Софийски Университет „Свети Климент Охридски“
Факултет по Математика и Информатика

Курсова работа

Тема: „Feed The Snake“

Учебна дисциплина: Обектно Ориентирано Програмиране със C#

Изготвил: Минко Добромиров Гечев, специалност Информационни системи, 2ри курс, фак. н. 71202

Съдържание

Увод

В настоящия документ ще разгледам пълната функционалност на играта, използваните алгоритми, технологии и способи на обектно-ориентирания дизайн в изготвената от мен курсова работа. За да бъде представата за проекта по-пълна съм подготвил и UML диаграми, които могат да бъдат разгледани надолу в текста. За диаграмите съм използвал Rational Software Architect, който ни бе предоставен от курса Информационни Системи – Теория и Практика. Представянето съм започнал с модела на класовете (тъй като той има водещо значение в трите модела), където съм разгледал релациите между тях. Следва подробно описание на всеки един от класовете. Продължил съм с state machine диаграма, където съм описал през какви състояния преминава класът Controller, след като бъде създаден. Описанието съм продължил с диаграма на вариантите на употреба. Там съм изложил всички възможности, които съм предоставил на потребителя, чрез главното меню. Най-сложният от всички варианти на употреба, разбира се е самата игра. Именно заради това с activity диаграма съм описал създаването на играта, игралния процес и края и.

Използвана литература

1. Кръстев Е. – Лекции по „Обектно-ориентирано програмиране със C#.NET“

2. Vector orientation - http://mathworld.wolfram.com/VectorOrientation.html

3. O’Reilly Programming WPF 2nd Edition August 2007

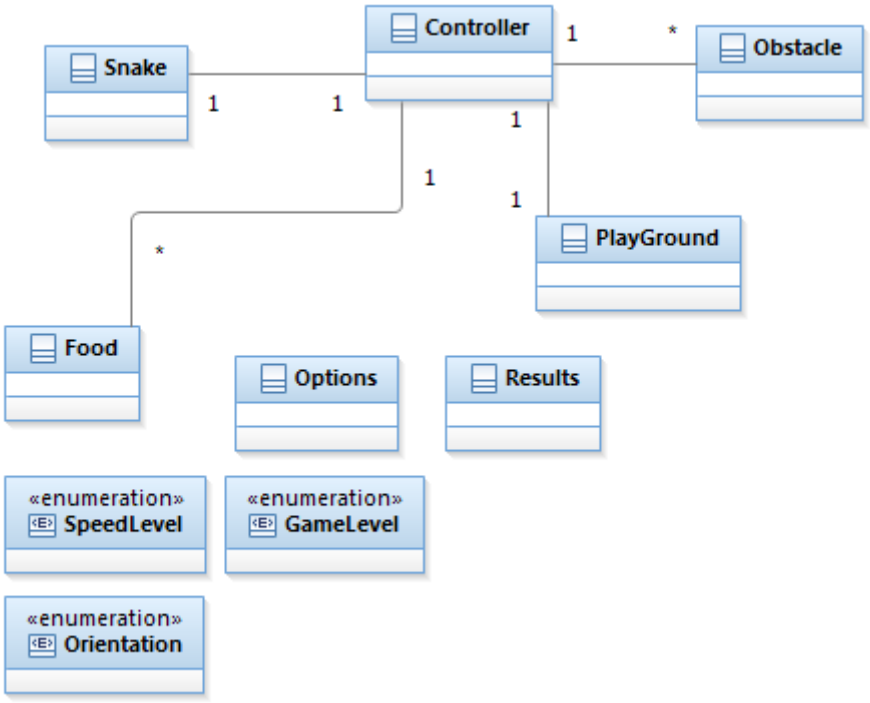
4. Статии за WPF в http://stackoverflow.com/

5. MSDN Library

Използвани технологии

Проектът е писан на C#.Net, като за разработката му съм използвал Microsoft Visual Studio 2008 Professional. За графичния потребителски интерфейс (GUI) съм използвал WPF (Windows Presentation Foundation).

