Ants Collecting Food

SOFT144: Assignment Report

10253311 - Brian Viviers

Contents

1 Introduction

Introduced within this report will be an explanation of the design choices and implementation of bringing project Ant Simulation into being. Each class along with their methods, only the ones of importance, will be explained in enough detail to understand the inside workings without the need to traverse through the project implementation. The inheritance hierarchy and associations will be discussed and three class diagrams are available (the first showing relationships, another showing only fields, methods and properties while the last shows everything).

The use of lists and why their use was of such importance is described. A critical evaluation will illuminate the areas that were felt to be a success and those that definitely needed a substantial amount of planning beforehand.

Towards the end of the report is a table with all the essential parameters used and also all the initial settings of the simulation upon starting for the first time. There is some simulation instructions included to assist the user in understanding applying settings.

1.1 The classes

There are 7 classes, not including *Program* and *Primitive2D* [3], used in this project. Class *AntSimulator* is the main game class and is composed of a class *Nest* twice. This is where it holds the nests, in lists, for the two types of ants. It is also composed of a class *Food* which is all the food placed by the user, held in a list. It is composed with each of the ant classes *BlackAnt* and *RedAnt* as well. These, too, are held in lists. Class *AntSimulator* also has one instance of class *Ant* but it will only ever contain either class *BlackAnt* or *RedAnt.* This polymorphism is used when one of the ant types has been marked to follow. Class *BlackAnt* and *RedAnt* both inherit from class *Ant*. Class *Ant* has a class *Nest* because it holds the known nest location of both ant types. Class *BlackAnt* has a class *Food* twice. The food it knows about and a potentially closer food source. The last class is *AntSimulationForm*. This has no relationships. Class *Program* is composed of class *AntSimulationForm* and *AntSimulator*.

1.2 Lists

The use of lists was of critical importance in the design of the ant simulation. The user can add as many nest and food objects to the simulation as they desire. To enable the user to do this, it needs to be possible to increase and decrease the amount of food and nests and this would be difficult without lists. Arrays mostly need to be given a fixed size and are difficult to resize once initialised. Lists provide the functionality to resize easily and during runtime.

Lists can easily have an item removed from within and it will dynamically shrink. They also allow for easy manipulation of a large amount of objects in minimal amount of code. One other benefit of lists, in this project, is that it allowed any amount of ants to be set during runtime. Without lists the amount of ants would probably need to be fixed at start and this would reduce the versatility of the simulation.

2 class Program

It was deemed important to firstly discuss this class as it has been modified from the original and this is where the game begins. An instance of the AntSimulationForm form is created and an initial window size set. To get the game to be drawn in a panel on the form required that the game object is initialised with a call to the forms `getDrawSurface()` method, as a parameter to the constructor of the game. This method returns the handle to the panel where the game window will be drawn. The code for passing the handle of the panel as a parameter was researched online [1].

3 class AntSimulationForm : Form

This form contains a panel that houses the game window. To the right of the panel is a variety of controls that enable the user to modify the simulation during runtime.

3.1 Returning the panel handle

A method, `getDrawSurface()`, returns the handle to the panel where the game window is drawn. This method is called in class `Program` when the game is created in order to give the game a handle to the place where to draw itself. This method, along with others, was researched online [1].

3.2 Adding and removing objects

The form has five radio buttons, to perform actions, and depending on which one is checked at the time of a user click a different action will be performed. They include adding food, adding a red or black ant nest, removing any of the added objects or placing a circle around an ant to follow it.

3.3 Changing food quantity

A text box provides the option to change the next placed food object quantity. When the apply button is pressed the text within the text box is parsed in a try-catch statement and if successful the integer is passed to the `AntSimulator` class which then will use that food quantity when it next adds a new food object. If unsuccessful then a message is shown requesting the user to enter an integer. The textbox colour will be green when the amount is successfully set. The food quantity cannot be less than 1.

3.4 Setting the ants count

The amount of black and red ants can also be changed by the user during runtime. Two text boxes provide the user with a place to enter the quantity of each ant type. Upon clicking the apply button the text within each text box is parsed in a try-catch statement and if successful the *LoadGame()* method, within the *AntSimulator* class, is called with the two integers as parameters. This restarts the simulation as if it had just been created but now with different amount of ants. LoadGame method will be discussed in more detail later. The ants count cannot be a negative value.

