

Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

Создание игры «SameGame» с помощью средств MFC и STL C++.

Выполнил
студент гр. в3530904/00030

В.С. Баганов

Руководитель
доцент.

С.К. Круглов

«_____» _____ 202__ г.

Санкт-Петербург
2023

Содержание

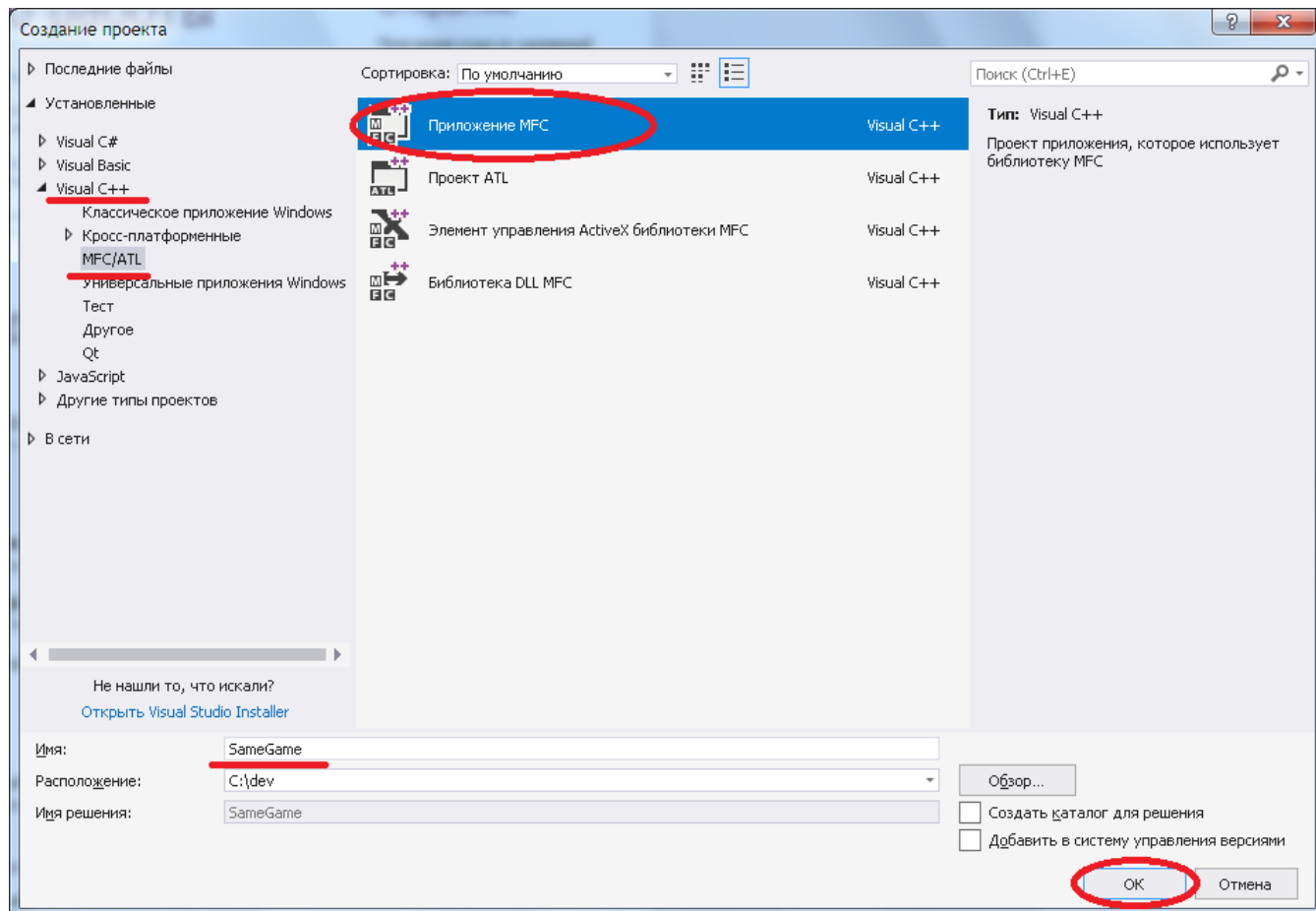
1. Правила игры «SameGame»	3
2. Создание проекта MFC в среде Visual Studio	4
3. Архитектура и хранение данных в игре «SameGame» на C++/MFC	8
4. Отрисовка игры «SameGame» на C++/MFC	13
5. Обработка событий в игре «SameGame» на C++/MFC	19
6. Работа с алгоритмом в игре «SameGame» на C++/MFC	25
7. Работа с меню в игре «SameGame» на C++/MFC	31
8. Добавление уровней сложности в игре «SameGame» на C++/MFC	33
9. Реализация функционала «Отмена/Повтор» игры «SameGame»	39
10. Заключение	49