Класове и релациите между тях

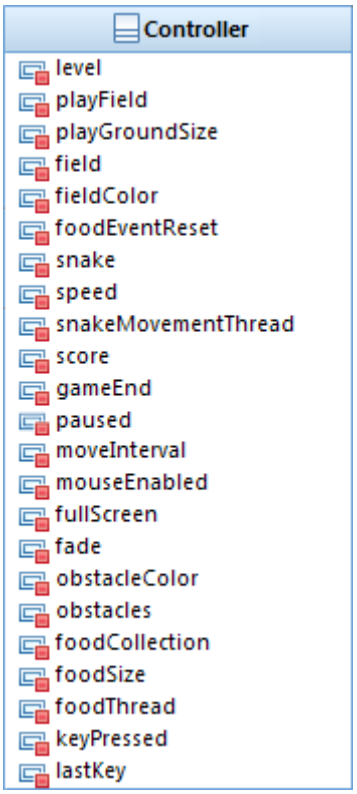


В горната диаграма съм показал основните класове, които използвам при реализацията на проекта. Класовете, които не съм добавил ще бъдат разгледани по-долу в текста.

При проектирането на Feed The Snake съм използвал шаблона MVC. Контролерът се забелязва, като към него с различни асоциации съм свързал и моделите (Snake, Obstacle, Food), както и View-то – PlayGround. Класовете, които не са в пряка релация с MVC (Options, Results) се грижат за допълнителни дейности, които не са пряко свързани с играта. Конструкторът на контролера играе и ролята на Factory method, за самата игра. В него се инициализират различните атрибути на Snake, PlayGround, Obstacle, стартира се нишката, грижеща се за периодично показващата се „храна“ и тн. Подробно самия конструктор на контролера ще бъда разгледан по-късно. SpeedLevel, GameLevel, Orientation са enum, които специфицират различни характеристики на играта. Имената им говорят сами за себе си. По-подробна информация за тях също ще намерите надолу в текста. Релацията между Snake и Controller е 1 към 1, поради факта, че за създаването на инстанция от тип Snake съм използвал design pattern-а Singleton, заради нуждата от само един обект от този тип за single player игра. Тъй като според различните нива на сложност, на полето предвидено за игра, има различен на брой препятствия съм решил асоциацията между Obstacle и Controller да бъде 1 към *. Поради неконстантния брой „храна“ на полето във всеки един момент, релацията между Food и Controller е със същата кратност. Заради възможността на играта да се извършва само върху едно игрово поле, класът Controller притежава един единствен PlayField.

Класът Controller

В настоящата секция ще разгледам класът Controller, който има главна, управляваща роля в самата игра.



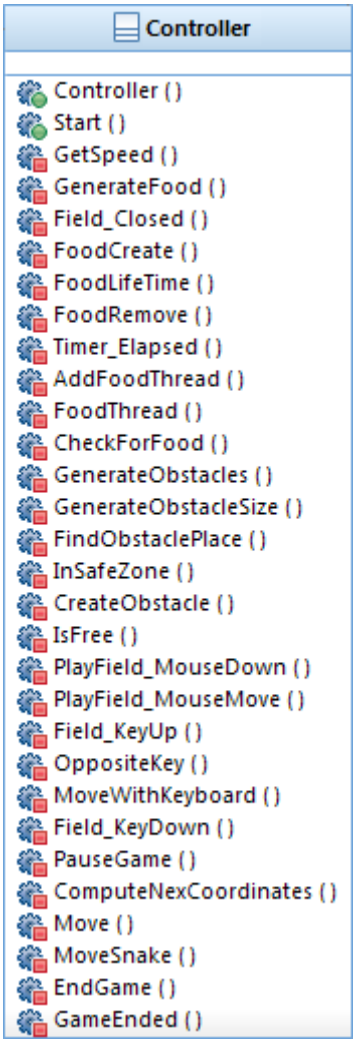
Тъй като Controller има голямо множество от атрибути и методи, ще започна с разглеждане на атрибутите и след това ще премина към методите. Променливата level е от тип GameLevel (enum) и съхранява текущото ниво, на което се играе. Нивото по подразбиране може да бъде променено от Options в главното меню. playField е променлива от тип Canvas. Върху този Canvas се играе самата игра. Размерът на текущото поле може да бъде променен при задаване на различна стойност на playGroundSize (от тип Size). Прозорецът в който се създава playField е от тип PlayGround и се идентифицира с променливата field. fieldColor (от тип Brush) предлага възможност за промяна на фона на Canvas (playField). snake е от тип Snake и представлява змията – основния актьор в играта. speed е от тип int, като получава стойността си след експлицитно кастване на променлива от тип SpeedLevel към int. Змията извършва своето движение в полето чрез нишката за движение – snakeMovementThread, която се стартира веднага след натискане на някоя от стрелките или клик на мишката – в случай, че мишката е активирана. В променливата score се съхранява текущия резултат. В началото score се инициализира със стойност 0, като в последствие в процеса на играта се увеличава (или не), според това колко „храна“ е изяла змията. Променливите от тип bool – palused и gameEnd показват дали играта е в режим на пауза или е приключила. Цветът на препятствията – obstacleColor и самия списък с препятствия obstacles (List<Obstacle>) съхраняват съответно цвета на препятствията и всички създадени препятствия.