3.5 Changing simulation speed

This functionality is provided on the form by two buttons, + & -. The user can click and hold the buttons which starts one of two timers. The timers have an event handler for every tick which is every 200 milliseconds. *PlusButtonTimer()* and *MinusButtonTimer()* are the event handlers. The timer stops when the button is released. Doing this means the user does not have to repeatedly click to increase the game speed. The two methods, while being called, then themselves call *UpdateSpeed()* with -0.1 or 0.1 as a parameter. This method then sends this speed change to the main game class along with showing the new speed on the form. It also clamps the min and max speed between 0 and 8. The reason for clamping the speed is that the simulation does not function as intended past these limits.

3.6 Play and pause

The form also has the option to pause and play the simulation through a button provided. The way that the game does this will be explained further later.

3.7 Changing resolution

Three options exist for a resolution size. 800x600, 1024x768 and a maximum which is the users screen resolution. To enable the form window and game window size to be changed, the form and the game are declared public in class *Program* so that their size properties can be accessed during runtime. In the *AntSimulator* class is a method *ChangeResolutionSize()* which changes the viewport bounds. The *GraphicsDeviceManager* object has its width and height set to the new values. Changing the resolution restarts the simulation back to default settings except for the resolution size.

3.8 Information window

Finally, in the form, is an info button that provides the user with a message box window with information about the simulation.

4 class Food

4.1 Methods

The only method is the draw method. Drawing the food to the screen requires first getting the scale size of the food so the image is proportionate to how much food is left.

4.2 Properties

This class has 5 properties. The first decreases the food quantity every time an ant takes food. Another returns the current scale size of the image. The scale is a decimal value between 0 and 1. Another property returns the food quantity remaining. The last two return the bounding rectangle and the food centre position.

5 class Nest

5.1 Methods

`Draw` is the only method in this class. There is no scaling or movement involved.

5.2 Properties

There are two properties in this class. One that returns the nests centre position and another that returns the nests bounding rectangle.

6 class AntSimulator

6.1 Constructor - *`AntSimulator(IntPtr drawSurface)`*

This has a parameter which is a pointer to where the game will be drawn. A bool variable *`restartCount`* is initialised to false. This is used in *`LoadGame()`* to only call the *`AddImages()`* method when the user restarts the simulation because this method is called automatically on first start. The viewport size is set to an initial size of 800x600 but this can be changed by the user on the form. This size is saved in a variable *viewPortBounds* which is passed to the ants, when created, so that they are able to take action when going over the viewport edges. When the resolution is changed by the user this variable is also changed accordingly.

Further, in the constructor, is another 3 lines of code which have been researched online [1]. These are where the draw surface is set to the pointer passed in as a parameter. Two event handlers are also created. The first, *`graphics_PreparingDeviceSettings()`*, captures the construction of a draw surface and makes sure that it gets redirected to the panel on the form marked by the pointer parameter. The second event handler *`AntSimulator_VisibleChanged()`* just ensures that the original game window is made invisible and stays that way.

6.2 Initialise

The initialize method has had all of its implementation refactored into a new method called *LoadGame()*. This new method is called from within *Initialize*. The reason for this is that when the ants count is adjusted by the user on the form, then to enable all the initialised variables and lists to be re-initialised, a method needs to be called to do this as if it was just being restarted.

6.3 Load game - *LoadGame(int blackAnts =300, int redAnts = 150)*

The two parameters are defaulted to their values as shown above. This is so that on first load the simulation has values for the ants count.

The first two lines of code set both *graphics.SynchronizeWithVerticalRetrace* and *IsFixedTimeStep* to false. This is to increase visual performance of the simulation. If they are not set to false then the simulation is almost unusable.

A *GameState* variable currentGameState which is an *enum* type - *GameState {paused, playing}* is initialised to playing. This is available so that the user can pause the simulation at any point if they desire via a button click on the form. The button click will change the *currentGameState* variable to either one of the states.

