SELFISH HERD WITH MOVING AND NON-MOVING VORONOI DIAGRAM IN TWO DIMENSIONS AND APPLICATION

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment of
the Requirements

for the Degree

Master of SCIENCE

by

Yasuharu Kai, B.S.

San Marcos, Texas
August 2004

ACKNOWLEDGEMENTS

I would like to acknowledge my gratitude to the supervisor of my master's thesis, Dr. Carol Hazlewood, for helpful comments and assistance. And also I wish to acknowledge the goodwill and patience of the rest of my thesis committee members, Dr. McCabe and Prof. Davis.

I also want to thank my friends, Brian Schutz and Misa Oshima, for their valuable support and encouragement.

Finally, I would like to express my sincere gratitude to my family for a chance to pursue a master's degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

SELFISH HERD WITH MOVING AND NON-MOVING VORONOI DIAGRAM IN TWO DIMENSIONS AND APPLICATION

By

Yasuharu Kai, B.S.

Texas State University-San Marcos

August 2004

SUPER VISING PROFESSOR: CAROL HAZLEWOOD

Voronoi Diagrams of moving points are applied to Selfish Herd problems. A portable C++ implementation with graphics is described. Data gathered from simulations using moving and non-moving Voronoi calculations are analyzed. The analysis shows that fewer complete Voronoi Diagrams are calculated using the moving Voronoi method. And also the results from simulations introduce some suggesstions to improve performance of the moving Voronoi Diagram.

INTRODUCTION

Voronoi Diagrams are useful and important for geometrical problems. They can be applied to various situations (Aurenhammer, 1991). The usefulness of Voronoi Diagrams in two dimensions has been found in the Selfish Herd problem (Hamilton, 1970). In the Selfish Herd problem, the objects, prey, move to reduce the chance of getting attacked by a predator. In Hamilton's paper, the prey moves to the closest neighbor to reduce the chance of getting attacked by a predator. The closest neighbor can be found efficiently by using a Voronoi Diagram. Simulations of Selfish Herd behavior have been done using non-moving Voronoi Diagrams (Viscido, n.d.). A different approach, which uses the method of maintaining the Voronoi Diagram of moving points over time (Albers, Guibas, Mitchell, & Roos, 1998), will be applied to the Selfish Herd problem in this study.

Main Contributions

One of the main contributions is a portable application for Selfish Herd simulation by use of the OpenGL graphics architecture and the popular C++ programming language. Another contribution is an improved simulation method using the moving Voronoi Diagram.

Organization

Section 2 summarizes the previous work related to this study. Section 3 describes the theory of Static and Kinetic Voronoi Diagrams in this study. Section 4 presents details of application. Experiments in Section 5 demonstrate that the Kinetic Voronoi Diagram is applicable to the Selfish Herd problem. Section 6 presents simulations for comparison of the speeds of Static and Kinetic Voronoi algorithms. In Section 7, we deduce our conclusions and introduce some sugesstions for improvement of the Kinetic algorithm.

PREVIOUS WORK

Voronoi Diagrams and Delaunay Triangluation

A Voronoi Diagram records information about objects that are close to other objects. Consider a finite set of points $Pi, i = 1, ..., n, (n \geq 3)$, called sites, in a given plane. Each site has a Voronoi cell, and every point in the Voronoi cell of Pi is closer to Pi than to any other site. The boundary of two sites Pi and Pj is part of the perpendicular bisector between Pi and Pj and this will be an edge of Voronoi Diagram. The points where the edges intersect will be the Voronoi vertices.

If the Voronoi Diagram is a graph G, the corresponding dual graph $G*$ is called a Delaunay triangulation. In figure 1, the dotted lines form the Voronoi Diagram and the solid lines show the Delaunay triangulation. The vertices in the graph G correspond to faces in the graph $G*$. Each of the faces in G corresponds to a vertex in the graph $G*$. If the two faces in G have a boundary edge, the corresponding two vertices in $G*$ are connected. If no four points are cocircular, the degree of the vertices is three in the Voronoi Diagram, and the dual graph is a collection of triangles. From the Delaunay triangulation, it is easy to find the convex hull, which is the smallest convex set containing all given points. The boundary of the convex hull is the boundary of the outer edges of the Delaunay triangulation.

In this study, Voronoi Diagrams are computed by use of Fortune's algorithm, which is a plane sweep algorithm (Fortune, 1987).

Voronoi Diagram of Moving Points

Voronoi Diagrams have been applied to geometry for moving points. There is a nice method of maintaining Voronoi Diagrams in two dimensions over time (Albers et al.,

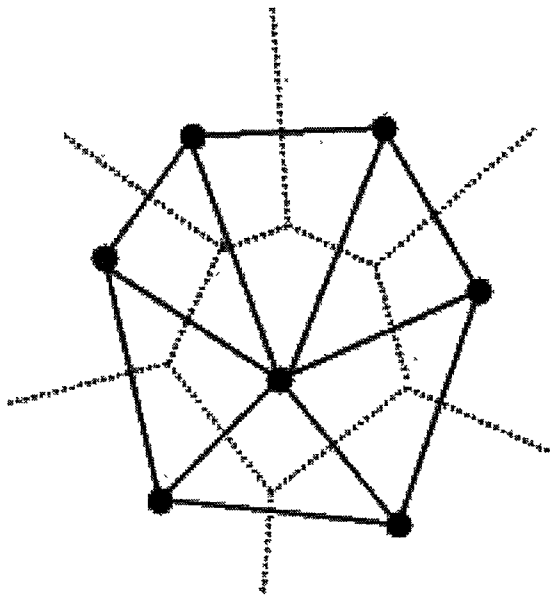

Figure 1. Voronoi Diagram and Delaunay Triangulation

1998).

Consider a finite set of points $Pi, i = 1, ..., n, (n \geq 3)$, each of which is moving along a given path. The points form a Voronoi Diagram. The Delaunay triangulation captures the topological structure of the Voronoi Diagram. After a small motion, the topological structure will be changed and the change will usually be local, that is, confined to a retriangulation of four points. Those changes are called topological events. The elementary changes are separated into two cases. One of the changes occurs inside a convex hull, and the other occurs on the convex hull. The Voronoi Diagram is not recalculated unless a topological event occurs. The topological events need to be stored in a priority queue according to time and the queue is updated when one of the events occurs. After an event, the Voronoi Diagram needs to be updated, but if the event is local, a partial update of the points related to the event needs to be completed. If the event is not local, recalculation of a Voronoi Diagram will be performed. This algorithm introduced better performance than the entire recalculation of Voronoi Diagram with

every motion of the points. The author concludes that every topological event needs $O(\log n)$ time and there are at most $O(n^4)$ topological events.

Topological Events

Topological structure is changed when a given point A's neighbor, called point B, moves away, and B is no longer A's neighbor. The elementary change of topological structure includes two cases.

First, consider two adjacent triangles $T1 = (A, B, C)$ and $T2 = (B, C, D)$. The triangle T1 represents that one point's neighbors are two other points. For example, B and C are neighbors of A, but D is not a neighbor of A. After an elementary change of topological structure, these two triangles will become triangles $T3 = (A, B, D)$ and $T4 = (B, C, D)$. Now, D is a neighbor of A, and C is not a neighbor of A. This event is called SWAP, which is shown in figure 2.

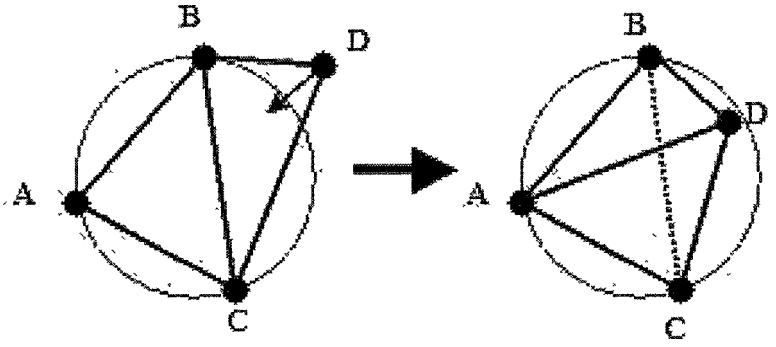

Figure 2. Topological Event SWAP

Second, consider two adjacent triangles $T1$ and $T2$, where points A, C, and D are

on the convex hull. The point C is moving to B and other points reside where they were previously found. Before C crosses line $L1 = (A, D)$, the neighbors' information is not changed, which means that two triangles $T1$ and $T2$ remain part of the triangulation. However, After C crosses line $L1$, those triangles become three separate triangles (A, C, B), (B, C, D), and (A, C, D). This event is shown in figure 3.

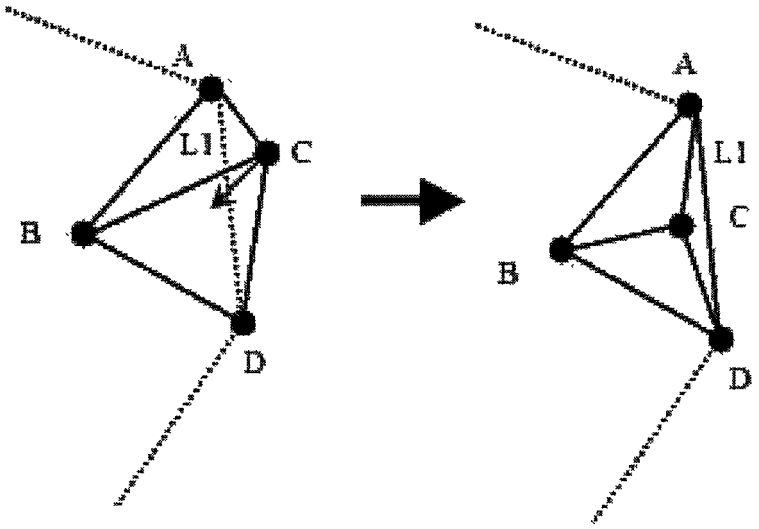

Figure 3. Topological Event on Convex Hull

Finding a Topological Event

There are two types of topological events, one inside the convex hull, the other on the convex hull. In each case a determinant can be used to compute the time of the event.

The first case is that the event will occur inside the convex hull and the InCircle determinant (Guibas & Stolfi, 1985) will be utilized:

$$InCircle(A,B,C,D) = \begin{vmatrix} x_a & y_a & x_a^2+y_a^2 & 1 \\ x_b & y_b & x_b^2+y_b^2 & 1 \\ x_c & y_c & x_c^2+y_c^2 & 1 \\ x_d & y_d & x_d^2+y_d^2 & 1 \end{vmatrix} \quad (1)$$

Consider two adjacent triangles $T1 = (A, B, C)$ and $T2 = (B, C, D)$. There are no points in the circle defined by those three points A, B, C because of a property of the Delaunay triangulation. After the point D moves into the circle $C1 = (A, B, C)$, this Delaunay triangulation property no longer holds true. When the InCircle determinant is 0, D is on the circle defined by three points A, B, and C. This means that the topological event just occurred. For all possible pairs of adjacent triangles, this InCircle determinant needs to be re-evaluated to discover all possible topological events. In this case, no other points exist on the circle $C1$.

Consider four points $A = (x_a, y_a)$, $B = (x_b, y_b)$, $C = (x_c, y_c)$, and $D = (x_d, y_d)$ with straight-line motion. Let each point's velocity be (x_{va}, y_{va}), (x_{vb}, y_{vb}), (x_{vc}, y_{vc}), and (x_{vd}, y_{vd}) respectively. If time t is considered to be when an event will occur, the InCircle determinant takes on the form shown in equation 2. In order to find the value of t, the fourth degree equation $InCircle(A, B, C, D) = 0$ must be solved.

$$InCircle(A,B,C,D) = \begin{vmatrix} x_a+x_{va}t & y_a+y_{va}t & (x_a+x_{va}t)^2+(y_a+y_{va}t)^2 & 1 \\ x_b+x_{vb}t & y_b+y_{vb}t & (x_b+x_{vb}t)^2+(y_b+y_{vb}t)^2 & 1 \\ x_c+x_{vc}t & y_c+y_{vc}t & (x_c+x_{vc}t)^2+(y_c+y_{vc}t)^2 & 1 \\ x_d+x_{vd}t & y_d+y_{vd}t & (x_d+x_{vd}t)^2+(y_d+y_{vd}t)^2 & 1 \end{vmatrix} \quad (2)$$

The second case is that the event will occur on the convex hull. In this case, the CCW determinant will be utilized. The determinant is the following:

$$CCW(A,B,C) = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix} \quad (3)$$

The $CCW(A, B, C)$ primitive (Guibas & Stolfi, 1985) is defined to be true when the points A, B, and C form a counterclockwise triangle. When the points A, B, and C form a clockwise triangle, $CCW(A, B, C) < 0$.

Consider one triangle (A, B, C) on the convex hull. The point C is moving toward the inside of the convex hull. A point C will cross the line defined by two points A and B. When C is on the line (A, B), these three points do not form a triangle. This means that a topological event is occurring. When the determinant CCW is 0, the three points A, B, and C are on the same line.

Consider four points $A = (x_a, y_a)$, $B = (x_b, y_b)$, and $C = (x_c, y_c)$ with straight-line motion. Let each point's velocity be (x_{va}, y_{va}), (x_{vb}, y_{vb}), and (x_{vc}, y_{vc}) respectively. If time t is considered to be when an event will occur, the CCW determinant will become equation 4. In order to find the value of t, the determinant need to be 0. For the equation $CCW(A, B, C) = 0$, the equation of degree 2 needs to be calculated.

$$CCW(A, B, C) = \begin{vmatrix} x_a + x_{va}t & y_a + y_{va}t & 1 \\ x_b + x_{vb}t & y_b + y_{vb}t & 1 \\ x_c + x_{vc}t & y_c + y_{vc}t & 1 \end{vmatrix} \quad (4)$$

Updating of Topological Events

There are two types of topological updates. A SWAP is performed if four points are cocircular. A triangulation is performed if more than four points are cocircular.

After all possible topological events are discovered, the updating of the topological event needs to be considered to manage the topological events. All events are stored in a priority queue. This queue is organized in order according to the time at which the event will occur. There are two cases for the updating of the topological events. One case is an update for an event, which is found by the InCircle test. For this event, we need to update the topological structure. The whole triangulation need not be done. The change of the topological structure is local to two adjacent triangles. Therefore, only those two triangles need to be updated. After the triangle update, the event queue also needs to be updated. The event that just occured will be removed from the queue. For the new event from the

two new adjacent triangles, the timing when the event will occur needs to be calculated, and the new event will be added to the event queue according to the timing.

There is another case for the updating of the topological event. In the previous case, only four points are cocircular. However, when more than five points are cocircular, the previous update SWAP cannot be applied. The reason is that it is difficult to find which point is close to another point after the event. One example is shown in figure 4. In this case, complete triangulation will be performed.

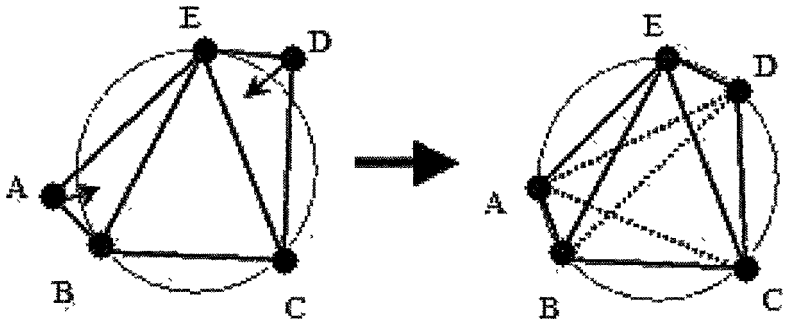

Figure 4. Five Points Are Cocircular

Voronoi Diagram in Selfish Herd

W. D. Hamilton has shown that a Voronoi Diagram is useful with regard to the Selfish Herd problem. A predator will attack the nearest prey, and all prey have their own region which is called the "domain of danger". Any points in a region of a given prey are closer to the owner of the region than any other prey. This region forms a Voronoi cell. If the domain is larger, the chance of attack will be higher.

One of the prey will notice that a predator is in its own region, and will try to reduce the chance of attack. The prey moves to its nearest neighbor, but all other prey stays in place during that moment. The nearest neighbor of a given prey is its closest neighbor among all neighbors of the prey. One property of this predator-prey problem is that a relation between prey and its nearest neighbor is not symmetric. Therefore, each prey has his own neighbor (figure 5).

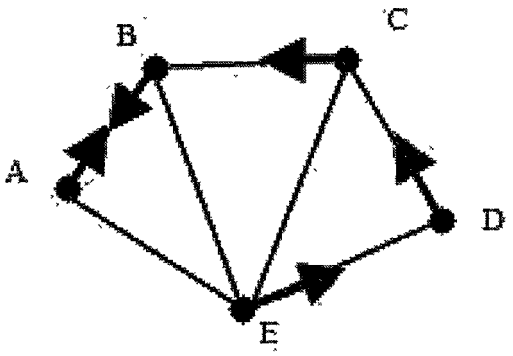

Figure 5. Relation between Prey and Prey's Closest Neighbor

From Hamilton's observation, such a prey will reduce its domain of danger, which means that the prey becomes safer in most cases. If the nearest neighbor is an isolated prey, its domain of danger may increase. If the Voronoi cell is simple (3 or 4 sides), such increases are more inclined to happen. Also, if the Vornoi cell has many sides, the domain of danger will tend to be reduced. Because, based on Hamilton's experimental analysis, the average number of sides of a Voronoi cell is six, and having a low number of sides is a rare case, he concluded that a given prey moving to its nearest neighbor is generally beneficial (Hamilton, 1970).

Simulation in Selfish Herd

S. Viscido tried Selfish Herd simulations with Voronoi Diagrams. The author implemented the code with a combination of Bourne shell, C, Maple V, and SAS. This simulation was based on time driven simulation.

The simulation field is a 500x500 space grid and the scale is in cm. He defined six movement rules. The first movement is chosen at random, and the distance moved is 1.4 cm for each frame. The second movement is also 1.4 cm each frame, but the direction is updated to their new closest neighbors, which is the method proposed by Hamilton. The third movement is to jump to a space between their closest neighbors and their closest neighbors' closest neighbors. The fourth movement is 1.4 cm each frame and the destination is the gap between the closest neighbors. The difference between the second and the fourth movement is whether the destination is updated every frame or not. The fifth movement is 1.4 cm each frame and the direction chosen is based on the entire herd's position. The sixth movement is influenced by a decay function which determines how much of an influence each prey has.

First in this simulation, all simulated animals' positions need to be defined. Next, triangulation will be executed via Bourne shell code. A Maple program calculates the domain of danger from the output of the triangulation program. This is the initialization section, and after that a loop for simulation is called. Calculations for the animal's movements, triangulation, and domain of danger will be repeated until enough data is collected to be analyzed. The analysis is to ascertain the difference between before and after the domain of danger is observed.

The codes are available on the web. However, the results of those calculations have never been published (Viscido, n.d.).

STATIC AND KINETIC VORONOI DIAGRAMS

When simulating a moving object, Voronoi Diagrams should be updated after every small motion of objects. In Selfish Herd, all prey move to their own closest neighbors. However, because neighbors are also moving, their neighbors and closest neighbor need to be updated. The difference between simulations using the Static and Kinetic Voronoi Diagrams is the way in which this information is managed over time.

Static Voronoi Diagrams

Voronoi Diagrams are calculated for initialization, and the topological structure of the Voronoi Diagram is stored in an array. The topological structure shows a prey's neighbor's information and the location of the closest neighbor in this application. After every small motion of all objects, Voronoi Diagrams are recalculated, and the prey's neighbors and closest neighbors information are also updated. In this simulation, recalculating and updating will be repeated every time after all prey moves. The algorithm is shown in figure 6. Every time the Voronoi Diagrams are calulated, the scene is considered to be Static. Therefore, this method is called the "Static Voronoi Diagram" in this study.

For calculating the topological structure, because Fortune's algorithm is utilized, the time complexity is $O(n \log n)$. Letting the prey move and printing the prey's positions takes $O(n)$. Recalculating the topological structure takes $O(n \log n)$. The overall time complexity for k time steps is $O(kn \log n)$.

Kinetic Voronoi Diagrams

The initialization portion is the same as a Static Voronoi Diagram, which is calculating a Voronoi Diagram and then storing the topological structure of the Diagram.

```
Initialize a topological structure of the Voronoi Diagram

LOOP
  all prey move
  print prey's positions
  recalculate topological structure
END LOOP
```

Figure 6. Static Voronoi Algorithm

In this simulation, only when the topological sturucture is changed, will it be updated. The change of topological structures are called topological events. The topological events need to be stored according to time. In this study, this method is called the "Kinetic Voronoi Diagram" because the topological structure of a Voronoi Diagram is managed over time. The algorithm is shown in figure 7.

The initialization section takes $O(n \log n)$ and is similar to other Static algorithms, with regard to time complexity. Letting the prey move and printing the prey's positions takes $O(n)$. Retriangulation takes $O(n \log n)$. For local updates, deleting an event takes $O(1)$ time, and putting a new event into a priority queue takes $O(\log n)$. Also, local updates need the topological structure update, which requires $O(1)$ time. The overall time complexity for e events is $O(mn \log n)$, where m is $max(n, e)$.

While retriangulation takes place during every simulation step in the Static Voronoi, an event does not take place during every simulation step in the Kinetic Voronoi. Therefore, the Kinetic Voronoi algorithm can be considered faster than the Static Voronoi algorithm.

```
Initialize a topological structure of the Voronoi Diagram

LOOP
  all prey move
  print prey's positions
  If events occur, retriangulation or local update SWAP
  Otherwise, do nothing
END LOOP
```

Figure 7. Kinetic Voronoi Algorithm

IMPLEMENTATION

An overview of the application will be introduced, followed by descriptions of the data structures, classes, and functions used in the application.

Survey of Application

This implementation is portable so that a user on a different system can utilize it for simulations. This program can be executed with or without animation according to the availability of OpenGL. Without OpenGL, the animation portion will be ignored when the program is compiled. With OpenGL, functions need to be registered for initializing a window, for drawing all prey's positions, for letting prey move, and so forth. A user-initialized flag determines if animation is included.

The program consists of two main parts: a simulation and an animation part (figure 8). The simulation section deals with each moving prey's position and the domain of danger, which is the Voronoi cell. For the animation, one member function in the herd class will take the preys' positions and neighbors' indices, and the animation part will display the map.