1. Правила игры «SameGame»

Задача игрока удалить все цветные блоки с игрового поля. Для того чтобы удалить блок, игрок должен кликнуть по любому блоку, который стоит рядом (вертикально или горизонтально) с другим блоком того же цвета. Таким образом уничтожится целая цепочка блоков одного цвета. Блоки, которые при этом находились сверху, упадут вниз, заполняя освободившееся пространство. При удалении всего столбца, стоящие справа столбцы сдвигаются влево, занимая пустое место. Игра заканчивается тогда, когда у игрока не остается больше возможных ходов. Цель игры состоит в том, чтобы как можно быстрее очистить доску от цветных блоков

2. Создание проекта MFC в среде Visual Studio

Создавать игру будем в Microsoft Visual Studio 2017. Все инструкции и описания, приведенные ниже, могут быть адаптированы и для других версий Visual Studio. Ниже скрины для настройки проекта MFC.



Выбирая опцию "Один документ" мы указываем нашему будущему приложению на необходимость использования архитектуры «Document/View». Следующий интересующий нас параметр — "Использование MFC", который содержит две опции. Выбор опции "Использовать MFC в общей библиотеке" подразумевает, что на компьютере конечного пользователя, который будет запускать наше приложение, уже имеются все необходимые для этого MFC-библиотеки. Если же мы выберем "Использовать MFC в статической библиотеке", то нужные MFC-библиотеки войдут в нашу программу на этапе компиляции, что в результате приведет к увеличению итогового размера нашей программы, но эта программа будет работать на любой машине с Windows.

Приложение MFC

Параметры типа приложения

Тип приложения

Свойства шаблона документа

Функции пользовательского интерфейса

Дополнительные функции

Созданные классы

Тип приложения:
Один документ

Параметры типа приложения:
☐ Документы с вкладками
☒ Поддержка архитектуры Document/View
☐ Проверки в жизненном цикле разработки защищенных приложений (SDL)

Параметры на основе диалогового окна:
<нет>

Поддержка составных документов:
<нет>

Параметры поддержки документов:
☐ Сервер активных документов
☐ Контейнер активных документов
☐ Поддержка составных файлов

Стиль проекта:
MFC standard

Визуальный стиль и цвета:
Windows Native/Default
☐ Разрешить смену визуального стиля

Язык ресурсов:
English (United States)

Использование MFC:
Использовать MFC в общей библиотеке

Назад

Далее

Готово

Отмена

Приложение MFC

Функции пользовательского интерфейса

Тип приложения

Свойства шаблона документа

Функции пользовательского интерфейса

Дополнительные функции

Созданные классы

Стили главного фрейма:
☐ Утолщенная граница фрейма
☒ Свернуть окно
☐ Развернуть окно
☐ Свернуто
☐ Развернуто
☒ Системное меню
☐ О программе
☐ Начальная строка состояния
☐ Разделенное окно

Стили дочерних фреймов:
☐ Кнопка свертывания дочернего окна
☐ Child maximize box
☐ Дочернее окно развернуто

Command bar (menu/toolbar/ribbon):
Использовать классическое меню
Classic menu options:
<нет>
Menu bar and toolbar options:
☐ Использовать пользовательские панели инструментов и изображения
☐ Настраиваемое поведение меню

Заголовок диалогового окна:
SameGame

Назад

Далее

Готово

Отмена

Приложение MFC

Параметры дополнительных функций

Тип приложения

Свойства шаблона документа

Функции пользовательского интерфейса

Дополнительные функции

Созданные классы

Дополнительные возможности:

☐ Печать и предварительный просмотр

☐ Автоматизация

☐ Элементы управления ActiveX

☐ Интерфейс MAPI (API сообщений)

☐ Сокеты Windows

☐ Active Accessibility

☒ Манифест стандартных элементов управления

☐ Поддержка диспетчера перезагрузки

☐ Открывать ранее открытые документы

☐ Поддержка восстановления приложения

Дополнительные области фреймов:

