens = "th generation";

     /* print generation */
    gcvt( (double)gen, 4, buffer);
    _settextposition( top-1 , left - 15);
    _outtext( strcat( buffer, gens ));

      /* print the frequency of each set */
    decimal = 3;
    for (i = 0; i < SETS; i++)
    {
        strcpy( buffer, format( (float) set_count[i] / size, decimal ));
        _settextposition( top + i, left - 20);
        _outtext( buffer );
    }

      /* print the frequency of each allele */
    decimal = 3;
    for ( i = 0; i < ALLELES; i++ )
     {
        strcpy( buffer, format( (float) allele_count[i] /
            GN_LEN / size, decimal ));
        _settextposition( top + i + SETS + 2, left - 20);
        _outtext( buffer );
     }

      /* print the frequency of loci with alleles identical by decent */
    decimal = 4;
    strcpy( buffer, format( (float) f_stat / size, decimal ));
    _settextposition( top + SETS + ALLELES + 3, left - 20);
    _outtext( buffer );

        /* plot the graph */
    column ++;
    for ( i = 0; i < SETS; i++)      /* for each allele */
    {
          /* add a dot to the graph of frequency */
        _setcolor( i + 1 );
        point =  graphbottom - (int)((float)set_count[i] / size * 100 );
```

```
        _setpixel(graphleft + 20 + column, point+1);
        _setpixel(graphleft + 20 + column, point);
        _setpixel(graphleft + 21 + column, point+1);
        _setpixel(graphleft + 21 + column, point);
    }
}

    /*************************************************************
    *                                                            *
    *  FILE:            G S _ C O M P . C                        *
    *                                                            *
    *                                                            *
    *************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "gs_const.h"
#include "gs_prots.h"


    /*************************************************************
                          C O M P E T E
          new individual tries to replace an existing individual

               Input:  parent   position of the parent individual
                       newgene  genotype of the new individual
               Return :   position of the defeated individual
                          or NULL is offspring fails
    *************************************************************/
Pos compete  (Pos parent, Genotype newgene, int gen )
{
 int  total_tries  = 0;          /* number of tries before failure */
 Pos  existing;                  /* position of an existing individual */
 int  tries;                     /* number of tries (including out of
                                          bounds) used to find the next space */
 Pos FAILURE = { -1, -1 };          /* flag for search failure */

    while (total_tries < 2 )      /* try 2 times */
    {       /* get the next position in the search grid */
        existing = next_space( parent, &tries, SEARCH );
        total_tries += tries;     /* keep track of the total number of tries */

              /* do not compete with parent */
        if ((existing.x != parent.x || existing.y != parent.y )
              /* new individual is fitter than existing */
            &&(fitness[get_set(newgene)] /
               (fitness[ind[D2(existing)].set] + fitness[get_set(newgene)])
               > get_fract() ))
         {
          return (existing);   /* existing individual will be replaced */
    }   }
    return (FAILURE);         /* flag for defeat */
}
```

APPENDIX A
Functional code

```
/*************************************************************
*                                                            *
*  FILE:            G S _ G E N E . C                        *
*                                                            *
*                                                            *
*************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include "gs_const.h"
#include "gs_prots.h"


    /*************************************************************
                        R E P R O D U C E
        create a new genotype using the two parents genotypes

        Input:      gn1      genotype of father
                    gn2,     genotype of mother
                    newgene  genotype of offspring
    *************************************************************/
void reproduce( Genotype gn1, Genotype gn2,  Genotype newgene )
{
 int i, j;

    for (i = 0; i < LOCI; i++)       /* for each locus */
    {
        j = i * STRANDS;
        newgene[j]   = gn1[getnum(0,1)+j]; /* one of the father's alleles */
        newgene[1+j] = gn2[getnum(0,1)+j]; /* one of the mother's alleles */
    }
}

    /*************************************************************
                        C O P Y   G E N E
                        copy gn2 to gn1

            Input:  gn1    gene copied to
                    gn2    gene copied from
    *************************************************************/
void genecpy( Genotype gn1,  Genotype gn2 )
{
 char i;
     for (i = 0; i < GN_LEN; i++) gn1[i] = gn2[i];
}


    /*************************************************************
                        O R D E R  G E N E
      arrange each locus with alleles in increasing order

            Input:      gn  genotype to be ordered
    *************************************************************/
void order_gene(Genotype  gn)
{
 int  i, j;
 Gene swap;
```

```
    for (i = 0; i < LOCI; i++)        /* for each locus */
    {
        j = i * STRANDS;
        if(gn[j].al > gn[j+1].al)     /* if the lower allele is second */
        {
            swap = gn[j];
            gn[j] = gn[j+1];
            gn[j+1] = swap;
}   }   }

    /*************************************************************
                            E Q U A L
      return true if two gene arrays are equal, false if not

            Input:  gn genotype to be checked
                    template  genotype to check against
    *************************************************************/
int equal( Genotype gn, Genotype template)
{
 char i;

    for (i = 0; i < GN_LEN; i++)
    {
        if (gn[i].al != template[i].al - '0' &&  template[i].al != '*')
        return( FALSE );
    }
    return (TRUE);
}

    /*************************************************************
                    I N I T   G E N E
     create genotypes with a monte carlo process using the setup
     probabilities for alleles
    *************************************************************/
void init_gene(Genotype gene)
{
 int i, j;
 float sum, mark;
 static unsigned int origin = 0;

    for ( i = 0; i < GN_LEN; i++)             /* for each allele */
    {
        mark = rand() / (float) (RAND_MAX+1);      /*  0 <= mark < 1  */
        j = 0;
        sum = 0;
        while (mark >= sum)
        {
            sum += ALL_INIT[j++];              /* sum the probabilities   */
        }
    gene[i].al = j;             /* allele is type j */
    if (j==0) getch();
    gene[i].origin = ++origin;  /* set the origin */
    }
}
```

APPENDIX A
Functional code

```
/*************************************************************
*                                                            *
*  FILE:            G S _ D R I V E R . C                    *
*                                                            *
*         contains the main loops for the program            *
*                                                            *
*                                                            *
*************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "gs_const.h"
#include "gs_prots.h"

/*************************************************************
                    D O   G E N E R A T I O N
  put every individual in the map through its annual cycle

          Input:   gen      number of the present generation
*************************************************************/
void do_generation( int gen  )
{
 Pos active;    /* the position of the active individual */
 char c;        /* key board input */
 int start = 1;