The simulation section uses object-oriented design with C++ in order for the user to be able to modify it easily. The application was implemented on Red Hat Linux 9.0, and the animation section is coded with OpenGL.

The top level algorithm with graphics is shown in figure 9.

The top level algorithm without graphics is shown in figure 10. The integer `loop_num` is for count how many simulation steps are executed.

Both programs take $O(n)$ running time, where n is the number of loop incarnations. The reason behind the $O(n)$ running time is that it iterates n times. In the pseudocode shown in figure 10, n is 20.

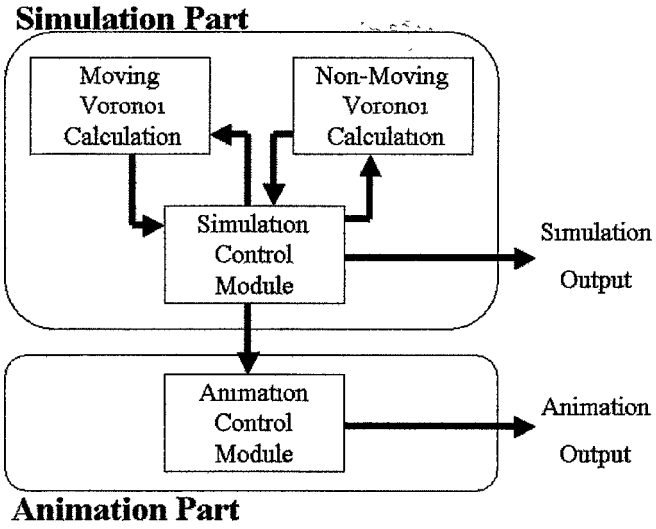

Figure 8. System Architecture

Data Structure

There are three data structures defined in the application. The "eventque" data structure is for an event queue. The "triangle" data structure stores a triangle's information. The "position" data structure is to store prey's postion.

- Event queue

```
typedef struct myeventque{
  int triangle1[3], triangle2[3];
  double time;
  struct myeventque *next;
}eventque;
```

This is stored in a linked list of eventque structures. The eventque structure includes two arrays of three integers which stores prey indices. Three prey form a triangle and two triangles will be related to one event (figure 2 and 3). This linked list is a priority queue based on when a given event will occur. A herd class includes an eventque linked list as private data. In a herd class's constructor, a event queue will be set up from

```
Initialize a herd.
main(){
  Initialize Window information using OpenGL Utility Toolkit (GLUT)
  registering display function
  registering Idle function
  myInit();
  glutMainLoop();
}
```

Figure 9. Top Level Algorithm with Graphics

Delaunay triangulation information. This queue will be updated when a event occurs. When the program finishes, this queue will be disapper with the herd class.

- Triangle

```
typedef struct mytriangle{
  int bot;
  int top;
  int right;
  struct mytriangle *next;
}triangle;
```

This is stored in a linked list of triangle structures. The triangle structure includes three integers which form a triangle. Also it has a pointer which contains the address of the next triangle. If there is no following triangle to be considered, the pointer will contain the NULL address. This triangle structure will be used when triangulation is taken place. Trianglulation information in triangle linked list will be used for discovering prey's neighbors and nearest neighbor or for finding all possible events. After the use for neighbors or events, the triangle linked list will be deleted.

- position

```
Initialize a herd;
Initialize loop_num integer;

MAIN(){
  LOOP
    Increment loop_num;
    if( loop_num > 20 ) break;
    herd_A.MoveAllPrey();
  END LOOP
}
```

Figure 10. Top Level Algorithm without Graphics

```
typedef struct myposition{
  double x;
  double y;
}position;
```

This position structure includes x and y coordinates and is used when prey's positions and "one step" of prey are stored. Prey's positions and "one step" of prey are stored as private variables in a prey class. Those variables are initialized when a prey is created in a herd class constructor and exist until a prey is destroyed.

Explanation for Class

There are four classes utilized in this application. The first two classes are codes from other programs, and the other two classes are prepared for this simulation. The first two classes are from Fortune's algorithm for performing triangulation, and from J-P Moreau's program for finding roots of an equation of degree 2, 3, and 4. The original code from other programs are modified for this application. The prey and herd classes are created for this aplication.

- Fortune Class

```
class Fortune{
private:

public:
  Fortune();
  int FortuneAlgo(position *allpos, triangle *head);
  void readsites(position *allpos) ;              /* from voronoi.c */
  void voronoi(Site *(*)(), triangle *head) ;     /* from voronoi.c */
};
```

This class is from a C program written by Steve J. Fortune (1987) and the code can be downloaded (Brook, 1999). The code is for the sweep line algorithm for Voronoi diagrams (Fortune, 1987). Minor points are modified for the input, output, and main functions. The majority of the main function was moved to a function called "FortuneAlgo", which is the only function to be called from other classes. This function is called when triangulation is needed. In this project, triangulation is needed in the Herd class. In order to pass prey's positions to the `FortuneAlgo()` function for input, a temporary array of position structure is used. A pointer of type "triangle structure" will be also passed to this function, and the result of this class will be next pointer of the triangle structure, which is a linked list.

- Root Class

```
class Root{
private:
  double a[5][5], r[5];
  int    i,n;
  double aa,b,c,d,ii,im,k,l,m,q,rr,s,sw;

public:
```

```
  Root();
  double f(double x);
  void Root_2();
  void Root_3();
  void Root_4();
  void CalcRoot(double x4, double x3, double x2, double x1, double x0,
      int *num_ans, double *smallest);
};
```

This class is from a C++ program written by J-P Moreau (Moreau, 2002). The reason that this program is utilized here is because the code is easily extendable due to the nature of the language used (C++). This is used to calculate the real or complex roots of algebraic equations of degrees 2, 3, and 4. To find when a given event will happen, an equation of degree 4 will be solved by this class. In this project, coefficients of time t's degree 4 are small in the usual case. Those small coefficients are ignored and this equation will be solved as an equation of degree 3. If coefficients are between -0.001 and 0.001, those coefficients are ignored.

- Prey Class

```
class Prey{

private:
  position pos, onestep;
  int myindex, neinum;
  int neighbors[PREYNUM];

public:
  Prey();
  void SetPosition(double x, double y);
  void SetIndex(int indx);
```

```
  void SetNeighbors(triangle *head);
  void CopyNeighbors(int max, int *inordernei);
  void SetClosest(double xclosest, double yclosest);
  void GetNeighbors(int *max, int **nei);
  void GetPosition(position *returnpos);
  void GetOnestep(position *returnpos);
  void Move();
  void PrintPos(void);
};
```

This class is for the prey. Prey must store their own position and desitination. In this project, prey have two position structures, which consist of their current positions and their "one steps". The "one step" contains x and y coordinates for one motion. In this study, the normalized vector from a prey's position to its nearest neighbor's postion is set up as the "one step". Prey also have an array of integers to store any neighbors's information. Prey are numbered from 0 to a given max number. The max number of prey is defined as "PREYNUM" in file "myheader.hpp". The prey class has five functions to set their private variables. These functions are to set their positions, indices, neighbor's triangulation information, neighbor's array, which is passed by argument to neighbors array in private variables, and to set "one step", which is their closest neighbor's direction. In order to get private variables, three functions are prepared, to get neighbors, positions, and to get "one step". Also, functions are available for printing their position, and for moving one step.

- Herd Class

```
class Herd{
private:
  Prey member[PREYNUM];
  double areaofdanger[PREYNUM];
  double old_areaofdanger[PREYNUM];
```

```
  eventque eventhead;

public:
  Herd();
  void SetAllClosest();
  void SetAreaOfDanger();
  void SetAllNeighbors(triangle *head);
  void SetEventQueue(triangle *head);
  void GetAllPosition(position *allpos);
  void GetAllOnestep(position *allpos);
  void MoveAllPrey(void);
  void PrintAllPrey(void);
  void DisplayHerd(void);
};
```

This class is for the herd. The herd stores an array of prey. It also has two arrays which store floating points, the first of which is for the current area of danger, which is the area of each prey's Voronoi diagram. The second is for the area of danger before the last event so that the herd can ascertain the difference between the new and old area of danger. The last private variable is an event queue structure which is constructed with a linked list. If the next pointer of the head event queue is NULL, the event queue is considered to be empty. To initialize the herd, which includes producing prey, the Herd constructor is prepared. Three functions are for setting area of danger, neighbors of all prey, and the event queue. To get information from private variables in the Herd class, two functions are written for getting positions of all prey, and getting the "one step" of all prey. In the simulation, prey are moved one step by a function called `MoveAllPrey()`. The last function is to print all prey positions.

Global Functions

These functions are defined globally. The function `Randomnum()` returns a random integer. The function `Distance()` calculates distance from one position to another. The function `NormToward()` normalizes a vector. Other functions explained in this section it to handle triangle linked list, and to handle an event queue.

```
int Randomnum(void){
  srand(time(NULL));

  return(rand());
}
```

This function simply returns a random integer. This is called when the prey's positions are initialized. In this function, srand and rand functions are called to get random numbers at any time. Simulators can specifically define this number if they want to use particular initial positions of prey.

```
  double Distance(position *from, position *to);
```

This function calculates distance from one position to another. The arguments are two position structure pointers and the returned value is a floating point value. When a prey picks the nearest neighbor among his neighbors, a distance between the prey and his neighbors will be calculated using this function.

```
  void   NormToward(position *from, position *to, position *one)
```

This function normalizes a vector. The arguments are three positional structure pointers. A vector from position "from" to position "to" can be created, and this is the vector to be normalized. The normalized vector will be stored where the position structure pointer "one" is pointing. This function is called when prey's "one step" is calculated. The "one step" can be calculated, when the prey's position is inserted into the "from" argument and the prey's neighbor's position is inserted into the "to" argument.

- `void AddNextTri(triangle *head, int bot, int top, int right);`
- `int  DelNextTri(triangle *head);`
- `void DelAllTri(triangle *head);`

The function `AddNextTri()` adds a triangle structure into triangle linked list. First, it allocates the space for triangle data structure. Second, it puts all three indices of prey into three integer variables in the triangle data structure. Then, this new triangle structure will be stored immediately after the "head" triangle structure. This function will be called when Delaunay triangulation has taken place and the result of the triangulation will be stored in the triangle linked list.

The function `DelNextTri()` deletes a triangle structure which the "head" triangle structure's "next" pointer is pointing to. This function will be called by a function `DelAllTri()` repeatedly.

The function `DelAllTri()` deletes an entire triangle linked list. After a calculation of topological structure, such as neighbors, closest neighbor, and a calculation of possible events, the triangle linked list will be entirely deleted by use of this function.

- `void AddInOrderEvent(eventque *head, int a1, int a2, int a3,`
  `int b1, int b2, int b3, double time);`
- `void AddNextEvent(eventque *head, int a1, int a2, int a3,`
  `int b1, int b2, int b3, double time);`
- `int  DelNextEvent(eventque *head);`
- `void DelAllEvent(eventque *head);`

The function `AddInOrderEvent()` adds an event into priority queue according to the variable "time". The "time" variable represents how many steps later an event will occur. Two triangles relate to an event, and are stored in eventque data structure with time variable. This function will be called when a new event needs to be added to the event queue.

The function `AddNextEvent()` adds a new event into the eventque linked list. A space for a new eventque structure will be allocated and the new eventque structure will

be stored in the linked list, where the "head" of eventque's "next" pointer is pointing to. This function is called by `AddInOrderEvent()`.

The function `DelNextEvent()` deletes an event, which "head" eventque structure's next pointer is pointing to. It will be called by the function `DelAllEvent()` repeatedly.

The function `DelAllEvent()` deletes a eventque linked list. When retriangulation needs to occur, an event queue is deleted using this function. Also, events which have already occured need to be deleted.

Global Functions to Find When an Event Occurs

```
- double coefficient_t_4(double xa, double xb, double xc, double xd,
                         double ya, double yb, double yc, double yd,
                         double xva, double xvb, double xvc, double xvd,
                         double yva, double yvb, double yvc, double yvd);
- double coefficient_t_3(double xa, double xb, double xc, double xd,
                         double ya, double yb, double yc, double yd,
                         double xva, double xvb, double xvc, double xvd,
                         double yva, double yvb, double yvc, double yvd);
- double coefficient_t_2(double xa, double xb, double xc, double xd,
                         double ya, double yb, double yc, double yd,
                         double xva, double xvb, double xvc, double xvd,
                         double yva, double yvb, double yvc, double yvd);
- double coefficient_t_1(double xa, double xb, double xc, double xd,
                         double ya, double yb, double yc, double yd,
                         double xva, double xvb, double xvc, double xvd,
                         double yva, double yvb, double yvc, double yvd);
- double coefficient_t_0(double xa, double xb, double xc, double xd,
                         double ya, double yb, double yc, double yd,
                         double xva, double xvb, double xvc, double xvd,
                         double yva, double yvb, double yvc, double yvd);
```

Let t be the time when an event will occur. The time t can be found using the determinant $Incircle(A, B, C, D)$. Four prey's positions are (x_a, y_a), (x_b, y_b), (x_c, y_c), and (x_d, y_d). Each prey moves straight toward its closest neighbor and each prey's velocity is (x_{va}, y_{va}), (x_{vb}, y_{vb}), (x_{vc}, y_{vc}), and (x_{vd}, y_{vd}) respectively. This velocity is each prey's "one step" variable, which is a private variable located in the prey class. The $Incircle(A, B, C, D)$ will be equation 2.

When an event occurs, the determinant will be 0. This equation can be solved by using the root class. Each degree's coefficients need to be passed to the member function CalcRoot() to solve an equation. Functions `coefficient_t_4()`, `coefficient_t_3()`, `coefficient_t_2()`, `coefficient_t_1()`, `coefficient_t_0()` are to retrieve the coefficients of t^4, t^3, t^2, t^1, and t^0 respectively. To expand the determinant $InCircle(A, B, C, D)$, Mathematica was utilized and the result coded in C++.

```
In[18]:=

          mycircle =
           {{xa + xva t, ya + yva t, (xa + xva t)^2 + (ya + yva t)^2, 1},
            {xb + xvb t, yb + yvb t, (xb + xvb t)^2 + (yb + yvb t)^2, 1},
            {xc + xvc t, yc + yvc t, (xc + xvc t)^2 + (yc + yvc t)^2, 1},
            {xd + xvd t, yd + yvd t, (xd + xvd t)^2 + (yd + yvd t)^2, 1}}

Out[18]=

          {{xa + t xva, ya + t yva, (xa + t xva)^2 + (ya + t yva)^2, 1},
           {xb + t xvb, yb + t yvb, (xb + t xvb)^2 + (yb + t yvb)^2, 1},
           {xc + t xvc, yc + t yvc, (xc + t xvc)^2 + (yc + t yvc)^2, 1},
           {xd + t xvd, yd + t yvd, (xd + t xvd)^2 + (yd + t yvd)^2, 1}}
```

Figure 11. Mathematica Result (Define variable)

In figure 11, an array is put into a variable "mycircle" to get the InCircle determinant. The determinant can be gotten a command "Det[mycircle]". Figure 12 shows expanding the determinant and putting it into a variable "Ans". To get formulas for coefficients of t^4, t^3, t^2, t^1, and t^0, commands "Coefficient[Ans, t, 4]", "Coefficient[Ans, t, 3]", "Coefficient[Ans, t, 2]", "Coefficient[Ans, t, 1]", "Coefficient[Ans, t, 0]" will be

In[21]:=

```
Ans = Expand[Det[mycircle]]
```

Out[21]=

$$x_b^2 x_c y_a - x_b x_c^2 y_a - x_b^2 x_d y_a + x_c^2 x_d y_a + x_b x_d^2 y_a - x_c x_d^2 y_a +$$
$$2 t x_b x_c x_{vb} y_a - t x_c^2 x_{vb} y_a - 2 t x_b x_d x_{vb} y_a + t x_d^2 x_{vb} y_a +$$
$$t^2 x_c x_{vb}^2 y_a - t^2 x_d x_{vb}^2 y_a + t x_b^2 x_{vc} y_a - 2 t x_b x_c x_{vc} y_a +$$
$$2 t x_c x_d x_{vc} y_a - t x_d^2 x_{vc} y_a + 2 t^2 x_b x_{vb} x_{vc} y_a - 2 t^2 x_c x_{vb} x_{vc} y_a +$$
$$t^3 x_{vb}^2 x_{vc} y_a - t^2 x_b x_{vc}^2 y_a + t^2 x_d x_{vc}^2 y_a - t^3 x_{vb} x_{vc}^2 y_a - t x_b^2 x_{vd} y_a +$$
$$t x_c^2 x_{vd} y_a + 2 t x_b x_d x_{vd} y_a - 2 t x_c x_d x_{vd} y_a - 2 t^2 x_b x_{vb} x_{vd} y_a +$$
$$2 t^2 x_d x_{vb} x_{vd} y_a - t^3 x_{vb}^2 x_{vd} y_a + 2 t^2 x_c x_{vc} x_{vd} y_a -$$
$$2 t^2 x_d x_{vc} x_{vd} y_a + t^3 x_{vc}^2 x_{vd} y_a + t^2 x_b x_{vd}^2 y_a - t^2 x_c x_{vd}^2 y_a +$$
$$t^3 x_{vb} x_{vd}^2 y_a - t^3 x_{vc} x_{vd}^2 y_a - x_a^2 x_c y_b + x_a x_c^2 y_b + x_a^2 x_d y_b -$$
$$x_c^2 x_d y_b - x_a x_d^2 y_b + x_c x_d^2 y_b - 2 t x_a x_c x_{va} y_b + t x_c^2 x_{va} y_b +$$
$$2 t x_a x_d x_{va} y_b - t x_d^2 x_{va} y_b - t^2 x_c x_{va}^2 y_b + t^2 x_d x_{va}^2 y_b - t x_a^2 x_{vc} y_b +$$
$$2 t x_a x_c x_{vc} y_b - 2 t x_c x_d x_{vc} y_b + t x_d^2 x_{vc} y_b - 2 t^2 x_a x_{va} x_{vc} y_b +$$
$$2 t^2 x_c x_{va} x_{vc} y_b - t^3 x_{va}^2 x_{vc} y_b + t^2 x_a x_{vc}^2 y_b - t^2 x_d x_{vc}^2 y_b +$$
$$t^3 x_{va} x_{vc}^2 y_b + t x_a^2 x_{vd} y_b - t x_c^2 x_{vd} y_b - 2 t x_a x_d x_{vd} y_b +$$
$$2 t x_c x_d x_{vd} y_b + 2 t^2 x_a x_{va} x_{vd} y_b - 2 t^2 x_d x_{va} x_{vd} y_b + t^3 x_{va}^2 x_{vd} y_b -$$
$$2 t^2 x_c x_{vc} x_{vd} y_b + 2 t^2 x_d x_{vc} x_{vd} y_b - t^3 x_{vc}^2 x_{vd} y_b - t^2 x_a x_{vd}^2 y_b +$$
$$t^2 x_c x_{vd}^2 y_b - t^3 x_{va} x_{vd}^2 y_b + t^3 x_{vc} x_{vd}^2 y_b - x_c y_a^2 y_b + x_d y_a^2 y_b -$$
$$t x_{vc} y_a^2 y_b + t x_{vd} y_a^2 y_b + x_c y_a y_b^2 - x_d y_a y_b^2 + t x_{vc} y_a y_b^2 -$$
$$t x_{vd} y_a y_b^2 + x_a^2 x_b y_c - x_a x_b^2 y_c - x_a^2 x_d y_c + x_b^2 x_d y_c + x_a x_d^2 y_c -$$
$$x_b x_d^2 y_c + 2 t x_a x_b x_{va} y_c - t x_b^2 x_{va} y_c - 2 t x_a x_d x_{va} y_c +$$

Figure 12. Mathematica Result (Expand determinant)

utilized, one of which is shown in figure 13.

Member Functions of Prey Class

The function `SetNeighbors()` in the prey class is described in this section. This function is used both for the Kinetic and for Static Voronoi algorithms.

```
void Prey::SetNeighbors(triangle *head)
```

This is a member function of the Prey class. The linked list of the triangle data structures will be used after getting the head pointer of the triangle structure, which is pointing to head of the linked list. From the linked list, this function will find triangles,

In[22]:=

```
Coefficient[Rhs, t, 4]
```

Out[22]=

$$
\begin{aligned}
&X_{vb}^2 X_{vc}\, Y_{va} - X_{vb} X_{vc}^2\, Y_{va} - X_{vb}^2 X_{vd}\, Y_{va} + X_{vc}^2 X_{vd}\, Y_{va} + \\
&X_{vb} X_{vd}^2\, Y_{va} - X_{vc} X_{vd}^2\, Y_{va} - X_{va}^2 X_{vc}\, Y_{vb} + X_{va} X_{vc}^2\, Y_{vb} + \\
&X_{va}^2 X_{vd}\, Y_{vb} - X_{vc}^2 X_{vd}\, Y_{vb} - X_{va} X_{vd}^2\, Y_{vb} + X_{vc} X_{vd}^2\, Y_{vb} - X_{vc}\, Y_{va}^2\, Y_{vb} + \\
&X_{vd}\, Y_{va}^2\, Y_{vb} + X_{vc}\, Y_{va}\, Y_{vb}^2 - X_{vd}\, Y_{va}\, Y_{vb}^2 + X_{va}^2 X_{vb}\, Y_{vc} - X_{va} X_{vb}^2\, Y_{vc} - \\
&X_{va}^2 X_{vd}\, Y_{vc} + X_{vb}^2 X_{vd}\, Y_{vc} + X_{va} X_{vd}^2\, Y_{vc} - X_{vb} X_{vd}^2\, Y_{vc} + X_{vb}\, Y_{va}^2\, Y_{vc} - \\
&X_{vd}\, Y_{va}^2\, Y_{vc} - X_{va}\, Y_{vb}^2\, Y_{vc} + X_{vd}\, Y_{vb}^2\, Y_{vc} - X_{vb}\, Y_{va}\, Y_{vc}^2 + X_{vd}\, Y_{va}\, Y_{vc}^2 + \\
&X_{va}\, Y_{vb}\, Y_{vc}^2 - X_{vd}\, Y_{vb}\, Y_{vc}^2 - X_{va}^2 X_{vb}\, Y_{vd} + X_{va} X_{vb}^2\, Y_{vd} + X_{va}^2 X_{vc}\, Y_{vd} - \\
&X_{vb}^2 X_{vc}\, Y_{vd} - X_{va} X_{vc}^2\, Y_{vd} + X_{vb} X_{vc}^2\, Y_{vd} - X_{vb} Y_{va}^2\, Y_{vd} + X_{vc}\, Y_{va}^2\, Y_{vd} + \\
&X_{va}\, Y_{vb}^2\, Y_{vd} - X_{vc}\, Y_{vb}^2\, Y_{vd} - X_{va}\, Y_{vc}^2\, Y_{vd} + X_{vb}\, Y_{vc}^2\, Y_{vd} + X_{vb}\, Y_{va}\, Y_{vd}^2 - \\
&X_{vc}\, Y_{va}\, Y_{vd}^2 - X_{va}\, Y_{vb}\, Y_{vd}^2 + X_{vc}\, Y_{vb}\, Y_{vd}^2 + X_{va}\, Y_{vc}\, Y_{vd}^2 - X_{vb}\, Y_{vc}\, Y_{vd}^2
\end{aligned}
$$

Figure 13. Mathematica Result (Coefficient of degree 4)

including the involving prey as one of the three points. Because the other two points will be neighbors of this prey, they are put into a temporary array if they are not currently stored in it. After checking all triangles from the linked list, the temporary array will be copied to the private array in prey class by using the CopyNeighbors member function of the prey class.