Next there are 5 lists (*List*<>) initialised. Two lists are for the ants and contain *BlackAnt* and *RedAnt*. The next two lists contain the nests for each type of ant and both lists contain *Nest*. The 5th list is the list of all the *Food* objects added by the user.

All the ant's widths and heights are set to 10 pixels. This is a fixed size because no scaling needs to be done on the ant images.

Two loops then create and add ants to the list with initial positions randomly generated. One loop is for *BlackAnt* and the other for *RedAnt*.

Lastly is a check to see if this is the first load of the simulation or if it is being done by the user's request. If it is the first start then there is no need to add images to the ant objects as this is automatically done on first load. On a second load, however, the *LoadContent()* method is not called but the content has already been loaded so the images only need to be added to the new lists of ants. This is done with a call to *AddImages().*

6.4 Load Content

Here all the images for the simulation are loaded into variables. Two arrays each of size 3 are initialised. They contain the image of ant with food, without food and the ant's nest image. The only two other images are the food image and the background/floor image. There is a method call to *AddImages()* which, as described previously, will add the images loaded to each ant object in the ants list. The *AddImages()* method contains two loops that iterate through each of the ant lists (red and black) and adds the images of an ant with and without food to each object in the lists.

6.5 Playing and pausing - *PausePlay()*

This method is called from within the *AntSimulationForm* and checks the current state of the game. If the game is paused then it sets the state to playing and vice versa. Then in the update method there is a check to see what the current state is before executing any code.

6.6 Removing finished food piles - *RemoveUsedFood()*

All this method does is iterate through the food list and check the quantity of each food object. If the quantity is zero or below then the food object is removed from the list which means it will not be available to be used by ants or drawn to the screen.

6.7 Adding food to the simulation - *AddFood(MouseEventArgs e)*

Firstly a check is done to see if the user is adding food on top of an existing pile of food. If they are then that food pile will have its quantity increased by the amount of the new quantity. If there is no existing food pile then a new one is created. Creating a new food object is done by declaring a *Food* object and a *Vector2* object. The X and Y coordinates of the mouse click are passed in from the form. These coordinates get placed into the *Vector2* object and are used as the centre point of the food when drawn onto the window. The food quantity currently set on the form will be used to set how much food this food pile will have. The newly created food object is then added to the list so that can be drawn onto the screen and access by other methods later.

6.8 Adding nests to the simulation - *AddNest(MouseEventArgs e, string antType)*

When the user clicks on the game window with either of the add nest radio buttons checked then a method call to this method occurs. A *Nest* object and a *Vector2* object are declared. A check is done to see what type of ant nest the user wants to add. It is either black or red ants nest. The X and Y coordinates of the mouse click are passed into the vector object and this is used as the centre position of this new nest. The nest is then added to its respective nest list.

6.9 Removing objects - *RemoveObject(MouseEventArgs e)*

In order to allow the user to remove nests or food from the simulation, this method was created. A *Point* object is created using the X and Y coordinates of the mouse click. Then the *blackNestList*, *redNestList*, and *foodList* are all iterated through in turn. A check is done to see if any of the objects in any of the lists have that point inside its bounding rectangle. If it does then it is removed from the list.

6.10 Changing the simulation speed - *ChangeSpeed(float speed)*

Both of the ant lists, *blackAntList* and *redAntList* are iterated through and their max speed is increased or decreased by 0.1 depending on which value is passed in as a parameter. This is achieved by calling the *MaxAntSpeed* property of each ant.

It is known that the speed could be increased by increasing the frequency with which the update method is called. Doing it this way required the two variables that have been set to false in `LoadGame` to be set to true. This decreases performance of the simulation considerably. For this reason it was chosen to increase the ant's maximum speed instead.

6.11 Changing food quantity

The food quantity of each food pile can be changed during runtime. A call to this property, from *`AntSimulationForm`*, sets *`foodQuantity`* variable to the amount passed in so that when the next food object is created and placed it will contain this value as its quantity.

6.12 Drawing all the images

All objects drawn on the viewport have their own draw methods except for the background image. When calling their draw methods, a *`SpriteBatch`* object is passed in to enable all the images to be drawn with the same settings. Images are drawn in order of what should be on top of others. The background is first then the nests which are followed by the food and lastly the ants.