   /* process the large region */
 for (active.x = start; active.x <= SIDE1; active.x++)/* for each column */
 {   /* for each row */
     for (active.y = start; active.y <= SIDE1; active.y++)
     {
         do_individual( active, gen );  /* put individual through cycle */
         if ( kbhit() )      /* check for quit signal */
             if ((c = getch()) == 'q') exit (3);
     }
 }
   /* process the small region */
 for (active.x = SIDE1 - SIDE2 + 1; active.x <= SIDE1; active.x++)
 {
     for (active.y = SIDE1 + 1; active.y <= SIDE; active.y++)
         do_individual( active, gen );
}    }

/*************************************************************
                    D O   I N D I V I D U A L
          put an individual through its annual cycle

          Input:  active     position of the active individual
                  gen        current generation
*************************************************************/
void do_individual( Pos active, int gen )
{
 Pos*        mate,          /* pointer to the position of the mate */
             new;           /* position invaded by the offspring */
 Genotype     newgene;      /* genotype of offspring */

 debug(active, 9);    /* for demo runs */
```

```
      /* get mate if available */
    if ( ( mate = find_mate( active )) != NULL
          /* active individual is at least one generation old */
        && ind[D2(active)].generation <= gen  )
    {
        debug(*mate, 10);   /* for demo runs */
          /* get the genotype of the offspring */
        reproduce(ind[D2(active)].genotype,
                  ind[D2(*mate)].genotype, newgene );
          /* offspring competes successfully */
        if ( in_bounds( new = compete( active, newgene, gen )))
        {
            debug(new, 11);   /* for demo runs */
            die(new);                    /* eliminate the individual at new */
            make( new, newgene, gen ); /* make a new individual */
}     }   }

      /*************************************************************
      *                                                            *
      *  FILE:            G S _ I N B O U . C                      *
      *                                                            *
      *                                                            *
      *************************************************************/

#include <stdio.h>
#include "gs_const.h"


    /*************************************************************
                        I N   B O U N D S

            Input:      pos   the position to be tested
            Return:     boolean flag for good position
    *************************************************************/
int in_bounds( Pos pos )
{
    if (
            (   pos.x >  0
                && pos.x <= SIDE1          /* pos is in the large area */
                && pos.y >  0
                && pos.y <= SIDE1
            )
         ||                                  /* or */
            (   pos.x >  SIDE1 - SIDE2
                && pos.x <= SIDE1
                && pos.y >  SIDE1          /* pos is in the small area */
                && pos.y <= SIDE
            )
        )  return (TRUE);

    return (FALSE);
}
```

APPENDIX A
Functional code

```
/*************************************************************
*                                                            *
*  FILE:            G S _ I N D I V . C                      *
*                                                            *
*   contains modules that modify values for an individual    *
*                                                            *
*************************************************************/

#include <stdio.h>
#include "gs_const.h"
#include "gs_prots.h"
#define _GFILLINTERIOR  3          /* graph.h not included in this file */
#define ADD              1
#define SUBTRACT        -1

     /*********************************************************
                            D I E
                    eliminate an individual

             Input:  out   position of dead individual
     *********************************************************/
void die(Pos out )
{
     if (ind[D2(out)].number > 0 )    /* individual is alive */
     {
         update_status( out, SUBTRACT );  /* do bookkeeping */
         ind[D2(out)].number = -1;       /* mark as dead */
}    }

     /*********************************************************
                            M A K E
                    create a new individual

             Input:  pos     position of new individual
                     gene    genotype of new individual
                     gen     current generation
     *********************************************************/
void make( Pos pos, Genotype gene,  int gen  )
{
 Ind  new;        /*  holder for parameters */
 static long number = 0;       /* unique number for individual */

       /* define the gene, number and generation of the new individual */
     genecpy(new.genotype, gene);
     new.number = number++;
     new.generation = gen;
     new.set = get_set(new.genotype);

     ind[D2(pos)] = new;          /* put individual in the position array */
     update_status(pos, ADD);     /* put individual in the model */
     display_ind(pos, new.set, _GFILLINTERIOR);  /* display individual */
}
     /*********************************************************
                              D 2
         return the offset to the individual specified by pos
     *********************************************************/
int D2 ( Pos pos )
{
     return ( (pos.x-1) * SIDE + pos.y-1 );}
```

APPENDIX A
Functional code

```
       /*************************************************************
       *                                                            *
       *  FILE:            G S _ I N I T . C                        *
       *                                                            *
       *         contains the map initialization routines           *
       *                                                            *
       *                                                            *
       *************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include "gs_const.h"      /* global constants and types */
#include "gs_prots.h"      /* module for prototypes */

     /*************************************************************
                        I N T I A L I Z E
                  fill the map with individuals
     *************************************************************/
void initialize()
{
 Pos        pos;                /* structure holding x, y position */
 int        generation = 0;     /* number of the current generation */
 Genotype   gn;                 /* structure holding an individual's alleles */


       /* fill the main area with individuals */
     for (pos.x = 1; pos.x <= SIDE1; pos.x++)  /* for each column */
     {
         for (pos.y = 1; pos.y <= SIDE1; pos.y++) /* for each row */
         {
             init_gene(gn);   /* create genes using a stochastic process */
               /* link position, gene and generation to make an individual */
             make(pos, gn, generation);
         }
     }
       /* fill the smaller area with individuals */
     ALL_INIT[1] = 1;     /* genotype contains only type 2 alleles */
     ALL_INIT[0] = 0;
     for (pos.x = SIDE1 - SIDE2 + 1; pos.x <= SIDE1; pos.x++)
     {
         for (pos.y = SIDE1 + 1; pos.y <= SIDE; pos.y++)
         {
             init_gene(gn);
             make(pos, gn, generation);
}  }  }

       /*************************************************************
       *                                                            *
       *  FILE:            G S _ M A T E . C                        *
       *                                                            *
       *                                                            *
       *************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "gs_const.h"
#include "gs_prots.h"
```

```
    /*************************************************************
                              F I N D   M A T E
        search the grid around the active individual for a mate
        until a mate is found or all the positions have been checked

        Input:  active    the position of the active individual

        Return:           pointer to the position of the mate
                              or NULL is no mate is found

    *************************************************************/
Pos* find_mate( Pos active  )
{
 int   search_area;            /* area of the search grid */
 static Pos   mate;            /* the position of the mate */
 int   tries;            /* the number of positions checked by next_space */
 int   total_tries = 0;   /* total number of the tries */