Member Functions of Herd Class

The functions `SetAllClosest()`, `SetEventQueue()`, `MoveAllPrey()`, and `MoveAllPreyStatic()` in the herd class are explained in this section. The function `MoveAllPrey()` is only for Kinetic Voronoi algorithm and the function `MoveAllPreyStatic()` is only for the Static Voronoi algorithm. The other two functions `SetAllClosest()` and `SetEventQueue()` are for both the Kinetic and Static Voronoi algorithms.

```
void Herd::SetAllClosest()
```

This is a member function of the Herd class. This function will be called after prey's neighbors are set. The neighbors will be set in clockwise order before setting the closest

prey. Then direction of movement of the prey will be set to the location of the closest neighbor. Neighbors will be divided into two sets, upper and lower parts of the prey. Each vector from the prey to neighbors will be calculated and will be normalized. The upper and lower neighbors, in order, can be read according to the x coordinates of the normalized vectors. After sorting them, the lower neighbors list will be added to the end of the upper list so that neighbors are listed in clockwise order. This list will be copied to a private array in the prey. After setting the neighbors in order, the prey's closest immediate neighbor will be set. The distance from the prey to the neighbors will be calculated and the neighbor with the shortest distance will be picked for the closest neighbor. The neighbor's position is passed to the member function SetClosest of the prey class. In this function, the vector from the prey to the neighbor is normalized and the normalized vector will be set to the "one step" of the prey. This is how all prey's closest neighbors are defined.

```
void Herd::SetAreaOfDanger()
```

This function is a member function of the Herd class. This function will be called every time the prey's area of danger needs to be calculated. When this function is called, the private array areaofdanger will be set. The private array `old_areaofdanger` stores the previous information of areaofdanger. For calculating the Voronoi cell, Voronoi vertices are needed. By using the function FindCenter() with the position of the prey itself and the neighbors' positions, all Voronoi vertices will be calculated. To calculate Voronoi cell area, Voronoi vertices will be set in clockwise order. After calculating the Voronoi cell area, it will be stored as an area of danger.

To calculate the Voronoi cell area from Voronoi vertices, consider Voronoi vertices $A = (x_1, y_1)$, $B = (x_2, y_2)$, $C = (x_3, y_3)$, $D = (x_4, y_4)$, and $E = (x_5, y_5)$. And also, there are $A' = (x_1, 0)$, $B' = (x_2, 0)$, $C' = (x_3, 0)$, $D' = (x_4, 0)$, and $E' = (x_5, 0)$. Voronoi vertices A, B, C, D, and E form a Voronoi cell, shown on figure 14.

There are trapezoids $T1 = A, B, A', B'$, $T2 = B, C, B', C'$, $T3 = C, D, C', D'$, $T4 = D, E, D', E'$, and $T5 = E, A, E', A'$. The area of the Voronoi cell can be calculated

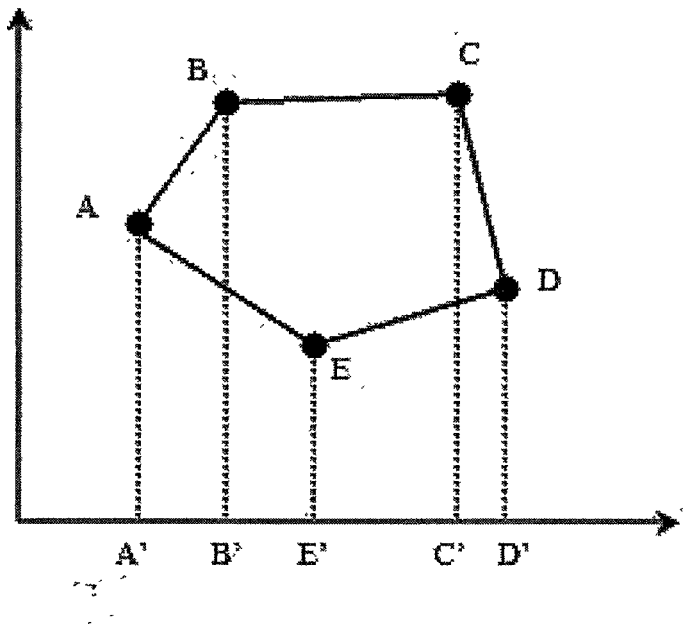

Figure 14. Calculate Area of Voronoi Cell

by addition of the areas of $T1$, $T2$, and $T3$ and subtraction of the areas of $T4$, and $T5$. This can be calculated by one formula 5.

$$VoronoiArea = \sum_{i=1}^{n-1} \frac{(x_{i+1} - x_i)(y_i + y_{i+1})}{2} + \frac{(x_1 - x_n)(y_n + y_1)}{2} \tag{5}$$

`void Herd::SetEventQueue(triangle *head)`

This function is a member function of Herd class. This function is called when the event queue is initialized or when retriangulation is performed. This function will find all possible events and put them into priority queue. First, it will find all pairs of adjacent triangles from triangle information, which is passed by an argument. When a pair of adjacent triangles is found, an equation $InCircle(A, B, C, D) = 0$, where A, B, C, and D are the points of triangles, will be calculated. When time t is found, this new event is inserted into a priority queue using `AddInOrderEvent()` function. After all possible events are found, the function `SetEventQueue()` is terminated.

`void Herd::MoveAllPrey(void)`

This is a member function of the Herd class. This is used for a Kinetic Voronoi simulation. This function is called when all prey move one complete simulation step. This function is called repeatedly until the simulation is finished. First, it lets all prey move using a member function Move() of the prey class, and prints all prey positions. All "time" variables in the event queue need to be updated so that at this moment, we will know when those events will occur. After "time" variable's updates, this function checks if any events have already occured. If the "time" variable is negative, this event has already occured. If any event does not occur, it calculates "area of danger" using `SetAreaOfDanger()`. Otherwise, The topological structure and an event queue will be updated. For updating the topological structure and an event queue, retriangulation or partial update, SWAP, needs to be done. In retriangulation, an event queue, neighbors information, and the closest neighbors will be set up again as they are initialized. The neighbors information, and the closest neighbors are overwritten, and the old event queue is deleted. In partial update neighbors of the four prey which are related to the event are changed. For example, if triangle $T1(A, B, C)$ and $T2(B, C, D)$ become $T3(A, B, D)$ and $T4(A, C, D)$, A's neighbors will be B and D insted of B and C, and so on. After neighbors are updated, their closest neighbor will be calculated. From the new triangles, an event will be created and the timing when it will occur will be calculated. If the event could occur based on the time, the event will be added to the event queue. After adding the event, an event, which has already occured, need to be deleted. `SetAreaOfDanger()` will be called to calculate "area of danger". The pseudocode of this function is shown in figure 15.

`void Herd::MoveAllPreyStatic(void)`

This is a member function of the Herd class. This is used for the Static Voronoi simulation. The function is called when all prey move one complete simulation step. This function is similar to MoveAllPrey(), except without event queue updates. First, it lets all prey move one step by using a member function Move() in the prey class. Then, all prey

```
MoveAllPrey(){
  For all prey, call Move()
  Call PrintAllPrey()
  Update Time variable in Event Queue

  If events occur,
    Perform SWAP or triangulation.
    Update topological structure
    Update Event Queue
  End of if

  Call SetAreaOfDanger()
}
```

Figure 15. Function MoveAllPrey

positions are printed out. After that, triangulation will take place in order to update the neighbors information and that of its closest neighbors. After the updating, "area of danger" will be calculated by using `SetAreaOfDanger()`. The pseudocode of this function is shown in figure 16.

```
MoveAllPreyStatic(){
  For all prey, call Move()
  Call PrintAllPrey()

  Perform triangulation
  Update topological structure

  Call SetAreaOfDanger()
}
```

Figure 16. Function MoveAllPreyStatic

EXPERIMENT

Experiment Explanation

The purpose of this experiment is to find out if the Kinetic Voronoi algorithm is as useful in application as the Static Voronoi algorithm, with regard to the Selfish Herd problem. The data sets of the first experiments are shown on the table 1.

Table 1

Data Set of the First Experiments

Simulation field	200x200
The number of prey	6, 9, 12, 15, 18, 21, 24, 27, 30
Simulation field	500x500
The number of prey	6, 9, 12, 15, 18, 21, 24, 27, 30

In these experiments, all prey move to their nearest neighbors. In the Static algorithm, the nearest neighbors of a given prey are decided when triangulation is performed, which means that the nearest neighbors are updated after every simulation step. In the Kinetic algorithm, the nearest neighbors are updated when triangulation is performed, or when an elementary event SWAP has occured. When triangulation is performed, all nearest neighbors are updated. However, a local update will be done when an elementary event SWAP occurs. Their closest neighbors are not updated when no events have occured.

The Static program outputs an area of danger for each prey, and the prey's positions after every simulation step. The Kinetic program outputs an area of danger for

each prey, the event list which is in order according to timing, and the prey's position after every simulation step.

Experiments are observed for 20 simulation steps because 20 steps are enough to see a general spread of events occuring.

Constraints for Kinetic Voronoi Algorithm

One of the important constraints for the Kinetic Voronoi algorithm is that all prey move to where their closest neighbors were located during the previous event. Because all prey are in motion, their targets are also in motion. However, in this application, all prey have the location where their closest neighbors were when the triangulation had taken place. The reason is that it is difficult to keep track of where the closest neighbors will be moving.

Another of the constraints for the Kinetic Voronoi algorithm is that a linked list is utilized for a priority queue which is an event queue. The time complexity of queue operations could be improved by use of a more complex data structure.

Result

The tables 2 and 3 are showing prey's indices, initial area of danger, and area of danger after 20 simulation steps in a simulation with 6 prey in field 200x200.

Table 2

Kinetic 6 prey 200x200

prey index	initial area of danger	area of danger after 20 steps
0	3600.35	6500.46
1	1000000.00	1000000.00
2	1538.29	101731.60
3	838.62	87.36
4	48.54	22.71
5	4487.54	4281.81

Table 3

Static 6 prey 200x200

prey index	initial area of danger	area of danger after 20 steps
0	3600.35	3804.14
1	1000000.00	1000000.00
2	1538.29	5110.40
3	838.62	75.40
4	48.54	25.29
5	4487.54	5231.56

Some areas are bigger than the simulation field. This means that the prey is on the convex hull and in this simulation this case is ignored. In this case, the prey 1's area of danger will be ignored in both algorithm. In the Kinetic algorithm, the prey 0's and 2's areas of danger have increased after 20 simulation steps. And the prey 3, 4, and 5 can reduce their areas of danger. In the Static algorithm, the prey 0, 2, and 5 increased their areas of danger and the prey 3, and 4 reduced their areas of danger.

The tables 4, 5, 6, and 7 are showing prey's indices, initial area of danger, and area of danger after 20 simulation steps in a simulation with 30 prey in field 500x500.

Table 4

Kinetic Voronoi 30 prey 500x500 (prey index from 0 to 14)

prey index	initial area of danger	area of danger after 20 steps
0	9422.15	7782.69
1	14984.97	17235.93
2	740138.18	329007.12
3	7752.34	7678.06
4	90285.35	112679.32
5	4980.92	7472.20
6	83382.73	45886.61
7	7683.28	4621.69
8	2088693.79	1006528.24
9	9884.99	9236.96
10	8006.70	7840.13
11	135883.17	89060.76
12	85166.49	-118053.10
13	45107.87	-1184118.03
14	16877.64	24088.45

The tables 8 and 9 are showing which prey have smaller areas of danger after 20 simulation steps in the Kinetic and Static Voronoi algorithms. The table 8 is calculated from the tables 4 and 5, and the table 9 is from the tables 6 and 7.

Although only two cases of first experiments are shown here, the Kinetic algorithm outputs almost the same result as the Static algorithm in all simulations. That means that the Kinetic algorithm can be applied to Selfish Herd problems.

Table 5

Kinetic Voronoi 30 prey 500x500 (prey index from 15 to 29)

prey index	initial area of danger	area of danger after 20 steps
15	8121.67	8472.85
16	7471.85	7744.21
17	6750.51	4975.86
18	9626.74	13380.17
19	334684.26	342788.00
20	12285.61	12696.41
21	180624.19	137591.71
22	7266.81	9743.90
23	6567.95	5331.97
24	279872.94	1028762.51
25	101401.76	60208.60
26	6778.96	6213.94
27	14653.31	13647.53
28	7879.07	7234.49
29	22798.92	24533.38

Table 6

Static Voronoi 30 prey 500x500 (prey index from 0 to 14)

prey index	initial area of danger	area of danger after 20 steps
0	9422.15	7780.59
1	14984.97	17238.09
2	740138.18	322871.51
3	7752.34	7627.75
4	90285.35	112295.86
5	4980.92	7819.25
6	83382.73	6285.94
7	7683.28	4620.90
8	2088693.79	7345.66
9	9884.99	9236.73
10	8006.70	7100.57
11	135883.17	87021.09
12	85166.49	1117.03
13	45107.87	726430.91
14	16877.64	24148.16

Table 7

Static Voronoi 30 prey 500x500 (prey index from 15 to 29)

prey index	initial area of danger	area of danger after 20 steps
15	8121.67	8445.45
16	7471.85	7740.12
17	6750.51	4917.82
18	9626.74	13452.37
19	334684.26	293983.65
20	12285.61	12697.34
21	180624.19	116099.47
22	7266.81	53604.94
23	6567.95	5332.51
24	279872.94	141105.43
25	101401.76	60165.64
26	6778.96	46004.60
27	14653.31	13581.78
28	7879.07	845562.80
29	22798.92	24533.66

Table 8

Prey Which Reduce or Increase Their Area of Danger (Kinetic)

Reduce	0, 2, 3, 6, 7, 8, 9, 10, 17, 21, 23, 25, 26, 27, 28
Increase	1, 4, 5, 14, 15, 16, 18, 19, 20, 22, 24, 29
Ignore	12, 13

Table 9

Prey Which Reduce or Increase Their Area of Danger (Static)

Reduce	0, 2, 3, 6, 7, 8, 9, 10, 11, 12, 16, 17, 19, 21, 23, 24, 25, 27
Increase	1, 4, 5, 13, 14, 15, 18, 20, 22, 26, 28, 29

COMPARISON WITH STATIC AND KINETIC

Explanation of Experiment

This experiment used the first experiment's data sets (6, 9, ..., 30) in the simulation field of 200x200 and 500x500 with additional data sets 100 and 1000 prey in the simulation field of 1000x1000. This experiment does not require animation because it compares with regard to the speeds of Static and Kinetic Voronoi algorithms.

In this experiment, the timing of how long it takes to execute one simulation step with regard to Static and Kinetic algorithms will be recorded 20 times, and the average of the timings will be calculated.

Timing Result of Experiment

The averages of the timing will be shown on the table 10. In most cases, the Static algorithm is faster than the Kinetic algorithm.

However, for smaller number of data sets (6 and 9), the Kinetic algorithm is faster than the Static algorithm. The table 11 shows the timing result of the simulations with 6 prey in 200x200 field.

When the Kinetic algorithm takes longer to proceed one simulation step than the Static algorithm, for example, step 16 or 17, the Kinetic algorithm is performing triangulation in addition to handling an event queue. In such a case, it is obvious that the Kinetic algorithm is slower than the Static algorithm because of the additional procedures.

When the Kinetic algorithm is faster than the Static one, the Kinetic algorithm does not perform a triangulation. If an event occurs, the Kinetic program is performing a

Table 10

Averages of the Timings

Data set	200x200 Kinetic	Static	500x500 Kinetic	Static	1000x1000 Kinetic	Static
6	167.77	220.11	109	239.83		
9	769.27	382.16	144.83	369.61		
12	1122.5	535.83	638.88	525		
15	1431.94	718.27	875.94	733.05		
18	1811.66	859.16	1382.16	894.16		
21	2317.77	1039.72	1495.94	1062.61		
24	2770.94	1237.22	1893.77	1213.38		
27	3091.27	1383.44	3111.16	1448.5		
30	3685.38	1596	3410.38	1616.38		
100					12776.44	3504.83
1000					328330.38	98608.38

SWAP update in addition to handling an event queue. Otherwise, the Kinetic program does not do anything except handling an event queue.

We can also observe that in Kinetic algorithm, a simulation with a larger data set, for example 1000 prey, performed more triangulations than a simulation with a smaller data set of 6 prey during 20 simulation steps.

Result of the Gprof Command

Other outputs are gleaned from the command "gprof". The command "gprof" is to know which functions are called, how many times those functions are called, and how long they take to reach completion. The result of the gprof show which portion of the program takes more time to be executed than other portions. Therefore, if the slower protion of the

program is improved, the program will be improved effectively (Foundation, 1998).

To get the output of the "gprof" command, when the programs are compiled, the "-pg" switch needs to be added to the "gprof" command. After compiling and executing the programs profiles will be produced using a command such as, "gprof ./kvoronoi > 6kprof.out". On the command line, the program name of the profile we want to get will be followed by the command name "gprof". The output will be stored in the file "6kprof.out" in this example.

The following is a portion of result with 1000 prey in simulation field 1000x1000 from the Kinetic algorithm.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 25.00      0.01     0.01   364156     0.00     0.00  Root::f(double)
 25.00      0.02     0.01     5735     0.00     0.00  coefficient_t_3(
double, double, ..., double, double)
 25.00      0.03     0.01     5732     0.00     0.00  Root::Root_3()
 25.00      0.04     0.01     2100     0.00     0.00  Prey::SetNeighbors(
mytriangle*)
  0.00      0.04     0.00    37998     0.00     0.00  Distance(myposition*,
 myposition*)
  0.00      0.04     0.00    30563     0.00     0.00  ref(tagSite*)
  0.00      0.04     0.00    26026     0.00     0.00  NormToward(
myposition*, myposition*, myposition*)
  0.00      0.04     0.00    21895     0.00     0.00  makefree(
tagFreenode*, tagFreelist*)
  0.00      0.04     0.00    20963     0.00     0.00  getfree(tagFreelist*)
  0.00      0.04     0.00    18340     0.00     0.00  deref(tagSite*)
```

The following is part of result with 1000 prey in simulation field 1000x1000 from the Static algorithm.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
50.00	0.01	0.01	11968	0.84	0.84	FindCenter(double, double, ..., double*)
50.00	0.02	0.01	21	476.19	952.38	Herd::SetAreaOfDanger()
0.00	0.02	0.00	38012	0.00	0.00	Distance(myposition*, myposition*)
0.00	0.02	0.00	30563	0.00	0.00	ref(tagSite*)
0.00	0.02	0.00	26040	0.00	0.00	NormToward(myposition*, myposition*, myposition*)
0.00	0.02	0.00	21895	0.00	0.00	makefree(tagFreenode*, tagFreelist*)
0.00	0.02	0.00	20963	0.00	0.00	getfree(tagFreelist*)
0.00	0.02	0.00	18340	0.00	0.00	deref(tagSite*)
0.00	0.02	0.00	15494	0.00	0.00	PQempty()
0.00	0.02	0.00	11972	0.00	0.00	intersect(tagHalfedge*, tagHalfedge*)

In the Kinetic algorithm, functions in Root class are called more often than other function. To find when an event occur, the root class is used and if those functions are improved, the Kinetic algorithm will be improved.

Table 11

Timing Result (6 prey in 200x200)

steps	Kinetic	Static
1	97.000	264.000
2	94.000	214.000
3	186.000	221.000
4	185.000	224.000
5	169.000	248.000
6	90.000	215.000
7	88.000	222.000
8	89.000	217.000
9	87.000	221.000
10	88.000	214.000
11	88.000	212.000
12	90.000	222.000
13	90.000	214.000
14	89.000	221.000
15	86.000	215.000
16	382.000	222.000
17	587.000	220.000
18	384.000	216.000
19	361.000	214.000
20	363.000	222.000

CONCLUSION

This paper proposed applying the Kinetic Voronoi Diagram algorithm to the Selfish Herd problem. The experiments showed that the Kinetic algorithm can apply to the Selfish Herd problem. From the timing results of the experiments, there are some improvements which can be implemented. If the number of occasions triangulation are reduced, the Kinetic algorithm is faster than the application presented in this paper.

There are four suggestions to improve the Kinetic Voronoi algorithm. The first suggestion is that once a single group of prey converges, they move to the other closest neighbor as a single prey. The second suggestion is moving to their closest neighbors instead of moving to where their closest used to be. The third suggestion is using more efficient data structure for priority queue, for example, by using heap instead of by using linked list. The fourth suggesstion is improving functions to find roots of equation of 4 th degree. These suggestions might let the execution time of Kinetic algorithm for one simulation step shorter than the application presented in this paper.

APPENDIX A (SOURCE CODES FOR PREY CLASS)

In this section, `prey.hpp` and `prey.cpp` are listed, and they are for prey class.

prey.hpp