Размерът на елипсите представляващи „храната“, се изчислява спрямо размера на игралното поле, съхраняващо се във foodSize. Всички създадени обекти от тип Food се съхраняват във foodCollection. Нишката, която се грижи за създаването на „храна“ през определени интервали от време и премахването на „храната“ след изтичане на случайно генериран интервал е foodThread. За да се избегнат проблеми свързани с играта, при използване на клавиатурата, съм добавил lastKey и keyPressed. Така се отстранява възможността при натискане и задържане на някоя от стрелките, скоростта на змията да е по-бърза от обикновено (чрез keyPressed). lastKey помага при опит за натискане на противоположни стрелки (например лява, след което дясна), защото тогава змията се счита за самопресякла се и играта приключва.

Ще продължа разглеждането на класа Controller с преминаване през неговите методи. Първо ще разгледам конструктора не особено подробно, тъй като описах всички атрибути на класа по-горе. След това ще направя кратко описание на всички методи в класа, последвано от

диаграма на състоянията, през които преминава класа и по-подробно описание на ключовите методи.

Както по-горе споменах, конструкторът на Controller представлява нещо кат Factory method pattern. Необходимите за инициализация на играта стойности, се набавят от файла options.dat, чрез статичния метод на класа Options – LoadOptions.



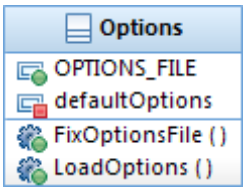
Методът Start, стартира самата игра, като стартира генерирането на препятствия и „храна“. GetSpeed изчислява скоростта на змията, според големината на игрището. GenerateFood инициализира таймер, при чийто Elapsed се създава „храна“. Field_Close прекратява генерирането на „храна“, като този метод се стартира при затваряне на формата, съдържаща игровото поле. FoodCreate създава нов обект от тип Food. FoodLifeTime е метод, който се грижи за регулацията на точките, които получава потребителят при изяждане на дадена „храна“ от неговата змия, според интервала от време през който е съществувала „храната“. FoodRemote се грижи за премахване на даден обект от тип Food. AddFoodThread е метод, който стартира нова нишка за даден обект от тип Food. CheckForFood е метод, който търси подходящо място за следващия обект от тип Food. Последният метод е управляващ съществуването на всички обекти от тип Food е FoodThread. Той се грижи за създаване и премахване на „храната“.

Методи грижещи се за генериране на препятствия: Obstacle създава определен брой препятствия, като негови помощни методи са – CreateObstacle (създаващ обект от тип Rectangle), FindObstaclePlace (търсещ подходящо място за препятствието), GenerateObstacleSize (генериращ размера на препятствието, според размерите на игрището).

Методът IsFree има две дефиниции, които се използват според своето предназначение. Едната се използва при генериране на препятствия, а другата при движението на змията и проверява дали змията е засегнала някое от препятствията. PlayField_MouseDown и PlayField_MouseMove са методи, които предизвикват някакво поведение в змията при натискане на клавиш на мишката или при движението и. Field_KeyUp задава стойност на keyPressed – false. MoveWithKeyBoard се грижи за движението на змията, чрез клавиатурата. PauseGame спира играта при натискане на бутона Pause от клавиатурата. Играта се стартира при повторно натискане на този бутон. MoveSnake придвижва змията, а Move има управляващо значение. Грижи се за изчисляване на координатите и извикване на метода MoveSnake. EndGame прекратява играта, а метода GameEnded проверява дали играта е приключила.

