**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**ОДЕССКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ ИМЕНИ И. И. МЕЧНИКОВА**

**ИНСТИТУТ МАТЕМАТИКИ, ЭКОНОМИКИ И МЕХАНИКИ**

Курсовая работа

по курсу "Разработка программного обеспечения"

Студента 2-го курса

Специальности прикладная математика

Никифорова Алексея

Преподаватель: Пенко В.Г.

Одесса 2015 г.

План работы:

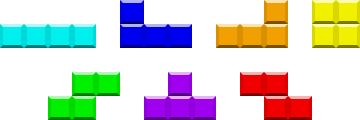
1. Постановка задачи
2. Рассмотрение модели решения.
3. Постановка ограничений решения задачи.
4. Написание библиотеки TetrisLibary. Рассмотрение ее основных положений.
5. Имплементация библиотеки улучшенной библиотеки в Windows Forms.
6. Пример работы библиотеки в WinForms and Unity.

\*Программы и их полный код можно будет найти на диске приложенном к курсовой работе.

Постановка задачи:

Написать графическое приложение – Тетрис. Исследовать модель его решения.

Случайные фигурки падают сверху в прямоугольный стакан шириной X и высотой Y клеток. В полёте игрок может поворачивать фигурку и двигать её по горизонтали. Также можно ускорять падение фигурки. Фигурка летит, пока не наткнётся на другую фигурку либо на дно стакана. Если при этом заполнился горизонтальный ряд из 10 клеток, он пропадает и всё, что выше него, опускается на одну клетку. Название игры происходит от количества клеток, из которых состоит каждая фигура. Игра заканчивается, когда новая фигурка не может поместиться в стакан. Игрок получает очки за каждый заполненный ряд, поэтому его задача — заполнять ряды, не заполняя сам стакан (по вертикали) как можно дольше, чтобы таким образом получить как можно больше очков.



Теоретические проблемы

Обычно игрок проигрывает из-за того, что не может справиться со слишком быстрым темпом игры, или потому, что данная реализация реагирует на клавиши слишком медленно по сравнению с ускоряющимся темпом падения фигурок, вследствие чего игрок уже не может в принципе приложить достаточное количество сдвигов к фигурке.

Была опубликована статья, автор которой доказывает, что даже если бы игрок реагировал мгновенно и всегда принимал правильные решения, то и в этом случае он бы в конечном счёте проиграл. Проблемой являются S- и Z-образные фигурки. Достаточно большое количество S-фигурок заставит игрока оставить дырку в правом нижнем углу. Достаточно большое количество Z-фигурок после этого заставит игрока оставить дырку в левом углу следующего ряда, не заполнив предыдущую дырку. Если после этого опять выпадет достаточно много S-фигурок, достаточно много Z-фигурок, и так много раз, заполнится (с дырками по краям) всё поле, и для следующей фигурки места не останется. Если генератор случайных чисел идеален и выдает дискретное равномерное распределение, любая (в том числе и такая) комбинация рано или поздно выпадет.

Однако среднее время, через которое выпадет такая комбинация, огромно и превышает время существования Вселенной. Тем не менее, не исключено, что существует какая-то другая, более трудная для доказательства причина, по которой идеальный игрок должен проиграть намного раньше указанной верхней границы.

Рассмотрение модели:

Для того чтобы задача была решена в полной мере необходимо написать библиотеку, которая реализуют все логику правил игры Тетриса. При этом на пути написания курсовой возникли два решения.

1) Библиотека как калькулятор, который будет высчитывать и передавать координаты.

2) Библиотека как оператор, который будет изменять объекты реализующие определенный интерфейс для работы с данной библиотекой.

Очевидно оба решения имеют право на существование, но при первом подходе работа с библиотекой превращается в громоздкий и не приятный процесс.   
Важно, что библиотека не должна быть привязана к графической среде, а лишь реализовать правила и структуры логики игры. Позже мы рассмотрим примеры реализации обеих подходов на Unity и WinForms.

