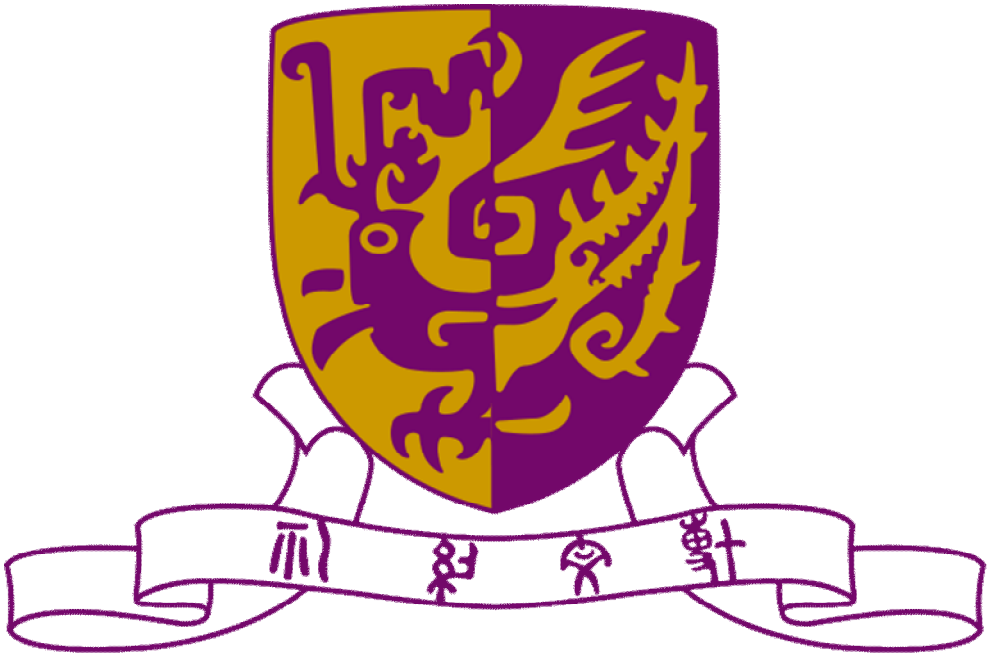


The Chinese University of Hong Kong

Department of Computer Science and Engineering

Final Year Project 2008-2009

1st Term Report

IPhone Application III

Group: IK0804

Supervisor: Prof. KING Kuo Chin, Irwin

Member: Ng Hon Pan (s06679724)

Wong Hung Ki (s06641853)

Index:

1. Introduction:

1.1 Overview:

Owing to the rapid growth of the microprocessor technology, the microprocessor can be made in smaller size with lower cost, and therefore it brings many Smartphone products to light. Apple, as a big company in Computer Industry, published iPhone which consists of many awesome features and they attract people's attention. iPhone not only opens up the Smartphone market, but also creates a new climate of developing iPhone applications.

In the project, we are going to develop a mini game called Monster Manager. In general, there are different kinds of monsters (tiny squares with different colors) in the game and they group around and form a large square. Player shall move the monsters in some particular ways so that monsters with the same type (same color) are grouped. This game is very challenging as players must recognize all possible patterns as fast as possible so as to reach higher level. The game provides many levels so that players would not feel bored very quickly, and this mini game can properly attract female players and is worth to spend in leisure time.

1.2 Motivation:

The idea of this game came from another game that we played several years ago. It is called Zoo Keeper. It is a puzzle game with many nice features, such as simple game rules, pretty cartoons, challenging, etc. This game attracted many females and people who spent lots of hours working in the offices per day, and it caused a huge mania at that time. Its huge success motivates us to development Monster Manager in iPhone platform.

Beside the above reason, there is several reasons cause us to make this game. First of all, based on the game idea of Zoo Keeper, we can think of any ways to implement our game. For example, it is up to us to design how players can move the monsters. Therefore, there are many chances for us to show our creativity. Secondly, base on the pros and cons of Zoo Keeper, we want to develop a comparable or even better game than Zoo Keeper. Thirdly, as iPhone is so popular nowadays, and Apple has released iPhone SDK, it is a good chance for us to gain some experience in developing mini game program on mobile device, and publish our game to the world to share our game ideas.

1.3 Game Review:

When we were designing our game, we reviewed several games that make use of the game idea of Zoo Keeper. Through understanding the ways how they are implemented, we managed to design the one that is as good as them with a partially or completely new design. We would like to keep their strengths, and reject their weaknesses. The reviewed games are Zoo Keeper, Trism, Diamond Twister and Jewel Quest II.

1.3.1 Zoo Keeper:

In Zoo Keeper, players need to achieve the required number of matches for each type of animal in each level, and the numbers are shown on the upper right corner. A match is made by grouping at least three animals of the same type horizontally or vertically. Players can only swap two adjacent animals. If there are no matches, the swapped animals will move back to original positions. If there is any match, the matched animals will disappear. The upper animals will then fall down and fill up the empty spaces. There is a time limit in each level, and the level goes up once the level requirement is completed.

1.3.2 Trism:

Trism is a puzzle game available on iPhone. The developers still bear the same idea from Zoo keeper, but they convert the square animals to bright and colorful triangles. All the triangles are put inside a frame. Players have to shift the whole line (formed by triangles) so as to group at least three same color triangles together. Although the front triangle and the back triangles of a line are not visually linked, they are actually linked during shifting. When the player rotates iPhone, a black arrow appears which indicates the direction of gravity. All the triangles inside the frame are under the force of this gravity.

Also, there are lots of puzzle levels for players to solve. In each level, some triangles are put inside an arbitrary shape frame, such as triangle shape, heart shape. Player needs to make use of the accelerometers of iPhone to move the triangles so that the triangles of the same color are grouped.

1.3.3 Diamond Twister:

Diamond Twister is very similar to Zoo Keeper, and is available in iPhone. The features that make it different from Zoo Keeper are the storyline, support of accelerometer, the dazzling visual effects and gem explosions. Rotating iPhone can change the direction of the gravity, and thus change the falling direction of the gems. This gives players more control on how to move the gems.

1.3.4 Jewel Quest II:

Jewel Quest II is another similar game to Zoo Keeper. In each level, players have to swap the gems in the same way as it is in the Zoo Keeper. However, once there is a match, the matched square become golden. A level is complete only if all the squares are golden.

2. Game Design Analysis:

2.1 General Game Idea:

All the above games work in different styles, but they in fact share the same idea. For our game design, we want our game to be built by a frame, where there are lots of monsters inside the frame. All the monsters have the same shape, and are packed nicely within the frame. The main target of players is to move the monsters so that several monsters of the same type (same color) are grouped together. The grouped monsters are removed, and new monsters come to fill in the empty space. This is just a general idea that every game discussed before has. For the implementation detail, we should ask ourselves the following important questions before make the final decision.

1. What is the shape of the monster?
2. How do the monsters move?
3. What is the definition of a match?
4. How to set the timer?
5. Is it a must to have at least one match per move?
6. How to fill up the empty spaces after a match?
7. How to set up different levels?