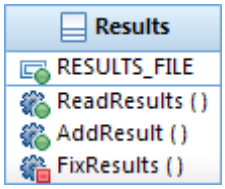
Накрая ще разгледам статичните методи на класовете Options и Results. Тъй като инстанции на тези два класа не са ни необходими, то в тях има само определен брой статични методи.

Нека първо разгледам класа Options.



Той притежава константа от тип стринг, в която се съхранява името на файла, в който са съхранени опциите на играта. Има и private статичен масив от стрингове, в който са записани настройките по подразбиране. Този масив се използва в метода FixOptionsFile в случай, че файлът с опциите бъде поразен. В FixOptionsFile, файлът OPTIONS_FILE се отваря за писане, като цялото му съдържание бива изтрито, като вместо старото му съдържание се добавят елементите от defaultOptions. Методът LoadOptions връща списък с всички опции.

Класът Results съдържа необходимите методи за работа с файла, съхраняващ резултатите от игрите. Публичната константа RESULTS_FILE пази името на този файл. ReadResults чете редовете от файла RESULTS_FILE по три на веднъж, като на първия ред е записано нивото на което се играе. Вторият ред е запазен за името на играча, а третият - е резултата му. Методът ReadResults връща обект от тип IEnumerable<KeyValuePair<int, KeyValuePair<int, string>>>, където int от първата двойка е нивото, int от втората двойка е резултата, а string е името на играча. Методът FixResults поправя файлът с резултатите, ако е поразен по някаква причина. AddResult добавя нов резултат в RESULTS_FILE.



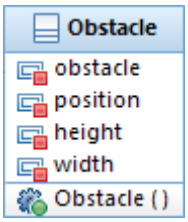
Класът Results се използва в StartWindow.xaml.cs. Тъй като зареждането и сортирането на всички резултати отнема време (около 5 секунди за 1 милион резултата, при процесор Intel Core 2 Duo T5800 2 GHz), изчислението пускам в отделна нишка и след като процесът приключи, визуализирам резултата. Ето самия метод:

```
ResultsForm resultsForm = new ResultsForm();
resultsForm.txtResults.Text = "Loading...";
resultsForm.Show();
Thread loadResults = new Thread(new ThreadStart(new Action(
()=>
{
      string result = ViewResults(5);
      Dispatcher.Invoke(System.Windows.Threading.DispatcherPrior
      ity.Normal, new Action(
         () =>
         {
               resultsForm.txtResults.Text = result;
         }));
})));
loadResults.Start();
```

Ще разгледам метода на сортиране на резултатите.

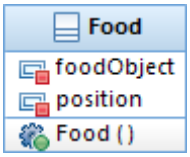
След извикването на метода AddResult, получавам списък от двойки. При обхождането на всички елементи от enum GameLevel, правя сортирване на двойките от втората част на двойката по брой точки. Ето как се получава това:

```
var sorted =
(from res in results
 where res.Key.Equals((int)i)
 orderby res.Value.Key descending
 select res);
```



Взимам първите пет резултата и ги добавям в StringBuilder. Така след като съм

обходил всички елементи от GameLevel, съм сортирал резултатите от игрите от всяко ниво. След това съм ги добавил в StringBuilder и мога да визуализирам резултата.