      /* get the number of spaces in the search grid */
    search_area = pow((2 * SEARCH + 1), 2);
      /* while there are unchecked positions */
    while (total_tries < search_area)
    {
         /* get the next individual to check */
        mate = next_space(active, &tries, SEARCH);
        total_tries += tries;
            /* mate is not the active individual  */
        if (( ind[D2(active)].number != ind[D2(mate)].number)
           /* mate is the same generation */
        && (ind[D2(mate)].generation == ind[D2(mate)].generation))

        return (&mate);  /* if this mate is usable, return its address */
    }
    return(NULL);  /* no mate was found in the search grid */
}


      /*************************************************************
      *                                                            *
      *  FILE:            G S _ N S P A C E . C                    *
      *                                                            *
      *                                                            *
      *************************************************************/


#include <stdio.h>
#include "gs_const.h"
#include "gs_prots.h"
```

```
     /*************************************************************
                          N E X T   S P A C E
          return the next position found in the search grid. Positions
          are found by indexing the offsets from the active position

          Input: active,   the position of the active individual
                   tries,    attempts to find a position that is in bounds
                   depth     the greatest search distance from active
     *************************************************************/
Pos next_space( Pos active, int* tries, char depth )
{
 int Xoffset, Yoffset;     /* offsets from active */
 Pos new;          /* new position defined by offsets and active */


     *tries = 0;

     Xoffset = getnum( -depth, depth);    /* get random search start position */
     Yoffset = getnum( -depth, depth);


     do
     {
          Xoffset ++;              /* index the X offset */
          if (Xoffset > depth)     /* if the X offset is outside the grid */
          {
              Xoffset = -depth;     /* reset the X offset to the other side */
              Yoffset ++;           /*  and index the Y offset */
              if (Yoffset > depth) /* if the Y offset is below the search grid
*/
              Yoffset = -depth;     /* reset the Y offset to the top of the grid
*/
          }

          new.x = Xoffset + active.x;     /* define the new position */
          new.y = Yoffset + active.y;
     }
     while ( ! in_bounds( new ));  /* loop until the position is on the map */

     (*tries)++;                   /* increment tries */
     return( new );                 /* return the new position */
}


     /*************************************************************
     *                                                            *
     *  FILE: gs_palet .c                                         *
     *                                                            *
     *        module for setting screen colors                    *
     *                                                            *
     *************************************************************/
#include <graph.h>
#include "gs_const.h"
#include "gs_prots.h"

#define GRID(x)  ((x) - 1) * box + border

extern int box;
extern int border;
```

APPENDIX A
Functional code

```
    /*************************************************************
                                S E T P A L E T T E
                              initialize display colors
    *************************************************************/
void setpalette()
{

                                 /* blue  green  red   */
    _remappalette (0, cllr(   0,    0,    0  ));     /* background color    */

    _remappalette (1, cllr(  22,   06,   02  ));     /* active colors       */
    _remappalette (2, cllr(  45,    0,    0  ));
    _remappalette (3, cllr(  35,   40,   25  ));
    _remappalette (4, cllr(  45,   20,   25  ));
    _remappalette (5, cllr(  35,    0,   35  ));
    _remappalette (6, cllr(  35,   35,   55  ));
    _remappalette (7, cllr(   0,   30,   63  ));
    _remappalette (8, cllr(   5,   60,   60  ));
    _remappalette (9, cllr(   0,    0,   60  ));

    _remappalette (10, cllr(   60,    60,    0  ));  /* debug colors */
    _remappalette (11, cllr(   60,     0,   60  ));
    _remappalette (12, cllr(    0,    60,   60  ));
    _remappalette (13, cllr(    0,     0,   60  ));

    _remappalette (14, cllr( 40,   40,   40  ));     /* frame color         */
    _remappalette (15, cllr( 50,   50,   50  ));     /* text color          */
}

    /*************************************************************
                           D I S P L A Y  I N D
            display individual on screen at position pos

            Input: pos        position of individual
                      color      individual's set
                      fill       flag for fill or border
    *************************************************************/
void display_ind( Pos pos, char color, char fill )
{
    _setcolor(color +1);
    _rectangle(fill,  GRID(pos.x), GRID(pos.y),
                          GRID(pos.x) + box  - 2, GRID(pos.y) + box   - 2);

}

      /*************************************************************
      *                                                            *
      *  FILE:            G S _ S E T . C                          *
      *                                                            *
      *                                                            *
      *************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "gs_const.h"
#include "gs_prots.h"
```

```
     /*************************************************************
                              G E T   S E T
                    return the set of the input genotype

               Input:      gn   genotype to check
               Uses:       geno  array of possible types
     *************************************************************/
int get_set(Genotype gn)
{
 char i;

     for (i = 0; i < SETS; i++)
          if ( equal( gn, geno[i] )) return (i);

     printf(" set types are incomplete ");    /* error trap */
     exit(255)
}



     /*************************************************************
     *                                                            *
     *  FILE: gs_setup .c                                         *
     *                                                            *
     *        module includes routines for reading run            *
     *        parameters from an external file                    *
     *                                                            *
     *************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include "gs_const.h"      /* global constants and types */
#include "gs_prots.h"      /* module for prototypes */

FILE    *infile;           /* the external file */


      /*************************************************************
                                   S E T   U P
        bring in parameters for this run from external text file

          Input:   numgen    pointer to the number of the last generation
                   filename  the literal name of the external file
       *************************************************************/
void setup( int* numgen ,char* filename)
{
 int     i, j;             /* loop variables */
 char    text[80];         /* raw text from the external file */

             /* open the setup file */
     if ((infile = fopen( filename, "rt" )) == NULL)
     {
          printf("unable to open %s\n", filename);
          exit(-1);
     }

       /* get the global parameters from the external file */
     SIDE1 = (int)get_next_param(   );          /* side of the larger area */
```

```
    SIDE2 = (int) get_next_param(  );        /* side of the larger area */
        SIDE = SIDE1 + SIDE2;
    ALLELES = (int) get_next_param(  );      /* number of allele types */
    LOCI = (int) get_next_param(  );         /* number of loci */
        GN_LEN =   LOCI * STRANDS ;      /* number of alleles per individual
*/
    SETS =   get_next_param(  );

    for (i = 0; i < SETS; i++)
    {
        fitness[i] =  get_next_param(  );   /* fitness for each set */
        fscanf( infile, "%s", text );
        for ( j = 0; j < GN_LEN; j++)
             geno[i][j].al =  text[j];
    }

    for (i = 0; i < ALLELES ; i++)
        ALL_INIT[i] = get_next_param(  );   /* initial frequency of alleles */

    RND = (int) get_next_param(  );      /* seed for pseudo random generator */
    SEARCH = (int) get_next_param(  );       /* search distance for mates */