7 class Ant, BlackAnt & RedAnt - Animating the ants

These three classes will be discussed together as black and red ants inherit from class Ant and they are very similar. They will be discussed during the explanation of the Update method - *`Update(GameTime gameTime).`*

In this method are two loops, one for the black and one for the red ants. The two loops are almost exactly the same. They both have the same calls to methods with the same name but the methods that deal with food for each ant are different internally because each type of ant collects from different food sources. Only one loop will be explained but where the methods differ then both of the methods will be discussed.

Black ants store an actual *`Food`* object in their memory whereas red ants store a food location using a *`Vector2`* object. This location is where they last collected food from or they have been told about.

7.1 Searching for and collecting new food

First a check is done to see if the ant has food by checking to see if their food object is not null and that they currently do not have food with them. If both are true then they search for food using the *`SearchForFood()`* method.

- **Black ants**

For the black ants if the food list doesn't contain food then they wander. If it does then the list is iterated through to see if the food is within their radius. The radius for all ants is 20. If

there is one then they are steered towards it using *MoveToFood(food).* This method is a seek method with an added extra. It first calculates a position on the edge of the food to sit and eat. The *CalculatePlaceToEat()* method is used to do this. Then the ant is steered towards it. Once they are close enough, distance varying according to their speed, to the food then they sit and eat using *PickUpFood(food).* What this method does is turn the ant to face the food with a call to *FaceFood()* which only adjusts their orientation. Then an integer is incremented for every time step. This simulates the ant eating. Once the int has reached 60 then their *HaveFood* bool variable becomes true and the food quantity is decreased by one. This *HaveFood* variable ensures the ant with food image is now drawn.

- **Red ants**

The red ants are slightly different. They use the black ants as a food source. So the black ants list it is iterated through to see if any black ants with food are in their radius. If true then they are steered towards the black ant with food using *MoveToFood(ant.AntPosition)* but without calculating a spot to eat. Once close enough, at a distance of 5, then they steal food using *StealFood(ant).* This sets the black ant to not having any food and the red ant to having food. It also sets their known food location to this location.

7.2 Collecting from a known food source

If the ants know of a food location but do not have food then their *CollectKnownFood()* method is used.

- **Black Ants**

As described above, this method also calculates a spot to eat on the food then checks the distance. If too far then the ant is moved closer. Once close enough then the food source is checked to see if it still exists because in the time they are away it could have been deleted by the user.

CheckFoodStillExists() is used to check if food is still there. It iterates through the food list to see if the known food source is equal to any in the food list.

If the food is still available then they pick up food as described above. Lastly a method call to *CheckFoodQuantity()* is done. This checks the remaining quantity of the food to see if any remains. If this isn't done then the ants get stuck at the old food source.

- **Red ants**

These ants will check to see the distance to the known food location and move closer using *ReturnToFood()* method. This is a seek method which seeks the known food location. Once they are in distance they forget that location which makes them search once more. So if the last ant they stole from was in a chain then there should be more to steal from again.

7.3 Searching for closer food

While collecting food both ants will still keep looking for other food locations that are closer to the nest. They both use a method called *`SearchForMoreFood()`* but which have different implementations in their own classes.

- **Black ants**

This iterates through the food list and checks if another food source is in their radius. If there is then a method call to *`CheckIfFoodCloser()`* happens. This method checks the distance from that food source to the nest. This distance is compared to their current known food source to the nest distance. If it is shorter, then the new food source becomes the one they collect from.

- **Red ants**

This is almost identical to the black ants except the distance is calculated to a black ant's position that has food.

7.4 Returning food to the nest

- **Black ants and Red ants**

`ReturnFood(List<Nest> nestList).` This method is in the *`abstract class Ant`* because both types of ant use the same method. A call to *`DepositFood(NestKnown)`* is first done. This is a seek method and steers the ants to the nest. Once the nest is within their radius then the nest is first checked to make sure the user has not removed it. This is done with a method *`CheckNestStillExists().`* It iterates through the nest list and compares every nest to the known nest. If none are the same then it has been deleted.

After this, if the nest is still there, the distance is checked and if the ant is within 3 pixels from the centre then their *`HaveFood`* variable becomes false.