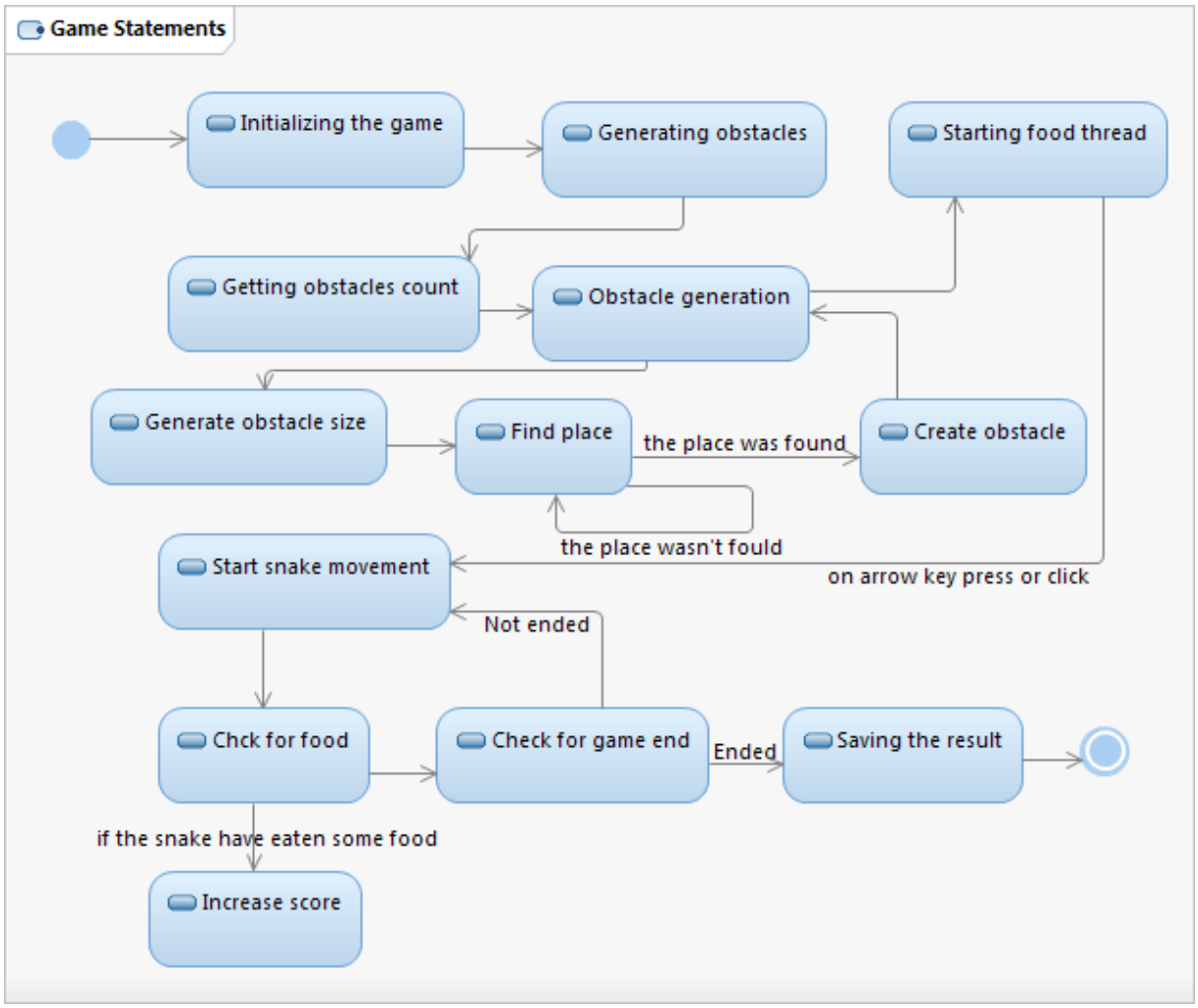


Нека разгледам класовете Obstacle и Food.

Obstacle съдържа обект от тип UIElement, position от тип Point, height от тип double и width от тип double. Идеята на този клас е да представлява по-добра абстракция и представяне на препятствията. Класът има като атрибути височина и широчина на препятствието, за по-лесно представяне при различните типове obstacle (например Line, Rectangle и други). Единственият метод в Obstacle е самият конструктор на класа.

Класът Food притежава атрибути: foodObject от тип Ellipse, position от тип Point (не използвам директно атрибутите на Ellipse, поради по-голямо удобство при използване на стойности в самия клас Food). Единственият метод е конструктора на класа.

Диаграма на състоянията



В тази секция ще разгледам състоянията, през които преминава Controller по време на игра. В началото е Initializing the game. Това е конструкторът (Factory method), който извлича настройките от файла options.dat, посредством статичния метод LoadOptions от класа Options. Освен съхранените настройки във файла има и други, които обаче зависят от вече заредените.