Постановка ограничений модели:

При написании библиотеки TetrisLibary поддерживалась идея SimpleCalculation. Все действия над фигурами выполнялись не зависимо от ограничений. Под ограничениями понимаем таблицу возможных ходов на игровом поле. После чего сделанный ход проверялся на корректность. Если ход не корректен, то отменяем его. Данный алгоритм применим в силу того, что у каждого действия в Тетрисе есть обратное (которое приводит его в прежнее состояние).

Таким образом выбранная стратегия расчетов не считает координаты движения фигуры, а лишь отображает в случае корректного хода. Таким образом алгоритм не будет зависеть от вида управляемой фигуры, ее размера и формы. А значит данная библиотека будет применима не только лишь стандартным набором фигур, а и к произвольным(хотя на нем мы и ограничимся).

В нашей задаче фигуры будут вращаться по модели SimpleSpin. То есть при вращении фигур мы не рассчитываем препятствия на ее пути.

Следует заметить, что графическая реализация данной задачи не является основной частью работы. Так как вся логика задачи переносится в ее библиотечную часть. Все что нам остается сделать это имплементировать корректным образом написанную библиотеку.

Так же установим, что мы не будем рассматривать имплементацию библиотеки – калькулятор. Как уже было сказано данный подход не является удачным. К тому же основная логика будет отражена и в библиотеке – операторе.

Написание библиотеки Tetris Library – оператор.

Библиотека состоит из следующих классов:

SimpleCube, Part,PartDefinitions, ITetrisable, GameFiled. Опишем каждый из них. После чего составим UML таблицу.

Как известно фигурки в Тетрисе состоят и кубов. Которые описывают в полной мере положение фигуры на игровом поле.

Введем такой куб на уровне логики Tetris Library:

// не полный код.

**public class SimpleCube**

{

Координаты куба.

int positionX;

int positionY;

Является ли куб центральным.

bool isCenter;

Каждый куб содержит ссылки на соседние, что позволяет описывать фигуру как массив кубов содержащих ссылки друг на друга.

SimpleCube[] Conj = new SimpleCube[4];

По смыслу вращение на уровне кубов в фигуре представляет собой замену ссылок кубов в зависимости от управления.

public void Rotate(bool direction)

{

if (direction)

{

SimpleCube tmpCube = Conj[Conj.Length - 1];

Array.Copy(Conj, 0, Conj, 1, Conj.Length - 1);

Conj[0] = tmpCube;

}

else

{

SimpleCube tmpCube = Conj[0];

Array.Copy(Conj, 1, Conj, 0, Conj.Length - 1);

Conj[Conj.Length - 1] = tmpCube; …

После построения игрового куба перейдем к описанию фигуры.

Вообще говоря нам не важно какой объект будет рассматриваться так как каждая фигура содержит игровую логику в массиве SimpleCube, над ними будут проводится операции логики Тетриса. Таким образом можно реализовать контейнерный тип. И для того чтобы поддержать графическое ограничение запросить у него реализацию интерфейса отвечающего за графическую часть приложения.

Сам интерфейс выглядит следующим образом:

public interface ITetrisable

{

Отвечает за координаты объекта в графической части.

void sync(SimpleCube sc);

Отвечает за прорисовку объекта в графической часть.

void render(SimpleCube sc);

Создает объект в графической части.

ITetrisable create(SimpleCube sc);

Удаляет объект из графической части

void delete();

}

Очень часто изменяя координаты объекта наблюдаем и его перерисовку в графической части. В нашем случае с WinForms, так и случилось, методы sync и render совпадают по своему смыслу.

На основе интерфейса выше, класс отвечающий фигуре в тетрисе выглядит так:

// не полный код

**public class Part<T> where T : ITetrisable, new()**

{

Кубы отвечающие за логику поведения фигуры

protected SimpleCube[] cubes;

Координаты фигуры равны координатам ее центрального куба.

protected int positionX;

protected int positionY;

Ссылка на центральнй куб.

protected SimpleCube centralCube;

Свойство фигуры отвечающие за способность к вращению.

private bool isRotatable;

Массив отображаемых объектов

protected T[] modelObjects;

Видим появление параметра PardDefinition, который отвечает за вид фигур в Тетрисе.

public Part(int positionX, int positionY, PartDefinitions.Definition def = PartDefinitions.Definition.T\_Detail, bool isRotatable = true)