☐ Закрепленная область обозревателя

☐ Закрепленная область вывода

☐ Закрепленная область свойств

☐ Область навигации

☐ Заголовок окна

☐ Пункты расширенного меню области фрейма показывают и активируют области

Число файлов в текущем списке файлов:

4

Назад

Далее

Готово

Отмена

Приложение MFC

Параметры созданных классов

Тип приложения

Свойства шаблона документа

Функции пользовательского интерфейса

Дополнительные функции

Созданные классы

Созданные классы:

View

App

MainFrame

View

Doc

Базовый класс:

CView

Файл заголовка:

SameGameView.h

CPP-файл:

SameGameView.cpp

Приложение MFC

Параметры созданных классов

Тип приложения

Свойства шаблона документа

Функции пользовательского интерфейса

Дополнительные функции

Созданные классы

Созданные классы:

View

Имя класса:

CSameGameView

Базовый класс:

CView

CEditView

CFormView

CHtmlEditView

CHtmlView

CListView

CRichEditView

CScrollView

CTreeView

CView

Файл заголовка:

SameGameView.h

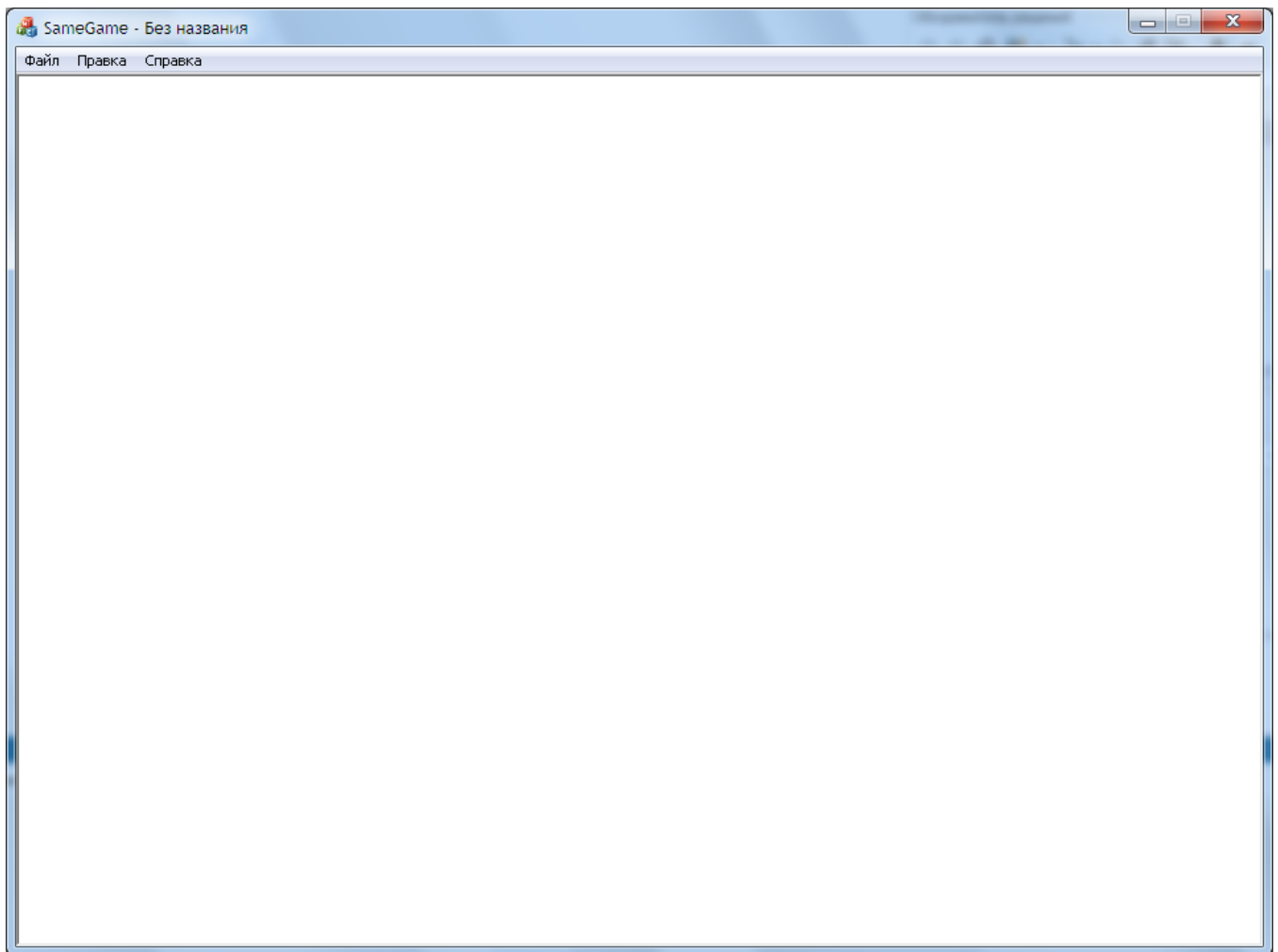
CPP-файл:

SameGameView.cpp

В результате будут автоматически сгенерированы 4 класса, которые станут основными классами в нашей игре. Класс CSameGameApp, который является классом-обёрткой для

всего нашего приложения и функции `main()` в том числе. Базовым для него всегда будет класс `CWinApp`.

В классе `CSameGameDoc`, в котором будут храниться все данные нашего приложения. Для него базовым будет класс `CDocument`. Класс `CMainFrame`, базовым для которого будет `CFrameWnd`. Он также является классом-обёрткой для окна нашей программы. Класс основного фрейма содержит меню и представление клиентской области. Клиентская область — это место, где отрисовывается наша игра. Класс `CSameGameView` — базовый класс, который представляет собой раскрывающийся список с набором общедоступных представлений, каждое из которых имеет свои особенности в использовании и применении. Тип представления по умолчанию — `CView`, который является общим представлением, где все отображения и взаимодействия с пользователем должны выполняться вручную.



Завершение работы мастера приложений MFC приведет к созданию и запуску приложения MFC. Ниже приведен скриншот того, как будет выглядеть базовое приложение (чтобы скомпилировать программу, нужно перейти в меню Visual Studio «Отладка» > «Запуск без отладки»):

3. Архитектура и хранение данных в игре «SameGame» на C++/MFC

Сначала мы создаем класс, представляющий игровую доску — CSameGameBoard. Вот код для заголовочного файла SameGameBoard.h:

```
1  #pragma once
2
3  class CSameGameBoard
4  {
5  public:
6      // Конструктор по умолчанию
7      CSameGameBoard(void);
8
9      // Деструктор
10     ~CSameGameBoard(void);
11
12     // Функция для случайной расстановки блоков в начале игры
13     void SetupBoard(void);
14
15     // Получаем цвет в определенном участке игрового поля
16     COLORREF GetBoardSpace(int row, int col);
17
18     // Геттеры для получения информации о параметрах игрового поля
19     int GetWidth(void) const { return m_nWidth; }
20     int GetHeight(void) const { return m_nHeight; }
21     int GetColumns(void) const { return m_nColumns; }
22     int GetRows(void) const { return m_nRows; }
23
24     // Функция для удаления игрового поля и освобождения памяти
25     void DeleteBoard(void);
26     private:
27     // Функция для создания игрового поля и выделения памяти под него
28     void CreateBoard(void);
29
30     // Указатель на двумерный массив
31     int** m_arrBoard;
32
33     // Список цветов: 0 — это цвет фона, 1-3 — это цвета блоков
34     COLORREF m_arrColors[4];
35
36     // Информация о размере игрового поля
37     int m_nColumns;
38     int m_nRows;
39     int m_nHeight;
40     int m_nWidth;
41 };
```

Этот класс содержит указатель `m_arrBoard` на двумерный массив целых чисел, которые представляют один из трех цветов (элементы с индексами 1-3), либо цвет фона (элемент с индексом 0). Затем мы добавляем переменные-члены, чтобы отслеживать строки (`m_nRows`), столбцы (`m_nColumns`), высоту (`m_nHeight`) и ширину (`m_nWidth`) в пикселях. Здесь также присутствуют функции для создания, настройки и удаления доски. Метод `CreateBoard()` нужен для выделения памяти под двумерный массив, в котором будет храниться игровое поле, и для инициализации всех блоков значением фонового