Те се пресмятат отново в контруктора на Controller. След преминаването през конструктора, играта се стартира и се преминава към генериране на препятствията. Методът GenerateObstacles (приемащ единствен аргумент число от тип int, което се получава при експлицитно кастване на GameLevel към int) се грижи за създаването на броя препятствия, според текущото ниво на играта. В един цикъл GenerateObstacles генерира случайна ориентация (хоризонтална или вертикална), размер на препятствието и намира подходящо място в следния цикъл:

```
while (!FindObstaclePlace(out newPosition, obstacleSize)) { }
```

Както се вижда цикълът се изпълнява, докато не е намерено място на препятствието с размер obstacleSize. При намиране на подходящото място newPosition се инициализира с генерираните координати. Следва генериране на самия обект, представляващ препятствието. Той трябва да имплементира интерфейсът UIElement. В случая съм използвал Rectangle, който се създава в метода CreateObstacle. Когато горните стъпки вече са преминали, се задават абсолютните координати на новия Rectangle в Canvas. Създава се нов обект от тип Obstacle и се добавя към списъка с препятствия.

След като преядствията са генерирани, се стартира метода грижещ се за генериране на „храна“. За това се грижи методът GenerateFood. Той създава обект от тип Timer, чийто интервал се задава според бързината на играта. Когато времето на таймера изтече (Timer_Elapsed), се добавя нова нишка „храна“ (AddFoodThread). В метода AddFoodThread в един ThreadPool се добавя нова нишка – FoodThread, в която се създава нов обект от тип Food. За целта се използва делегат от тип CreateFood. Методът, който приема като стойност екземпляра на този делегат е FoodCreate. В нишката на GUI се включва метода FoodCreate, който генерира цвят за съответната елипса (чрез GenerateColor), генерира позиция на обекта от тип Ellipse и създава нов обект от тип Food, който добавя във foodCollection. След като обектът от тип Food е създаден и видим за потребителя, започва неговото обратно броене. „Храната“ има определен интервал от време, през който е видима и е необходимо това време да изтече преди да я премахнем. Това обратно броене се извършва в метода FoodLifeTime. Той генерира продължителността на „живот“ на „храната“ :

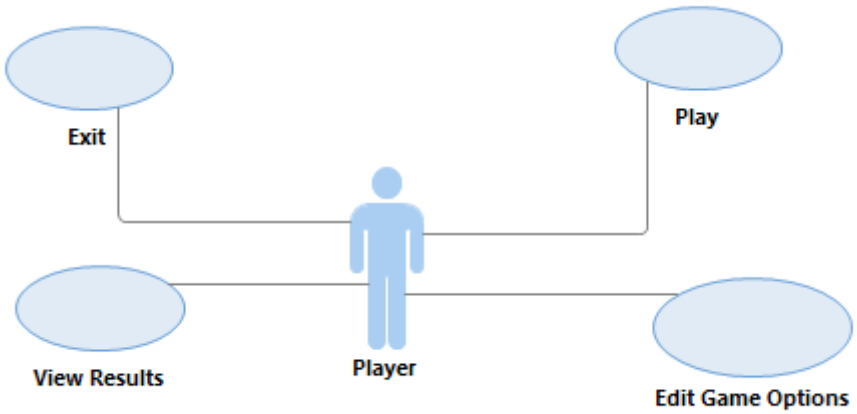
```
Random randomNumber = new Random();
//generating randome food life time in seconds
int lifeTime = randomNumber.Next(((int)speed) / 6,
((int)speed) / 6 + ((int)speed) / 6);
```

След което започва обратно броене:

```
for (int i = 0; i < lifeTime; i++)
{
     Thread.Sleep(1000);
     if (foodCollection.ContainsKey(foodObject))
     {
          foodCollection[foodObject]++;
     }
}
```

foodCollection е от тип Dictionary<Food, int>. Така за елемента на речника с ключ foodObject (текущия обект от тип Food), int стойността се увеличава на всяка секунда. Колкото е по-голяма стойността на елемента с ключ foodObject, толкова по-малко точки трябва да получи потребителя, когато змията му „изяде“ този обект. Затова точките за „изяден“ обект от тип Food се получават по следния начин *score += 100 / (obj.Value + 1);*. След като цикълът прикючи своето изпълнение, преминаваме към метода FoodRemote, приемащ като аргумент обект от тип Food - foodObject. От Canvas се премахва съответстващата на обекта елипса, а от Dictionary самата двойка <Food, int>, където ключът е foodObject.