```
#ifndef _PREY_H
#define _PREY_H

#include "myheader.hpp"

class Prey{

private:
  position pos, /*destination,*/ onestep;
  int myindex, neinum;
  int neighbors[PREYNUM];

public:
  Prey();
  void SetPosition(double x, double y);
  void SetIndex(int indx);
  void SetNeighbors(triangle *head);
  void CopyNeighbors(int max, int *inordernei);
  void SetClosest(double xclosest, double yclosest);
  void GetNeighbors(int *max, int **nei);
  void GetPosition(position *returnpos);
  void GetOnestep(position *returnpos);
  void Move();
  void PrintPos(void);
};
#endif
```

prey.cpp

```
#include "prey.hpp"

/************************************************/
/* Constructor   */
/************************************************/

Prey::Prey(){
  pos.x = 0;
  pos.y = 0;
  neinum = 0;
  for(int i=0;i<PREYNUM;i++){
    neighbors[i] = EMPTY;
  }
}


/************************************************/
/* SetIndex(int indx)   */
/************************************************/

void Prey::SetIndex(int indx){
  myindex = indx;
}


/************************************************/
/* SetNeighbors()   */
/************************************************/

void Prey::SetNeighbors(triangle *head){
  triangle *temp;
  int in[2], inneighbor = FALSE;
  int temp_neighbors[PREYNUM], temp_num = 0;
```

```
  temp = head->next;

  while(temp != NULL){

    if(myindex == temp->bot || myindex == temp->top
       || myindex == temp->right){

      if(myindex == temp->bot) {
in[0] = temp->top;
in[1] = temp->right;
      }
      if(myindex == temp->top) {
in[0] = temp->bot;
in[1] = temp->right;
      }
      if(myindex == temp->right) {
in[0] = temp->bot;
in[1] = temp->top;
      }

      for(int i=0;i<2;i++){
inneighbor = FALSE;
for(int j=0; j<temp_num;j++){
  if(in[i] == temp_neighbors[j]){
    inneighbor = TRUE;
    j = temp_num;
  }
}

if(inneighbor == FALSE){
  temp_neighbors[temp_num] = in[i];
  temp_num++;
}
      }
    }
    temp = temp->next;
  }

  CopyNeighbors(temp_num, temp_neighbors);
```

```
}

/************************************************/
/* InOrderNeighbors()   */
/************************************************/

void Prey::CopyNeighbors(int max, int *inordernei){
  neinum = max;
  for(int i=0;i<neinum;i++){
    neighbors[i] = inordernei[i];
  }
}



/************************************************/
/* SetClosest()   */
/************************************************/

void Prey::SetClosest(double xclosest, double yclosest){
  position destination;

  destination.x = xclosest;
  destination.y = yclosest;

  /* Call NormToward() to Set onestep */

  NormToward(&pos, &destination, &onestep);
}


/************************************************/
/*  GetNeighbors()  return indexes of Neighbors  */
/************************************************/



void Prey::GetNeighbors(int *max, int **nei){
  *max = neinum;
```

```
  for(int i=0;i<neinum;i++){
    nei[i]=&neighbors[i];
  }
}


/************************************************/
/* SetPosition(double x, double y)   */
/************************************************/

void Prey::SetPosition(double x, double y){
  pos.x = x;
  pos.y = y;
}


/************************************************/
/* PrintPos(void)   */
/************************************************/

void Prey::PrintPos(void){
  printf("Index %2d x = %7.3f  y = %7.3f\n", myindex, pos.x, pos.y);
}


/************************************************/
/*  GetPosition()  return position              */
/************************************************/

void Prey::GetPosition(position *returnpos){
  returnpos->x = pos.x;
  returnpos->y = pos.y;
}


/************************************************/
/*  GetOnestep()  return position               */
/************************************************/
```

```
void Prey::GetOnestep(position *returnpos){
  returnpos->x = onestep.x;
  returnpos->y = onestep.y;
}



/************************************************/
/*  Move()  Prey is moving toward closest       */
/************************************************/

void Prey::Move(){
  pos.x += onestep.x;
  pos.y += onestep.y;
}
```

APPENDIX B (SOURCE CODES FOR HERD CLASS)

In this section, `herd.hpp`, `herd.cpp`, and `herdstatic.cpp` are listed. The files `herd.cpp` and `herd.hpp` are for kinetic Voronoi algorithm, and the files `herdstatic.cpp` and `herdstatic.hpp` are for static Voronoi algorithm.

herd.hpp

```
#ifndef _HERD_H
#define _HERD_H


#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include "prey.hpp"
#include "fortune.hpp"

#ifdef MYOPENGL
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#endif

class Herd{
private:
  Prey member[PREYNUM];
  double areaofdanger[PREYNUM];
  double old_areaofdanger[PREYNUM];
  eventque eventhead;
```

```
public:
  Herd();
  void SetAllClosest();
  void SetAreaOfDanger();
  void SetAllNeighbors(triangle *head);
  void SetEventQueue(triangle *head);
  void GetAllPosition(position *allpos);
  void GetAllOnestep(position *allpos);
  void MoveAllPrey(void);
  void PrintAllPrey(void);
#ifdef MYOPENGL
  void DisplayHerd(void);
#endif
};
#endif
```

herd.cpp

```
#include "herd.hpp"
#include "root.hpp"


/*********************************************************************/
/**  Herd()  constructor                                            **/
/*********************************************************************/


Herd::Herd(){
  Fortune fort;
  position allpos[PREYNUM];
  triangle trihead, *temp;
  eventque *etemp;
  int i, j;

  /*--- fill prey's position and index ---*/
  for(int i=0;i<PREYNUM;i++){
    member[i].SetPosition(Randomnum()%1000,Randomnum()%1000);
```

```
    member[i].SetIndex(i);
    old_areaofdanger[i] = 0;
    areaofdanger[i] = 0;
  }

  eventhead.next = NULL;
  trihead.next  = NULL;


  GetAllPosition(allpos);
  fort.FortuneAlgo(allpos, &trihead);


  // Set Neighbors
  SetAllNeighbors(&trihead);

  SetAllClosest();
  SetAreaOfDanger();
  SetEventQueue(&trihead);
  DelAllTri(&trihead);


  etemp = eventhead.next;

  printf("\n\n+++++++++++++++  Event List  +++++++++++++++\n\n");
  i = 0;
  while(etemp != NULL){
    printf("%3d -- (%2d,%2d,%2d) (%2d,%2d,%2d)  time %12.5f\n",
   i++, etemp->triangle1[0],
   etemp->triangle1[1],etemp->triangle1[2],
   etemp->triangle2[0],etemp->triangle2[1],
   etemp->triangle2[2],etemp->time);
    etemp = etemp->next;
  }
}


/*****************************************************************/
/**  SetAllClosest()                                            **/
```

```
/*********************************************************************/

void Herd::SetAllClosest(){
  int max=0, tempclosest, index, tempin;
  int *neighbors[PREYNUM];
  int tempnei[2][PREYNUM];
  int tempnei2[2][PREYNUM];
  int tempnum[2], inorder[PREYNUM];
  position tempnorm[2][PREYNUM];
  position allpos[PREYNUM], temppos;
  double mindistance, tempdist;


  GetAllPosition(allpos);


  for(int i=0;i<PREYNUM;i++){
    // Get Neighbors and Pick Closest.

    member[i].GetNeighbors(&max, neighbors);


    //////////////     make neighbor list in order
    tempnum[0]=0;
    tempnum[1]=0;

    for(int k=0;k<max;k++){
      if(allpos[i].y < allpos[*neighbors[k]].y){ // upper
tempnei[0][tempnum[0]] = *neighbors[k];
NormToward(&allpos[i],&allpos[*neighbors[k]],&tempnorm[0][tempnum[0]]);
tempnum[0]++;
      }else{                                        // lower
tempnei[1][tempnum[1]] = *neighbors[k];
NormToward(&allpos[i],&allpos[*neighbors[k]],&tempnorm[1][tempnum[1]]);
tempnum[1]++;
      }
    }
```

```
    ///   sort
    for(int n=0;n < 2; n++){
      for(int k=0;k<tempnum[n]-1;k++){
for(int m=k+1;m<tempnum[n];m++){
  if(tempnorm[n][k].x > tempnorm[n][m].x){
    temppos.x = tempnorm[n][k].x;
    temppos.y = tempnorm[n][k].y;
    tempnorm[n][k].x = tempnorm[n][m].x;
    tempnorm[n][k].y = tempnorm[n][m].y;
    tempnorm[n][m].x = temppos.x;
    tempnorm[n][m].y = temppos.y;
    tempin = tempnei[n][k];
    tempnei[n][k] = tempnei[n][m];
    tempnei[n][m] = tempin;
  }
}
      }
    }


    // After sort

    for(int k=0;k<tempnum[0];k++){
      inorder[k] = tempnei[0][k];
    }
    for(int k=0;k<tempnum[1];k++){
      inorder[tempnum[0]+tempnum[1]-k-1] = tempnei[1][k];
    }

    member[i].CopyNeighbors(max,inorder);



    mindistance = 1000000;
    tempclosest = EMPTY;

    for(int j=0;j<max;j++){
      tempdist = Distance(&(allpos[i]), &(allpos[*neighbors[j]]));
```

```
      if(tempdist < mindistance){
tempclosest = j;
mindistance = tempdist;
      }
    }
    member[i].SetClosest((allpos[*neighbors[tempclosest]]).x,
 (allpos[*neighbors[tempclosest]]).y);
  }
}




void Herd::SetAreaOfDanger(){
  int max=0;
  int *neighbors[PREYNUM];
  position allpos[PREYNUM];
  position areapos[PREYNUM];
  position tempareapos[2][PREYNUM];
  position tempnorm[2][PREYNUM];
  position temppos;
  position inorder[PREYNUM];
  int tempnum[2];

  double t;

  GetAllPosition(allpos);

  for(int i=0;i<PREYNUM;i++){
    old_areaofdanger[i] = areaofdanger[i];
    areaofdanger[i] = 0;

    member[i].GetNeighbors(&max, neighbors);


    if(max >= 3){
      for(int j=1;j<max;j++){
FindCenter((allpos[*neighbors[j-1]]).x, (allpos[*neighbors[j-1]]).y,
```

```
   (allpos[*neighbors[j]]).x, (allpos[*neighbors[j]]).y,
   allpos[i].x, allpos[i].y,
   &(areapos[j-1].x),&(areapos[j-1].y));
      }

      FindCenter((allpos[*neighbors[max-1]]).x,
 (allpos[*neighbors[max-1]]).y,
 (allpos[*neighbors[0]]).x, (allpos[*neighbors[0]]).y,
 allpos[i].x, allpos[i].y,
 &(areapos[max-1].x),&(areapos[max-1].y));


      tempnum[0]=0;
      tempnum[1]=0;

      for(int j=0;j<max;j++){
if(allpos[i].y < areapos[j].y){ // upper
  tempareapos[0][tempnum[0]].x = areapos[j].x;
  tempareapos[0][tempnum[0]].y = areapos[j].y;
  NormToward(&allpos[i],&areapos[j],&tempnorm[0][tempnum[0]]);
  tempnum[0]++;
}else{                          // lower
  tempareapos[1][tempnum[1]].x = areapos[j].x;
  tempareapos[1][tempnum[1]].y = areapos[j].y;
  NormToward(&allpos[i],&areapos[j],&tempnorm[1][tempnum[1]]);
  tempnum[1]++;
}

      }

      ///////                Sort   ///////////////
      for(int n=0;n < 2; n++){
for(int k=0;k<tempnum[n]-1;k++){
  for(int m=k+1;m<tempnum[n];m++){
    if(tempnorm[n][k].x > tempnorm[n][m].x){
      temppos.x = tempareapos[n][k].x;
      temppos.y = tempareapos[n][k].y;
      tempareapos[n][k].x = tempareapos[n][m].x;
      tempareapos[n][k].y = tempareapos[n][m].y;
```

```
      tempareapos[n][m].x = temppos.x;
      tempareapos[n][m].y = temppos.y;

      temppos.x = tempnorm[n][k].x;
      temppos.y = tempnorm[n][k].y;
      tempnorm[n][k].x = tempnorm[n][m].x;
      tempnorm[n][k].y = tempnorm[n][m].y;
      tempnorm[n][m].x = temppos.x;
      tempnorm[n][m].y = temppos.y;
    }
  }
}
      }


      for(int k=0;k<tempnum[0];k++){
inorder[k].x = tempareapos[0][k].x;
inorder[k].y = tempareapos[0][k].y;
      }
      for(int k=0;k<tempnum[1];k++){
inorder[tempnum[0]+tempnum[1]-k-1].x = tempareapos[1][k].x;
inorder[tempnum[0]+tempnum[1]-k-1].y = tempareapos[1][k].y;
      }


      for(int j=1;j<max;j++){
t =((inorder[j]).x - (inorder[j-1]).x)
  * ((inorder[j]).y + (inorder[j-1]).y) / 2.0;
areaofdanger[i] += t;
      }
      t =((inorder[0]).x - (inorder[max-1]).x)
  * ((inorder[0]).y + (inorder[max-1]).y) / 2.0;
      areaofdanger[i] += t;


    }else{
      areaofdanger[i] = INFINITE;
    }
  }
```

```
  printf("\nprey  Area of Danger      Difference between\n"
 "                                old and new one \n");
  for(int i=0;i<PREYNUM;i++){
    printf("%2d    %10.2f          %10.2f\n", i, areaofdanger[i],
   areaofdanger[i]-old_areaofdanger[i]);
  }
  printf("\n\n");
}



void Herd::SetAllNeighbors(triangle *head){
  for(int i=0;i<PREYNUM;i++){
    member[i].SetNeighbors(head);
  }
}



void Herd::SetEventQueue(triangle *head){
  int bot, top, right, found, eventindex=0, x1,x2,x3,x4;
  double xx0, xx1, xx2, xx3, xx4;
  double xa, xb, xc, xd, ya, yb, yc, yd;
  double xva, xvb, xvc, xvd, yva, yvb, yvc, yvd;
  position allpos[PREYNUM];
  position allonestep[PREYNUM];
  triangle trihead, *temp1, *temp2, *temp3;
  Root findroot;
  int num_ans;
  double smallest;

  temp1 = head->next;
  GetAllPosition(allpos);
  GetAllOnestep(allonestep);


  while(temp1 != NULL){
```

```
    bot = temp1->bot;
    top = temp1->top;
    right = temp1->right;

    temp2 = temp1->next;
    while(temp2 != NULL){

if(
   ((bot == temp2->bot || bot == temp2->top || bot == temp2->right) &&
   (top == temp2->bot || top == temp2->top || top == temp2->right))
   ||
   ((top == temp2->bot || top == temp2->top || top == temp2->right)&&
    (right == temp2->bot||right == temp2->top||right == temp2->right))
   ||
   ((bot == temp2->bot || bot == temp2->top || bot == temp2->right)&&
    (right == temp2->bot||right == temp2->top||right == temp2->right))){


  x1 = top;
  x2 = bot;
  x3 = right;
  if(top != temp2->top && bot!= temp2->top && right!= temp2->top){
    x4 = temp2->top;
  }else if(top != temp2->bot && bot!= temp2->bot && right!= temp2->bot){
    x4 = temp2->bot;
  }else if(top!=temp2->right&& bot!=temp2->right&& right!=temp2->right){
    x4 = temp2->right;
  }

  xa  = allpos[x1].x;
  ya  = allpos[x1].y;
  xb  = allpos[x2].x;
  yb  = allpos[x2].y;
  xc  = allpos[x3].x;
  yc  = allpos[x3].y;
  xd  = allpos[x4].x;
  yd  = allpos[x4].y;
  xva = allonestep[x1].x;
  yva = allonestep[x1].y;
```

```
  xvb = allonestep[x2].x;
  yvb = allonestep[x2].y;
  xvc = allonestep[x3].x;
  yvc = allonestep[x3].y;
  xvd = allonestep[x4].x;
  yvd = allonestep[x4].y;


  xx4 = coefficient_t_4(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx3 = coefficient_t_3(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx2 = coefficient_t_2(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx1 = coefficient_t_1(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx0 = coefficient_t_0(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);

  findroot.CalcRoot(xx4,xx3,xx2,xx1,xx0, &num_ans, &smallest);

  if(smallest!=MAXNUM){
    AddInOrderEvent(&eventhead, top, bot, right, temp2->top,
    temp2->bot, temp2->right,smallest);
  }
}
      temp2 = temp2->next;
    }
    temp1 = temp1->next;
  }
}




void Herd::GetAllPosition(position *allpos){
  for(int i=0;i<PREYNUM;i++){
    member[i].GetPosition(&allpos[i]);
```

```
  }
}



void Herd::GetAllOnestep(position *allpos){
  for(int i=0;i<PREYNUM;i++){
    member[i].GetOnestep(&allpos[i]);
  }
}



void Herd::MoveAllPrey(void){
  double mindistance, tempdist;
  double xx0, xx1, xx2, xx3, xx4;
  double xa, xb, xc, xd, ya, yb, yc, yd;
  double xva, xvb, xvc, xvd, yva, yvb, yvc, yvd;
  int count, a, b, c, d, i, j, max, after, tempclosest;
  int count_update;
  int *neighbors[PREYNUM], updateneighbor[PREYNUM];
  eventque *updateque, *temp, *etemp;
  position allpos[PREYNUM];
  position allonestep[PREYNUM];
  Root findroot;
  int num_ans;
  double smallest;
  int eventnum=0;



  for(int i=0;i<PREYNUM;i++){
    member[i].Move();
  }
  PrintAllPrey();
```

```
  SetAreaOfDanger();


  if(eventhead.next!=NULL){

    temp = eventhead.next;
    // Updating time variable in the priority queue
    while(temp->next != NULL){
      temp->time--;
      temp = temp->next;
    }

    if(eventhead.next->time < 0){  // When an event occur
      GetAllPosition(allpos);

      updateque = eventhead.next;
      temp = eventhead.next;
      count_update = 1;

      // check if elementary topological change or retriangulate
      while(temp->next != NULL && temp->next->time < 0){
count_update++;
temp = temp->next;
      }
      eventhead.next = temp->next;
      temp->next = NULL;

      temp = updateque;

      if(temp != NULL){

if(count_update > 1){

  /*************************************************************/
  //**********          Retriangulate            ************//
  /*************************************************************/

  Fortune fort;
  triangle trihead, *temptri;
```

```
  trihead.next = NULL;

  fort.FortuneAlgo(allpos, &trihead);

  if(trihead.next==NULL){
  }
  temptri = trihead.next;

  printf("--------- retriangulation ---------\n");


  SetAllNeighbors(&trihead);

  SetAllClosest();
  DelAllEvent(&eventhead);
  SetEventQueue(&trihead);

  DelAllTri(&trihead);


}else{

  /*************************************************************/
  /*************************************************************/
  if(temp->triangle1[0] != temp->triangle2[0] &&
     temp->triangle1[0] != temp->triangle2[1] &&
     temp->triangle1[0] != temp->triangle2[2]){
    a = temp->triangle1[0];
    b = temp->triangle1[1];
    c = temp->triangle1[2];
  }else if(temp->triangle1[1] != temp->triangle2[0] &&
   temp->triangle1[1] != temp->triangle2[1] &&
   temp->triangle1[1] != temp->triangle2[2]){
    a = temp->triangle1[1];
    b = temp->triangle1[0];
    c = temp->triangle1[2];
  }else if(temp->triangle1[2] != temp->triangle2[0] &&
   temp->triangle1[2] != temp->triangle2[1] &&
   temp->triangle1[2] != temp->triangle2[2]){
```

```
    a = temp->triangle1[2];
    b = temp->triangle1[0];
    c = temp->triangle1[1];
  }

  if( b != temp->triangle2[0] && c != temp->triangle2[0]){
    d = temp->triangle2[0];
  }else if( b != temp->triangle2[1] && c != temp->triangle2[1]){
    d = temp->triangle2[1];
  }else if( b != temp->triangle2[2] && c != temp->triangle2[2]){
    d = temp->triangle2[2];
  }

  printf("An elementary topological event\n"
 " (a,b,c,d) = (%2d,%2d,%2d,%2d)\n", a, b, c, d);


  /*--------------------------------------*/
  /*--      Updating neighbors of a      --*/
  /*--   insert d into neighbors list    --*/
  /*--------------------------------------*/
  member[a].GetNeighbors(&max, neighbors);

  for(i=0;i<max;i++){
    if(b == *neighbors[i] || c == *neighbors[i]){
      updateneighbor[i] = *neighbors[i];
      updateneighbor[++i] = d;
      for(;i<max;i++){
updateneighbor[i+1] = *neighbors[i];
      }
    }else{
      updateneighbor[i] = *neighbors[i];
    }
  }
  max++;

  mindistance = 1000000;
  tempclosest = EMPTY;
```

```
for(int j=0;j<max;j++){
  tempdist = Distance(&(allpos[a]), &(allpos[updateneighbor[j]]));

  if(tempdist < mindistance){
    tempclosest = j;
    mindistance = tempdist;
  }
}

member[a].SetClosest((allpos[updateneighbor[tempclosest]]).x,
     (allpos[updateneighbor[tempclosest]]).y);

xa  = allpos[a].x;
ya  = allpos[a].y;
xva = (allpos[updateneighbor[tempclosest]]).x;
yva = (allpos[updateneighbor[tempclosest]]).y;


printf("prey a's Neighbors (before) -- ");
for(int k=0;k < max-1; k++){
  printf("%3d",*neighbors[k]);
}
printf("\nprey a's Neighbors (after)  -- ");
for(int k=0;k < max; k++){
  printf("%3d",updateneighbor[k]);
}
printf("\n");
member[a].CopyNeighbors(max,updateneighbor);




/*--------------------------------------*/
/*--      Updating neighbors of b      --*/
/*--   remove c from neighbors list    --*/
/*--------------------------------------*/
member[b].GetNeighbors(&max, neighbors);
```

```
  for(i=0;i<max;i++){
    if(c == *neighbors[i]){
      i++;
      for(;i<max;i++){
updateneighbor[i-1] = *neighbors[i];
    }
    }else{
      updateneighbor[i] = *neighbors[i];
    }
  }
  max--;

  mindistance = 1000000;
  tempclosest = EMPTY;

  for(int j=0;j<max;j++){
    tempdist = Distance(&(allpos[b]), &(allpos[updateneighbor[j]]));

    if(tempdist < mindistance){
      tempclosest = j;
      mindistance = tempdist;
    }
  }

  member[b].SetClosest((allpos[updateneighbor[tempclosest]]).x,
       (allpos[updateneighbor[tempclosest]]).y);

  xb  = allpos[b].x;
  yb  = allpos[b].y;
  xvb = (allpos[updateneighbor[tempclosest]]).x;
  yvb = (allpos[updateneighbor[tempclosest]]).y;



  printf("prey b's Neighbors (before) -- ");
  for(int k=0;k < max+1; k++){
    printf("%3d",*neighbors[k]);
  }
  printf("\nprey b's Neighbors (after)  -- ");
```