2.2 Shape of Monster:

The shape of the monster can be a square, a circle, a triangle, etc. Difficult shape can affect the way of our implementation. For example, we need to think of a nice way or data structure to store the positions of all the objects. Also, we must pay attention to the movement of the monsters, and avoid them to overlap with others. Moreover, we also want a data structure that let us retrieve and set a particular monster object, or change the position of the monsters efficiently. Comparing triangle with square and circle, it is obvious that triangle shape require more deep consideration so as to meet the above requirements. On the other hand, square shape or circle shape is easier to implement as they are uniform in shape, although they may be less attractive than triangle shape or other non-uniform shape.

2.3 Move Monsters:

It is a huge topic on discussing how player can move the monsters. In order to find the best one, we managed to analyze the pros and cons of the existing methods that are used by the games talked above, and we have also proposed some new ways too. The following would talk about all the possible methods that we thought. Here, we talk all of them so as to show how we come up with the final decision.

2.3.1 Swap:

The first one that we want to discuss is Swap, and it is adopted in Zoo Keeper, Diamond Twister, and Jewel Quest II. The definition of Swap is the following. Players are required to select any monster inside the frame first, and then select a second monster which is next to the first monster (up, bottom, left, right). Afterwards, the first monster moves to the position of the second monster, and the second monster move to the position of the first monster.

Pros: This method is very simple. The program code to handle swapping is easy to write as we only need to swap the positions of the two adjacent monsters. Moreover, detecting matches after swapping can perform extremely fast as we can start checking at the positions of the swapped monsters. That means we first choose one of the swapped monsters, and then check whether its neighbors have the same type as itself. Then, do the same thing for another swapped monster.

Cons: This set of game rule is already implemented by quite a lot of game, such as Zoo Keeper, Diamond-Twister, Jewel Quest II, etc. Players are restricted to swapping only, and it seems to be too straight forward. In Chinese Chess, any single move can subsequently affect the next move, or the move following the next move, and it is the reason why Chinese Chess is so fun. However, the Swap method does not show this feature significantly.

2.3.2 Pick and Push 1:

Pick and Push 1, players first select a monster only from the edge, which means only the monsters that are not surrounded by four monsters can be selected. This action is called “pick”. Afterwards, an empty space is left. Players then need to “push” the picked monster back to the frame, starting from the edge again. For Example, sixteen monsters group around to form a 4×4 square. Since only the

monsters at edges can be picked, the four monsters at the center are fixed. The following is a simple graph showing the map of the sixteen monsters, where only the monsters marked with symbol (o) can be picked up, while the monsters marked as (x) cannot be picked. (+) is the frame.

+	+	+	+	+	+
+	o	o	o	o	+
+	o	x	x	o	+
+	o	x	x	o	+
+	o	o	o	o	+
+	+	+	+	+	+

If a monster (m) at the edge is picked like the following…

+	+	+	+	+	+
+	o	o	o	o	+
+	m	x	x	o	+
+	o	x	x	o	+
+	o	o	o	o	+
+	+	+	+	+	+

The (s) are the positions where the monster can be pushed back…

+	s	+	+	+	+
+	o	o	o	o	+
+	m	x	x	o	s
+	o	x	x	o	+
+	o	o	o	o	+
+	s	+	+	+	+

After the push, the monsters on the same line will be shifted together.

Pros: As far as we know, no one has implemented this. This method would create more variations compared to the Swap method. The way of making a matches in the Swap method is direct, while it is indirect in this Pick and Push 1 method; Players need to think a little bit to find out how to shift the monsters in order to obtain match. This method also gives players more thinking space how to move so that there will be another possible matches waiting for players.

Cons: This method requires shifting the entire line and all the monsters on the line have to change their positions according to the shifting direction, and the computation time of checking matches is much larger than the Swap method. The biggest defeat of this method is that player cannot find a way to group the following monsters (e) together. Monsters (o) can be any type.

+	+	+	+	+	+	+
+	o	o	o	o	o	+
+	o	e	o	e	e	+
+	o	o	o	o	o	+
+	o	o	o	o	o	+
+	+	+	+	+	+	+

But, the Swap method allows players to match those monsters.

2.3.3 Pick and Push 2:

Pick and Push 2, This set of game rule is the same as Pick and Push 1 rule except one thing. Any monster can be picked inside the frame. Let's see the above example where the monster (e) cannot be linked.

+	+	+	+	+	+	+
+	o	o	o	o	o	+
+	o	e	o	e	e	+
+	o	o	o	o	o	+
+	o	o	o	o	o	+
+	+	+	+	+	+	+

The monster at ($) is picked, and is pushed at (#).

+	+	+	+	+	+	+
+	o	o	o	o	o	+
+	o	e	$	e	e	#
+	o	o	o	o	o	+
+	o	o	o	o	o	+
+	+	+	+	+	+	+

Pros: It solves the problem shown in Pick and Push 1.

Cons: It seems that there is a loophole in this rule. Let's consider the following example. Monsters (a, b, c and #) are four different types of monsters. Monster (o) can be any type, except the types of monsters a, b, c and #.

+	+	+	@	+	+	+	+
+	o	o	o	#	o	o	+
+	o	o	$	o	o	o	+
+	o	$	c	$	o	o	+
+	o	o	a	o	o	o	+
+	o	o	#	o	o	o	+
+	o	o	b	o	#	o	+
+	+	+	+	+	+	+	+

Consider the monsters ($), although monster a, b or c are also fine, and push it at position (@), we can link them up by picking a monster (#).

+	+	+	@	+	+	+	+
+	o	o	#	#	o	o	+
+	o	o	o	o	o	o	+
+	o	$	$	$	o	o	+
+	o	o	c	o	o	o	+
+	o	o	a	o	o	o	+
+	o	o	b	o	#	o	+
+	+	+	+	+	+	+	+

If the empty spaces of the monsters ($) are filled by replacing them with new monsters (o), the map becomes the following.

+	+	+	+	+	+	+	+
+	o	o	#	#	o	o	+
+	o	o	o	o	o	o	+
+	o	o	o	o	o	o	+
+	o	o	c	o	o	o	+
+	o	o	a	o	o	o	+
+	o	o	b	o	#	o	+
+	+	+	+	+	+	+	+

It is clear that players can obtain another match next. This is the trick that players can apply in the game, and it significantly reduces the interest of the game.

2.3.4 Pick and Push 3:

Pick and Push 3, in this method, players need to select one output arrow, and one input arrow, where the arrows are located around all the monsters. The monsters will shift according to the input and output arrows. For example, let (i) stands for input arrow, and (o) stands for outputs arrow.

+	+	+	+	+	+	+	+
+	n	n	n	n	n	n	+
o	x	n	n	n	y	z	i
+	n	n	n	n	n	n	+
+	+	+	+	+	+	+	+

The whole line will be shifted to the left. A randomly assigned monster will come behind monster (z).

Pros: Player will no longer be able to play tricks any more.

Cons: It seems that shifting an entire line is very straight forward.

2.3.5 Pick and Push 4:

Pick and Push 4, this method is extended from the Pick and Push 3. Now, players are able to shift monsters not only in one straight line, but also in two straight lines.

+	+	+	+	+	+	+	+
o	s	s	s	s	n	n	+
+	n	n	n	x	n	n	+
+	n	n	n	x	n	n	+
+	+	+	+	i	+	+	+

Monsters (s) will be shifted to the left first, and the monsters (x) will then shifted to the north in order to fill up the empty space.