Варианти на употреба



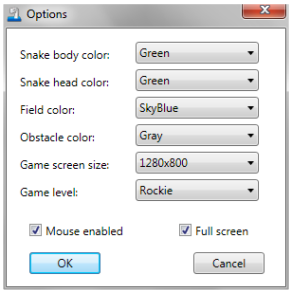
Тук ще разгледам основните варианти на употреба, които предоставя моят вариант на играта Feed The Snake.

След като потребителя отвори играта, ще попадне на менюто от Фигура 1.

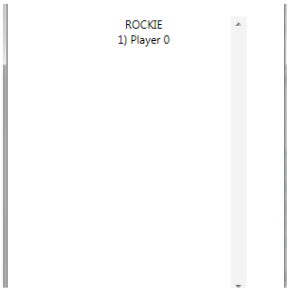
Фигура 1

От тук може да бъде започната нова игра. При това положение контролерът ще премине през състоянията описани по-горе. Потребителят може да избере и бутон Options. От появилото се меню (Фигура 2) ще могат да бъдат променяни основните атрибути на Controller, като например – цвят на тялото на змията, цвят на главата на змията, цвят на полето, цвят на препятствията, големина на игралното поле, трудност, дали змията да може да бъде управлявана от мишката, разрешаване или забраняване на пълен екран. Друг вариант на употреба е преглеждането на точките (Фигура 3). Ако потребителят избере Help ще види прозорец като този на Фигура 4.