{

this.positionX = positionX;

this.positionY = positionY;

if (def == PartDefinitions.Definition.O\_Detail)

{

isRotatable = false;

}

else

{

this.isRotatable = isRotatable;

}

this.cubes = PartDefinitions.GetPartDefinition(def);

modelObjects = new T[cubes.Length];

findCentralCube();

SetPos(positionX, positionY);

for (int i = 0; i < cubes.Length; i++)

{

После получения шаблона приводим в соответствие координаты.

modelObjects[i] = (T)new T().create(cubes[i]);

}

}

Вспомогательный метод для синхронизации фигуры в целом, а не отдельных ее частей.

public void sync()

{

for (int i = 0; i < cubes.Length; i++)

{

modelObjects[i].sync(cubes[i]);

}

}

Удаляем фигуру по элементно в ее графическом представлении

public void delete()

{

for (int i = 0; i < cubes.Length; i++)

{

modelObjects[i].delete();

}

}

Ищем центральный куб со свойством isCentral.

protected void findCentralCube()

{

int i = 0;

if (cubes != null)

while (!cubes[i++].isCentral) { }

centralCube = cubes[--i];

}

Устанавливаем фигуру, а значит и ее центральный куб в указанную позицию.

public void SetPos(int X, int Y)

{

centralCube.SetPos(X, Y);

this.positionX = X;

this.positionY = Y;

UpdatePos();

}

Аналогично строим для других простейших операций над фигурой.

public void MoveDown()

{

this.SetPos(positionX, positionY + 1);

}

Основная внутренняя операция для фигуры. В виду построения данной структуры мы можем рекурсивно обновить состояния фигуры полагая, что фигуры связаны однозначно.

В силу построения массива SimpleCube в PartDefinitions каждая фигура представляется как ацикличный граф. Поэтому для корректных входных массивов данный метод работает исправно.

private void UpdatePos()

{

UpdatePos(centralCube);

}

private void UpdatePos(SimpleCube SimCubeInstance)

{

int currPosX = SimCubeInstance.getPosX;

int currPosY = SimCubeInstance.getPosY;

SimpleCube[] ArrConj = SimCubeInstance.getConj;

for (int i = 0; i < 4; i++)

{

if (ArrConj[i] != null)

{

switch (i)

{

case 0:

ArrConj[0].SetPos(currPosX - 1, currPosY);

break;

case 1:

ArrConj[1].SetPos(currPosX, currPosY - 1);

break;

case 2:

ArrConj[2].SetPos(currPosX + 1, currPosY);

break;

case 3:

ArrConj[3].SetPos(currPosX, currPosY + 1);

break;

}

UpdatePos(ArrConj[i]);

}

}

}

}

Посмотрим как выглядит класс PartDefinitions:

**public static class PartDefinitions**

{

public static Random r = new Random();

public enum Definition { O\_Detail, I\_Detail, J\_Detail, L\_Detail, S\_Detail, Z\_Detail, T\_Detail };

public static SimpleCube[] GetPartDefinition(Definition name)

{

SimpleCube[] TetraPart = new SimpleCube[4];

for (int i = 1; i < 4; i++)

{

TetraPart[i] = new SimpleCube();

}

TetraPart[0] = new SimpleCube(0, 0);

switch (name)

{

case Definition.O\_Detail:

TetraPart[0].SetConj(null, null, TetraPart[1]);

TetraPart[1].SetConj(null, null, null, TetraPart[2]);

TetraPart[2].SetConj(TetraPart[3]);

break;

…

}

return TetraPart;

}

public static Definition GerDefinedRandomPart()

{

int range = Enum.GetValues(typeof(Definition)).Length;

int ind = r.Next(range);

return (Definition)ind;

}

}

}

После построения логики фигуры осталось описать логику игры в тетрис. Наш Controller выглядит следующим образом:

**public class GameField<T> where T : ITetrisable, new()**

{

Высота и ширина игрового поля измеряемая в клетках.

protected int height;

protected int width;

Количество набранных очков.

protected int score;

Точка создания фигур.

protected int spawnPointX;

protected int spawnPointY;

Количество очков начисляемых за сложение линии.

private static int lineScore = 100;

protected SimpleCube[,] GameBoard;

private T[,] GameBoardObj;

Имеем прямоугольные массивы размерности width x height для отображение логики и графической части.