Pros: This method does not look straight forward, and there are lots of variations in real time.

Cons: In our opinion, this method enforces players to think of the consequence of all different positions of input and output arrows. Player thus cannot quickly determine the next move, and the situation is similar as Chinese Chess. Player may feel hard to play, and the game becomes not exciting as the thinking time is long. For example, consider the following case.

In order to match monsters (x), we can have totally nine possible moves.

+	+	+	+	+	+	+	+
+	n	x	n	n	n	n	+
+	n	n	x	n	n	n	+
+	n	x	n	n	n	n	+
+	+	+	+	+	+	+	+

Under nine possible moves, players can easily get confused, and they then try hard to think of the best move. The consequence is that the game becomes hard to play, and not exciting anymore.

In fact, we have other set of game rules related to the Pick and Push approach, but they are no better than the one describing below.

2.3.6 Rotation:

Rotation, players first select a monster in the frame as the center of rotation. Then, players should choose the direction of rotation, either clockwise or anti-clockwise. Then, the monsters surrounding the monster at the selected center are shifted according to the direction players desired. Let's see an example. Monster (5) is selected as the center, and the surrounding monsters (1, 2, 3, 4, 6, 7, 8 and 9) rotate in clockwise direction.

+	+	+	+	+	+	+	+
+	o	1	2	3	o	o	+
+	o	4	5	6	o	o	+
+	o	7	8	9	o	o	+
+	o	o	o	o	o	o	+
+	+	+	+	+	+	+	+

After rotation…

+	+	+	+	+	+	+	+
+	o	4	1	2	o	o	+
+	o	7	5	3	o	o	+
+	o	8	9	6	o	o	+
+	o	o	o	o	o	o	+
+	+	+	+	+	+	+	+

It is possible to select a center at the edge, as there are monsters hide behind the frame. In another word, the (+) symbols are in fact be considered as monsters, but they are not visible to players, and they cannot be selected as the center of rotation.

Pros: It game rule does not have the cons discussed above.

Cons: It is not possible to link up the monster with the same type if they are located like the following map.

+	+	+	+	+	+	+	+
+	o	x	o	o	o	o	+
+	x	o	o	o	o	o	+
+	x	o	o	o	o	o	+
+	o	o	o	o	o	o	+

+	+	+	+	+	+	+	+

However, since the above case does not happen frequently, therefore, this drawback does not have significant impact.

2.4 Definition of Match:

After designing the way of movement of monsters, we need to decide the definition of a match. A match can be the followings.

1. At least three monsters of the same type grouped horizontally or vertically
2. A square with four monsters with the same type.

There are still many other definitions of a match, but the above two should be the most reasonable. By comparing these two, the first seems to be the better one. It is not only because the second one requires four monsters, while the first one requires only three, but also the first one requires a one dimensional line, while the second one requires a two dimensional square. Let's consider the following case, and we use the Swap method and the second definition of a match.

+	+	+	+	+	+	+	+
+	o	o	o	o	o	o	+
+	o	o	x	x	o	o	+
+	o	o	x	o	#	o	+
+	o	o	o	#	o	o	+
+	+	+	+	+	+	+	+

The monsters (x) can match together if either one of the monster (#) is (x), and there are only two choices.
Let's consider another case, and we use the Swap method and the first definition of a match.

+	+	+	+	+	+	+	+
+	o	o	o	o	o	o	+
+	o	o	o	#	o	o	+
+	o	x	x	o	#	o	+
+	o	o	o	#	o	o	+
+	+	+	+	+	+	+	+

The monsters (x) can match together if either one of the monster (#) is (x), and there are three choices. In conclusion, the second definition is much harder to satisfy.

2.5 Timer:

About the timer, we immediately think of a timer gauge. Sooner or later we think of another interesting way to represent time. There is no global time gauge. On the other hand, each monster would have a local time gauge. When a monster arrives inside the frame, a random value is assign to that monster. This value is just the time that the monster can live. The timer of a monster stops once the monster match with others. The game is over if there is the time gauge of any monster inside the frame drops to zero. The time gauge of each monster will increase its value once players succeed finding out a match. In order to indicate the timer of each monster, we design to create a set of monster images. When the timer of a monster reaches a certain value, the image of the monster would change accordingly. This method is good as the old fashioned timer gauge can be removed. The drawback is that the game program must keep check of all the timer of the monsters in a new thread, and lots of monsters need to change its image if the total number of monster is huge.

2.6 One Move One Match:

Is it a must to have at least one match per move? This question may need some further explanation before answering it. After receiving player's input, the game program needs to check whether there is at least one match, and there are two choices if no match happens. The first one is to move the moved monsters back to their original positions. The second one is just do nothing and start receiving next player's input. If we adopt the first one, the game program guarantees player must be able to find a match in one move; otherwise, player will never be able to reach high level. If we adopt the second one, the game program does not need to guarantee this. However, the second one would properly change the style of how to play our game as players can move the monsters that they want without any restrictions. In order to prevent this to happen, we can introduce a new variable called life. The purpose of the life variable is to restrict player to move monsters without thinking. If no match happens after a move, life will decrease by one. If life reaches zero, the game will be over. The life variable also induces a good side effect which player is allowed to have some freedom to find a match in more than one move.

2.7 Filling up Empty Space:

When a match happens, the matched monsters must be removed, and there are no other choices. However, we have some choices on how to remove them. In fact, this is also the time when the program but not the player can change the positions of the monsters in the frame. This is important as if the game program does not change some of the monsters′ positions periodically, player will soon get familiar with the arrangement of the monsters, and also the probability of successfully finding a match will properly decrease when more and more matches are found. In fact, all of the reviewed game, such as Zoo Keeper, Trism, etc, use the way described below.

Once there is a match, the monsters involved must be removed. The empty spaces left are filled by the monsters above the empty spaces. It is just the same as the situation always happens in real life. Something will drop if there are nothings below it. New monsters arrive so as to compensate the loss of monsters which are removed after a match.

It is really interesting to see the monsters falling under the force of gravity. Also, by using the accelerometer installed in iPhone, players can change the direction of gravity. Trism, Diamond Twister and Jewel Quest II have made this idea to reality. However, to be honest, we both think that this idea have two big problems.
The implementation is quite difficult. We not only need to care the falling distance of the monster, but also the falling speed, or even acceleration. Therefore, a particular algorithm is necessary to first calculate the falling distance, speed and acceleration. Then, it should change the moved monsters′ positions in the data structure and detect any possible match.

Controlling the direction of gravity sounds interesting, but it is not practical. As the game is like Zoo Keeper, the main objective of the player is to find matches in the shortest time. If we allow player to change gravity direction, they may need to spend some time and think of it and the game flow must be slower. Notice that there is a timer in the game which aims to compel player to find matches as fast as possible and make the game more exciting. It seems that the control of direction of gravity contradicts the game philosophy.

Instead of the above idea, we can simply replace the matched monster with new monsters. In the implementation aspect, this method is extremely easy. We only need to change the type of matched monsters with a new one, and then change the image representing the new type of monsters. However, we should make sure that the new types do not induce a new match immediately. Now, the question is, how can we

ensure this. Let's consider the following case. Frame is (+), Monsters (o) are ignored and Monsters (1, 2, 3 and 4) are different types of monsters. Symbols (#) are the places where new monster should be located.