    *numgen =  (int) get_next_param(  );  /* number of generations in run */

    if ((i = (int) get_next_param()) != -999) /* -999 is a flag for success */
    {
        printf("too many parameters in Setup ");
        exit (-3);
    }
    fclose( infile );
}


     /*************************************************************
                    G E T   N E X T   P A R A M
            return the float value of the string following
            the next colon in the parameter file
      *************************************************************/
float get_next_param(void)
{
 char    text[10];    /* raw text from the external file */
 char    c;           /* input character */

    while ( (c = fgetc(infile)) != ':'   /* read to the next colon */
           && c != EOF);
    if ( c == EOF )
    {
        printf("input error:  to few values");
        exit(-2);
    }
     fscanf( infile, "%s", text );          /* read in text value */
     return (atof( text ));                 /* convert to float and return */
}
```

```
   /*************************************************************
   *                                                            *
   *  FILE:            G S _ U T I L . C                        *
   *                                                            *
   *                contains utility functions                  *
   *                                                            *
   *                                                            *
   *************************************************************/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "gs_const.h"
#include "gs_prots.h"



   /*************************************************************
                        G E T  N U M
     returns a random integer between (and including ) high and low

              Input:  low      lower bound
                      high     upper bound
              Returns:         random integer
   *************************************************************/
int getnum( int low, int high )
{
  return ( floor( get_fract() * (high - low + 1) + low));
}



     /*************************************************************
                        G E T  F R A C T
     returns a random number between (but not including) 0 and 1

              Input:      none
              Returns:    random float
     *************************************************************/
float get_fract(void)
{
  return (rand() / (float)(RAND_MAX+1 ));
}

   /*************************************************************
   *                                                            *
   *  FILE: gs_video .c                                         *
   *                                                            *
   *        module includes all video routines                  *
   *                                                            *
   *************************************************************/

#include <stdio.h>
#include <graph.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "gs_const.h"
```

```
#include "gs_prots.h"

#define GRID(x)  ((x) - 1) * box + border  /* translate map position to
                                              pixel position */

struct videoconfig vc;

int box ;                          /* side length of individual square   */
int border = 10;                   /* width of outline                    */
int edge = 0;                     /* graph border                        */

int graphbottom = 430;             /* graph position parameters           */
int graphdepth  = 100;
int graphleft   = 130;


int left = 70;                     /* text position parameters            */
int top  = 3;

int Tvrt, Thor;                    /* text to pixel conversions           */



   /*************************************************************
                          D E B U G
         marks the active grid position with a colored boarder,
         waits for a keystroke, then erases the mark.

         Input:  pos      position to be marked
                 color    color of the boarder

   *************************************************************/
void debug(Pos pos, short color)
{

     _setcolor(color +1);
        /* print a three pixel wide border at pos */
     _rectangle(_GBORDER,  GRID(pos.x), GRID(pos.y),
                             GRID(pos.x) + box  - 2, GRID(pos.y) + box  - 2);

     _rectangle(_GBORDER,  GRID(pos.x)+1, GRID(pos.y)+1,
                             GRID(pos.x) + box  - 3, GRID(pos.y) + box  - 3);

     _rectangle(_GBORDER,  GRID(pos.x)+2, GRID(pos.y)+2,
                             GRID(pos.x) + box  - 4, GRID(pos.y) + box  - 4);

      if (getch() == 'q' ) exit(1);  /* pause */
        /* erase border */
      display_ind(pos, ind[D2(pos)].set, _GFILLINTERIOR );
}
```

```
   /*************************************************************
                          I N I T   V I D
                       configure video system
   *************************************************************/
void init_vid()
{

   _clearscreen(_GCLEARSCREEN);

   if(!_setvideomode(_VRES16COLOR))      /* initialize video mode */
   {
      _setvideomode(_DEFAULTMODE);       /* exit program if VGA   */
      printf("VGA not present\n");       /*  is not available     */
      getch();
      exit(-1);
   }
   _getvideoconfig( &vc);                /* get video configuration */

      /* define pixel to text conversion */
   Tvrt = vc.numypixels / vc.numtextrows;
   Thor = vc.numxpixels / vc.numtextcols;


     box = 300 / (SIDE1+SIDE2);     /* set the size (in pixels) of a position */

     setpalette();                  /* initialize display colors */

}

   /*************************************************************
                              C L L R
            use color intensities to build the VGA code
   *************************************************************/
long cllr(int blue, int green, int red)
{
   return (blue * 0x10000 + green * 0x100 + red);
}



   /*************************************************************
                        D O   L E G E N D
                  display set symbols and colors
   *************************************************************/
void do_legend (void)
{
 int        i, j ;           /* loop control variables */
 char       eq[] = " = ";      /* equals sign */
 char       buffer[6];         /* array far output */

     for (i = 0; i < SETS; i++)
   {
         /* place set color */
      _setcolor(i + 1);
      _rectangle(_GFILLINTERIOR,
                 left * Thor, Tvrt * (i + top - 1),
                 left * Thor + Tvrt, Tvrt * (i + top));

       /* output set symbol */
```

```
    _settextposition(top + i, left - 2 - GN_LEN);

    for ( j = 0; j < GN_LEN; j++ )
        _outtext( &geno[i][j].al );   /* print the set symbol */
     _outtext( eq );
    _settextposition( top + i, left + 6 );
    _outtext( gcvt( fitness[i], 3, buffer ));   /* print the set fitness */

    buffer[1] = 0;
    for ( j = 0; j < ALLELES; j++ )
    {
        _settextposition( top + j +  SETS + 2, left - 10);
        _outtext(" type ");
        buffer[0] =  '1' + j;
        _outtext( buffer );
        _outtext(" alleles");
    }
    _settextposition( top + 3 + SETS + ALLELES, left - 10);
    _outtext(" identical decent ");

}  }

  /*************************************************************
                  F O R M A T   D I S P L A Y
               create color frames on the screen
  *************************************************************/
void format_display()
{
 int i;

    _setcolor(14);   /* fill main area */
    _rectangle(_GFILLINTERIOR, 0, 0, SIDE1 * box + 2 * border,
                               SIDE1 * box + 2 * border);
    _setcolor(14);   /* fill small area */
    _rectangle(_GFILLINTERIOR, (SIDE1 - SIDE2) * box,
                               SIDE1 * box + 2 * border,
                               SIDE1 * box + 2 * border,
                               (SIDE1+SIDE2) * box + 2 * border);

    _setcolor(0);    /* fill main background */
    _rectangle(_GFILLINTERIOR, border, border, (SIDE1) * box + border,
                               (SIDE1) * box + border);