7.5 Wandering

- **Black ants and Red ants**

The method *`Wander()`* is called from many different places all depending on what the ant is doing at the time. To prevent it from being called more than once in a time step another method called *`WanderLimiter()`* is used. This has a reference variable passed in from the main game class. Once this variable is set to true then Wander cannot be called again in this time step.

7.6 Searching for a nest

- **Black ants and Red ants**

SearchForNest(). This method is in the *abstract class Ant* because both ant types use it. First a call to *ForgetNestLocation()* occurs. This checks if the timers, for how long it has been since they last saw or where told a nest location, are past 20 seconds. If they are then the ant will forget the nest.

Then a loop iterates through the ant specific nest list. If a nest is in its radius then one of the following is done. If the ant currently knows of a nest location then a check is done to see if the new nest is closer to the food. This is done the same way as checking if food is closer. If no known nest location is known then this becomes their nest. Whenever they learn about a new nest, one of the two forget timers are started in the *InitForgetTimer()* method. One is for when they last saw a nest and the other for when they were last told about a nest.

7.7 Asking for food and nest locations

To increase the amount of ants the simulation can have, only a third of the ants in the list will ask for food or nests in one time step. A method *CalculateAskingLoop()* works out which ants should ask. Ants will always be asking for food and nest locations irrespective of what they are doing.

- **Food locations**

Both ants have a method called *AskFoodLocation()* but their implementation is different because they use different food sources.

  - **Black ants**

If their radius has another ant in it and the other ant knows of a food location then one of the two following things happen. If this ant doesn't know of any food then this ant arrives to the other ant by using *ArriveAndAsk().* When they are a distance of 10 away then they get told about the food source in a *Food* object. If this ant does know of a food source then it is checked to see if the new food is closer as described earlier.

  - **Red ants**

Red ants do the same as black except they ask for a food location in a *Vector2* rather than a *Food* object.

- **Nest locations**

  - **Black ants and Red ants**

Both ant types use the same method *AskNestLocation()* in the *abstract class Ant*. They use their own nest list though. This method is almost the same as asking for food except a *Nest* object is passed if they get told about it. Also the last told forget timer is started.

8 Evaluation

Overall the feelings are that the project was a success. The ants are able to easily find the shortest path between food and nests. This means they form realistic chains while collecting and returning food. Having the game window inside a form made the simulation a lot more customizable. This provided the user with the ability to adjust the amount of food and ants which meant that the user can experiment. With the option to change resolution means the ants can be restricted to less space which in effect is like having more ants available on a larger resolution. Being able to change the speed is a nice feature too, because the patterns formed, when sped up, are quite pleasing to watch.

A problem that has been noticed is that when ants are supposed to sit around food objects and collect food, sometimes they vibrate on the spot. The reason for this is still unknown but it wasn't a big enough issue to spend more time on.

With a simulation such as this, having so many ants means loops need to run for a long time. This decreases the performance of the simulation. This problem was encountered and a solution was found by dividing the lists into sections and only performing actions on one section at a time. This solution is not the best idea and more effort would be put into working out a better solution.

With regards to doing things differently it would have been a lot more beneficial to have first planned every class before implementing them. Having designed only one ant class and then requiring another meant that a lot of time was spent moving code around and fixing methods so that two new subclasses could inherit from the abstract Ant class. Trying to make a class able to inherit is a lot more difficult that designing inheritance from the beginning.

It could have possibly been possible to make a class from which the food and nest classes inherited. By doing this the two classes would each have had less implementation individually and code could have been shared by inheriting from a parent class. This was only considered towards the end of the project and thus was not attempted.

This project was the first attempt at anything object orientated and this approached was fundamental to getting the project working. It would be very inefficient to have to code the same code for 500 ants which is an approach that probably would have been taken beforehand.