+	+	+	+	+	+	+
+	o	o	2	o	o	+
+	o	o	2	o	o	+
+	1	1	#	4	4	+
+	o	o	3	o	o	+
+	o	o	3	o	o	+
+	+	+	+	+	+	+

In order to avoid inducing new match once after replacement, the new monster cannot be monsters (1, 2, 3 and 4). Therefore, we must ensure that there are at least five types of monsters available in game.

2.8 Level Settings:

Usually a puzzle game comprises of many levels, and each level must be set appropriate so that the majority of players feel the game is challenging but not extremely hard to play. According to the reviewed games, they all together propose three set of rules level settings.

In Zoo Keeper, players need to achieve the required number of matches for each type of animal in each level, and the level goes up once the level requirement is completed. For example, in level 1, player is asked to match 6 elephants, 9 lions and 3 crocodiles. Then, player must achieve this requirement before the time is running out in order to go to the next level.

In Trism and Diamond Twister, player must get the required points in order to go to the next level. There is a set of rule to count the number of point obtain for each successful match.

In Jewel Quest II, once there is a match, the matched square become golden. A level is completed only if all the squares are golden.

2.9 The particulars of Monster Manager:

After the detail discussion of the design, we finally concluded the particulars of our game as follow. Monster Manager is a puzzle game available on iPhone. In the game, 117 monsters group together and form a large (13x9) rectangle. The outermost monsters are not visible. There are eight different colors of monster, and they represent eight different types. The main objective is to group at least three monsters of the same type horizontally or vertically and gain enough score within a limited amount of time. In order to move the monsters, player can either use single tap or double tap. If player do single tap on a monster, the monsters surrounding it are shifted so that they perform clockwise rotation around the tapped monster. If player do double tap to a monster, the same things happen except anti-clockwise rotation is performed. After rotation, if no match happens, the life will decrease by one. The game is over once the life becomes zero. On the other hand, if at least one match occurs, the matched monsters will be replaced by new monsters. Then, the game will randomly pick one monster as the center and perform a bonus rotation which is either clockwise or anti-clockwise. The same process continues if there is at least one match afterwards. Each matched monster can score 100 points times the number of successive matches. In each out of ten different levels, player must obtain enough score specified by that level. Higher level would require more point within less amount of time.

2.10 Compare Zoo Keeper with Monster Manager:

2.10.1 Similarities:

1. Both require players to move monsters and group them up, but they present the idea in two different ways.
2. Both have timer and score.
3. A match may trigger other matches. This can keep the whole map changing periodically.

2.10.2 Differences:

1. In Zoo Keeper, player moves monsters by simply swapping two monsters which are next to each other.
   In Monster Manager, player moves monsters by selecting a monster as a center, and all the adjacent monsters will shift so that they seem to rotate around the center monster.
   (Swapping only involve two monsters, while rotation involve eight monsters, and so it is more challengeable.)
2. In Zoo Keeper, Once three or more monsters join together and form a straight line, they disappear, and then the upper monster fall out and fill up the empty space.
   In Monster Manager, after they disappear, the empty space is filled by simply put new monsters to those locations.
   (Falling is a good idea. However, this idea has been repeatedly used for many times, and the implementation is not a piece of cake. Therefore, we apply the simplest method, which is replacement method.)
3. In Zoo Keeper, if there are no matches after swapping, the moved monsters will move back to their original positions. In Monster Manager, the rotation cannot be undone. In order to prohibit players from tapping without thinking, a counter called "life" is introduced. The life value is initially set to three. If there are no matches after rotation, the life value will be decreased by one. The game is over once the life value becomes zero.
   (There are several advantages of using life counter. First, the game no longer need to keep checking whether there is always at least one match that can be made within one rotation in the game. It saves computation power. Second,

it allows players to get scores by two rotations during extreme condition.)

3. Class Description:

3.1 Overview

Our game program comprises the following classes.

1. Board – The object of this class is the container of all the monsters. That means it contains a two dimension array to store all monsters. It also has methods to set and get particular monster, create all monster before the game start, initialize the positions of monsters and update the timer, life and score.
2. ImageLoader – The object of this class helps loading all the monsters' images to memory, and allows other objects to retrieve them.
3. Monster – This class inherits the class Square which represents any square object in the game. An object of this class represents a monster. It has a type attribute, and methods to change the image of monster.
4. Square – This class inherits the class UIImageView which is a class in iPhone SDK representing an image. The object of this class stores its location in the game and the object itself is stored in a two dimension array which is located inside the Board object.
5. MonsterManagerAppDelegate – This is the class that must exist. In general, the object of this class creates all the necessary objects in order to run the application.
6. MonsterManagerViewController – The object of this class is created by MonsterManagerAppDelegate object. It is responsible to add the elements of the game, such as all monsters, background, labels to the window object.
7. TheGame – The object of this class creates all object used in the game, such as Board and ImageLoader. It is also responsible to handle the game logic and animations.

3.2 UML Class Diagram: (Only the important information is stated here.)

3.2.1 TheGame:

TheGame
- imageLoader : ImageLoader - board : Board - score : int - life : int - timeLeft: int - hits : int
+ gameInit () : id + rotationAtCenter (monster : Monster , direction : int) : void - rotationAnimationDidStop (animationID : NSString , finished : NSNumber , context : void) : void - shakeMonsters (matchedMonsters : NSMutableArray , counter : int) : void - shakeAnimationDidStop (animationID : NSString , finished : NSNumber , context : void) : void - diminishAnimationDidStop (animationID : NSString , finished : NSNumber , context : void) : void - magnifyAnimationDidStop: (animationID : NSString , finished : NSNumber , context : void) : void

3.2.1.1 Instance variables:

1. imageLoader – The singleton ImageLoader object
2. board – The singleton Board object
3. score – The current score that player get
4. life – The life left. In another words, the number of rotation allowed without any matches afterwards.
5. Hits – The numbers of successive matches by either a single tap or double tap.

3.2.1.2 Instance methods:

1. gameInit – This method initializes the game object and returns it. It also creates the ImageLoader and Board objects, and initializes all the instance variables.

2. rotationAtCenter – This method starts the procedure of rotating the monsters surrounding the center monster (the first parameter) in either clockwise or anti-clockwise direction (the second parameter).
3. rotationAnimationDidStop – This method is called after rotation animation completes, and it checks whether matches happen.
4. shakeMonsters – This method performs shaking animation of all the monsters int the array (first parameter). A shaking animation in fact comprises of many single small translation in any direction within a short period of time. This method just performs a small translation in a randomly assigned direction. In order to have a complete shaking animation, this method must be called many times. It is done by assigning a method that the small translation animation will call after finishing. The called animation then will simply call this method again. The third parameter is a counter which store the current iteration number, and is used for terminating the loop.
5. shakeAnimationDidStop - This method is called after a small translation animation completes. It simply calls the shake monsters again.
6. diminishAnimationDidStop – This method is called after diminishing animation completes. This method is also responsible to assign the types of the replaced monsters.
7. magnifyAnimationDidStop – This method is called after magnifying animation completes.