    _setcolor(0);    /* fill small background */
    _rectangle(_GFILLINTERIOR, (SIDE1 - SIDE2) * box + border,
                               SIDE1 * box ,
                               SIDE1 * box + border,
                               (SIDE1+SIDE2) * box +  border);

    _setcolor(14);   /* fill graph background */
    _rectangle(_GFILLINTERIOR, graphleft - edge,
                                    graphbottom - graphdepth - edge,
                                    vc.numxpixels - 1,
                                    graphbottom + edge );
    do_legend();    /* display gene set symbols and colors */
}
```

APPENDIX A
Functional code

```
    /*************************************************************
                          E X I T   V I D
                      reset video mode for exit
   *************************************************************/
void exit_vid()
{
   _setvideomode(_DEFAULTMODE);
}

    /*************************************************************
                              F O R M A T
    return a string formatted to 'decimal' decimal places avoiding
    the scientific notation returned by gcvt()

         Input: num      number to be converted   0 < num <= 1
                decimal  number of places to the right of the decimal point

         Return:          formatted string
    *************************************************************/
char* format(float num, int decimal )
{
 static char buffer[20];    /* return string */

        /* put 'decimal' number of places to the left of the decimal point */
      num = (num * pow(10, decimal)) + pow(10, decimal);
        /* convert to a string */
      gcvt( (float) num , decimal + 2, buffer);
        /* put decimal point at beginning of string */
      buffer[0] = '.';
        /* check for num = 1 */
      if (num == 2 * pow(10, decimal) ) strcpy (buffer, "1.00 " );
        /* make buffer a string */
      buffer[decimal+1] = 0;

      return( buffer );
}
```

APPENDIX B

Object-oriented code

```
//
//                              M A I N . C P P
//
//
//
//


#include <list.h>
#include "global.h"
#include "being.h"
#include "map.h"
#include "video.h"
#include "grf_clss.h"
#include <stdio.h>
#include <conio.h>  // for kbhit()



List popu;
Window* clss[3];

main()
{
 void doCycle(  Object& obj, void * );
 Map map;
 Video V;
 Pos pos;

     global::readIn();     // bring in parameters
     V.initVid();          // initialize video

     while ((pos.x != SIDE1) || (pos.y != SIDE) )  // initialize the map
     {
         pos = map.nextSquare();                    // get the next square
         Being* bptr = new Being( pos );           // create a being
         bptr->make();                              // enter the being
                                                    //into the model
     }
     for (int gen = 1; gen <= GEN; gen++ )   // for each generation
     {
         popu.forEach( doCycle, &gen );      // put each being through
                                           //its annual cycle
         for( int i = LEDG; i <= GRPH; i++ )
             clss[i]->update( Being::getNumInSet() );
     }
     getch();
}

//''''''''''''''''''''''''''''''''''''''''''''
//           D O   C Y C L E
//
//   put a being through its annual cycle
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void doCycle(Object& obj, void* gen)
{
   Being& b = (Being&) obj;        // cast obj as a Being b
      b.cycle( *(int*)gen );        // put b through cycle
      if (kbhit()) exit(1);         // exit on key press
}
```

APPENDIX B
Object-oriented code

```
//
//
//                          B E I N G . H
//
//
//


#if !defined __BEING_H
#define  __BEING_H

#define beingClass __firstUserClass

#include "gene.h"
#include <object.h>

typedef struct Pos
{
     int x;
     int y;
};


class Being : public Object, public Gene
{
     private:

          Pos pos;
          int generation;
          long serialNumber;              // unique number for each Being

          static long Number;             // number of living Beings
          static long GserialNumber;      // total number of Beings created
          static int numInSet[10];        // number of being in each set


     public:


            // default constructor
          Being(void);

            //initial constructor
          Being( Pos pos ): Gene( "init" )  // "init" is a flag for Gene
          {                                 //initializing constructor
              setUp( pos );
              generation = 0;
          }
            // make constructor
          Being( Pos pos, int gen, Gene  gn ): Gene(gn)
          {
            setUp( pos );
            generation = gen;
          }

            // destructor
          ~Being(){ Number--;}

          void    make( void );
          void    die( void );
          void    setUp( Pos newpos );
          void    cycle( int gen );
```

```
        int      getNumber(void){ return serialNumber; }
        void     show( void );
        Pos      getPos( void ){ return pos; }

        static int* getNumInSet() { return numInSet; }
        static int getSize() { return Number; }


          // functions required for classLib

        void Being::afunc(void);
        classType isA() const { return beingClass; }
        char* nameOf() const { return "being"; }
        void printOn( ostream& outputStream )const {} ;
        hashValueType hashValue() const
            { return (hashValueType) Number; }
        int isEqual( const Object& testBeing ) const
            { return (serialNumber == ((Being&) testBeing).serialNumber);}
};


#endif

//
//
//                              B E I N G . C P P
//
//
//


#include "being.h"
#include "map.h"
#include "video.h"
#include "global.h"
#include "grf_clss.h"
#include <list.h>

long Being::Number = 0;              // static variable declared in the
long Being::GserialNumber = 0;       //header file must be redeclared
int Being::numInSet[];               //here
extern List popu;                  // the linked list from classLib
extern Window* clss[];             // array of screen windows


//''''''''''''''''''''''''''''''''''''''''
//              S E T U P
//
// utility function for Being constructors
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Being::setUp( Pos newpos )
{
       pos.x = newpos.x;                  // set the position
       pos.y = newpos.y;

       GserialNumber++;                   // increment the global serial number
       serialNumber = GserialNumber;      // set the being's serial number
}
```

APPENDIX B

Object-oriented code

```
//''''''''''''''''''''''''''''''''''''''''
//              C Y C L E
//
//      annual cycle for a being
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Being::cycle(int gen)
{
 Being* mate;
 Gene    newgene;
 Being* existing;
 float exfit, newfit;

     mate = Map::search( pos );                   // find a mate
     newgene = reproduce( mate->getGeno() );      // create a new gene
     newfit = FITNESS[newgene.getSet()];          // get its fitness
     existing = Map::search( pos );               // find a being to compete with
     exfit = FITNESS[existing->getSet()];         // get its fitnesses

     if ( (( newfit / ( newfit+ exfit ))           // newgene is successful
             > global::getFract())                 //and it is not competing
       && (existing->getNumber() != getNumber())        //with itself
       && (existing->getNumber() != mate->getNumber()))//or its mate
     {
         Pos exPos = existing->getPos();            // destroy the looser  and
         existing->die();                           // save its position

         Being* bptr = new Being( exPos, gen, newgene );// put the new being
         bptr->make();                                  //into the model at
}    }                                                  //that position