цвета. Метод SetupBoard() сбрасывает игровое поле путем случайного выбора цвета для каждого элемента поля на доске. Метод DeleteBoard() освобождает память, которую мы используем для игровой доски (в противном случае, это может привести к утечкам памяти).

```
1  #include "stdafx.h" // в более новых версиях Visual Studio эта строка не
   ↳ нужна
2  #include "SameGameBoard.h"
3
4  CSameGameBoard::CSameGameBoard(void)
5      : m_arrBoard(NULL),
6        m_nColumns(15), m_nRows(15),
7        m_nHeight(35), m_nWidth(35)
8  {
9      m_arrColors[0] = RGB(0, 0, 0);
10     m_arrColors[1] = RGB(255, 0, 0);
11     m_arrColors[2] = RGB(255, 255, 64);
12     m_arrColors[3] = RGB(0, 0, 255);
13 }
14
15 CSameGameBoard::~CSameGameBoard(void)
16 {
17     // Просто удаляем нашу доску
18     DeleteBoard();
19 }
20
21 void CSameGameBoard::SetupBoard(void)
22 {
23     // При необходимости создаем доску
24     if (m_arrBoard == NULL)
25         CreateBoard();
26
27     // Устанавливаем каждому блоку случайный цвет
28     for (int row = 0; row < m_nRows; row++)
29         for (int col = 0; col < m_nColumns; col++)
30             m_arrBoard[row][col] = (rand() % 3) + 1;
31 }
32
33 COLORREF CSameGameBoard::GetBoardSpace(int row, int col)
34 {
35     // Проверяем границы массива
36     if (row < 0 || row ≥ m_nRows || col < 0 || col ≥ m_nColumns)
37         return m_arrColors[0];
38     return m_arrColors[m_arrBoard[row][col]];
39 }
40
41 void CSameGameBoard::DeleteBoard(void)
42 {
43     if (m_arrBoard ≠ NULL)
44     {
45         for (int row = 0; row < m_nRows; row++)
46         {
47             if (m_arrBoard[row] ≠ NULL)
48             {
49                 // Сначала удаляем каждую отдельную
50                 ↳ строку
51                 delete[] m_arrBoard[row];
52                 m_arrBoard[row] = NULL;
53             }
54         }
55     }
```

```

53         }
54         // В конце удаляем массив, содержащий строки
55         delete[] m_arrBoard;
56         m_arrBoard = NULL;
57     }
58 }
59
60 void CSameGameBoard::CreateBoard(void)
61 {
62     // Если у нас осталась доска с предыдущего раза, то удаляем её
63     if(m_arrBoard != NULL)
64         DeleteBoard();
65
66     // Создаем массив для хранения строк
67     m_arrBoard = new int*[m_nRows];
68
69     // Создаем отдельно каждую строку
70     for (int row = 0; row < m_nRows; row++)
71     {
72         m_arrBoard[row] = new int[m_nColumns];
73
74         // Устанавливаем для каждого блока значение цвета, равное
75         //   →   цвету фона
76         for (int col = 0; col < m_nColumns; col++)
77             m_arrBoard[row][col] = 0;
78     }
79 }

```

В классе, представляющем игровую доску, также присутствует массив `m_arrColors[]` элементов типа `COLORREF`. Тип `COLORREF` — это 32-битный целочисленный тип `unsigned`, который содержит значение цвета блока в формате RGBA. Элемент массива `m_arrColors[0]` хранит фоновый цвет игровой доски, а элементы `m_arrColors[1]`, `m_arrColors[2]` и `m_arrColors[3]` — цвет блоков. В конструкторе, приведенном ниже, мы будем использовать макрос `RGB` для создания значения `COLORREF` из трех целых чисел, представляющих собой числовое обозначение красного, зеленого и синего цветов. Ниже приведена реализация класса `CSameGameBoard` в `SameGameBoard.cpp`:

Игровая доска инкапсулирована в объект, можем создать экземпляр этого объекта в классе документа. Класс документа содержит все игровые данные и отделен от кода View. Вот заголовочный файл `SameGameDoc.h`:

```

1  #pragma once
2
3  #include "SameGameBoard.h"
4
5  class CSameGameDoc : public CDocument
6  {
7  protected: // создаем только из сериализации
8      CSameGameDoc();
9      virtual ~CSameGameDoc();
10     DECLARE_DYNCREATE(CSameGameDoc)
11
12     // Атрибуты
13     public:
14
15     // Операции
16     public:

```

```

17      // Геттеры для получения информации о параметрах игрового поля
18      COLORREF GetBoardSpace(int row, int col)
19      {
20          return m_board.GetBoardSpace(row, col);
21      }
22      void SetupBoard(void) { m_board.SetupBoard(); }
23      int GetWidth(void) { return m_board.GetWidth(); }
24      int GetHeight(void) { return m_board.GetHeight(); }
25      int GetColumns(void) { return m_board.GetColumns(); }
26      int GetRows(void) { return m_board.GetRows(); }
27      void DeleteBoard(void) { m_board.DeleteBoard(); }
28
29
30
31      // Переопределения
32      public:
33          virtual BOOL OnNewDocument();
34
35      protected:
36          // Экземпляр объекта нашей игровой доски
37          CSameGameBoard m_board;
38
39
40      // Генерация функции сообщений
41      protected:
42          DECLARE_MESSAGE_MAP()
43      };

```

Document,, является простой обёрткой для нашего класса. Добавили экземпляр нашего класса и семь функций, которые вызывают аналогичные функции класса SameGameBoard. Благодаря этому View сможет получить доступ к данным игровой доски, хранящимся в Document. Исходный файл для Document (SameGameDoc.cpp):

```

1  #include "stdafx.h" // в более новых версиях Visual Studio эта строка не
   ↪ нужна
2  #include "SameGame.h"
3
4  #include "SameGameDoc.h"
5
6  #ifdef _DEBUG
7  #define new DEBUG_NEW
8  #endif
9
10 // CSameGameDoc
11 IMPLEMENT_DYNCREATE(CSameGameDoc, CDocument)
12 BEGIN_MESSAGE_MAP(CSameGameDoc, CDocument)
13 END_MESSAGE_MAP()
14
15 // Создание CSameGameDoc
16 CSameGameDoc::CSameGameDoc()
17 {
18 }
19
20 // Уничтожение CSameGameDoc
21 CSameGameDoc::~CSameGameDoc()
22 {
23 }
24

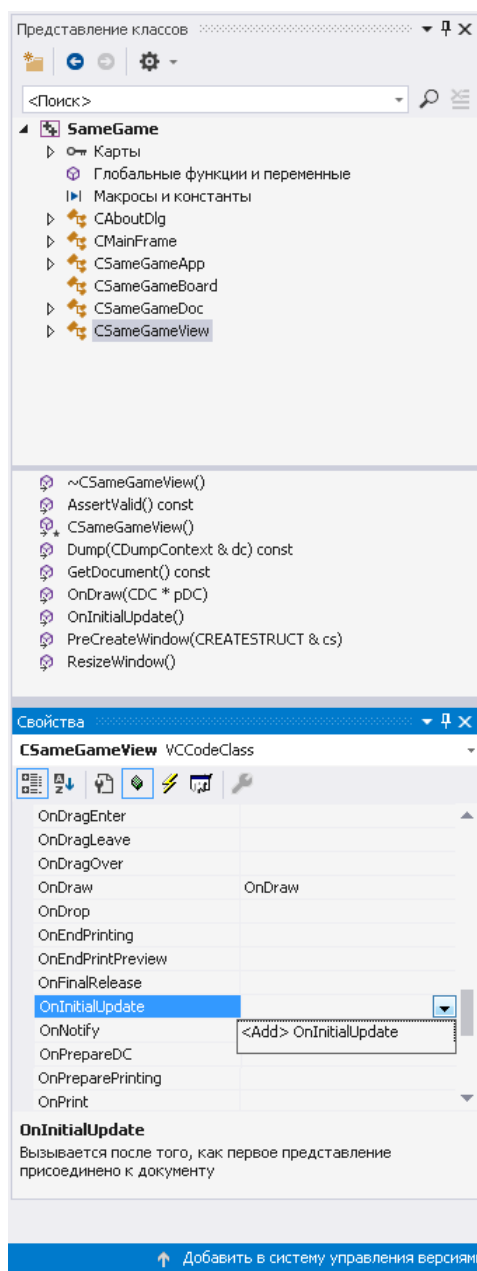
```

```
25  BOOL CSameGameDoc::OnNewDocument()  
26  {  
27      if (!CDocument::OnNewDocument())  
28          return FALSE;  
29  
30      // Установка (или сброс) параметров доски  
31      m_board.SetupBoard();  
32  
33      return TRUE;  
34  }
```

Добавили вызов функции SetupBoard() в обработчике OnNewDocument. Это позволяет пользователю начать новую игру с помощью нажатия Ctrl+N или из меню File > New