```
  for(int k=0;k < max; k++){
    printf("%3d",updateneighbor[k]);
  }
  printf("\n");

  member[b].CopyNeighbors(max,updateneighbor);

  /*--------------------------------------*/
  /*--      updating neighbors of c      --*/
  /*--   remove b from neighbors list    --*/
  /*--------------------------------------*/
  member[c].GetNeighbors(&max, neighbors);

  for(i=0;i<max;i++){
    if(b == *neighbors[i]){
      i++;
      for(;i<max;i++){
updateneighbor[i-1] = *neighbors[i];
      }
    }else{
      updateneighbor[i] = *neighbors[i];
    }
  }
  max--;

  mindistance = 1000000;
  tempclosest = EMPTY;


  for(int j=0;j<max;j++){
    tempdist = Distance(&(allpos[c]), &(allpos[updateneighbor[j]]));

    if(tempdist < mindistance){
      tempclosest = j;
      mindistance = tempdist;
    }
  }
```

```
  member[c].SetClosest((allpos[updateneighbor[tempclosest]]).x,
       (allpos[updateneighbor[tempclosest]]).y);


  xc  = allpos[c].x;
  yc  = allpos[c].y;
  xvc = (allpos[updateneighbor[tempclosest]]).x;
  yvc = (allpos[updateneighbor[tempclosest]]).y;

  printf("prey c's Neighbors (before) -- ");
  for(int k=0;k < max+1; k++){
    printf("%3d",*neighbors[k]);
  }
  printf("\nprey c's Neighbors (after)  -- ");
  for(int k=0;k < max; k++){
    printf("%3d",updateneighbor[k]);
  }
  printf("\n");
  member[c].CopyNeighbors(max,updateneighbor);


  /*--------------------------------------*/
  /*--      updating neighbors of d      --*/
  /*--   insert a into neighbors list    --*/
  /*--------------------------------------*/
  member[d].GetNeighbors(&max, neighbors);

  for(i=0;i<max;i++){
    if(b == *neighbors[i] || c == *neighbors[i]){
      updateneighbor[i] = *neighbors[i];
      updateneighbor[++i] = a;
      for(;i<max;i++){
updateneighbor[i+1] = *neighbors[i];
      }
    }else{
      updateneighbor[i] = *neighbors[i];
    }
  }
  max++;
```

```
  mindistance = 1000000;
  tempclosest = EMPTY;

  for(int j=0;j<max;j++){
    tempdist = Distance(&(allpos[d]), &(allpos[updateneighbor[j]]));

    if(tempdist < mindistance){
      tempclosest = j;
      mindistance = tempdist;
    }
  }

  member[d].SetClosest((allpos[updateneighbor[tempclosest]]).x,
       (allpos[updateneighbor[tempclosest]]).y);

  xd  = allpos[d].x;
  yd  = allpos[d].y;
  xvd = (allpos[updateneighbor[tempclosest]]).x;
  yvd = (allpos[updateneighbor[tempclosest]]).y;


  printf("prey d's Neighbors (before) -- ");
  for(int k=0;k < max-1; k++){
    printf("%3d",*neighbors[k]);
  }
  printf("\nprey d's Neighbors (after)  -- ");
  for(int k=0;k < max; k++){
    printf("%3d",updateneighbor[k]);
  }
  printf("\n");
  member[d].CopyNeighbors(max,updateneighbor);


  xx4 = coefficient_t_4(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx3 = coefficient_t_3(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx2 = coefficient_t_2(xa, xb, xc, xd, ya, yb, yc, yd,
```

```
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx1 = coefficient_t_1(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  xx0 = coefficient_t_0(xa, xb, xc, xd, ya, yb, yc, yd,
xva, xvb, xvc, xvd, yva, yvb, yvc, yvd);
  findroot.CalcRoot(xx4,xx3,xx2,xx1,xx0, &num_ans, &smallest);

  if(smallest!=MAXNUM){
    AddInOrderEvent(&eventhead, b, a, d, a, d, c,smallest);
  }
}

/*************************************************************/
/*************************************************************/

printf("\n\n+++++++++++++++  Event List  ++++++++++++++\n\n");
etemp = eventhead.next;
eventnum =0;
while(etemp != NULL){
  printf("%2d (%2d,%2d,%2d) (%2d,%2d,%2d)  time %12.5f\n", eventnum++,
 etemp->triangle1[0],
 etemp->triangle1[1],etemp->triangle1[2],
 etemp->triangle2[0],etemp->triangle2[1],
 etemp->triangle2[2],etemp->time);
  etemp = etemp->next;
}
printf("\n\n");
      }

      DelAllEvent(updateque);


    } //  End of Event
  }
}
```

```
void Herd::PrintAllPrey(void){
  printf("\n");
  for(int i=0;i<PREYNUM;i++){
    member[i].PrintPos();
  }
  printf("\n");
}
```

```
#ifdef MYOPENGL
void Herd::DisplayHerd(void){
  int max;
  position allpos[PREYNUM];
  position allonestep[PREYNUM];
  int *neighbors[PREYNUM], updateneighbor[PREYNUM];

  glClear(GL_COLOR_BUFFER_BIT);     // clear the screen
  glColor3f(0, 0, 1);

  GetAllPosition(allpos);

  // Points represents preys' positions
  glBegin(GL_POINTS);
  for(int i=0;i<PREYNUM;i++){
    glVertex2d(allpos[i].x, allpos[i].y);
  }
  glEnd();

  //  Lines between neighbors
  glColor3f(1, 0, 0);

  for(int i=0;i<PREYNUM;i++){
    member[i].GetNeighbors(&max,neighbors);

    for(int j=0;j<max;j++){
```

```
      glBegin(GL_LINES);
      glVertex2d(allpos[i].x, allpos[i].y);
      glVertex2d(allpos[*neighbors[j]].x, allpos[*neighbors[j]].y);
      glEnd();
    }
  }

  glFlush();            // send all output to display
}
#endif
```

herdstatic.hpp

```
#ifndef _HERD_H
#define _HERD_H


#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include "prey.hpp"
#include "fortune.hpp"

#ifdef MYOPENGL
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#endif

class Herd{
private:
  Prey member[PREYNUM];
  double areaofdanger[PREYNUM];
  double old_areaofdanger[PREYNUM];


public:
```

```
  Herd();
  void SetAllClosest();
  void SetAreaOfDanger();
  void SetAllNeighbors(triangle *head);
  void GetAllPosition(position *allpos);
  void GetAllOnestep(position *allpos);
  void PrintAllPrey(void);
  void MoveAllPreyStatic(void);
#ifdef MYOPENGL
  void DisplayHerd(void);
#endif
};
#endif
```

herdstatic.cpp

```
#include "herdstatic.hpp"
#include "root.hpp"


/*****************************************************************/
/**  Herd()  constructor                                        **/
/*****************************************************************/


Herd::Herd(){
  Fortune fort;
  position allpos[PREYNUM];
  triangle trihead, *temp;
  // eventque *etemp;
  int i, j;

  /*--- fill prey's position and index ---*/
  for(int i=0;i<PREYNUM;i++){
    member[i].SetPosition(Randomnum()%200,Randomnum()%200);
    member[i].SetIndex(i);
```

```
    old_areaofdanger[i] = 0;
    areaofdanger[i] = 0;
  }

  trihead.next  = NULL;

  GetAllPosition(allpos);
  fort.FortuneAlgo(allpos, &trihead);


  // Set Neighbors
  SetAllNeighbors(&trihead);

  SetAllClosest();
  SetAreaOfDanger();
  // SetEventQueue(&trihead);
  DelAllTri(&trihead);

}


/*************************************************************************/
/**  SetAllClosest()                                                    **/
/**  This function will set closest prey for all prey.                  **/
/**                                                                     **/
/**                                                                     **/
/**                                                                     **/
/*************************************************************************/


void Herd::SetAllClosest(){
  int max=0, tempclosest, index, tempin;
  int *neighbors[PREYNUM];
  int tempnei[2][PREYNUM];
  int tempnei2[2][PREYNUM];
  int tempnum[2], inorder[PREYNUM];
  position tempnorm[2][PREYNUM];
  position allpos[PREYNUM], temppos;
  double mindistance, tempdist;
```

```
  GetAllPosition(allpos);


  for(int i=0;i<PREYNUM;i++){
    // Get Neighbors and Pick Closest.

    member[i].GetNeighbors(&max, neighbors);


    //////////////     make neighbor list in order
    tempnum[0]=0;
    tempnum[1]=0;

    for(int k=0;k<max;k++){
      if(allpos[i].y < allpos[*neighbors[k]].y){ // upper
tempnei[0][tempnum[0]] = *neighbors[k];
NormToward(&allpos[i],&allpos[*neighbors[k]],&tempnorm[0][tempnum[0]]);
tempnum[0]++;
      }else{                                          // lower
tempnei[1][tempnum[1]] = *neighbors[k];
NormToward(&allpos[i],&allpos[*neighbors[k]],&tempnorm[1][tempnum[1]]);
tempnum[1]++;
      }
    }


    ///   sort
    for(int n=0;n < 2; n++){
      for(int k=0;k<tempnum[n]-1;k++){
for(int m=k+1;m<tempnum[n];m++){
  if(tempnorm[n][k].x > tempnorm[n][m].x){
    temppos.x = tempnorm[n][k].x;
    temppos.y = tempnorm[n][k].y;
    tempnorm[n][k].x = tempnorm[n][m].x;
    tempnorm[n][k].y = tempnorm[n][m].y;
    tempnorm[n][m].x = temppos.x;
    tempnorm[n][m].y = temppos.y;
```

```
    tempin = tempnei[n][k];
    tempnei[n][k] = tempnei[n][m];
    tempnei[n][m] = tempin;
  }
}
      }
    }


    // After sort

    for(int k=0;k<tempnum[0];k++){
      inorder[k] = tempnei[0][k];
    }
    for(int k=0;k<tempnum[1];k++){
      inorder[tempnum[0]+tempnum[1]-k-1] = tempnei[1][k];
    }

    member[i].CopyNeighbors(max,inorder);



    mindistance = 1000000;
    tempclosest = EMPTY;

    for(int j=0;j<max;j++){
      tempdist = Distance(&(allpos[i]), &(allpos[*neighbors[j]]));

      if(tempdist < mindistance){
tempclosest = j;
mindistance = tempdist;
      }
    }
    member[i].SetClosest((allpos[*neighbors[tempclosest]]).x,
 (allpos[*neighbors[tempclosest]]).y);
  }
}
```

```
void Herd::SetAreaOfDanger(){
  int max=0;
  int *neighbors[PREYNUM];
  position allpos[PREYNUM];
  position areapos[PREYNUM];
  position tempareapos[2][PREYNUM];
  position tempnorm[2][PREYNUM];
  position temppos;
  position inorder[PREYNUM];
  int tempnum[2];

  double t;

  GetAllPosition(allpos);

  for(int i=0;i<PREYNUM;i++){
    old_areaofdanger[i] = areaofdanger[i];
    areaofdanger[i] = 0;

    member[i].GetNeighbors(&max, neighbors);


    if(max >= 3){
      for(int j=1;j<max;j++){
FindCenter((allpos[*neighbors[j-1]]).x, (allpos[*neighbors[j-1]]).y,
   (allpos[*neighbors[j]]).x, (allpos[*neighbors[j]]).y,
   allpos[i].x, allpos[i].y,
   &(areapos[j-1].x),&(areapos[j-1].y));
      }

      FindCenter((allpos[*neighbors[max-1]]).x,
 (allpos[*neighbors[max-1]]).y,
 (allpos[*neighbors[0]]).x, (allpos[*neighbors[0]]).y,
 allpos[i].x, allpos[i].y,
 &(areapos[max-1].x),&(areapos[max-1].y));
```

```
      tempnum[0]=0;
      tempnum[1]=0;

      for(int j=0;j<max;j++){
if(allpos[i].y < areapos[j].y){ // upper
  tempareapos[0][tempnum[0]].x = areapos[j].x;
  tempareapos[0][tempnum[0]].y = areapos[j].y;
  NormToward(&allpos[i],&areapos[j],&tempnorm[0][tempnum[0]]);
  tempnum[0]++;
}else{                          // lower
  tempareapos[1][tempnum[1]].x = areapos[j].x;
  tempareapos[1][tempnum[1]].y = areapos[j].y;
  NormToward(&allpos[i],&areapos[j],&tempnorm[1][tempnum[1]]);
  tempnum[1]++;
}

      }

      ///////              Sort   ///////////////
      for(int n=0;n < 2; n++){
for(int k=0;k<tempnum[n]-1;k++){
  for(int m=k+1;m<tempnum[n];m++){
    if(tempnorm[n][k].x > tempnorm[n][m].x){
      temppos.x = tempareapos[n][k].x;
      temppos.y = tempareapos[n][k].y;
      tempareapos[n][k].x = tempareapos[n][m].x;
      tempareapos[n][k].y = tempareapos[n][m].y;
      tempareapos[n][m].x = temppos.x;
      tempareapos[n][m].y = temppos.y;

      temppos.x = tempnorm[n][k].x;
      temppos.y = tempnorm[n][k].y;
      tempnorm[n][k].x = tempnorm[n][m].x;
      tempnorm[n][k].y = tempnorm[n][m].y;
      tempnorm[n][m].x = temppos.x;
      tempnorm[n][m].y = temppos.y;
    }
  }
}
```

```
      }


      for(int k=0;k<tempnum[0];k++){
inorder[k].x = tempareapos[0][k].x;
inorder[k].y = tempareapos[0][k].y;
      }
      for(int k=0;k<tempnum[1];k++){
inorder[tempnum[0]+tempnum[1]-k-1].x = tempareapos[1][k].x;
inorder[tempnum[0]+tempnum[1]-k-1].y = tempareapos[1][k].y;
      }


      for(int j=1;j<max;j++){
t =((inorder[j]).x - (inorder[j-1]).x)
  * ((inorder[j]).y + (inorder[j-1]).y) / 2.0;
areaofdanger[i] += t;
      }
      t =((inorder[0]).x - (inorder[max-1]).x)
  * ((inorder[0]).y + (inorder[max-1]).y) / 2.0;
      areaofdanger[i] += t;


    }else{
      areaofdanger[i] = INFINITE;
    }
  }


  printf("\nprey  Area of Danger      Difference between\n"
 "                              old and new one \n");
  for(int i=0;i<PREYNUM;i++){
    printf("%2d    %10.2f          %10.2f\n", i, areaofdanger[i],
   areaofdanger[i]-old_areaofdanger[i]);
  }
  printf("\n\n");
}
```

```
void Herd::SetAllNeighbors(triangle *head){
  for(int i=0;i<PREYNUM;i++){
    member[i].SetNeighbors(head);
  }
}



void Herd::GetAllPosition(position *allpos){
  for(int i=0;i<PREYNUM;i++){
    member[i].GetPosition(&allpos[i]);
  }
}



void Herd::GetAllOnestep(position *allpos){
  for(int i=0;i<PREYNUM;i++){
    member[i].GetOnestep(&allpos[i]);
  }
}


void Herd::MoveAllPreyStatic(void){
  double mindistance, tempdist;
  double xx0, xx1, xx2, xx3, xx4;
  double xa, xb, xc, xd, ya, yb, yc, yd;
  double xva, xvb, xvc, xvd, yva, yvb, yvc, yvd;
  int count, a, b, c, d, i, j, max, after, tempclosest;
  int count_update;
  int *neighbors[PREYNUM], updateneighbor[PREYNUM];
  position allpos[PREYNUM];
  position allonestep[PREYNUM];
  int num_ans;
  double smallest;
  int eventnum=0;
```

```
  for(int i=0;i<PREYNUM;i++){
    member[i].Move();
  }
  PrintAllPrey();


  /*************************************************************/
  //**********          Retriangulate             ************//
  /*************************************************************/

  Fortune fort;
  triangle trihead, *temptri;
  trihead.next = NULL;

  GetAllPosition(allpos);
  fort.FortuneAlgo(allpos, &trihead);

  if(trihead.next!=NULL){
    temptri = trihead.next;

    printf("--------- retriangulation ---------\n");

    SetAllNeighbors(&trihead);
    SetAllClosest();
    DelAllTri(&trihead);
    SetAreaOfDanger();
  }
}



void Herd::PrintAllPrey(void){
  printf("\n");
  for(int i=0;i<PREYNUM;i++){
    member[i].PrintPos();
  }
  printf("\n");
}
```

```
#ifdef MYOPENGL
void Herd::DisplayHerd(void){
  int max;
  position allpos[PREYNUM];
  position allonestep[PREYNUM];
  int *neighbors[PREYNUM], updateneighbor[PREYNUM];

  glClear(GL_COLOR_BUFFER_BIT);     // clear the screen
  glColor3f(0, 0, 1);

  GetAllPosition(allpos);

  // Points represents preys' positions
  glBegin(GL_POINTS);
  for(int i=0;i<PREYNUM;i++){
    glVertex2d(allpos[i].x, allpos[i].y);
  }
  glEnd();

  //  Lines between neighbors
  glColor3f(1, 0, 0);

  for(int i=0;i<PREYNUM;i++){
    member[i].GetNeighbors(&max,neighbors);

    for(int j=0;j<max;j++){
      glBegin(GL_LINES);
      glVertex2d(allpos[i].x, allpos[i].y);
      glVertex2d(allpos[*neighbors[j]].x, allpos[*neighbors[j]].y);
      glEnd();
    }
  }

  glFlush();                  // send all output to display
}
#endif
```

APPENDIX C (SOURCE CODES FOR ROOT CLASS)

In this section, `root.hpp` and `root.cpp` are listed, and they are for root class. The original programs from J-P Moreau can be found on the web (Moreau, 2002).

root.hpp

```
#ifndef _ROOT_H
#define _ROOT_H

#include "myheader.hpp"


class Root{
private:
  double a[5][5], r[5];
  int    i,n;
  double aa,b,c,d,ii,im,k,l,m,q,rr,s,sw;

public:
  Root();
  double f(double x);
  void Root_2();
  void Root_3();
  void Root_4();
  void CalcRoot(double x4, double x3, double x2, double x1, double x0,
    int *num_ans, double *smallest);
};
#endif
```

root.cpp

```
#include "root.hpp"


Root::Root(){

}


double Root::f(double x){
  return x * x * x + a[3][2] * x * x + a[3][1] * x + a[3][0];
}


/*****************************************
* This Subroutine calculates the roots of *
*    X^2 + A(2,1)*X + A(2,0) = 0          *
*    W = Determinant of equation          *
*****************************************/
void Root::Root_2()  {

  double q1, q2, w;
  w = a[2][1] * a[2][1] - 4 * a[2][0];
  q1 = -a[2][1] / 2.0;
  if (w < 0) {
    q2 = sqrt(-w) / 2.0;
    im = q2;
    r[1] = q1; r[2] = q1;
  }
  else if (w==0) {
    r[1] = q1; r[2] = q1; im = 0.0;
  }
  else if (w > 0) {
    q2 = sqrt(w) / 2.0; im = 0.0;
    r[1] = q1 + q2; r[2] = q1 - q2;
  }
}
```

```
/*****************************************
* This subroutine calculates the roots of  *
* X^3 + A(3,2)*X^2 + A(3,1)*X + A(3,0) = 0 *
*****************************************/
void Root::Root_3()  {
  //Labels e100,e200,e500
  double am, er, te, tt, xa, xb, xc, y1, y2;
  // one root equals zero
  if (a[3][0] == 0)  {
    xc = 0; goto e500;
  }
  // looking for( maximum coefficient in fabsolute magnitude
  am = fabs(a[3][0]);
  for (i = 1; i<4; i++) {
    tt = fabs(a[3][i]);
    if (am < tt) am = tt;
  }
  //Define interval where a real root exists
  // according to sign of A(3,0)
  if (a[3][0] > 0) {
    aa = -am - 1;
    b = 0;
    goto e100;
  }
  aa = 0; b = am + 1;

e100:
  // Searching for( xc = real root in interval (a,b)
  //   (by Bisection method)

  // Define segment (xa,xb) including the root
  xa = aa; xb = b; y1 = f(aa); y2 = f(b);

  er = 0.000001;
  if (y1 == 0) {
    xc = aa; goto e500;
  }
```

```
  if (y2 == 0) {
    xc = b; goto e500;
  }
  // xc : middle of segment
  e200: xc = (xa + xb) / 2.0; te = f(xc);
  if (te == 0) goto e500;
  if ((xb - xa) < er) goto e500;
  if (f(xa) * te > 0) {
    xa = xc; goto e200;
  }
  xb = xc; goto e200;
  // r[3] is a real root
e500: r[3] = xc;
  if (sw == -1) return;
  // Calculates the roots of remaining quadratic equation
  // Define the equation coefficients
  a[2][1] = a[3][2] + xc;
  a[2][0] = a[3][1] + a[3][2] * xc + xc * xc;
  Root_2();
  return;
}



// Root search main subroutine
void Root::Root_4()  {
  // Labels: e100,e200,e300
  // Normalization of coefficients
  for (i = 0; i<n; i++) {
    a[n][i] /= a[n][n];
  }
  // Branching according to degree n
  if (n==2) {
    Root_2();
    return;
  }
  else if (n==3) {
    Root_3();
```