3.2.2 Board:

Board
- allSquares : NSMutableArray - map : Point2D[][] - gameTimer : UILabel - gameScore : UILabel - gameLife : UILabel
+ initTheBoard (g : TheGame) : id - generateMap: (numOfPotentialMatches : int , typeMap : int[][]) : void + getSquareWithLocation (x : int , y : int) : id + getSquareWithMapLocation: (x : int , y : int) : id + getPointInMapWithLocation: (x : int , y : int) : Point2D + setPointInMapAtLocation: (x : int , y : int , point : Point2D) : void + getMonsterTypeWithMapLocation: (x : int , y : int) : int + updateTimer: (t : int) : void + updateScore: (s : int) : void + updateLife: (l : int) : void

3.2.2.1 Instance variables:

1. allSquares – A mutable array that stores all the monsters. The indexes in this array are not the actual positions of monster show in the game. There is another two dimension array called map which store the positions of all the monsters in this 2D array, and its indexes are the positions of monsters in the game. For instance, the monster at map[0][0] is located at the allSquares[map[0][0].x][map[0][0].y].
2. map – It is a two dimension array storing the positions of all monsters in allSquares. The indexes of this 2D array are the actual positions of the monsters in the game. Point2D is a C structure with two integer values. They are x and y which stand for the x and y coordinates respectively.
3. gameTimer – It is a label showing the time remained.
4. gameScore – It is a label showing the player's score.
5. gameLife – It is a label showing the life remained.

3.2.2.2 Instance methods:

1. initTheBoard – In this method, the positions of all monsters are arranged porperly in the 2D array map. The monster objects are created here and their types are also assigned. This method in fact runs an algorithm, and it ensures that once the monsters' types are defined, no matches would happen immediatelty. Moreover, it also ensures that player must be able to find out at least a number of matches which is set by programmmers.
2. generateMap – This is the method that runs the algorithm discussed above. Roughly speaking, for each of the position, it first creates a monster object and then it checks all possible types. The possible types mean the types that do not match with the neighbours.
3. getSquareWithLocation – This method returns the square object at allSquare[x][y].
4. getSquareWithMapLocation – This method returns the square object at allSquare[map[x][y].x][map[x][y].y]
5. getPointInMapWithLocation – This method returns the C structure Point2D at map[x][y].
6. setPointInMapAtLocation – This method assigns the argument point to map[x][y].
7. getMonsterTypeWithMapLocation – This method returns the type of the monster at allSquare[map[x][y].x][map[x][y].y].
8. updateTimer – update timer.
9. updateScore – update score.
10. updateLife – update score.

3.2.3 ImageLoader:

ImageLoader
- backgroundImage : UIImage - monsterImages : NSArray
+ initWithAllImagesLoad () : id + getMonsterImagesWithNumber (type : int) : void

3.2.3.1 Instance variables:

1. backgroundImage – A UIImageView object representing the background.
2. monsterImages – A one dimension array storing all the images of different types of monsters.

3.2.3.2 Instance methods:

1. initWithAllImagesLoad – This is the constructor of ImageLoader object. It loads all the monsters' images into the memory, and return the object itself.
2. getMonsterImagesWithNumber – This method returns the pointer to an UIImageView object according to the type (fisrt parameter).

3.2.4 Monster:

Monster
- type : int
+ initMonster (game : TheGame , type : int , mapX : int , mapY : int) : id + changeType (type : int) : void - touchesBegan (touches : NSSet , event : UIEvent) : void - touchesEnded (touches : NSSet , event : UIEvent) : void - clockwiseRotation : void - antiClockwiseRotation : void

3.2.4.1 Instance variables:

1. type – The type of this monster.

3.2.4.2 Instance methods:

1. initMonster – This method creates and return the monster object with specified type and position in the 2D array map.
2. changeType – This method changes the type of this monster to new type (first parameter) and changes the image accordingly.
3. touchesBegan – This method is called once player's finger touches this monster on the screen.
4. touchesEnded – This method is called once player releases his finger from this monster on the screen.
5. clockwiseRotation – This method calls the method rotationAtCenter in TheGame object for performing clockwise rotation.
6. antiClockwiseRotation - This method calls the method rotationAtCenter in TheGame object for performing anti-clockwise rotation.

3.2.5 Square:

Square
- locationInMap : Point2D - game : TheGame
+ setNewImage (image : UIImage) : void

3.2.5.1 Instance variables:

1. locationInMap – This is a C structure, and consists of two integer values x and y. It tells us where we can find the position of this square in the map array in Board object.
2. game – The game object

3.2.5.2 Instance methods:

1. setNewImage – This method replaces the old image of this square with the new one (first parameter).

3.2.6 MonsterManagerAppDelegate:

MonsterManagerAppDelegate
- window : IBOutlet UIWindow - viewController : IBOutlet MonsterManagerViewController

3.2.6.1 Instance variables:

1. window – This is the highest object in the view hierarchy. We can add children (sub-views) to this object.
2. viewController – This is an object consists of a view object, and methods to handle this view object.

3.2.7 MonsterManagerViewController:

MonsterManagerViewController
- game : TheGame
+ setGame : void + viewDidLoad : void

3.2.7.1 Instance variables:

1. game –The game object.

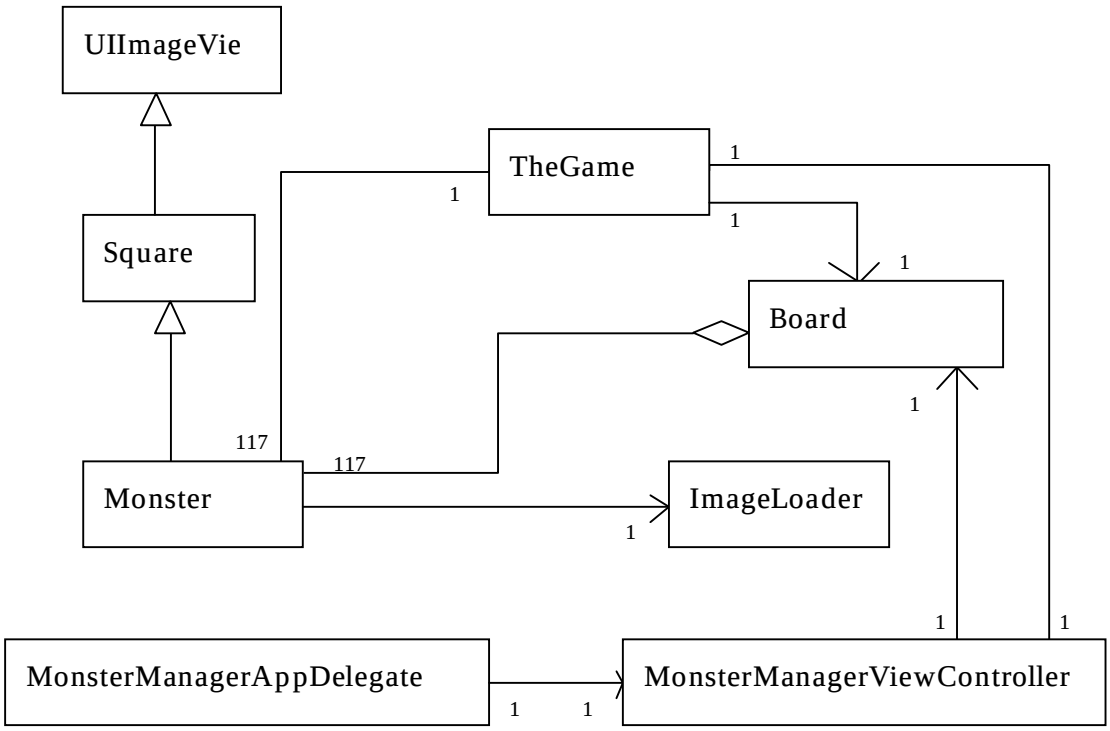
3.2.7.2 Instance methods:

1. setGame –This method simple crate a singleton TheGame object.