4. Отрисовка игры «SameGame» на C++/MFC

Добавляем код для изменения параметров окна до нужного размера. Сейчас окно имеет размер, заданный по умолчанию. Исправим это в переопределяемом методе `OnInitialUpdate()`. Класс `View` наследует базовый метод `OnInitialUpdate()`, который задает представление документа, и мы должны переопределить этот метод, чтобы получить возможность изменять размеры окна. Для того чтобы это реализовать, нам нужно открыть окно свойств заголовочного файла `CSameGameView` (который фактически будет называться `SameGameView.h`): для этого нужно сделать `Alt+Enter` или в меню "Вид" > "Окно свойств"). Найти опцию `OnInitialUpdate` и выбрать `<Add> OnInitialUpdate`, как показано на скриншоте ниже:



Добавляем переопределенный метод `OnInitialUpdate()` к нашему View с небольшим содержанием по умолчанию для вызова функции `ResizeWindow()`. Таким образом, заголовочный файл `SameGameView.h` будет иметь следующий вид:

```

1  #pragma once
2
3  class CSameGameView : public CView
4  {
5  protected:
6      CSameGameView();
7      DECLARE_DYNCREATE(CSameGameView)
8
9      // Атрибуты
10     public:
11         CSameGameDoc* GetDocument() const;
12
13     // Переопределения
14     public:
15         virtual void OnDraw(CDC* pDC); // переопределяем, чтобы
16         ↪ нарисовать этот View
17         virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
18     protected:
19
20     // Реализация
21     public:
22
23         void ResizeWindow();
24
25         virtual ~CSameGameView();
26 #ifdef _DEBUG
27         virtual void AssertValid() const;
28         virtual void Dump(CDumpContext& dc) const;
29 #endif
30
31     // Генерируем функцию сообщений
32     protected:
33         DECLARE_MESSAGE_MAP()
34     public:
35         virtual void OnInitialUpdate();
36     };
37
38 #ifndef _DEBUG // версия debug в SameGameView.cpp
39 inline CSameGameDoc* CSameGameView::GetDocument() const
40 {
41     return reinterpret_cast<CSameGameDoc*>(m_pDocument);
42 }
43 #endif

```

добавляем код отрисовки в класс CSameGameView. Заголовочные и исходные файлы для View уже содержат переопределение функции OnDraw(). Ниже приведен полный исходный код для SameGameView.cpp:

```

1  #include "stdafx.h"
2  #include "SameGame.h"
3
4  #include "SameGameDoc.h"
5  #include "SameGameView.h"
6
7  #ifdef _DEBUG
8  #define new DEBUG_NEW

```

```

9  #endif
10
11  // CSameGameView
12  IMPLEMENT_DYNCREATE(CSameGameView, CView)
13  BEGIN_MESSAGE_MAP(CSameGameView, CView)
14  END_MESSAGE_MAP()
15
16  // Конструктор CSameGameView
17  CSameGameView::CSameGameView()
18  {
19  }
20
21  // Деструктор CSameGameView
22  CSameGameView::~~CSameGameView()
23  {
24  }
25
26  BOOL CSameGameView::PreCreateWindow(CREATESTRUCT& cs)
27  {
28      return CView::PreCreateWindow(cs);
29  }
30
31  // Отрисовка игры
32  void CSameGameView::OnDraw(CDC* pDC) // MFC закомментирует имя аргумента
    ↪ по умолчанию. Раскомментируйте это
33  {
34      // В начале создаем указатель на Document
35      CSameGameDoc* pDoc = GetDocument();
36      ASSERT_VALID(pDoc);
37      if (!pDoc)
38          return;
39
40      // Сохраняем текущее состояние контекста устройства
41      int nDCSave = pDC->SaveDC();
42
43      // Получаем размеры клиентской области
44      CRect rcClient;
45      GetClientRect(&rcClient);
46      COLORREF clr = pDoc->GetBoardSpace(-1, -1);
47
48      // Сначала отрисовываем фон
49      pDC->FillSolidRect(&rcClient, clr);
50
51      // Создаем кисть для рисования
52      CBrush br;
53      br.CreateStockObject(HOLLOW_BRUSH);
54      CBrush* pbrOld = pDC->SelectObject(&br);
55
56      // Рисуем блоки
57      for (int row = 0; row < pDoc->GetRows(); row++)
58      {
59          for (int col = 0; col < pDoc->GetColumns(); col++)
60          {
61
62              clr = pDoc->GetBoardSpace(row, col);
63
64              // Вычисляем размер и позицию игрового
        ↪ пространства
65              CRect rcBlock;
66              rcBlock.top = row * pDoc->GetHeight();