//''''''''''''''''''''''''''''''''''''''''
//                 M A K E
//
//        establish the new being
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Being::make(void) // create a new being
{
 Video V;

     Number++;                          // increment the total number of beings
     numInSet[ getSet() ]++ ;              // increment the per set counter
     Map::insert( getPos(), this );        // put it in the map array
     popu.add (*this);                     // put it in the list
     int nums[3];
     nums[0] = pos.x;
     nums[1] = pos.y;
     nums[2] = getSet();
     clss[VMAP]->update( nums );           // display it on the screen
}
```

APPENDIX B

Object-oriented code

```
//''''''''''''''''''''''''''''''''''''''''''
//              D I E
//
//          destroy a being
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Being::die( void )
{
        numInSet[ getSet() ] --;    // decrement the per set counter
        popu.destroy( *this ) ;     // remove being from list and
                                   //free its heap space
}



//
//
//                              G E N E . H
//
//

#if !defined __GENE_H
#define __GENE_H

#include <string.h>

typedef char Genotype[5];

class Gene
{
     private:

         Genotype geno;
         int set;

     public:
        Gene(void){};
          // initialize constructor
        Gene( char* init ) { initGeno( ); }
          // copy constructor
        Gene( Gene& gn ) { *this = gn; }

        char* getGeno(void ) { return geno;}
        int   getSet(void) { return set; }
        void  setSet(void);
        Gene  reproduce(  Genotype );
        void  order_gene( Gene );
        int   equal( Gene, Gene );
        void  initGeno( );
};
#endif
```

APPENDIX B

Object-oriented code

```
//
//
//                         G E N E . C P P
//
//
//

#include "global.h"
#include "gene.h"
#include <stdio.h>
#include <stdlib.h>

//''''''''''''''''''''''''''''''''''''''''
//              S E T   S E T
//
// find the set that this genotype belongs to
//      by totaling the type 2 alleles
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Gene::setSet(void)
{
 int i;
 int sum = 0;

     for(i = 0; i < GNLEN; i++ )
     {
         sum += geno[i] - 1;
     }
     set = sum;
}


//''''''''''''''''''''''''''''''''''''''''
//          R E P R O D U C E
//
// create a new gene from this gene and gn1
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

Gene Gene::reproduce( Genotype gn1 )
{
 int i, j;
 Gene newgene;
     for (j = 0; j < LOCI * 2; j += 2 )                      // for each locus
     {                                                       //select
         newgene.geno[j]   = gn1[global::getnum(0,1)+j]; //one of the father's
                                                                //alleles
         newgene.geno[1+j] = geno[global::getnum(0,1)+j];//and one of the
     }                                                          //mother's alleles
     newgene.geno[GNLEN] = 0;    // for debug display
     newgene.setSet();                                  // define the set
     return( newgene );
}
```

APPENDIX B
Object-oriented code

```
//''''''''''''''''''''''''''''''''''''''''
//          O R D E R  G E N E
//
//    sort the alleles at each locus
//''''''''''''''''''''''''''''''''''''''''

void Gene::order_gene(Gene  gn)
{
 int  i, j;
 int swap;

     for (i = 0; i < LOCI; i++)              // for each locus
     {
         j = i * STRANDS;

         if(gn.geno[j] > gn.geno[j+1])       // put the lower valued
         {                                   //allele first
             swap = gn.geno[j];
             gn.geno[j] = gn.geno[j+1];
             gn.geno[j+1] = swap;
}    }    }



//''''''''''''''''''''''''''''''''''''''''
//           E Q U A L
//
//  returns TRUE if two genes are equal
//''''''''''''''''''''''''''''''''''''''''

int Gene::equal( Gene gn, Gene template )
{
 char i;

     for (i = 0; i < GNLEN; i++)                  // for each allele
     {
         if (gn.geno[i] != template.geno[i]    // if they are not equal
             &&  template.geno[i] != '*')      //and template is not a wild
         return( FALSE );                     //card, return false
     }
     return (TRUE);
}



//''''''''''''''''''''''''''''''''''''''''
//           I N I T   G E N E
//
//       build a new gene using the
//        initialization frequencies
//''''''''''''''''''''''''''''''''''''''''

void Gene::initGeno( )
{
 int i, j;
 float sum, mark;

     for ( i = 0; i < GNLEN; i++)              // for each allele
     {
         mark = rand() / (float) RAND_MAX;       //  0 <= mark < 1
         j = 0;
```

```
        sum = 0;
        while (mark > sum)
        {                                   // sum the probabilities
            sum += INIT[j++];               //until they are greater
        }                                   //than mark
        geno[i] = j;                    // allele is type j
    }
    geno[GNLEN] = 0;                    // for debug display
    setSet();                           // define the set
}


//
//
//                    G L O B A L . H
//
//
//



#if !defined __GLOBAL_H
#define   __GLOBAL_H

#include <fstream.h>
#include <math.h>
#include <stdlib.h>

const int MAX = 10;
const int STRANDS = 2;

const int VMAP = 0;
const int LEDG = 1;
const int GRPH = 2;

enum {FALSE, TRUE};


  // definitions making code for access to constants more compact
# define INIT        global::getInit()
# define SETS        global::getSets()
# define SIDE1       global::getSide1()
# define SIDE2       global::getSide2()
# define SIDE        global::getSide()
# define LOCI        global::getLoci()
# define ALLELES global::getAlleles()
# define RND     global::getRnd()
# define SEARCH  global::getSearch()
# define GEN     global::getGen()
# define GNLEN       global::getGnLen()
# define FITNESS global::getFitness()
# define GSYM       global::getGenSym()


class global
{

    private:      // global constants

        static  float   Init[MAX];
        static  float   Fitness[MAX];
        static  int     Sets;
```

```
        static  int     Side1;
        static  int     Side2;
        static  int     Side;
        static  int     Loci;
        static  int     Alleles;
        static  int     GnLen;
        static  int     Rnd;
        static  int     Search;
        static  int     Gen;

        static  char    GenSym[MAX][MAX];


        static  float   getNextParam( void );
        static  void    openFile( char* );

    public:

        static  void    readIn(void);


          // functions returning global constants
        static  float*  getInit(void)       { return Init  ; }
        static  float*  getFitness(void)    { return Fitness; }
        static  int     getSets(void)       { return Sets  ; }
        static  int     getSide1(void)          { return Side1 ; }
        static  int     getSide2(void)          { return Side2 ; }
        static  int     getSide(void)       { return Side  ; }
        static  int     getLoci(void)       { return Loci  ; }
        static  int     getAlleles(void)    { return Alleles; }
        static  int     getRnd(void)        { return Rnd   ; }
        static  int     getSearch(void)     { return Search    ; }
        static  int     getGen(void)        { return Gen   ; }
        static  int     getGnLen(void)          { return GnLen ; }