9 Parameters

Ants radius	20
Initial max ant speed	4
Ants slow radius	20
Ants asking radius	30
Time for ants to forget a nest	20 seconds
Maximum rotation	1 radian per time step
Black ants collecting food time	3 seconds
Black ants distance to stop from food	Calculated according to food size. Max = 25
Red ant stealing distance	5 - from centre of black ant
Ants width and height	10x10
Food width and height	40x40
Nest width and height	80x80

Simulation Initial Settings and Limits

Food quantity – initial	200 – 1 to unlimited
Black Ants – initial	500 – 0 to unlimited
Red Ants – initial	200 – 0 to unlimited
Speed – initial	4.0 – 0.0 to 8.0 – incremented by 0.1
Resolution – initial	1024x768 – 800x600 and maximum

10 Simulation Instructions

When entering value into textboxes, if the back colour turns red then this indicates that the new value has not been applied. Once a confirm button has been clicked then the back colour turns green which indicates that the quantity is ready for use. This also applies to the resolution setting.

- **Setting resolution** – This should be done first. When the resolution is changed all settings are reset to default. Select the required resolution and click apply.
- **Changing ants count** – Doing this will restart the simulation and should be done after changing resolution. It will restore all initial settings to the default except for the required amount of ants and the resolution. Enter the required amount of ants for both and click the apply button.
- **Placing food and nests** – Ensure the required object is selected and then click anywhere within the game window.
- **Mark an ant to follow it** – Check the 'Follow Ant' button and then click on an ant. To remove the mark, click on an open space.
- **Changing food quantity** – Enter an amount in the text box provided and click apply. The next food object placed will contain the entered amount.
- **Changing speed** – Clicking once on the + or – button will increase or decrease the speed by 0.1. Clicking and holding a button will increase or decrease the speed by 0.1 every 200 milliseconds.
- **Play/Pause** – Click once and the simulation will be frozen as is. Click once more and the simulation continues.

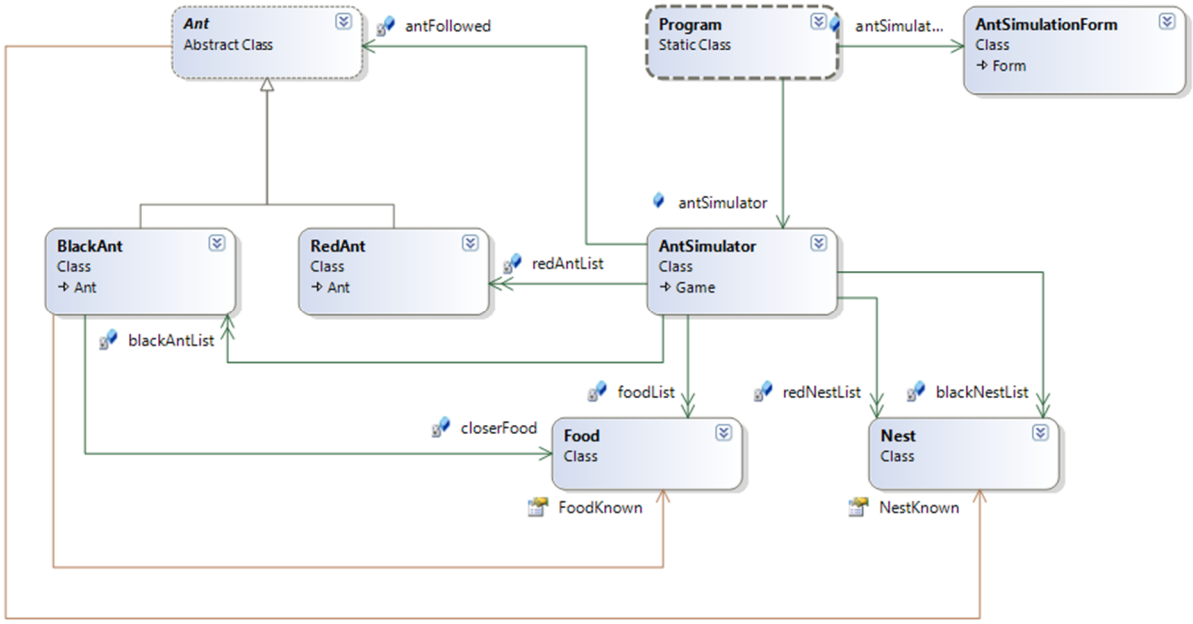
11 References

1. Triesscheijn, R. (2008) *XNA 3.0 and Winforms, the easy way*, Available at: http://royalexander.wordpress.com/2008/10/09/xna-30-and-winforms-the-easy-way/ [Accessed: 09 April 2014]