```
    return;
  }
  else if (n==4) {
    aa = a[4][3];
    b = a[4][2];
    c = a[4][1];
    d = a[4][0];
    q = b - (3.0 * aa * aa / 8.0);
    rr = c - (aa * b / 2) + (aa * aa * aa / 8.0);
    s = d - (aa * c / 4) + (aa * aa * b / 16.0) -
(3 * aa * aa * aa * aa / 256.0);

    a[3][2] = q / 2.0;
    a[3][1] = (q * q - 4 * s) / 16.0;
    a[3][0] = -(rr * rr / 64.0);
    // Calculate a real root of this equation
    if ((rr != 0) || (a[3, 1] >= 0)) goto e100;
    // Particular case when this equation is of 2nd order
    a[2][1] = a[3][2]; a[2][0] = a[3][1];
    Root_2();
    // One takes the positive root
    r[3] = r[1];
    goto e200;
    // Calling Root_3 with sw=-1 to calculate one root only
e100: sw = -1;
    Root_3();
    // real root of above equation
e200: k = sqrt(r[3]);
    // Calculate L and M if k=0
    if (k == 0) {
      rr = sqrt(q * q - 4 * s); goto e300;
    }
    q = q + (4 * r[3]);
    rr = rr / (2 * k);
e300: l = (q - rr) / 2.0; m = (q + rr) / 2.0;
    // Solving two equations of degree 2
    a[2][1] = 2 * k; a[2][0] = l;
    // 1st equation
    Root_2();
```

APPENDIX D (SOURCE CODES FOR FORTUNE CLASS)

The files `fourtune.hpp`, `fourtune.cpp`, `vdefs.h`, `edgelist.c`, `geometry.c`, `heap.c`, and `memory.c` are for fortune class. Those codes are from Steve J. Fortune, and can be found (Brook, 1999).

fourtune.hpp

```
#ifndef _FORTUNEGLO_H
#define _FORTUNEGLO_H

#include "voronoi/vdefs.h"
#include "myheader.hpp"


extern int sorted, triangulate, plot, debug, nsites, siteidx ;
extern float xmin, xmax, ymin, ymax ;
extern Site * sites ;
extern Freelist sfl ;
/* main.c

extern int sorted, triangulate, plot, debug, nsites, siteidx ;
extern float xmin, xmax, ymin, ymax ;
extern Site * sites ;
extern Freelist sfl ;

 int sorted, triangulate, plot, debug, nsites, siteidx ;
 float xmin, xmax, ymin, ymax ;
 Site * sites ;
 Freelist sfl ;
 */
```

```
  a[2][1] = x1;
  a[2][0] = x0;
}else if(x1<-0.001 || x1 > 0.001){
  *smallest = MAXNUM;
  temp = x0 / x1;

  if(temp >= 0){
    *smallest = temp;
  }
  return;
}else{
  return;
}




// calling root search main subroutine
Root_4();
// writing results
if (n==2)  {
  //case n=2
  if (im == 0) {
    answer[(*num_ans)++] = r[1];
    answer[(*num_ans)++] = r[2];
  }
}
else if (n==3)  {
  //case n=3
  answer[(*num_ans)++] = r[3];
  if (im == 0) {
    answer[(*num_ans)++] = r[1];
    answer[(*num_ans)++] = r[2];
  }
}
else if (n==4) {
  //case n=4
  if (im==0) {
    answer[(*num_ans)++] = r[1];
```

```
      answer[(*num_ans)++] = r[2];
    }
    if (ii==0) {
      answer[(*num_ans)++] = r[3];
      answer[(*num_ans)++] = r[4];
    }
  }

  *smallest = MAXNUM;

  for(int i=0;i<*num_ans;i++){

    if(answer[i]>=0){
      if(*smallest>answer[i]){
*smallest = answer[i];
      }
    }
  }
}
```

APPENDIX D (SOURCE CODES FOR FORTUNE CLASS)

The files `fourtune.hpp`, `fourtune.cpp`, `vdefs.h`, `edgelist.c`, `geometry.c`, `heap.c`, and `memory.c` are for fortune class. Those codes are from Steve J. Fortune, and can be found (Brook, 1999).

fourtune.hpp

```
#ifndef _FORTUNEGLO_H
#define _FORTUNEGLO_H

#include "voronoi/vdefs.h"
#include "myheader.hpp"


extern int sorted, triangulate, plot, debug, nsites, siteidx ;
extern float xmin, xmax, ymin, ymax ;
extern Site * sites ;
extern Freelist sfl ;
/* main.c

extern int sorted, triangulate, plot, debug, nsites, siteidx ;
extern float xmin, xmax, ymin, ymax ;
extern Site * sites ;
extern Freelist sfl ;

 int sorted, triangulate, plot, debug, nsites, siteidx ;
 float xmin, xmax, ymin, ymax ;
 Site * sites ;
 Freelist sfl ;
 */
```

```
class Fortune{
private:

public:
  Fortune();
  int FortuneAlgo(position *allpos, triangle *head);
  void readsites(position *allpos) ;              /* from voronoi.c */
  void voronoi(Site *(*)(), triangle *head) ;     /* from voronoi.c */


};
#endif
```

fourtune.cpp

```
#include "fortune.hpp"

int sorted, triangulate, plot, debug, nsites, siteidx ;
float xmin, xmax, ymin, ymax ;
Site * sites ;
Freelist sfl ;


Site * readone(void) ;
Site * nextone(void) ;


Fortune::Fortune(){


}

int Fortune::FortuneAlgo(position *allpos, triangle *head){

    Site *(*next)() ;

    sorted = triangulate = plot = debug = 0 ;
```

```
    triangulate = 1;        /* All I need is triangulation*/

    freeinit(&sfl, sizeof(Site)) ;

    readsites(allpos) ;
    next = nextone ;

    siteidx = 0 ;
    geominit() ;

    voronoi(next, head) ;
    return (0) ;
}



/*** sort sites on y, then x, coord ***/

int
scomp(const void * vs1, const void * vs2)
    {
    Point * s1 = (Point *)vs1 ;
    Point * s2 = (Point *)vs2 ;

    if (s1->y < s2->y)
        {
        return (-1) ;
        }
    if (s1->y > s2->y)
        {
        return (1) ;
        }
    if (s1->x < s2->x)
        {
        return (-1) ;
        }
    if (s1->x > s2->x)
        {
```

```
        return (1) ;
        }
    return (0) ;
    }



/*** return a single in-storage site ***/

Site *
nextone(void)
    {
    Site * s ;

    if (siteidx < nsites)
        {
        s = &sites[siteidx++];
        return (s) ;
        }
    else
        {
        return ((Site *)NULL) ;
        }
    }




/*** read all sites, sort, and compute xmin, xmax, ymin, ymax ***/

void
Fortune::readsites(position *allpos)
    {
    int i ;

    nsites = 0 ;
    sites = (Site *) myalloc(PREYNUM * sizeof(Site));

    /*-------- Input part --------*/
```

```
    for(i=0; i<PREYNUM; i++){
      sites[nsites].coord.x = allpos[i].x;
      sites[nsites].coord.y = allpos[i].y;
      sites[nsites].sitenbr = nsites ;
      sites[nsites++].refcnt = 0 ;
    }


    qsort((void *)sites, nsites, sizeof(Site), scomp) ;
    xmin = sites[0].coord.x ;
    xmax = sites[0].coord.x ;
    for (i = 1 ; i < nsites ; ++i)
        {
        if(sites[i].coord.x < xmin)
            {
            xmin = sites[i].coord.x ;
            }
        if (sites[i].coord.x > xmax)
            {
            xmax = sites[i].coord.x ;
            }
        }
    ymin = sites[0].coord.y ;
    ymax = sites[nsites-1].coord.y ;
    }




/*** read one site ***/

Site *
readone(void)
    {
    Site * s ;

    s = (Site *)getfree(&sfl) ;
    s->refcnt = 0 ;
    s->sitenbr = siteidx++ ;
```

```
    if (scanf("%f %f", &(s->coord.x), &(s->coord.y)) == EOF)
        {
        return ((Site *)NULL ) ;
        }
    return (s) ;
    }


extern Site * bottomsite ;
extern Halfedge * ELleftend, * ELrightend ;

/*** implicit parameters: nsites, sqrt_nsites, xmin, xmax, ymin, ymax,
 : deltax, deltay (can all be estimates).
 : Performance suffers if they are wrong; better to make nsites,
 : deltax, and deltay too big than too small.  (?)
 ***/

void
Fortune::voronoi(Site *(*nextsite)(void), triangle *head)
    {
    Site * newsite, * bot, * top, * temp, * p, * v ;
    Point newintstar ;
    int pm ;
    Halfedge * lbnd, * rbnd, * llbnd, * rrbnd, * bisector ;
    Edge * e ;

    PQinitialize() ;
    bottomsite = (*nextsite)() ;
    out_site(bottomsite) ;
    ELinitialize() ;
    newsite = (*nextsite)() ;
    while (1)
        {
        if(!PQempty())
            {
            newintstar = PQ_min() ;
            }
        if (newsite != (Site *)NULL && (PQempty()
            || newsite -> coord.y < newintstar.y
```

```
        || (newsite->coord.y == newintstar.y
        && newsite->coord.x < newintstar.x))) {
        {
        out_site(newsite) ;
        }
    lbnd = ELleftbnd(&(newsite->coord)) ;
    rbnd = ELright(lbnd) ;
    bot = rightreg(lbnd) ;
    e = bisect(bot, newsite) ;
    bisector = HEcreate(e, le) ;
    ELinsert(lbnd, bisector) ;
    p = intersect(lbnd, bisector) ;
    if (p != (Site *)NULL)
        {
        PQdelete(lbnd) ;
        PQinsert(lbnd, p, dist(p,newsite)) ;
        }
    lbnd = bisector ;
    bisector = HEcreate(e, re) ;
    ELinsert(lbnd, bisector) ;
    p = intersect(bisector, rbnd) ;
    if (p != (Site *)NULL)
        {
        PQinsert(bisector, p, dist(p,newsite)) ;
        }
    newsite = (*nextsite)() ;
    }
else if (!PQempty())   /* intersection is smallest */
        {
        lbnd = PQextractmin() ;
        llbnd = ELleft(lbnd) ;
        rbnd = ELright(lbnd) ;
        rrbnd = ELright(rbnd) ;
        bot = leftreg(lbnd) ;
        top = rightreg(rbnd) ;

        // out_triple(bot, top, rightreg(lbnd)) ;
/*****************************************/
// bot->sitenbr, top->sitenbr, rightreg(lbnd)->sitenbr
```

```
// These three numbers are indexes of triple

AddNextTri(head, bot->sitenbr,
   top->sitenbr, rightreg(lbnd)->sitenbr);


/*****************************************/
        v = lbnd->vertex ;
        makevertex(v) ;
        endpoint(lbnd->ELedge, lbnd->ELpm, v);
        endpoint(rbnd->ELedge, rbnd->ELpm, v) ;
        ELdelete(lbnd) ;
        PQdelete(rbnd) ;
        ELdelete(rbnd) ;
        pm = le ;
        if (bot->coord.y > top->coord.y)
            {
            temp = bot ;
            bot = top ;
            top = temp ;
            pm = re ;
            }
        e = bisect(bot, top) ;
        bisector = HEcreate(e, pm) ;
        ELinsert(llbnd, bisector) ;
        endpoint(e, re-pm, v) ;
        deref(v) ;
        p = intersect(llbnd, bisector) ;


        if (p  != (Site *) NULL)
            {
            PQdelete(llbnd) ;
            PQinsert(llbnd, p, dist(p,bot)) ;
            }
        p = intersect(bisector, rrbnd) ;
        if (p != (Site *) NULL)
            {
            PQinsert(bisector, p, dist(p,bot)) ;
```

```
                }
            }
        else
            {
            break ;
            }
        }   // end of while

    }
```

vdefs.h

```
#ifndef __VDEFS_H
#define __VDEFS_H

#ifndef NULL
#define NULL 0
#endif

#define DELETED -2

typedef struct tagFreenode
    {
    struct tagFreenode * nextfree;
    } Freenode ;


typedef struct tagFreelist
    {
    Freenode * head;
    int nodesize;
    } Freelist ;

typedef struct tagPoint
    {
    float x ;
    float y ;
```

```
    } Point ;

/* structure used both for sites and for vertices */

typedef struct tagSite
    {
    Point coord ;
    int sitenbr ;
    int refcnt ;
    } Site ;


typedef struct tagEdge
    {
    float a, b, c ;
    Site * ep[2] ;
    Site * reg[2] ;
    int edgenbr ;
    } Edge ;

#define le 0
#define re 1

typedef struct tagHalfedge
    {
    struct tagHalfedge * ELleft ;
    struct tagHalfedge * ELright ;
    Edge * ELedge ;
    int ELrefcnt ;
    char ELpm ;
    Site * vertex ;
    float ystar ;
    struct tagHalfedge * PQnext ;
    } Halfedge ;
```

```
/* edgelist.c */
void ELinitialize(void) ;
Halfedge * HEcreate(Edge *, int) ;
void ELinsert(Halfedge *, Halfedge *) ;
Halfedge * ELgethash(int) ;
Halfedge * ELleftbnd(Point *) ;
void ELdelete(Halfedge *) ;
Halfedge * ELright(Halfedge *) ;
Halfedge * ELleft(Halfedge *) ;
Site * leftreg(Halfedge *) ;
Site * rightreg(Halfedge *) ;
extern int ELhashsize ;
extern Site * bottomsite ;
extern Freelist hfl ;
extern Halfedge * ELleftend, * ELrightend, **ELhash ;

/* geometry.c */
void geominit(void) ;
Edge * bisect(Site *, Site *) ;
Site * intersect(Halfedge *, Halfedge *) ;
int right_of(Halfedge *, Point *) ;
void endpoint(Edge *, int, Site *) ;
float dist(Site *, Site *) ;
void makevertex(Site *) ;
void deref(Site *) ;
void ref(Site *) ;
extern float deltax, deltay ;
extern int nsites, nedges, sqrt_nsites, nvertices ;
extern Freelist sfl, efl ;

/* heap.c */
void PQinsert(Halfedge *, Site *, float) ;
void PQdelete(Halfedge *) ;
int PQbucket(Halfedge *) ;
int PQempty(void) ;
Point PQ_min(void) ;
Halfedge * PQextractmin(void) ;
void PQinitialize(void) ;
extern int PQmin, PQcount, PQhashsize ;
```

```
extern Halfedge * PQhash ;

/* main.c */
extern int sorted, triangulate, plot, debug, nsites, siteidx ;
extern float xmin, xmax, ymin, ymax ;
extern Site * sites ;
extern Freelist sfl ;

/* memory.c */
void freeinit(Freelist *, int) ;
char *getfree(Freelist *) ;
void makefree(Freenode *, Freelist *) ;
char *myalloc(unsigned) ;

#endif
```

edgelist.c

```
/*** EDGELIST.C ***/

#include "vdefs.h"

int ELhashsize ;
Site * bottomsite ;
Freelist hfl ;
Halfedge * ELleftend, * ELrightend, **ELhash ;

int ntry, totalsearch ;

void
ELinitialize(void)
    {
    int i ;

    freeinit(&hfl, sizeof(Halfedge)) ;
    ELhashsize = 2 * sqrt_nsites ;
```

```
    ELhash = (Halfedge **)myalloc( sizeof(*ELhash) * ELhashsize) ;
    for (i = 0  ; i < ELhashsize  ; i++)
        {
        ELhash[i] = (Halfedge *)NULL ;
        }
    ELleftend = HEcreate((Edge *)NULL, 0) ;
    ELrightend = HEcreate((Edge *)NULL, 0) ;
    ELleftend->ELleft = (Halfedge *)NULL ;
    ELleftend->ELright = ELrightend ;
    ELrightend->ELleft = ELleftend ;
    ELrightend->ELright = (Halfedge *)NULL ;
    ELhash[0] = ELleftend ;
    ELhash[ELhashsize-1] = ELrightend ;
    }

Halfedge *
HEcreate(Edge * e, int pm)
    {
    Halfedge * answer ;

    answer = (Halfedge *)getfree(&hfl) ;
    answer->ELedge = e ;
    answer->ELpm = pm ;
    answer->PQnext = (Halfedge *)NULL ;
    answer->vertex = (Site *)NULL ;
    answer->ELrefcnt = 0 ;
    return (answer) ;
    }

void
ELinsert(Halfedge * lb, Halfedge * newh)
    {
    newh->ELleft = lb ;
    newh->ELright = lb->ELright ;
    (lb->ELright)->ELleft = newh ;
    lb->ELright = newh ;
    }

/* Get entry from hash table, pruning any deleted nodes */
```

```
Halfedge *
ELgethash(int b)
    {
    Halfedge * he ;

    if ((b < 0) || (b >= ELhashsize))
        {
        return ((Halfedge *)NULL) ;
        }
    he = ELhash[b] ;
    if ((he == (Halfedge *)NULL) || (he->ELedge != (Edge *)DELETED))
        {
        return (he) ;
        }
    /* Hash table points to deleted half edge.  Patch as necessary. */
    ELhash[b] = (Halfedge *)NULL ;
    if ((--(he->ELrefcnt)) == 0)
        {
        makefree((Freenode *)he, (Freelist *)&hfl) ;
        }
    return ((Halfedge *)NULL) ;
    }

Halfedge *
ELleftbnd(Point * p)
    {
    int i, bucket ;
    Halfedge * he ;

    /* Use hash table to get close to desired halfedge */
    bucket = (int)((p->x - xmin) / deltax * ELhashsize) ;
    if (bucket < 0)
        {
        bucket = 0 ;
        }
    if (bucket >= ELhashsize)
        {
        bucket = ELhashsize - 1 ;
```

```
    }
he = ELgethash(bucket) ;
if  (he == (Halfedge *)NULL)
    {
    for (i = 1 ; 1 ; i++)
        {
        if ((he = ELgethash(bucket-i)) != (Halfedge *)NULL)
            {
            break ;
            }
        if ((he = ELgethash(bucket+i)) != (Halfedge *)NULL)
            {
            break ;
            }
        }
    totalsearch += i ;
    }
ntry++ ;
/* Now search linear list of halfedges for the corect one */
if (he == ELleftend || (he != ELrightend && right_of(he,p)))
    {
    do  {
        he = he->ELright ;
        } while (he != ELrightend && right_of(he,p)) ;
    he = he->ELleft ;
    }
else
    {
    do  {
        he = he->ELleft ;
        } while (he != ELleftend && !right_of(he,p)) ;
    }
/*** Update hash table and reference counts ***/
if ((bucket > 0) && (bucket < ELhashsize-1))
    {
    if (ELhash[bucket] != (Halfedge *)NULL)
        {
        (ELhash[bucket]->ELrefcnt)-- ;
        }
```

```
        ELhash[bucket] = he ;
        (ELhash[bucket]->ELrefcnt)++ ;
        }
    return (he) ;
    }

/*** This delete routine can't reclaim node, since pointers from hash
 : table may be present.
 ***/

void
ELdelete(Halfedge * he)
    {
    (he->ELleft)->ELright = he->ELright ;
    (he->ELright)->ELleft = he->ELleft ;
    he->ELedge = (Edge *)DELETED ;
    }

Halfedge *
ELright(Halfedge * he)
    {
    return (he->ELright) ;
    }

Halfedge *
ELleft(Halfedge * he)
    {
    return (he->ELleft) ;
    }

Site *
leftreg(Halfedge * he)
    {
    if (he->ELedge == (Edge *)NULL)
        {
        return(bottomsite) ;
        }
    return (he->ELpm == le ? he->ELedge->reg[le] :
        he->ELedge->reg[re]) ;
```

```
    }

Site *
rightreg(Halfedge * he)
    {
    if (he->ELedge == (Edge *)NULL)
        {
        return(bottomsite) ;
        }
    return (he->ELpm == le ? he->ELedge->reg[re] :
        he->ELedge->reg[le]) ;
    }
```

geometry.c

```
/*** GEOMETRY.C ***/

#include <math.h>
#include "vdefs.h"

float deltax, deltay ;
int nedges, sqrt_nsites, nvertices ;
Freelist efl ;

void
geominit(void)
    {
    freeinit(&efl, sizeof(Edge)) ;
    nvertices = nedges = 0 ;
    sqrt_nsites = (int) sqrt(nsites+4) ;
    deltay = ymax - ymin ;
    deltax = xmax - xmin ;
    }

Edge *
bisect(Site * s1, Site * s2)
    {
```

```
    float dx, dy, adx, ady ;
    Edge * newedge ;

    newedge = (Edge *)getfree(&efl) ;
    newedge->reg[0] = s1 ;
    newedge->reg[1] = s2 ;
    ref(s1) ;
    ref(s2) ;
    newedge->ep[0] = newedge->ep[1] = (Site *)NULL ;
    dx = s2->coord.x - s1->coord.x ;
    dy = s2->coord.y - s1->coord.y ;
    adx = dx>0 ? dx : -dx ;
    ady = dy>0 ? dy : -dy ;
    newedge->c = s1->coord.x * dx + s1->coord.y * dy + (dx*dx +
dy*dy) * 0.5 ;
    if (adx > ady)
        {
        newedge->a = 1.0 ;
        newedge->b = dy/dx ;
        newedge->c /= dx ;
        }
    else
        {
        newedge->b = 1.0 ;
        newedge->a = dx/dy ;
        newedge->c /= dy ;
        }
    newedge->edgenbr = nedges ;
    out_bisector(newedge) ;
    nedges++ ;
    return (newedge) ;
    }

Site *
intersect(Halfedge * el1, Halfedge * el2)
    {
    Edge * e1, * e2, * e ;
    Halfedge * el ;
    float d, xint, yint ;
```

```
int right_of_site ;
Site * v ;

e1 = el1->ELedge ;
e2 = el2->ELedge ;
if ((e1 == (Edge*)NULL) || (e2 == (Edge*)NULL))
    {
    return ((Site *)NULL) ;
    }
if (e1->reg[1] == e2->reg[1])
    {
    return ((Site *)NULL) ;
    }
d = (e1->a * e2->b) - (e1->b * e2->a) ;
if ((-1.0e-10 < d) && (d < 1.0e-10))
    {
    return ((Site *)NULL) ;
    }
xint = (e1->c * e2->b - e2->c * e1->b) / d ;
yint = (e2->c * e1->a - e1->c * e2->a) / d ;
if ((e1->reg[1]->coord.y < e2->reg[1]->coord.y) ||
    (e1->reg[1]->coord.y == e2->reg[1]->coord.y &&
    e1->reg[1]->coord.x < e2->reg[1]->coord.x))
    {
    el = el1 ;
    e = e1 ;
    }
else
    {
    el = el2 ;
    e = e2 ;
    }
right_of_site = (xint >= e->reg[1]->coord.x) ;
if ((right_of_site && (el->ELpm == le)) ||
   (!right_of_site && (el->ELpm == re)))
    {
    return ((Site *)NULL) ;
    }
v = (Site *)getfree(&sfl) ;
```

```
    v->refcnt = 0 ;
    v->coord.x = xint ;
    v->coord.y = yint ;
    return (v) ;
    }