    //  static  char*[10]  getGenSym(void)     { return GenSym ; }


         // random number utility functions
        static  int     getnum( int low, int high )
            {return ( floor( getFract() * (high - low + 1) + low));}

        static  float   getFract(void)
            {return (rand() / (float)(RAND_MAX+1 ));}


};

#endif
```

APPENDIX B
Object-oriented code

```
//
//
//                    G L O B A L . C P P
//


#include  "global.h"
#include  <stdlib.h>

ifstream infile;                    // construct the file object


float    global::Init[MAX];              // declare static variables
float    global::Fitness[MAX];
char global::GenSym[MAX][MAX];
int      global::Sets;
int      global::Side1;
int      global::Side2;
int      global::Side;
int      global::Loci;
int      global::Alleles;
int      global::GnLen;
int      global::Rnd;
int      global::Search;
int      global::Gen;

//''''''''''''''''''''''''''''''''''''''''
//          O P E N F I L E
//
//       open the setup file
//''''''''''''''''''''''''''''''''''''''''


void global::openFile( char* inFileName )
{
     infile.open(inFileName, ios::nocreate );
     if (! infile)
     {
         cerr << "could not open " << inFileName;
         exit (-1);
     }
}


//''''''''''''''''''''''''''''''''''''''''
//         R E A D  I N
//
//    get the global parameters
//''''''''''''''''''''''''''''''''''''''''

void global::readIn(void)
{
 char    text[80];

     global::openFile( "gs_setup.h" );

     Side1 = (int)getNextParam();        // side of the larger area
     Side2 = (int) getNextParam();       // side of the larger area
         Side = Side1 + Side2;
     Alleles = (int) getNextParam();     // number of allele types
```

```
    Loci = (int) getNextParam();          // number of loci
        GnLen =   Loci * STRANDS ;        // number of alleles per individual
    Sets =   getNextParam();

    for (int i = 0; i < SETS; i++)
    {
        Fitness[i] =  getNextParam();     // fitness for each genotype set
        infile.get( text, 80 );
      //    for ( int j = 0; j < GnLen; j++)
       //     Geno[i][j] = text[j];        // symbol for each genotype set
    }

    for ( i = 0; i < ALLELES ; i++)
    {
          Init[i] = getNextParam();       // initial frequency of alleles
    }


    Rnd = (int) getNextParam();           // seed for pseudo random generator
    Search = (int) getNextParam();        // search distance

    Gen =  (int) getNextParam();

      // error trap
    if ((i = (int) getNextParam()) != -999) // -999 is a flag for success
    {
        cout < "too many parameters in Setup ";
        exit (-3);
    }
}


//''''''''''''''''''''''''''''''''''''''''''''
//     G E T   N E X T   P A R A M
//
// return the string following the next colon
//''''''''''''''''''''''''''''''''''''''''''''


float global::getNextParam(void)
{

 char    text[800];    // raw text from the external file
 char    c;            // input character


    infile.get( text, 800, ':' );  // read to the next colon
    infile.get(c);
    if (infile.eof())
        { cout < "too few input values"; exit(6);}
    infile.get( text, 10 );        // read in text value
    return (atof( text ));         // convert to float and return
}
```

APPENDIX B
Object-oriented code

```
//
//              G R F   C L S S   . H
//
//        A file containing the graphics classes.


#if !defined __GRF_CLSS_H
#define      __GRF_CLSS_H

#include <graphics.h>
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>

//////////  C L A S S    S C R E E N   E L E M E N T

class ScreenElement
{
     protected:
         int posX, posY;      // position
         int color;           // color

     public:
         ScreenElement(){}

         ScreenElement( int x, int y, int clr )
         {
             posX = x; posY = y; color = clr;
             setcolor( color );
         }

};


//////////  C L A S S   P I X E L

class Pixel:  public ScreenElement
{

     public:
         Pixel(): ScreenElement() {}
         Pixel( int x, int y, int clr ): ScreenElement( x, y, clr )
             { display(); }

         void display()
             { putpixel( posX, posY, color ); }
};


//////////  C L A S S   R E C T A N G L E

class Rectangle: public ScreenElement
{
     protected:
         int sideX, sideY;

     public:
         Rectangle(): ScreenElement() {}
```

```
        Rectangle( int x, int y, int sX, int sY, int clr )
            : ScreenElement( x, y, clr )
            { sideX = sX; sideY = sY; display(); }

        virtual void display() {}

};


////////// C L A S S  F I L L E D

class Filled: public Rectangle
{
    public:
        Filled(): Rectangle() {}
        Filled( int x, int y, int sX, int sY, int clr )
            : Rectangle( x, y, sX, sY, clr )
            {  display(); }

        void display()
            {
                setfillstyle(SOLID_FILL, color );
                bar( posX, posY, posX+sideX, posY+sideY );
            }
};

////////// C L A S S  F R A M E

class Frame: public Rectangle
{
    public:
        Frame(): Rectangle() {}
        Frame( int x, int y,  int sX, int sY, int clr )
            : Rectangle( x, y, sX, sY, clr )
            { display(); }

        void display()
        {
            setlinestyle( SOLID_LINE, 0, THICK_WIDTH );
            rectangle( posX, posY, posX + sideX, posY + sideY );
        }
};

////////// C L A S S  T E X T  E L E M E N T

class textElement: public ScreenElement
{
    protected:
        int row, col;

    public:
        textElement(): ScreenElement() {}
        textElement( int x, int y, int clr )
            : ScreenElement( x, y, clr )
            {
                row = (float)posY / getmaxy() * 25;
                col = (float)posX / getmaxx() * 80;
            }
        virtual void display(){};

};
```

APPENDIX B
Object-oriented code

```
//////////  C L A S S   N U M B E R S

class Numbers: public textElement
{
     private:
          float num;

     public:
          Numbers( int x, int y, float value, int clr = 7)
               : textElement( x, y, clr)
               { num = value; display(); }

          void display()
          {
               gotoxy(col, row);
               cout.precision(3);
               cout << num ;
          }
};



//////////  C L A S S   C A P T I O N

class Caption: public textElement
{
     private:
          char buffer[80];

     public:
          Caption(int x, int y, int clr, char* text )
               : textElement( x, y, clr)
               { strcpy( buffer, text ); display(); }

          void display(){ gotoxy(col, row); cout << buffer; }
};