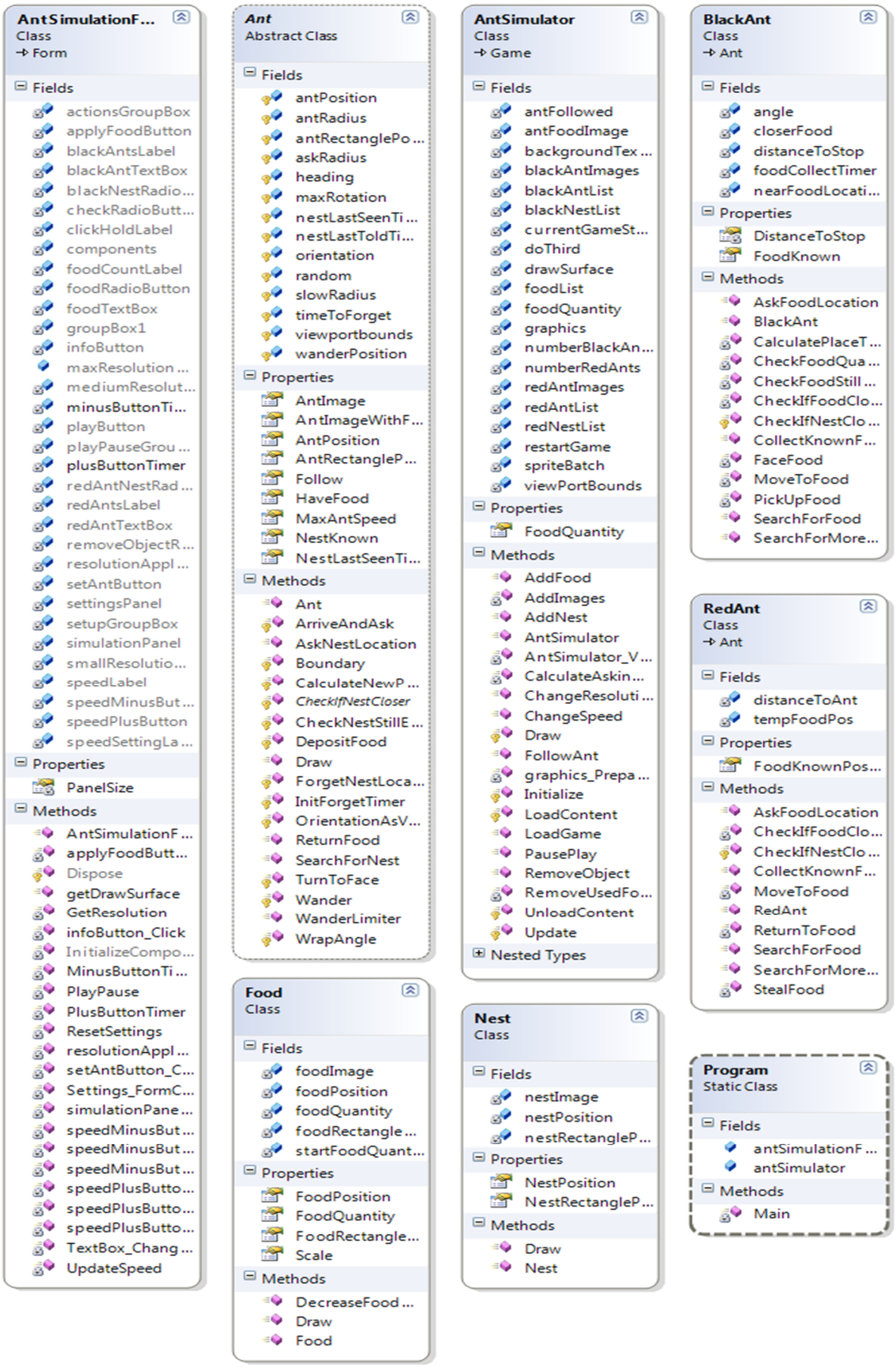
2. Game Icon: Available at: http://www.clipartbest.com/clipart-nTBG6KnTA/ [Accessed: 12 April 2014]