2. viewDidLoad – This method is responsible to add the UIImageView objects (all the monsters, background), and UILabel objects (the timer, score and life lables) to the UIView object at this controller object.

3.3 Class Diagram Relationship:

A class diagram showing the relationship between objects:



At the left hand side of this class diagram, there is a hierarchical relation. UIImageView is the class representing an image. Square class inherits UIImageView class and Monster class inherits Square class. The reason why Monster class does not directly inherit UIImageView class is that we can easily add other classes as the children of Square class. Also, we can easily introduce other classes, such as Triangle or Circle classes as the children of UIImageView class.

At the bottom of the class diagram, MonsterManagerAppDelegate object creates MonsterManagerViewController, and MonsterManagerViewController object in turn creates TheGame object.

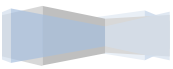
TheGame object and Monster object are associated with each other because once a touch event occurs; the monster object that is touched would call the rotationAtCenter method in TheGame object so as to run the rotation procedure. On the other hand, TheGame object needs to call the method changeType of all the matched monsters after matches happen.

During initialization or replacement of monsters, the monsters object must get the correct image corresponding to its type by calling the getMonsterImagesWithNumber method in ImageLoader object. Their relationship is uni-directional as ImageLoader object does not know about monster class, while all monster objects know about ImageLoader class.

Monster class also has the basic aggregation relationship with the Board class. All the monsters can be easier obtained by calling the methods in a Board object.

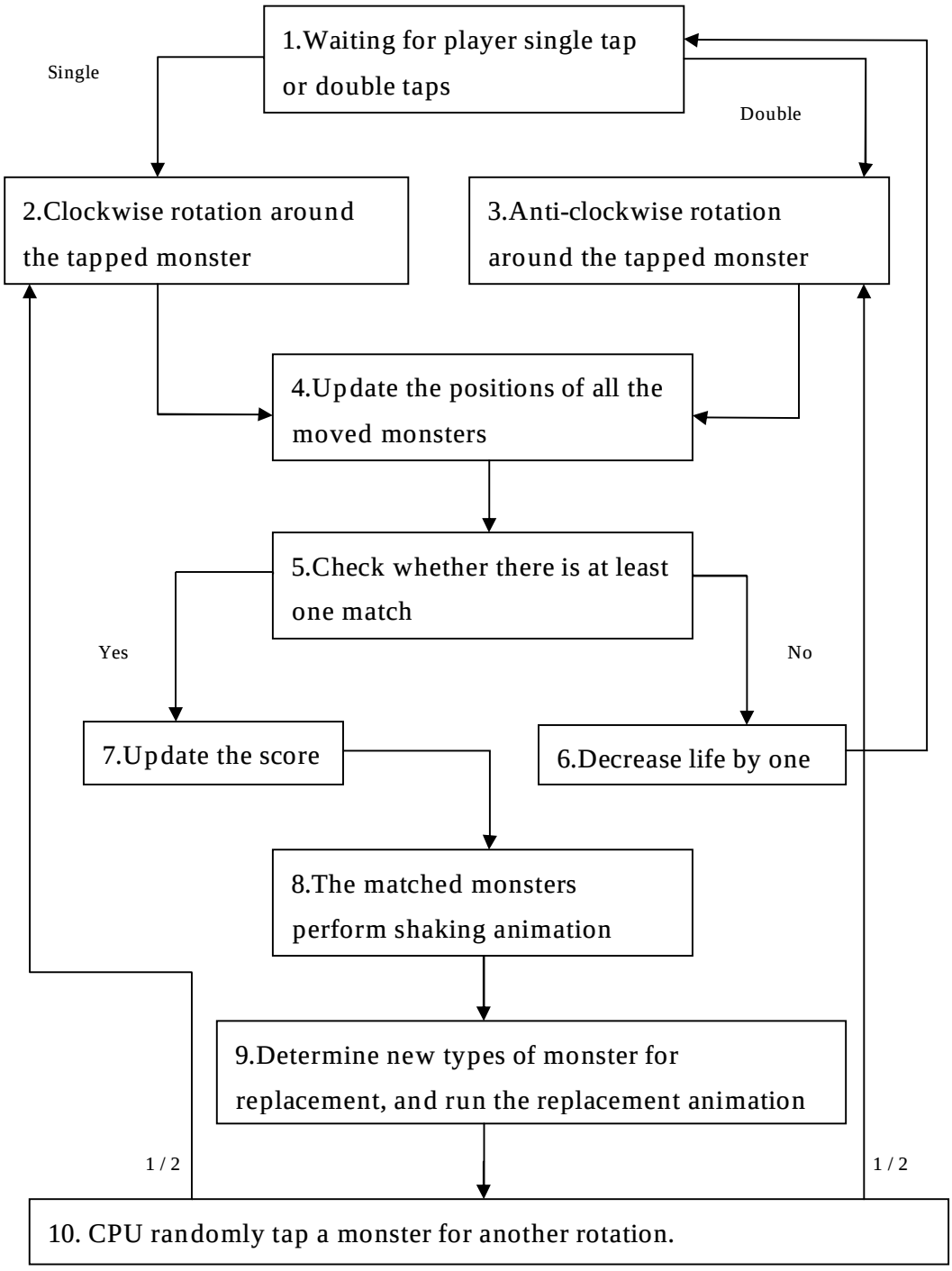
TheGame class is associated with Board class. TheGame object that run the game logic usually needs to take a look at the arrangement of monsters inside the frame, and that information can only be retrieved via the Board object.

At the bottom right corner, MonsterManagerViewController class is associated with Board class. In the controller object, there is an important method (viewDidLoad) that adds the background, monsters and labels into the controller's view which can only be obtained at Board object.



4. Game Flow:

4.1 Control flow diagram:



The above diagram shows the all the existing steps that the game flow may come across.

1. Once the player's input is received, either single tap or double tap a monster on the screen.
2. If a single tap is received, then performs clockwise rotation around the tapped monster.
3. If a double tap is received, then performs anti-clockwise instead.
4. Afterwards, the game program must update all the moved monsters' position before any checking on matches.
5. Then, check the neighbors of all the moved monsters and see whether they have the same type as the moved monsters, and at least three of them are linked horizontally or vertically.
6. If no match happens, the life is decreased by one, and the game would then wait for another player's tap. If the life is zero, terminates the game.
7. If there is at least one match, first update the score
8. The matched monsters perform the shaking animation in order to notice that they are matched.
9. Then replace the matched monsters to new monsters with animation.
10. After the replacement animation, the program would randomly select a center monster and perform either clockwise or anti-clockwise rotation around it and the same procedure runs again.