//////////  C L A S S   M A P

class Vmap: public Window
{
     private:
          int box;
          int border;

     public:
          Vmap( int x, int y, int sX, int sY, int clr )
               : Window(x,  y,  sX,  sY,  clr)
               {
                    border = 2;
                    box = 300 / SIDE;
                    Filled fl(posX, posY, sideX, sideY, color );
               }
          void update(int nums[MAX])
          {
               Filled be( posX + (nums[0]-1) * box + border,
                          posY + (nums[1]-1) * box + border,
                          box - border,
                          box - border,
                          nums[2] );
}          };
```

APPENDIX B
Object-oriented code

```
//////////  C L A S S   G R A P H

class Graph: public Window
{
     private:
          int column;

     public:
          Graph( int x, int y, int sX, int sY, int clr )
               : Window(x,  y,  sX,  sY,  clr)
               {
                    column = 10;
                    Filled fl(posX, posY, sideX, sideY, color );
               }

          void update(int nums[MAX]);
};



//////////  C L A S S    L E G D G E R

class Ledger: public Window

{
     private:

     public:
          Ledger( int x, int y, int sX, int sY, int clr = 0 )
               : Window(x, y, sX, sY, clr)
               {
                    Frame fr(posX, posY, sideX, sideY, color );
                    initLedger();
               }
          void initLedger();
          void update(int *);
};
#endif


//////////  C L A S S   W I N D O W

class Window: public Rectangle
{
     public:
          Window( int x, int y, int sX, int sY, int clr = 0 )
               : Rectangle(x,  y,  sX,  sY,  clr)  { }
     virtual void update( int[MAX] ){};  };
```

APPENDIX B
Object-oriented code

```
//
//                    V I D E O  .  H
//
//
//


#if !defined __VIDEO_H
#define __VIDEO_H

#include <stdio.h>
#include <graphics.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include "map.h"

#define GRID(x)  (x-1) * box + border  // translate map position to
                                       //pixel position
         const int border = 10;         // width of outline
         const int graphtop    = 300;   // graph position parameters
         const int graphbottom = 400;
         const int graphleft   = 380;
         const int left = 70;           // text position parameters
         const int top  = 3;

class Video
{
     private:
         int box ;                 // side length of individual square
         int Tvrt, Thor;           // text to pixel conversions

         char*   format(float num, int decimal );
         void    do_legend (void);
         void    format_display();

     public:

         void initVid(void);
         void display_ind( Pos, char  );
         void debug( Pos pos, short color );
         void exit_vid();
};
#endif

//
//                    V I D E O  .  C P P
//
//
//

#include "video.h"
#include "grf_clss.h"

extern Window* clss[3];
```

APPENDIX B

Object-oriented code

```
//''''''''''''''''''''''''''''''''''''''''''
//            I N I T   V I D
//
//          configure video system
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Video::initVid()
{

 int gmode, errorcode, gdriver = DETECT;  // DETECT --> find highest mode

     initgraph(&gdriver, &gmode, "\\tc\\bgi");  // initialize graphics mode


     errorcode = graphresult();          // read result of initialization

     if (errorcode != grOk)      // an error occurred
     {
        cout << "Graphics error: " << grapherrormsg(errorcode) << endl;
        cout << "Press any key to halt:" ;
        getch();
        exit(255);               // return with error code
     }
     clss[VMAP] = new Vmap   ( 10, 10, 304, 304, 15);    // display the windows
     clss[LEDG] = new Ledger ( 400, 20, 200, 200, 14);
     clss[GRPH] = new Graph  ( 30, 320, 500, 100, 8);


}


//'''''''''''''''''''''''''''''''''''''''''''''''''
//            E X I T   V I D
//
//       reset video mode for exit
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
void Video::exit_vid()
{
     closegraph();
}



//'''''''''''''''''''''''''''''''''''''''''''''''''
//            F O R M A T
//
//   return a string formatted to 'decimal'
//          decimal places
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

char* Video::format(float num, int decimal )
{
 static char buffer[20];    // return string

        // put 'decimal' number of places to the left of the decimal point
      num = (num * pow(10, decimal)) + pow(10, decimal);
        // convert to a string
      gcvt( (float) num , decimal + 2, buffer);
        // put decimal point at beginning of string
      buffer[0] = '.';
        // check for num = 1
```

```
     if (num == 2 * pow(10, decimal) ) strcpy (buffer, "1.0" );
        // make buffer a string
     buffer[decimal+1] = 0;

     return( buffer );
}

//''''''''''''''''''''''''''''''''''''''''''''''
//        LEDGER :: I N I T L E D G E R
//
//       print the set colors and symbols
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Ledger::initLedger()
{
 int Th = 16;
    for (int i = 0; i < SETS; i++ )
    {
        Filled rt( posX + 140, posY + 30 + i * (Th-2), Th+3, Th-6, i+1 );
        Caption wr(posX + 100, posY + 75 + i * Th, 7, "here" );
    }
}

//''''''''''''''''''''''''''''''''''''''''''''''
//        LEDGER ::  U P D A T E
//
//       print the set frequencies
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Ledger::update(int nums[MAX])
{
 int Th = 16;
    for (int i = 0; i < SETS; i++ )
    {
        Numbers n(posX + 25, posY + 75 + i * Th,
            nums[i] / (float)Being::getSize() );
}   }
//''''''''''''''''''''''''''''''''''''''''''''''
//       GRAPH :: U P D A T E
//
//  plot a pixel for each set frequency
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

void Graph::update(int nums[MAX])
{
    for (int i = 0; i < SETS; i++ )
        Pixel( posX + column,
            posY + sideY - nums[i] / (float)Being::getSize() * sideY,
            i+1 );
    column++;
}
```

```
//
//
//                      M A P . H
//
//
//


#if !defined __MAP_H
#define __MAP_H


#include "being.h"
#include "global.h"

const int side = 30;


class Map
{
     private:

          static Pos    present;
          static Being* frame[side][side];
          static int    inBounds( Pos pos );


     public:

          static Being*   search( Pos active );
          static Pos      nextSquare(void);
          static void     insert( Pos active, Being* bptr )
                              { frame[active.x][active.y] = bptr; }
          static Being*   getBptr( Pos active )
                              { return frame[active.x][active.y]; }
};
#endif


//
//
//                    M A P . C P P
//

#include "map.h"

Being* Map::frame[side][side];


//''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
//          N E X T   S Q U A R E
//
//  move cursor to the next position in
//  right to left, top to bottom order
//,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,

Pos Map::nextSquare(void)
{
 static Pos cursor = {0, 0};
     do
     {
          if ( ! (++cursor.x %= SIDE1+1) )  // move cursor to the right
```