3. Class Primitive2D: Available at: https://bitbucket.org/C3/2d-xna-primitives/wiki/Home

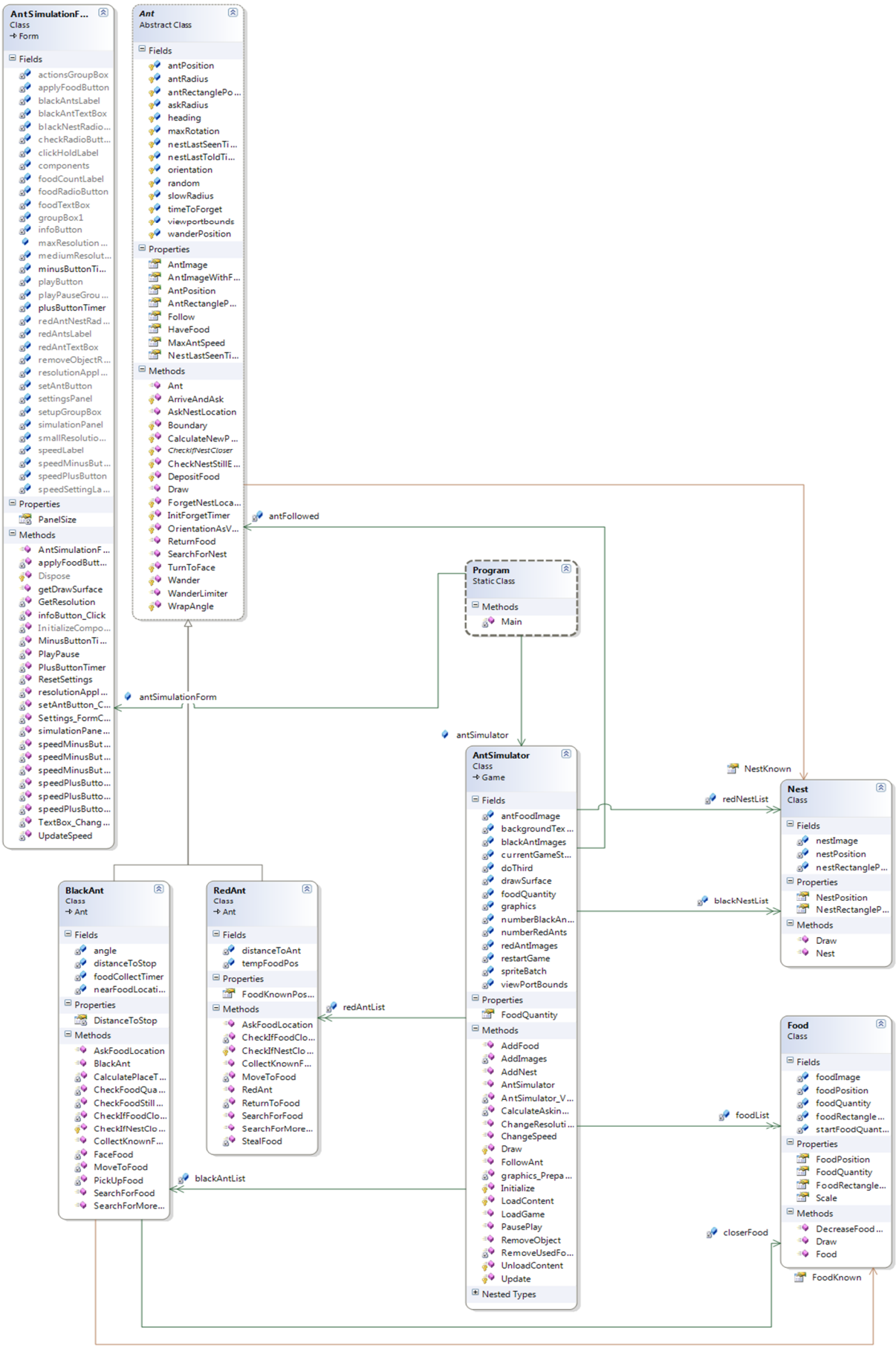
12 Class Diagrams



Class diagram only showing relationships.



Class diagram showing fields, methods and properties but no relationships.



Full class diagram showing all fields, properties, methods and associations.