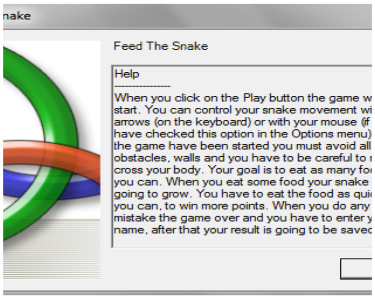
Фигура 2



Фигура 3

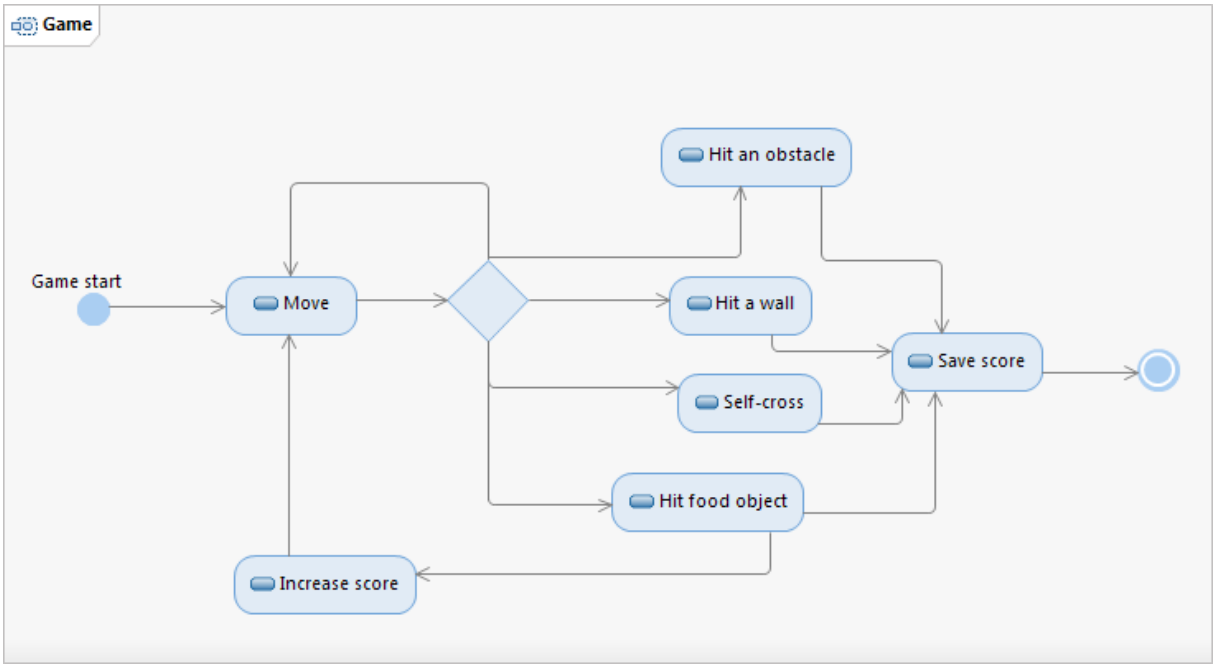


Фигура 4



Диаграма на дейността

В диаграмата на дейността по-долу, съм описал типичния сценарий, през който преминава една игра.



В началото на играта змията е неподвижна. При натискане на някоя от стрелките на клавиатурата или при клик на мишката върху игралното поле, змията започва да се движи. За да се изчисли следващото положение в което ще се намира първата точка от Polyline (изобразяваща змията), се пресмята уравнението на правата преминаваща през текущото положение на тази точка и курсора на мишката (при клик). Нека наречем тази права *a*. Следва пресмятане на дължината на вектор, колинеарен с правата *a.*

```
double currentLength = moveInterval / Math.Sqrt(a * a + b * b);
```

Настоящите координати се пресмятат със следния код:

```
u = -b * currentLength + lastHeadPositionX;
v = a * currentLength + lastHeadPositionY;
```

Тъй като първата точка от Polyline не съвпада с центъра на кръга представляващ главата на змията, то този център трябва да бъде изчислен. Това става на следните редове:

```
double x1 = position.X - ((snake.Head.Width +
snake.Head.Width) / 3) * Math.Cos(45 * Math.PI / 180);
double y1 = position.Y - ((snake.Head.Height +
snake.Head.Height) / 3) * Math.Sin(45 * Math.PI / 180);
```

Когато всички изчисления са вече направени, настоящата позиция на змията е известна и следователно можем да проверим дали тя е ударила някое препятствие или е попаднала на „храна“. И в двата случая алгоритъма е подобен. Използва се цикъл, с който се обхождат всички елементи от съответната колекция (с обекти от тип Food или Obstacle). Ще разгледам проверката за храна. Със следния булев израз проверявам дали главата на змията е засегнала някой активен обект от тип Food:

```
!(foodObj.Key.Position.X > x || foodObj.Key.Position.X +
foodObj.Key.FoodObject.Width < x) &&
!(foodObj.Key.Position.Y > y || foodObj.Key.Position.Y +
foodObj.Key.FoodObject.Height < y)
```

Тук *x* и *y* са координатите на главата на змията, а foodObj е обект от Dictionary съдържащ обекти от тип Food и интервала от време, през който дадения обект е съществувал.
Другата проверка, която се прави е дали змията е напуснала очертанията на игралното поле. Тя представлява следния булев израз:

```
snakeX >= playField.Width || snakeY >= playField.Height ||
snakeX <= 0 || snakeY <= 0
```

Ако очертанията бъдат напуснати, булевият израз връща true - в противен случай false. По-сложна е проверката за самопресичане. В нея използвам познатите от аналитичната геометрия ориентирани равнини. Самият метод, проверяващ за самопресичане се намира в класа Snake. Методът се нарича IsSelfCrossing, като използва помощните методи LineIntersects и Clockwise. IsSelfCrossing обхожда точките, съставящи змията по двойки, като на всеки един етап проверява дали отсечката образувана от двойката точки се пресича от отсечката образувана от първата и втората точка изграждащи Polyline. Нека означа първите две точки с (p1, p2), а вторите две (p3, p4). За да се пресичат тези две отсечки трябва p1 и p2 да лежат от различни страни на отсечката (p3, p4), т.е. векторите p1-p3 и p2-p3 трябва да имат противоположна ориентация с вектора p4-p3. Аналогично, p3 и p4 трябва да лежат от различни страни на (p1, p2).

В случай, че играта приключи, програмата подканва играча да въведе името си, след което резултатът му бива записан, чрез статичния метод на класа Results – AddResult. Ако играчът не въведе своето име, играта го записва като Anonymous.