Простейшее игровое событие, передаваемая фигура движется вниз на одну позицию, в зависимости от логической ситуации возвращаем true or false.

public bool playGame(ref Part<T> part)

{

if(part==null)

{ // в случае если фигура не инициализирована.

part = part = new Part<T>(

PartDefinitions.GerDefinedRandomPart());

spawnPart(part);

}

else

{

if (!MoveDown(part))

{

// проверяем была ли заполнена по крайней мере одна линия.

scoreCondition();

part = new Part<T>(

PartDefinitions.GerDefinedRandomPart());

spawnPart(part);

// проверка на условие завершения игры

if (loseCondition(part))

return false;

}

}

return true;

}

// Синхронизация логической таблицы и таблицы объектов графического интерфейса.

private void sync()

{

for (int i = 0; i < GameBoard.GetLength(0); i++)

{

for (int j = 0; j < GameBoard.GetLength(1); j++)

{

if (GameBoard[i, j] != null && GameBoardObj == null)

{

GameBoardObj[i, j] = (T)new T().create(GameBoard[i, j]);

}

else if (GameBoard[i, j] == null && GameBoardObj[i, j] != null)

{

GameBoardObj[i, j].delete();

GameBoardObj[i, j] = default(T);

}

}

}

}

// return true if player scored a line

private bool scoreCondition()

{

int multiScore = 0;

bool scored;

bool[] scoredLine = new bool[height];

for (int i = 0; i < height; i++)

{

//проверяем была ли заполнена линия целиком.

scored = true;

for (int j = 0; j < width; j++)

{

if (GameBoard[j, i] == null)

{

scored = false;

break;

}

}

Далее нужно обработать сдвиг и multiscorecondition

if (scored)

{//multi line bonus score

multiScore++;

scoredLine[i] = true;

}

}

// ввели булевский массив для контроля сдвигов на разных уровнях, именно в зависимости от его //значения определяется необходимый сдвиг.

int downOffset = 0;

if (multiScore > 0)

for (int i = height - 1; i > 0; i--)

{

if (scoredLine[i])

{

downOffset++;

continue;

}

if (downOffset > 0)

{

for (int j = 0; j < width; j++)

{// после расчета всех линий проводим сдвиг элементов в логической части Controller’a

GameBoard[j, i + downOffset] = GameBoard[j, i];

GameBoard[j, i] = null;

}

}

}

//end:

score += multiScore \* lineScore;

if (multiScore > 0)

{

// проводим синхронизацию

sync();

return true;

}

return false;