5. Problems Encountered:

5.1 Design Problems:

5.1.1 Monsters' Images:

Originally, we set the size of all square monster images to 32x32. The reason is that the width and height of screen are 320 and 480 respectively which are both dividable by 32. By doing simple division, we know that we can at most put 10 rows and 15 columns of monsters on the screen. We both believe that the size of the image is set appropriately. However, once we try to show the image in iPhone, we find out there is a big problem.

The image is blurred and darkened in iPhone. Also the color of the image is not exactly the same as the original one shown on PC as the size of pixel on iPhone is smaller than that of pixel on the LCD monitor. Since the image becomes smaller, it is hard to see the image in detail, and touching to a particular monster becomes difficult too.

Hence, we enlarge the monsters' images to 40x40, adjust the brightness, and sharpen them. Also, we have considered the size of each image and make sure that it is not too big in order to avoid significant reduction of speed during loading and animation.

5.1.2 Game Rules:

Game rules (how to play the game) are the essence of a game, and any game cannot get rid of them. As programmers, the game rules are also our main concern. It is because game rules can affect the difficulty of the implementation, the simplicity of the game play and the value of the game. When we were discussing the rules of our game a few weeks before, we did not consider much about the above issues. We only focus on the uniqueness and attractiveness of the rules, and did not think about the effects of a particular set of game rule to the entire game. As a result, at the beginning and middle of the semester, we in fact kept on designing and then dropping the game rules.

We now understand the truth that the thing we imagine in mind is usually a piece of puzzle; there should still lots of related issues that we do not imagine. Therefore, besides imagining idea, analysis is also important. For example, during

designing our game, we wanted to make use of the accelerometer installed on iPhone. We thought that it would be an interesting idea to allow player to control the direction of gravity or allow player to move monsters to fill up the empty space. However, when we started bringing our idea to reality, many problems arouse. First of all, we did not really estimate the value and side effects of this feature during the game play. After a match happened, it is seldom that player would really try to think of the direction of gravity in order to obtain a match. It is because the chance of getting a match in this way is not very high. On the other hand, player would have a higher chance to find out a match if they concentrate on the matches via rotation. Moreover, allowing player to change the direction of gravity would induce side effect. As we have talked before in the analysis of game design part, player would require some time to think of which direction is the best, and this properly slow down the game flow, and the game becomes unexciting. The second problem is the implementation difficulties. It is worthless to spend a lot of time to work on solving the difficulties of a low value feature.

For the design of how to move the monsters, we also spend a lot of time. Many nice and interesting ideas are proposed by us, but most of them have unacceptable defeats. The reason behind is that we only focus on the uniqueness, attractiveness of idea, but do not think of whether the idea is practical or not.

5.1.3 Rotation Problem:

It is impossible to select the outermost monsters at the edges as the center of rotation.

+	+	+	+	+	+	+
+	x	x	x	x	x	+
+	x	o	o	o	x	+
+	x	o	o	o	x	+
+	x	o	o	o	x	+
+	x	x	x	x	x	+
+	+	+	+	+	+	+

The monsters (x) do not have exact eight monsters around their. Therefore, they cannot act as the center of rotation. No matter how large the frame is, the problem still exists. In order to solve this problem, we do not make the outermost monsters visible to player, and they are hidden behind the frame (+).

#	#	#	#	#	#	#
#	x	o	o	o	o	#
#	o	o	o	o	o	#
#	o	o	o	o	o	#
#	o	o	o	o	o	#
#	o	o	o	o	o	#
#	#	#	#	#	#	#

Positions (#) still representing frame, but there are monsters hidden under them. If monster x is selected, the monsters at positions highlighted in red will rotate around monster (x), and the monsters under (red #) will rotate accordingly.

5.2 Implementation Problems:

5.2.1 Data Structure to Store All Monsters:

As all the monsters are packed to form a large rectangle, it seems that we can simply use a two dimension array to store the pointers to different monsters. However, in object-oriented point of view, this method is not a good idea. Therefore, we manage to find an alternative way which no array is needed.

The alternative approach is to make use of pointers. Each monster object has four additional pointers. They point to the adjacent monsters in north, east, south and west direction respectively. If there is no monster in some direction, the pointer of that direction will be assign to null. The following diagram illustrates this idea.
In the following example, there are nine monsters inside the frame.

Monster1	Monster2	Monster3
north = null east = monster2 south = monster4 west = null	north = null east = monster3 south = monster5 west = monster1	north = null east = null south = monster6 west = monster2
Monster4	**Monster5**	**Monster6**
north = monster1 east = monster5 south = monster7 west = null	north = monster2 east = monster6 south = monster8 west = monster4	north = monster3 east = null south = monster9 west = monster5
Monster7	**Monster8**	**Monster9**
north = monster4 east = monster8 south = null west = null	north = monster5 east = monster9 south = null west = monster7	north = monster6 east = null south = null west = monster8

Given a monster, it is very easy to get the pointer of the monster related to the given monster. For example, if we want to get the monster next to monster 5 in north direction, we can simple access the instance variable north of monster5

(monster5.north). If we want to get monster6 from monster4, calls (monster4.east.east). This method does not require any array, and allows further growth of number of monsters in real time, while a standard array does not allow changing its size in real time. If array is used, when a monster needs to get its neighbors, it must access the object where the array is stored. On the other hand, the retrieving of monsters can be done within the Monster object.

However, we do not use this method as it is not suitable for the condition that monsters are required to change their positions all the time. If a monster moves to other location, we must update all its pointers at the same time. It is not a big problem if only one monster moves. However, it is a complicated problem if lots of monsters move together. On the other hand, a two dimension array does not have this problem.

5.2.2 Initialization of Type Map:

Before creating monster objects, we first need to generate a type map (a 2D array storing the types of the monsters at each position). By referring to the generated type map, the program can create monster with type stored in type map. The simplest way to generate the type map is using a random number generator. However, there is a big problem. The type map generated may cause some monsters already matched. We must avoid this to happen. Therefore we design an algorithm to generate a type map that no match happen at the beginning and also ensure that player can find matches at the beginning.

There are in a lot of patterns that player can obtain a match in one rotation, and some of them are… (Cross stands for one type of monster, circle stands for monster other than monster (x))

o	o	o
x	x	o
o	o	x

o	o	x
x	x	o
o	o	o

o	x	o
o	x	o
x	o	o

o	x	o
o	x	o
o	o	x

x	o	o
o	x	x
o	o	o

o	o	o
o	x	x
x	o	o

x	o	o
o	x	o
o	x	o

o	o	x
o	x	o
o	x	o

x	x	o
o	o	x
o	o	o

o	o	o
o	o	x
x	x	o

o	o	o
x	o	o
o	x	x

o	x	x
x	o	o
o	o	o

o	x	o
x	o	o
x	o	o

o	x	o
o	o	x
o	o	x

x	o	o
x	o	o
o	x	o

o	o	x
o	o	x
o	x	o

In order to guarantee player to be able to find matches, we should put some of the patterns like the above to the type map. Therefore, our algorithm consists of two

parts. First, insert a number of patterns to the type map. Then, fill up the rest of the types.