/*** returns 1 if p is to right of halfedge e ***/

int
right_of(Halfedge * el, Point * p)
    {
    Edge * e ;
    Site * topsite ;
    int right_of_site, above, fast ;
    float dxp, dyp, dxs, t1, t2, t3, yl ;

    e = el->ELedge ;
    topsite = e->reg[1] ;
    right_of_site = (p->x > topsite->coord.x) ;
    if (right_of_site && (el->ELpm == le))
        {
        return (1) ;
        }
    if(!right_of_site && (el->ELpm == re))
        {
        return (0) ;
        }
    if (e->a == 1.0)
        {
        dyp = p->y - topsite->coord.y ;
        dxp = p->x - topsite->coord.x ;
        fast = 0 ;
        if ((!right_of_site & (e->b < 0.0)) ||
         (right_of_site & (e->b >= 0.0)))
            {
            fast = above = (dyp >= e->b*dxp) ;
            }
        else
            {
```

```
            above = ((p->x + p->y * e->b) > (e->c)) ;
            if (e->b < 0.0)
                {
                above = !above ;
                }
            if (!above)
                {
                fast = 1 ;
                }
            }
        if (!fast)
            {
            dxs = topsite->coord.x - (e->reg[0])->coord.x ;
            above = (e->b * (dxp*dxp - dyp*dyp))
                    <
                    (dxs * dyp * (1.0 + 2.0 * dxp /
                    dxs + e->b * e->b)) ;
            if (e->b < 0.0)
                {
                above = !above ;
                }
            }
        }
    else  /*** e->b == 1.0 ***/
        {
        yl = e->c - e->a * p->x ;
        t1 = p->y - yl ;
        t2 = p->x - topsite->coord.x ;
        t3 = yl - topsite->coord.y ;
        above = ((t1*t1) > ((t2 * t2) + (t3 * t3))) ;
        }
    return (el->ELpm == le ? above : !above) ;
    }

void
endpoint(Edge * e, int lr, Site * s)
    {
    e->ep[lr] = s ;
    ref(s) ;
```

```
    if (e->ep[re-lr] == (Site *)NULL)
        {
        return ;
        }
    out_ep(e) ;
    deref(e->reg[le]) ;
    deref(e->reg[re]) ;
    makefree((Freenode *)e, (Freelist *) &efl) ;
    }

float
dist(Site * s, Site * t)
    {
    float dx,dy ;

    dx = s->coord.x - t->coord.x ;
    dy = s->coord.y - t->coord.y ;
    return (sqrt(dx*dx + dy*dy)) ;
    }

void
makevertex(Site * v)
    {
    v->sitenbr = nvertices++ ;
    out_vertex(v) ;
    }

void
deref(Site * v)
    {
    if (--(v->refcnt) == 0 )
        {
        makefree((Freenode *)v, (Freelist *)&sfl) ;
        }
    }

void
ref(Site * v)
    {
```

```
    ++(v->refcnt) ;
    }
```

heap.c

```
/*** HEAP.C ***/

#include "vdefs.h"

int PQmin, PQcount, PQhashsize ;
Halfedge * PQhash ;

void
PQinsert(Halfedge * he, Site * v, float offset)
    {
    Halfedge * last, * next ;

    he->vertex = v ;
    ref(v) ;
    he->ystar = v->coord.y + offset ;
    last = &PQhash[ PQbucket(he)] ;
    while ((next = last->PQnext) != (Halfedge *)NULL &&
      (he->ystar  > next->ystar  ||
      (he->ystar == next->ystar &&
      v->coord.x > next->vertex->coord.x)))
        {
        last = next ;
        }
    he->PQnext = last->PQnext ;
    last->PQnext = he ;
    PQcount++ ;
    }

void
PQdelete(Halfedge * he)
    {
```

```
    Halfedge * last;

    if(he ->  vertex != (Site *) NULL)
        {
        last = &PQhash[PQbucket(he)] ;
        while (last -> PQnext != he)
            {
            last = last->PQnext ;
            }
        last->PQnext = he->PQnext;
        PQcount-- ;
        deref(he->vertex) ;
        he->vertex = (Site *)NULL ;
        }
    }

int
PQbucket(Halfedge * he)
    {
    int bucket ;


    if     (he->ystar < ymin)  bucket = 0;
    else if (he->ystar >= ymax) bucket = PQhashsize-1;
    else
       bucket = (int)((he->ystar - ymin)/deltay * PQhashsize);
    if (bucket < 0)
        {
        bucket = 0 ;
        }
    if (bucket >= PQhashsize)
        {
        bucket = PQhashsize-1 ;
        }
    if (bucket < PQmin)
        {
        PQmin = bucket ;
        }
    return (bucket);
```

```
    }

int
PQempty(void)
    {
    return (PQcount == 0) ;
    }


Point
PQ_min(void)
    {
    Point answer ;

    while (PQhash[PQmin].PQnext == (Halfedge *)NULL)
        {
        ++PQmin ;
        }
    answer.x = PQhash[PQmin].PQnext->vertex->coord.x ;
    answer.y = PQhash[PQmin].PQnext->ystar ;
    return (answer) ;
    }

Halfedge *
PQextractmin(void)
    {
    Halfedge * curr ;

    curr = PQhash[PQmin].PQnext ;
    PQhash[PQmin].PQnext = curr->PQnext ;
    PQcount-- ;
    return (curr) ;
    }

void
PQinitialize(void)
    {
    int i ;
```

```
    PQcount = PQmin = 0 ;
    PQhashsize = 4 * sqrt_nsites ;
    PQhash = (Halfedge *)myalloc(PQhashsize * sizeof *PQhash) ;
    for (i = 0 ; i < PQhashsize; i++)
        {
        PQhash[i].PQnext = (Halfedge *)NULL ;
        }
    }
```

memory.c

```
/*** MEMORY.C ***/

#include <stdio.h>
#include <stdlib.h>  /* malloc(), exit() */

#include "vdefs.h"

extern int sqrt_nsites, siteidx ;

void
freeinit(Freelist * fl, int size)
    {
    fl->head = (Freenode *)NULL ;
    fl->nodesize = size ;
    }

char *
getfree(Freelist * fl)
    {
    int i ;
    Freenode * t ;
    if (fl->head == (Freenode *)NULL)
        {
        t =  (Freenode *) myalloc(sqrt_nsites * fl->nodesize) ;
        for(i = 0 ; i < sqrt_nsites ; i++)
            {
```

```
            makefree((Freenode *)((char *)t+i*fl->nodesize), fl) ;
            }
        }
    t = fl->head ;
    fl->head = (fl->head)->nextfree ;
    return ((char *)t) ;
    }

void
makefree(Freenode * curr, Freelist * fl)
    {
    curr->nextfree = fl->head ;
    fl->head = curr ;
    }

int total_alloc ;

char *
myalloc(unsigned n)
    {
    char * t ;
    if ((t=(char *)malloc(n)) == (char *) 0)
        {
        fprintf(stderr,
        "Insufficient memory processing site %d (%d bytes in use)\n",
        siteidx, total_alloc) ;
        exit(0) ;
        }
    total_alloc += n ;
    return (t) ;
    }
```

APPENDIX E (SOURCE CODES FOR MAIN FUNCTION)

In this section, `kvoronoi.cpp` and `svoronoi.cpp` are listed. The file `kvoronoi.cpp` includes the `main()` function for kinetic Voronoi algorithm and `svoronoi.cpp` includes the `main()` function for static Voronoi algorithm

kvoronoi.cpp

```
#include <iostream.h>

#ifdef MYOPENGL
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#endif

#include <sys/time.h>
#include <time.h>
#include <unistd.h>


#include "herd.hpp"

Herd herd_A;
int loop_num = 0;

#ifdef MYOPENGL
//<<<<<<<<<<<<<<<<<<<<<<< myInit >>>>>>>>>>>>>>>>>>>
void myInit(void)
{
  glClearColor(1.0,1.0,1.0,0.0);          // set white background color
  glColor3f(0.0f, 0.0f, 0.0f);            // set the drawing color
```

```
  glPointSize(4.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, 200.0, 0.0, 200.0);
}



//<<<<<<<<<<<<<<<<<<<<<<<< myDisplay >>>>>>>>>>>>>>>>>
void myDisplay(void)
{
  herd_A.DisplayHerd();
}


void myIdle(void){
  sleep(3);
  loop_num++;
  if(loop_num>20) exit(0);

  herd_A.MoveAllPrey();
  glutPostRedisplay();
}


//<<<<<<<<<<<<<<<<<<<<<<<< keyboard >>>>>>>>>>>>>>>>>
void keyboard(unsigned char key, int x, int y)
{
  switch(key) {
  case 27:
  case 'q':
    exit(0);
  }
  glutPostRedisplay();
}
#endif
```

```
int main(int argc, char** argv){
  int i, j, movecount, count;
  position allpos[PREYNUM];

#ifdef MYOPENGL
  glutInit(&argc, argv);          // initialize the toolkit
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); // set display mode
  glutInitWindowSize(200,200);     // set window size
  glutInitWindowPosition(100, 100); // set window position on screen
  glutCreateWindow("Simulation for Selfish Herd");
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(myDisplay);     // register redraw function
  glutIdleFunc(myIdle);
  myInit();

  glutMainLoop();       // go into a perpetual loop
#endif

#ifndef MYOPENGL
  struct timeval tv1,tv2;
  double diff_time[20];

  while(1){
    loop_num++;
    if(loop_num>20){
      printf("End of simulation\n");
      printf("Time\n");
      for(int i=0;i<20;i++){
printf("%12.3f \n", diff_time[i]);
      }
      exit(0);
    }
    gettimeofday(&tv2,NULL);
    herd_A.MoveAllPrey();
    gettimeofday(&tv1,NULL);
    diff_time[loop_num-1] = (tv1.tv_sec-tv2.tv_sec)*1000000
      +(tv1.tv_usec-tv2.tv_usec);
  }
#endif
```

```
}
```

svoronoi.cpp

```
#include <iostream.h>

#ifdef MYOPENGL
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#endif

#include <sys/time.h>
#include <time.h>
#include <unistd.h>



#include "herdstatic.hpp"

Herd herd_A;
int loop_num = 0;

#ifdef MYOPENGL
//<<<<<<<<<<<<<<<<<<<<<<< myInit >>>>>>>>>>>>>>>>>>>
void myInit(void)
{
  glClearColor(1.0,1.0,1.0,0.0);          // set white background color
  glColor3f(0.0f, 0.0f, 0.0f);            // set the drawing color
  glPointSize(4.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, 200.0, 0.0, 200.0);
}
```

```
//<<<<<<<<<<<<<<<<<<<<<<<< myDisplay >>>>>>>>>>>>>>>>>>
void myDisplay(void)
{
  herd_A.DisplayHerd();
}


void myIdle(void){
  sleep(3);
  loop_num++;
  if(loop_num>20) exit(0);

  herd_A.MoveAllPreyStatic();
  glutPostRedisplay();
}


//<<<<<<<<<<<<<<<<<<<<<<<< keyboard >>>>>>>>>>>>>>>>>>
void keyboard(unsigned char key, int x, int y)
{
  switch(key) {
  case 27:
  case 'q':
    exit(0);
  }
  glutPostRedisplay();
}
#endif



int main(int argc, char** argv){
  int i, j, movecount, count;
  position allpos[PREYNUM];



#ifdef MYOPENGL
```

```
  glutInit(&argc, argv);          // initialize the toolkit
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB); // set display mode
  glutInitWindowSize(200,200);     // set window size
  glutInitWindowPosition(100, 100); // set window position on screen
  glutCreateWindow("Simulation for Selfish Herd");
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(myDisplay);     // register redraw function
  glutIdleFunc(myIdle);
  myInit();

  glutMainLoop();       // go into a perpetual loop
#endif

#ifndef MYOPENGL
  struct timeval tv1,tv2;
  double diff_time[20];

  while(1){
    loop_num++;
    if(loop_num>20){
      printf("End of simulation\n");
      printf("Time\n");
      for(int i=0;i<20;i++){
printf("%12.3f \n", diff_time[i]);
      }
      exit(0);
    }
    gettimeofday(&tv2,NULL);
    herd_A.MoveAllPreyStatic();
    gettimeofday(&tv1,NULL);
    diff_time[loop_num-1] = (tv1.tv_sec-tv2.tv_sec)*1000000
      +(tv1.tv_usec-tv2.tv_usec);
  }
#endif
}
```

APPENDIX F (OTHER SOURCE CODES)

The source codes `myheader.hpp` and `myheader.cpp` are listed. Global functions or variables are defined in the files.

myheader.hpp

```
#ifndef _MYHEADER_H
#define _MYHEADER_H

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>

#define PREYNUM 1000
#define TRUE  1
#define FALSE 0
#define EMPTY -2
#define MAXNUM   100000
#define INFINITE 1000000

typedef struct myposition{
  double x;
  double y;
}position;

typedef struct mytriangle{
  int bot;
  int top;
  int right;
```

```
  struct mytriangle *next;
}triangle;

typedef struct myeventque{
  int triangle1[3], triangle2[3];
  double time;
  struct myeventque *next;
}eventque;


int    Randomnum(void);
double Distance(position *from, position *to);
void   NormToward(position *from, position *to, position *one);

void AddNextTri(triangle *head, int bot, int top, int right);
int  DelNextTri(triangle *head);
void DelAllTri(triangle *head);


void AddInOrderEvent(eventque *head, int a1, int a2, int a3,
     int b1, int b2, int b3, double time);
void AddNextEvent(eventque *head, int a1, int a2, int a3,
  int b1, int b2, int b3, double time);
int  DelNextEvent(eventque *head);
void DelAllEvent(eventque *head);


void FindCenter(double xa, double ya, double xb, double yb,
double xc, double yc, double *x, double *y);


/*********************************************/
/*  Functions To Find coefficient            */
/*********************************************/

double coefficient_t_4(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd);
```

```
double coefficient_t_3(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd);
double coefficient_t_2(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd);
double coefficient_t_1(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd);
double coefficient_t_0(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd);


#endif
```

myheader.cpp

```
#include "myheader.hpp"


/*-----------------------------------------*/
/*- randomnum() will generate random number. -*/
/*-----------------------------------------*/

int Randomnum(void){
  // srand(time(NULL));
  srand(rand());

  return(rand());
}
```

```
/*-----------------------------------------*/
/*- Distance(position *from, position *to)   -*/
/*- calculates distance from position "from" -*/
/*- to position "to".                        -*/
/*-----------------------------------------*/

double Distance(position *from, position *to){
  double xx, yy;

  xx = (to->x - from->x);
  yy = (to->y - from->y);

  return sqrt ( (xx * xx) + (yy * yy) );
}


/*-----------------------------------------------------------*/
/*- NormToward(position *from, position *to, position *one) -*/
/*-  calculates structure position "one" which includes x   -*/
/*-  and y of each move.                                    -*/
/*-----------------------------------------------------------*/

void NormToward(position *from, position *to, position *one){
  double dis;

  dis = Distance(from, to);
  one->x = (to->x - from->x) / dis;
  one->y = (to->y - from->y) / dis;
}


void AddNextTri(triangle *head, int bot, int top, int right){
  triangle *temp, *newtri;

  temp = head;
  newtri = (triangle *)malloc(sizeof(triangle));

  newtri->bot = bot;
  newtri->top = top;
```

```
  newtri->right = right;

  newtri->next = temp->next;
  temp->next = newtri;
}



int DelNextTri(triangle *head){
  triangle *temp = head;

  if(temp->next!=NULL){
    //    printf("Delete Triangle next.\n");
    temp = temp->next;
    head->next = temp->next;
    delete temp;
    return TRUE;
  }
  return FALSE;
}




void DelAllTri(triangle *head){
  while(DelNextTri(head));
}


void AddInOrderEvent(eventque *head, int a1, int a2, int a3,
     int b1, int b2, int b3, double time){
  eventque *temp = head;

  // if temp->next.time    < time
  while(temp->next != NULL &&  temp->next->time < time){
    temp = temp->next;
  }
  AddNextEvent(temp, a1, a2, a3, b1, b2, b3, time);
```

```
}


void AddNextEvent(eventque *head, int a1, int a2, int a3,
  int b1, int b2, int b3, double time){
  eventque *temp, *newevent;

  temp = head;
  newevent = (eventque *)malloc(sizeof(eventque));

  newevent->triangle1[0] = a1;
  newevent->triangle1[1] = a2;
  newevent->triangle1[2] = a3;
  newevent->triangle2[0] = b1;
  newevent->triangle2[1] = b2;
  newevent->triangle2[2] = b3;
  newevent->time = time;


  newevent->next = temp->next;
  temp->next = newevent;
}



int DelNextEvent(eventque *head){
  eventque *temp = head;

  if(temp->next!=NULL){
    temp = temp->next;
    head->next = temp->next;
    delete temp;
    return TRUE;
  }
  return FALSE;
}
```

```
void DelAllEvent(eventque *head){
  while(DelNextEvent(head));
}



void FindCenter(double xa, double ya, double xb, double yb,
double xc, double yc, double *x, double *y){
  double temp;

  // if(xa-xb > -0.5 && xa-xb < 0.5){
  if(xa-xb > -0.01 && xa-xb < 0.01){
  // if(xa==xb){
    if(ya-yc > -0.01 && ya-yc < 0.01){
    // if(ya==yc){
      // ==========  First case  ========== //
      (*x) = (xa + xc)/2.0;
      (*y) = (ya + yb)/2.0;
    }else{
      // ==========  Second case  ========== //
      (*y) = (ya + yb)/2.0;
      (*x) = ((*y)-(ya+yc)/2.0)*((-yc+ya)/(xc-xa)) + (xa+xc)/2.0;
    }
  }else{
    if(ya-yb > -0.01 && ya-yb < 0.01){
    // if(ya==yb){
       if(xa-xc > -0.01 && xa-xc < 0.01){
      // if(xa==xc){
      // ==========  Third case  ========== //
(*x) = (xa + xb)/2.0;
(*y) = (ya + yc)/2.0;
      }else{
      // ==========  Fourth case  ========== //
(*x) = (xa + xb)/2.0;
```

```
(*y) = ((*x)-(xa+xc)/2.0)*((-xc+xa)/(yc-ya))+(ya+yc)/2.0;
      }
    }else{
      // ==========  Fifth case  ========== //
      // xa!=xb && ya!=yb && xa!=xc && ya!=yc

      (*x) = (xc*xc-(xa*xa))/(2.0*(yc-ya))
     -(xb*xb-(xa*xa))/(2.0*(yb-ya)) +(yc-yb)/2.0;
      temp = (xc-xa)/(yc-ya)-(xb-xa)/(yb-ya);
      (*x) /= temp;

      (*y) = ((*x)-(xa+xc)/2.0)*((-xc+xa)/(yc-ya))+(ya+yc)/2.0;

    }
  }
}
```

```
/*********************************

CForm[Coefficient[Ans, t, 4]]

*********************************/

double coefficient_t_4(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd){

  return
    xvb*xvb*xvc*yva - xvb*xvc*xvc*yva - xvb*xvb*xvd*yva + xvc*xvc*xvd*yva +
    xvb*xvd*xvd*yva - xvc*xvd*xvd*yva - xva*xva*xvc*yvb + xva*xvc*xvc*yvb +
    xva*xva*xvd*yvb - xvc*xvc*xvd*yvb - xva*xvd*xvd*yvb + xvc*xvd*xvd*yvb -
    xvc*yva*yva*yvb + xvd*yva*yva*yvb + xvc*yva*yvb*yvb - xvd*yva*yvb*yvb +
    xva*xva*xvb*yvc - xva*xvb*xvb*yvc - xva*xva*xvd*yvc + xvb*xvb*xvd*yvc +
    xva*xvd*xvd*yvc - xvb*xvd*xvd*yvc + xvb*yva*yva*yvc - xvd*yva*yva*yvc -
```

```
    xva*yvb*yvb*yvc + xvd*yvb*yvb*yvc - xvb*yva*yvc*yvc + xvd*yva*yvc*yvc +
    xva*yvb*yvc*yvc - xvd*yvb*yvc*yvc - xva*xva*xvb*yvd + xva*xvb*xvb*yvd +
    xva*xva*xvc*yvd - xvb*xvb*xvc*yvd - xva*xvc*xvc*yvd + xvb*xvc*xvc*yvd -
    xvb*yva*yva*yvd + xvc*yva*yva*yvd + xva*yvb*yvb*yvd - xvc*yvb*yvb*yvd -
    xva*yvc*yvc*yvd + xvb*yvc*yvc*yvd + xvb*yva*yvd*yvd - xvc*yva*yvd*yvd -
    xva*yvb*yvd*yvd + xvc*yvb*yvd*yvd + xva*yvc*yvd*yvd - xvb*yvc*yvd*yvd;
}


/*********************************

CForm[Coefficient[Ans, t, 3]]

*********************************/

double coefficient_t_3(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd){

  return
    xvb*xvb*xvc*ya - xvb*xvc*xvc*ya - xvb*xvb*xvd*ya + xvc*xvc*xvd*ya +
    xvb*xvd*xvd*ya - xvc*xvd*xvd*ya - xva*xva*xvc*yb + xva*xvc*xvc*yb +
    xva*xva*xvd*yb - xvc*xvc*xvd*yb - xva*xvd*xvd*yb + xvc*xvd*xvd*yb +
    xva*xva*xvb*yc - xva*xvb*xvb*yc - xva*xva*xvd*yc + xvb*xvb*xvd*yc +
    xva*xvd*xvd*yc - xvb*xvd*xvd*yc - xva*xva*xvb*yd + xva*xvb*xvb*yd +
    xva*xva*xvc*yd - xvb*xvb*xvc*yd - xva*xvc*xvc*yd + xvb*xvc*xvc*yd +
    xc*xvb*xvb*yva - xd*xvb*xvb*yva + 2*xb*xvb*xvc*yva - 2*xc*xvb*xvc*yva -
    xb*xvc*xvc*yva + xd*xvc*xvc*yva - 2*xb*xvb*xvd*yva + 2*xd*xvb*xvd*yva +
    2*xc*xvc*xvd*yva - 2*xd*xvc*xvd*yva + xb*xvd*xvd*yva - xc*xvd*xvd*yva -
    xvc*yb*yva*yva + xvd*yb*yva*yva + xvb*yc*yva*yva - xvd*yc*yva*yva -
    xvb*yd*yva*yva + xvc*yd*yva*yva - xc*xva*xva*yvb + xd*xva*xva*yvb -
    2*xa*xva*xvc*yvb + 2*xc*xva*xvc*yvb + xa*xvc*xvc*yvb - xd*xvc*xvc*yvb +
    2*xa*xva*xvd*yvb - 2*xd*xva*xvd*yvb - 2*xc*xvc*xvd*yvb
    + 2*xd*xvc*xvd*yvb -
    xa*xvd*xvd*yvb + xc*xvd*xvd*yvb - 2*xvc*ya*yva*yvb + 2*xvd*ya*yva*yvb +
    2*xvc*yb*yva*yvb - 2*xvd*yb*yva*yvb - xc*yva*yva*yvb + xd*yva*yva*yvb +
    xvc*ya*yvb*yvb - xvd*ya*yvb*yvb - xva*yc*yvb*yvb + xvd*yc*yvb*yvb +
    xva*yd*yvb*yvb - xvc*yd*yvb*yvb + xc*yva*yvb*yvb - xd*yva*yvb*yvb +
```