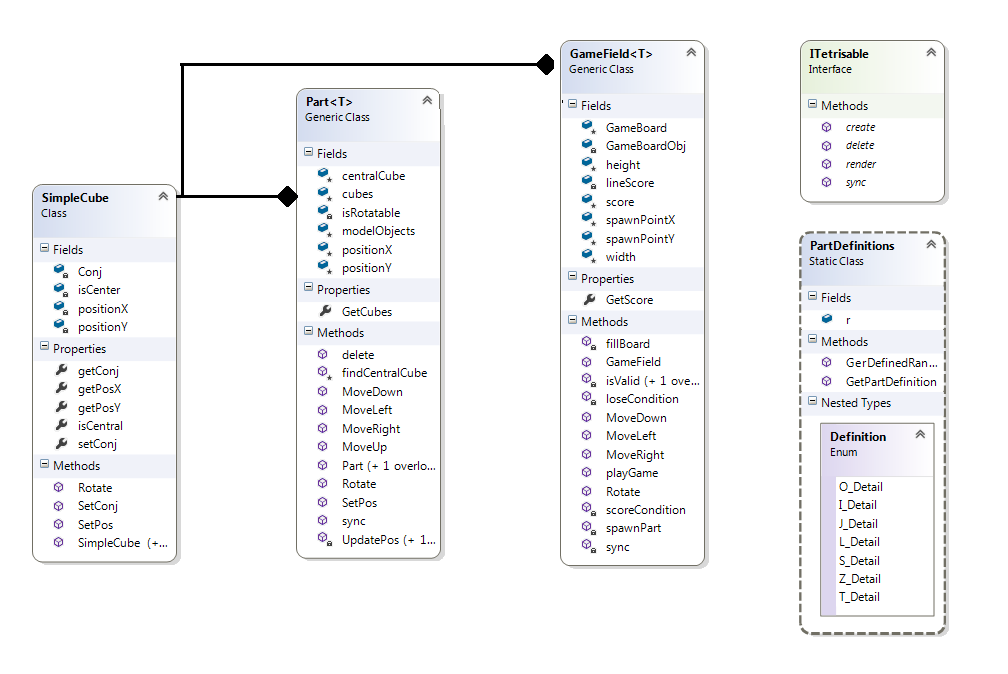
}

#region Actions

// wrappers for valid moves Далее в коде идут реализации простейших операций над фигурой на уровне допустимости хода с учетом игровой таблицы. Основная идея которых была описана выше. Поэтому опустим рассмотрение данной части.

}

Далее перейдем к UML таблице которая описывает взаимодействие и взаимоотношение объектов в рассматриваемой библиотеке.



Если опустить создание графической части интерфейса для корректного взаимодействия с пользователем, задача написания графического приложения Тетрис сводится к следующим шагам.

1. Реализация выше упомянутого интерфейса ITetrisable с корректной реализацией его методов.
2. Написание простого управляющего шаблона.

В качестве графического отображения напишем класс SimpleCubePicBox, он наследует графический элемент PictureBox и имплементирует ITetrisable.

public void sync(SimpleCube sc)

{

render(sc);

}

public void render(SimpleCube sc)

{

this.Left = sc.getPosX \* boxWidth;

this.Top = sc.getPosY \* boxHeight+fieldOffset;

if(!isShowed)

{

gameBoardstandard.Controls.Add(this);

isShowed = true;

}

}

public void delete()

{

this.Dispose();

gameBoardstandard.Controls.Remove(this);

}

public ITetrisable create(SimpleCube sc)

{

this.render(sc);

return this;

}

После чего в специальной игровой форме определим следующие поля:

private GameField<SimpleCubePicBox> gameTetris;

private Part<SimpleCubePicBox> currPart

private Timer gameSpeedTimer;

Тогда написанную нами библиотеку можно использовать следующим образом:

void gameSpeedTimer\_Tick(object sender, EventArgs e)

{

Если игра закончена настраиваем отображение в форме.

if (!gameTetris.playGame(ref currPart))

{

GameInfo.IsInGame = false;

gameSpeedTimer.Stop();

ladderBoard.AddValid(new KeyValuePair<string, int>("Foo", gameTetris.GetScore));

MessageBox.Show("You lost :(");

}

else

{

scoreLabel.Text = scoreLabel.Text.Substring(0, 7) + gameTetris.GetScore.ToString();

}

}

Обработчик нажатий:

void Form2\_KeyPress(object sender, KeyPressEventArgs e)

{

if (this.gameInfo.IsInGame)

{

if (e.KeyChar == 'a' || e.KeyChar == 'ф')

{

gameTetris.MoveLeft(currPart);

}

…

else if (e.KeyChar == 'q' || e.KeyChar == 'й')

{

gameTetris.Rotate(currPart, false);

}

else if (e.KeyChar == 's' || e.KeyChar == 'ы')

{

gameSpeedTimer\_Tick(gameSpeedTimer, e);

}

}

}

GameBoard

CurrPart

Form

INPUT

Tetris\_Event[Timer]

Схема выше описывает шаблон управления через библиотеку. Есть два источника воздействия на текущую управляемую деталь. Деталь и действие над ней передается в экземпляр класса GameField текущей сессии. В свою очередь GameFiled влияет на состояние (в данном случае еще и графическое) формы.

Таким образом мы построили класс который позволяет управлять игровой логикой построив лишь простой шаблон с делегирование текущей рассматриваемой игровой фигурой в игровой сессии.

Стоит заметить, что не всегда есть необходимость использовать SimpleCube со всей его функциональностью, поэтому ресурсные затраты в данном решении относительно велики.