After inserting several patterns, the next step is to fill in the rest of the blanks in the type map.

```
for each s in type map
    if s == NULL
        while (true)
            newType = random() % 8;
            // check north direction
            if match happens by considering north direction (1)
                continue;
            if match happens by considering south direction (2)
                continue;
            if match happens by considering west direction (3)
                continue;
            if match happens by considering east direction (4)
                continue;
            if match happens by considering north and south directions (5)
                continue;
            if match happens by considering east and west directions (6)
                continue;
            s = newType;
            break;
```

Statements (1), (2), (3) and (4) mean that checking is performed along the north, south, west and east directions respectively. If there are at least two successive monsters have the type (newType), newType is not suitable and the while loop will restart again.

If statements (5) and (6) mean that checking is performed along the north-south and east-west directions respectively. If there are at least one monster in north direction and at least one monster in south direction, the while loop will restart again.

Also, if there are at least one monster in east direction and at least one monster in west direction, the while loop will restart again.

In fact, there is a simpler algorithm than the above one, but the result is not as good as it. This simpler algorithm first gets the adjacent monsters' types (A) in all four directions. The algorithm then randomly selects a type in the set (all types - A). This algorithm is fast but the generated type map does not contain any two monsters of the same type are adjacent with each other, and it is unacceptable.

5.2.3 Mutable Array problem:

Since it is annoying to create a two dimension array using NSArray class, we use NSMutableArray class. The main difference between them is that NSMutableArray can change its size, add elements and remove elements in real time. Therefore, we use an NSMutableArray object to create our two dimension array to store the monsters. However, there is a problem when we want to shift several monsters in the NSMutableArray object to other positions.

If we remove an object from an NSMutableArray, all elements beyond the gap are moved by subtracting 1 from their index. If we add an object from an NSMutableArray, all elements beyond the inserted object are moved by adding 1 from their index. These features of a mutable array would cause us to think carefully how to move a monster from one position to another position in the mutable array correctly.

On the other hand, what we want is a two dimension array that its size does not change. Therefore, we design to use a C language 2D array to store the positions of pointers, while we use NSMutableArray to store the pointers to different monsters.

(1, 0)	(2, 0)	(1, 2)
(0, 0)	(0, 2)	(1, 1)
(2, 2)	(0, 1)	(2, 1)

Monster 1	Monster 2	Monster 3
Monster 4	Monster 5	Monster 6
Monster 7	Monster 8	Monster 9

The left hand side is a 3x3 2D array storing the positions of pointers, and its indexes represent the actual positions of monsters in the game, while the right hand side is a 3x3 2D array storing the pointers of monsters. If we want to know what is monster's type at location (0, 0) in the game. First, we need to know where we can obtain the pointer to that monster. By referring to the 2D array at the left, the position

of pointer is (1, 0). Then, by referring to the 2D array at the right, the monster located in (0, 0) in the game is Monster 2.

If we want to swap the monsters at (0, 1) and (1, 1) in the game, change the 2D array on the left hand side to this.

(1, 0)	(2, 0)	(1, 2)
(0, 2)	(0, 0)	(1, 1)
(2, 2)	(0, 1)	(2, 1)

This approach can avoid changing the content of the NSMutableArray, and thus prevent the problem described above.

5.2.4 Checking Matches Algorithm:

After rotation, the game program must check if matches happen. The algorithm of checking matches is quite similar to the algorithm of generating a type map.

```
matchedMonsters = {}
for each moved monster m
    if match happens by considering north direction at m
        for each monster x along north direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
    if match happens by considering south direction at m
        for each monster x along south direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
    if match happens by considering west direction at m
        for each monster x along west direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
    if match happens by considering east direction at m
        for each monster x along east direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
    if match happens by considering north and south directions at m
        for each monster x along north direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
        for each monster x along south direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
    if match happens by considering east and west directions at m
        for each monster x along east direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
        for each monster x along west direction at m
            if m.type == x.type && x is not in the set matchedMonsters
                add x to matchMonsters
```

5.2.5 The Animation:

The main problem in handling the animation of our game is not how to create the animations, but how we can link up the animations properly. In our game, players can see the following animations.

1. Clockwise rotation animation – When player tap a monster once, the other monsters around the tapped monster would perform clockwise rotation.
2. Anti-clockwise rotation animation – When player double tap a monster, the other monsters around the tapped monster would perform anti-clockwise rotation.
3. Shaking animation – When there are matched monsters, they perform shaking animation so as to inform player that he or she has successfully found out matches.
4. Diminishing animation – The first half of the replacement animation.
5. Magnifying animation – The second half of the replacement animation.

Each animation should run one after the other. In another words, after an animation completes, it should triggers another animation or other procedures. Fortunately, iPhone SDK allows programmers to assign a particular method for each animation. That method is called just after the animation completes. Hence, we can easy setup the control flow of the game program. Otherwise, we may need to use flag variables together with while loop so as to wait for the completion of any animations.

6. Conclusion:

6.1 Accomplishment:

In this semester, we have successfully developed a playable game, and almost all ideas in our mind have come true now. During the development, we learnt a lot about how to write an iPhone application by using Objective-C programming language and Xcode.

6.2 Further Suggestions:

In fact, we have planned to develop a game mode for two, three or even four players. In this game mode, iPhones are connected via Wi-Fi. Players will face to the same frame with lots of monsters inside. Each player will own part of the types of monsters available in game. The main target of each player is to find matches of his or her own types of monsters as much as possible. Since there are multiple players, they will take turns to select a monster for rotation within 10 seconds. After the time is over, the player with the highest scores is the winner. This game mode is interesting as player not only needs to care about finding matches, but also think about how to prohibit the opposites to score.

In addition, since we are developing a game on iPhone, we should try to make use of the special features provided by it, such as accelerometers, multi-touch, camera, etc. For example, we can use multi-touch feature to allow player to select multiple monsters and perform several rotations at the same time.

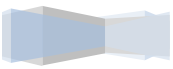
Also, as the time is not enough, we have not implement the timer, sound effects of the game, and design the settings of each level. So, maybe we can finish them later.

6.3 Reflection:

In this semester, we have read through a lot about Objective-C programming and iPhone OS Programming. The experience in implementation of Monster Manager has also provided us a general view of game design. Being a developer for the world latest technology product is a wonderful experience. During our implementation, the iPhone application market is ongoing growing at the same time. Different creative applications are blooming, and they simulate us to do our best in this final year project. The objective of our final year project is to explore and learn.

In particular, we have explored the new product, iPhone and learnt the objective-C language, which was brand-new to us. In the process of development, we understand the importance of self-learning. As technology changes with each passing day, self-learning skill can get us updated.

By the end of this semester, it is not only the break of our final year project development; it is also a time for us to plan for our next stage. For the next semester, with the technique learnt, we can develop to something more challenging and user-friendly application for the iPhone users. Except a simple game can be made, more innovative application can be developed. All other details would be settled down in the near future.



7. Acknowledgement:

We would like to express our heartfelt thank to our project supervisor, Professor Prof. KING Kuo Chin, Irwin who has given us many useful advices on the project.

8. References:

1. Useful PDF: (available in http://developer.apple.com/iphone/)
   - iPhone OS Programming Guide
   - The Objective-C 2.0 Programming Language
   - UIKit Framework Reference
   - Model Object Implementation Guide

2. Useful websites:
   - iPhone Development Center - http://developer.apple.com/iphone/
   - iPhone Development Central - http://www.iphonedevcentral.org/home.php