```
    xb*xva*xva*yvc - xd*xva*xva*yvc + 2*xa*xva*xvb*yvc - 2*xb*xva*xvb*yvc -
    xa*xvb*xvb*yvc + xd*xvb*xvb*yvc - 2*xa*xva*xvd*yvc + 2*xd*xva*xvd*yvc +
    2*xb*xvb*xvd*yvc - 2*xd*xvb*xvd*yvc + xa*xvd*xvd*yvc - xb*xvd*xvd*yvc +
    2*xvb*ya*yva*yvc - 2*xvd*ya*yva*yvc - 2*xvb*yc*yva*yvc
    + 2*xvd*yc*yva*yvc +
    xb*yva*yva*yvc - xd*yva*yva*yvc - 2*xva*yb*yvb*yvc + 2*xvd*yb*yvb*yvc +
    2*xva*yc*yvb*yvc - 2*xvd*yc*yvb*yvc - xa*yvb*yvb*yvc + xd*yvb*yvb*yvc -
    xvb*ya*yvc*yvc + xvd*ya*yvc*yvc + xva*yb*yvc*yvc - xvd*yb*yvc*yvc -
    xva*yd*yvc*yvc + xvb*yd*yvc*yvc - xb*yva*yvc*yvc + xd*yva*yvc*yvc +
    xa*yvb*yvc*yvc - xd*yvb*yvc*yvc - xb*xva*xva*yvd + xc*xva*xva*yvd -
    2*xa*xva*xvb*yvd + 2*xb*xva*xvb*yvd + xa*xvb*xvb*yvd - xc*xvb*xvb*yvd +
   2*xa*xva*xvc*yvd - 2*xc*xva*xvc*yvd - 2*xb*xvb*xvc*yvd
   + 2*xc*xvb*xvc*yvd -
   xa*xvc*xvc*yvd + xb*xvc*xvc*yvd - 2*xvb*ya*yva*yvd + 2*xvc*ya*yva*yvd +
   2*xvb*yd*yva*yvd - 2*xvc*yd*yva*yvd - xb*yva*yva*yvd + xc*yva*yva*yvd +
   2*xva*yb*yvb*yvd - 2*xvc*yb*yvb*yvd - 2*xva*yd*yvb*yvd
   + 2*xvc*yd*yvb*yvd +
   xa*yvb*yvb*yvd - xc*yvb*yvb*yvd - 2*xva*yc*yvc*yvd + 2*xvb*yc*yvc*yvd +
   2*xva*yd*yvc*yvd - 2*xvb*yd*yvc*yvd - xa*yvc*yvc*yvd + xb*yvc*yvc*yvd +
   xvb*ya*yvd*yvd - xvc*ya*yvd*yvd - xva*yb*yvd*yvd + xvc*yb*yvd*yvd +
   xva*yc*yvd*yvd - xvb*yc*yvd*yvd + xb*yva*yvd*yvd - xc*yva*yvd*yvd -
   xa*yvb*yvd*yvd + xc*yvb*yvd*yvd + xa*yvc*yvd*yvd - xb*yvc*yvd*yvd;
}


/*********************************

CForm[Coefficient[Ans, t, 2]]

*********************************/

double coefficient_t_2(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd){

  return
    xc*xvb*xvb*ya - xd*xvb*xvb*ya + 2*xb*xvb*xvc*ya - 2*xc*xvb*xvc*ya -
    xb*xvc*xvc*ya + xd*xvc*xvc*ya - 2*xb*xvb*xvd*ya + 2*xd*xvb*xvd*ya +
```

```
2*xc*xvc*xvd*ya - 2*xd*xvc*xvd*ya + xb*xvd*xvd*ya - xc*xvd*xvd*ya -
xc*xva*xva*yb + xd*xva*xva*yb - 2*xa*xva*xvc*yb + 2*xc*xva*xvc*yb +
xa*xvc*xvc*yb - xd*xvc*xvc*yb + 2*xa*xva*xvd*yb - 2*xd*xva*xvd*yb -
2*xc*xvc*xvd*yb + 2*xd*xvc*xvd*yb - xa*xvd*xvd*yb + xc*xvd*xvd*yb +
xb*xva*xva*yc - xd*xva*xva*yc + 2*xa*xva*xvb*yc - 2*xb*xva*xvb*yc -
xa*xvb*xvb*yc + xd*xvb*xvb*yc - 2*xa*xva*xvd*yc + 2*xd*xva*xvd*yc +
2*xb*xvb*xvd*yc - 2*xd*xvb*xvd*yc + xa*xvd*xvd*yc - xb*xvd*xvd*yc -
xb*xva*xva*yd + xc*xva*xva*yd - 2*xa*xva*xvb*yd + 2*xb*xva*xvb*yd +
xa*xvb*xvb*yd - xc*xvb*xvb*yd + 2*xa*xva*xvc*yd - 2*xc*xva*xvc*yd -
2*xb*xvb*xvc*yd + 2*xc*xvb*xvc*yd - xa*xvc*xvc*yd + xb*xvc*xvc*yd +
2*xb*xc*xvb*yva - xc*xc*xvb*yva - 2*xb*xd*xvb*yva + xd*xd*xvb*yva +
xb*xb*xvc*yva - 2*xb*xc*xvc*yva + 2*xc*xd*xvc*yva - xd*xd*xvc*yva -
xb*xb*xvd*yva + xc*xc*xvd*yva + 2*xb*xd*xvd*yva - 2*xc*xd*xvd*yva -
2*xvc*ya*yb*yva + 2*xvd*ya*yb*yva + xvc*yb*yb*yva - xvd*yb*yb*yva +
2*xvb*ya*yc*yva - 2*xvd*ya*yc*yva - xvb*yc*yc*yva + xvd*yc*yc*yva -
2*xvb*ya*yd*yva + 2*xvc*ya*yd*yva + xvb*yd*yd*yva - xvc*yd*yd*yva -
xc*yb*yva*yva + xd*yb*yva*yva + xb*yc*yva*yva - xd*yc*yva*yva -
xb*yd*yva*yva + xc*yd*yva*yva - 2*xa*xc*xva*yvb + xc*xc*xva*yvb +
2*xa*xd*xva*yvb - xd*xd*xva*yvb - xa*xa*xvc*yvb + 2*xa*xc*xvc*yvb -
2*xc*xd*xvc*yvb + xd*xd*xvc*yvb + xa*xa*xvd*yvb - xc*xc*xvd*yvb -
2*xa*xd*xvd*yvb + 2*xc*xd*xvd*yvb - xvc*ya*ya*yvb + xvd*ya*ya*yvb +
2*xvc*ya*yb*yvb - 2*xvd*ya*yb*yvb - 2*xva*yb*yc*yvb
+ 2*xvd*yb*yc*yvb +
xva*yc*yc*yvb - xvd*yc*yc*yvb + 2*xva*yb*yd*yvb - 2*xvc*yb*yd*yvb -
xva*yd*yd*yvb + xvc*yd*yd*yvb - 2*xc*ya*yva*yvb + 2*xd*ya*yva*yvb +
2*xc*yb*yva*yvb - 2*xd*yb*yva*yvb + xc*ya*yvb*yvb - xd*ya*yvb*yvb -
xa*yc*yvb*yvb + xd*yc*yvb*yvb + xa*yd*yvb*yvb - xc*yd*yvb*yvb +
2*xa*xb*xva*yvc - xb*xb*xva*yvc - 2*xa*xd*xva*yvc + xd*xd*xva*yvc +
xa*xa*xvb*yvc - 2*xa*xb*xvb*yvc + 2*xb*xd*xvb*yvc - xd*xd*xvb*yvc -
xa*xa*xvd*yvc + xb*xb*xvd*yvc + 2*xa*xd*xvd*yvc - 2*xb*xd*xvd*yvc +
xvb*ya*ya*yvc - xvd*ya*ya*yvc - xva*yb*yb*yvc + xvd*yb*yb*yvc -
2*xvb*ya*yc*yvc + 2*xvd*ya*yc*yvc + 2*xva*yb*yc*yvc
- 2*xvd*yb*yc*yvc -
2*xva*yc*yd*yvc + 2*xvb*yc*yd*yvc + xva*yd*yd*yvc - xvb*yd*yd*yvc +
2*xb*ya*yva*yvc - 2*xd*ya*yva*yvc - 2*xb*yc*yva*yvc
+ 2*xd*yc*yva*yvc - 2*xa*yb*yvb*yvc + 2*xd*yb*yvb*yvc +
2*xa*yc*yvb*yvc - 2*xd*yc*yvb*yvc -
xb*ya*yvc*yvc + xd*ya*yvc*yvc + xa*yb*yvc*yvc - xd*yb*yvc*yvc -
xa*yd*yvc*yvc + xb*yd*yvc*yvc - 2*xa*xb*xva*yvd + xb*xb*xva*yvd +
```

```
    2*xa*xc*xva*yvd - xc*xc*xva*yvd - xa*xa*xvb*yvd + 2*xa*xb*xvb*yvd -
    2*xb*xc*xvb*yvd + xc*xc*xvb*yvd + xa*xa*xvc*yvd - xb*xb*xvc*yvd -
    2*xa*xc*xvc*yvd + 2*xb*xc*xvc*yvd - xvb*ya*ya*yvd + xvc*ya*ya*yvd +
    xva*yb*yb*yvd - xvc*yb*yb*yvd - xva*yc*yc*yvd + xvb*yc*yc*yvd +
    2*xvb*ya*yd*yvd - 2*xvc*ya*yd*yvd - 2*xva*yb*yd*yvd + 2*xvc*yb*yd*yvd +
    2*xva*yc*yd*yvd - 2*xvb*yc*yd*yvd - 2*xb*ya*yva*yvd + 2*xc*ya*yva*yvd +
    2*xb*yd*yva*yvd - 2*xc*yd*yva*yvd + 2*xa*yb*yvb*yvd - 2*xc*yb*yvb*yvd -
    2*xa*yd*yvb*yvd + 2*xc*yd*yvb*yvd - 2*xa*yc*yvc*yvd + 2*xb*yc*yvc*yvd +
    2*xa*yd*yvc*yvd - 2*xb*yd*yvc*yvd + xb*ya*yvd*yvd - xc*ya*yvd*yvd -
    xa*yb*yvd*yvd + xc*yb*yvd*yvd + xa*yc*yvd*yvd - xb*yc*yvd*yvd;
}



/*********************************

CForm[Coefficient[Ans, t, 1]]

*********************************/

double coefficient_t_1(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd){

  return
    2*xb*xc*xvb*ya - xc*xc*xvb*ya - 2*xb*xd*xvb*ya + xd*xd*xvb*ya +
    xb*xb*xvc*ya - 2*xb*xc*xvc*ya + 2*xc*xd*xvc*ya - xd*xd*xvc*ya -
    xb*xb*xvd*ya + xc*xc*xvd*ya + 2*xb*xd*xvd*ya - 2*xc*xd*xvd*ya -
    2*xa*xc*xva*yb + xc*xc*xva*yb + 2*xa*xd*xva*yb - xd*xd*xva*yb -
    xa*xa*xvc*yb + 2*xa*xc*xvc*yb - 2*xc*xd*xvc*yb + xd*xd*xvc*yb +
    xa*xa*xvd*yb - xc*xc*xvd*yb - 2*xa*xd*xvd*yb + 2*xc*xd*xvd*yb -
    xvc*ya*ya*yb + xvd*ya*ya*yb + xvc*ya*yb*yb - xvd*ya*yb*yb +
    2*xa*xb*xva*yc - xb*xb*xva*yc - 2*xa*xd*xva*yc + xd*xd*xva*yc +
    xa*xa*xvb*yc - 2*xa*xb*xvb*yc + 2*xb*xd*xvb*yc - xd*xd*xvb*yc -
    xa*xa*xvd*yc + xb*xb*xvd*yc + 2*xa*xd*xvd*yc - 2*xb*xd*xvd*yc +
    xvb*ya*ya*yc - xvd*ya*ya*yc - xva*yb*yb*yc + xvd*yb*yb*yc -
    xvb*ya*yc*yc + xvd*ya*yc*yc + xva*yb*yc*yc - xvd*yb*yc*yc -
    2*xa*xb*xva*yd + xb*xb*xva*yd + 2*xa*xc*xva*yd - xc*xc*xva*yd -
```

```
    xa*xa*xvb*yd + 2*xa*xb*xvb*yd - 2*xb*xc*xvb*yd + xc*xc*xvb*yd +
    xa*xa*xvc*yd - xb*xb*xvc*yd - 2*xa*xc*xvc*yd + 2*xb*xc*xvc*yd -
    xvb*ya*ya*yd + xvc*ya*ya*yd + xva*yb*yb*yd - xvc*yb*yb*yd -
    xva*yc*yc*yd + xvb*yc*yc*yd + xvb*ya*yd*yd - xvc*ya*yd*yd -
    xva*yb*yd*yd + xvc*yb*yd*yd + xva*yc*yd*yd - xvb*yc*yd*yd +
    xb*xb*xc*yva - xb*xc*xc*yva - xb*xb*xd*yva + xc*xc*xd*yva +
    xb*xd*xd*yva - xc*xd*xd*yva - 2*xc*ya*yb*yva + 2*xd*ya*yb*yva +
    xc*yb*yb*yva - xd*yb*yb*yva + 2*xb*ya*yc*yva - 2*xd*ya*yc*yva -
    xb*yc*yc*yva + xd*yc*yc*yva - 2*xb*ya*yd*yva + 2*xc*ya*yd*yva +
    xb*yd*yd*yva - xc*yd*yd*yva - xa*xa*xc*yvb + xa*xc*xc*yvb +
    xa*xa*xd*yvb - xc*xc*xd*yvb - xa*xd*xd*yvb + xc*xd*xd*yvb -
    xc*ya*ya*yvb + xd*ya*ya*yvb + 2*xc*ya*yb*yvb - 2*xd*ya*yb*yvb -
    2*xa*yb*yc*yvb + 2*xd*yb*yc*yvb + xa*yc*yc*yvb - xd*yc*yc*yvb +
    2*xa*yb*yd*yvb - 2*xc*yb*yd*yvb - xa*yd*yd*yvb + xc*yd*yd*yvb +
    xa*xa*xb*yvc - xa*xb*xb*yvc - xa*xa*xd*yvc + xb*xb*xd*yvc +
    xa*xd*xd*yvc - xb*xd*xd*yvc + xb*ya*ya*yvc - xd*ya*ya*yvc -
    xa*yb*yb*yvc + xd*yb*yb*yvc - 2*xb*ya*yc*yvc + 2*xd*ya*yc*yvc +
    2*xa*yb*yc*yvc - 2*xd*yb*yc*yvc - 2*xa*yc*yd*yvc + 2*xb*yc*yd*yvc +
    xa*yd*yd*yvc - xb*yd*yd*yvc - xa*xa*xb*yvd + xa*xb*xb*yvd +
    xa*xa*xc*yvd - xb*xb*xc*yvd - xa*xc*xc*yvd + xb*xc*xc*yvd -
    xb*ya*ya*yvd + xc*ya*ya*yvd + xa*yb*yb*yvd - xc*yb*yb*yvd -
    xa*yc*yc*yvd + xb*yc*yc*yvd + 2*xb*ya*yd*yvd - 2*xc*ya*yd*yvd -
   2*xa*yb*yd*yvd + 2*xc*yb*yd*yvd + 2*xa*yc*yd*yvd - 2*xb*yc*yd*yvd;
}

/*********************************

CForm[Coefficient[Ans, t, 0]]

*********************************/

double coefficient_t_0(double xa, double xb, double xc, double xd,
       double ya, double yb, double yc, double yd,
       double xva, double xvb, double xvc, double xvd,
       double yva, double yvb, double yvc, double yvd){

  return
    xb*xb*xc*ya - xb*xc*xc*ya - xb*xb*xd*ya + xc*xc*xd*ya + xb*xd*xd*ya -
    xc*xd*xd*ya - xa*xa*xc*yb + xa*xc*xc*yb + xa*xa*xd*yb - xc*xc*xd*yb -
```

```
    xa*xd*xd*yb + xc*xd*xd*yb - xc*ya*ya*yb + xd*ya*ya*yb + xc*ya*yb*yb -
    xd*ya*yb*yb + xa*xa*xb*yc - xa*xb*xb*yc - xa*xa*xd*yc + xb*xb*xd*yc +
    xa*xd*xd*yc - xb*xd*xd*yc + xb*ya*ya*yc - xd*ya*ya*yc - xa*yb*yb*yc +
    xd*yb*yb*yc - xb*ya*yc*yc + xd*ya*yc*yc + xa*yb*yc*yc - xd*yb*yc*yc -
    xa*xa*xb*yd + xa*xb*xb*yd + xa*xa*xc*yd - xb*xb*xc*yd - xa*xc*xc*yd +
    xb*xc*xc*yd - xb*ya*ya*yd + xc*ya*ya*yd + xa*yb*yb*yd - xc*yb*yb*yd -
    xa*yc*yc*yd + xb*yc*yc*yd + xb*ya*yd*yd - xc*ya*yd*yd - xa*yb*yd*yd +
    xc*yb*yd*yd + xa*yc*yd*yd - xb*yc*yd*yd;
}
```

APPENDIX G (MAKEFILE)

The file`Makefile` is listed in this section. When a simulation with graphics needs to be executed, "OPENGL=-DMYOPENGL" is reuired in this file.

Makefile

```
CC = g++ -ansi -Wno-deprecated

LIBS =  -lGLU -lGL -lm -lglut -L/usr/X11R6/lib -L/usr/lib/X11 -lXi -lXmu

VOR=voronoi/edgelist.c voronoi/geometry.c voronoi/heap.c voronoi/memory.c
voronoi/output.c  fortune.cpp

OBJ=prey.o herd.o myheader.o fortune.o edgelist.o geometry.o
heap.o memory.o output.o root.o

SOBJ=prey.o herdstatic.o myheader.o fortune.o edgelist.o geometry.o
heap.o memory.o output.o

OPENGL=
#-DMYOPENGL


all: kinetic static

static: $(SOBJ)
$(CC) $(SOBJ) $(OPENGL) svoronoi.cpp -pg -o svoronoi $(LIBS)

kinetic: $(OBJ)
$(CC) $(OBJ) $(OPENGL) kvoronoi.cpp -pg -o kvoronoi $(LIBS)
```

```
prey.o: prey.cpp prey.hpp myheader.hpp
$(CC) -pg -c prey.cpp

myheader.o: myheader.cpp myheader.hpp
$(CC) -pg -c myheader.cpp

herd.o: herd.cpp herd.hpp prey.hpp myheader.hpp
$(CC) -pg $(OPENGL) -c herd.cpp

herdstatic.o: herdstatic.cpp herdstatic.hpp prey.hpp myheader.hpp
$(CC) -pg $(OPENGL) -c herdstatic.cpp

fortune.o: $(VOR) voronoi/vdefs.h fortune.hpp
$(CC) -pg -c $(VOR)

root.o: myheader.hpp root.hpp root.cpp
$(CC) -pg -c root.cpp
```

REFERENCES

Albers, G., Guibas, L., Mitchell, J., & Roos, T. (1998). Voronoi diagram of moving points. *International Journal of Computational Geometry and Applications, 8*, 365–380.

Aurenhammer, F. (1991). Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys, 23*(3), 345–405.

Berg, M. de, Kreveld, M. van, Overmars, M., & Schwarzkopf, O. (2000). *Computational geometry: Algorithms and applications.* Springer-Verlag.

Brook, S. (1999). *Fortune's 2d voronoi diagram code.* from http://www.cs.sunysb.edu/ algorith/implement/fortune/implement.shtml.

Devillers, O., & Golin, M. (1998). Dog bites postman: Point location in the moving Voronoi diagram and related problems. *International Journal of Computational Geometry and Applications, 8*(3), 321–342.

Fortune, S. (1987). A sweepline algorithm for voronoi diagrams. *Algorithmica 2, 2*, 153–174.

Foundation, F. S. (1998). *Gnu gprof - table of contents.* from http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

Guibas, L. J., & Stolfi, J. (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics, 4*(2), 74-123.

Hamilton, W. D. (1970). Geometry for the selfish herd. *Journal of theoretical Biology, 31*, 295–311.

Moreau, J.-P. (2002). *Roots of a real function in c/c++.* from http://perso.wanadoo.fr/jean-pierre.moreau/c_roots.html.

Mount, D. M. (2002). *Cmsc 754 computational geometry.* from http://www.cs.umd.edu/ mount/754/754lects.pdf.

O'Rourke, J. (1998). *Computational geometry in C.* Cambridge University Press.

Reynolds, C. W. (2001). *Boids (flocks, herds, and schools: a distributed behavioral model.* from http://www.red3d.com/cwr/boids/.

Viscido, S. (n.d.). *The paradox of the selfish herd: the search for a realistic movement rule.* from http://tbone.biol.sc.edu/ steven/paradox.html.

VITA

Yasuharu Kai was born in Kitakyushu, Fukuoka, Japan on February 13, 1978, the son of Toshio Kai and Marumi Kai. After completing his work at High School attached to Kyushu International University, Kyushu, Fukuoka, in 1996, he entered the University of Aizu. He received the degree of Bachelor of Science from the University of Aizu in March 2000. In September 2000, he entered the Graduate College of Texas State University-San Marcos.

Permanent Address: 3-3-6 Takasuhigashi Wakamatsu-ku
Kitakyushu, Fukuoka, Japan, 808-0144

This thesis was typed by Yasuharu Kai.