```

```

67         rcBlock.left = col * pDoc→GetWidth();
68         rcBlock.right = rcBlock.left + pDoc→GetWidth();
69         rcBlock.bottom = rcBlock.top + pDoc→GetHeight();
70
71         // Заполняем блок соответствующим цветом
72         pDC→FillSolidRect(&rcBlock, clr);
73
74         // Рисуем контур
75         pDC→Rectangle(&rcBlock);
76     }
77 }
78 // Восстанавливаем контекст устройства
79 pDC→RestoreDC(nDCSave);
80 br.DeleteObject();
81 }
82
83 // Диагностика CSameGameView
84 #ifdef _DEBUG
85 void CSameGameView::AssertValid() const
86 {
87     CView::AssertValid();
88 }
89
90 void CSameGameView::Dump(CDumpContext& dc) const
91 {
92     CView::Dump(dc);
93 }
94
95 // Версия non-debug
96 CSameGameDoc* CSameGameView::GetDocument() const
97 {
98     ASSERT(m_pDocument→IsKindOf(RUNTIME_CLASS(CSameGameDoc)));
99     return (CSameGameDoc*)m_pDocument;
100 }

```

Чтобы нарисовать игровую доску будем перебирать каждую строку столбец за столбцом и рисовать цветной прямоугольник. У функции OnDraw() есть один аргумент — указатель на CDC. Класс CDC является базовым классом для всех контекстов устройства. Контекст устройства — это обобщённый интерфейс устройства вывода, такого как экран или принтер. Вначале инициализируем указатель на Document, чтобы иметь возможность получить данные игрового поля. Далее вызываем функцию SaveDC() из контекста устройства. Эта функция сохраняет состояние контекста устройства, чтобы мы могли восстановить его после того, как закончим.

```

1 // Получаем клиентский прямоугольник
2 CRect rcClient;
3 GetClientRect(&rcClient);
4
5 // Получаем фоновый цвет доски
6 COLORREF clr = pDoc→GetBoardSpace(-1, -1);
7
8 // Сначала рисуем фон
9 pDC→FillSolidRect(&rcClient, clr);

```

Покрасим фон клиентской области в черный цвет. Нужно получить размеры клиентской

области — вызываем `GetClientRect()`. Вызов `GetBoardSpace(-1,-1)` в `Document` возвратит цвет фона, а `FillSolidRect()` заполнит клиентскую область фоновым цветом.

```
1 // Создаём кисть для рисования
2 CBrush br;
3 br.CreateStockObject(HOLLOW_BRUSH);
4 CBrush* pbrOld = pDC->SelectObject(&br);
5
6 ...
7 // Восстановление настроек контекста устройства
8 pDC->RestoreDC(nDCSave);
9 br.DeleteObject();
```

Рисуем отдельные прямоугольники. Сначала нарисуем цветной прямоугольник, а затем обведем его черным контуром. Нужно создать объект кисти, чтобы сделать контур. Кисть `HOLLOW_BRUSH`, которую создаем, называется `hollow` (в переводе «пустой»), потому что, когда мы рисуем прямоугольник, MFC захочет заполнить его внутренность каким-нибудь цветом. Поэтому будем использовать `HOLLOW_BRUSH`. Создание кисти приводит к выделению GDI-памяти, которую позднее нужно будет очистить.

```
1 // Рисуем квадраты
2 for (int row = 0; row < pDoc->GetRows(); row++)
3 {
4     for (int col = 0; col < pDoc->GetColumns(); col++)
5     {
6         // Получаем цвет для пространства доски
7         clr = pDoc->GetBoardSpace(row, col);
8
9         // Рассчитываем размер и положение пространства доски
10        CRect rcBlock;
11        rcBlock.top = row * pDoc->GetHeight();
12        rcBlock.left = col * pDoc->GetWidth();
13        rcBlock.right = rcBlock.left + pDoc->GetWidth();
14        rcBlock.bottom = rcBlock.top + pDoc->GetHeight();
15
16        // Заполняем блок правильным цветом
17        pDC->FillSolidRect(&rcBlock, clr);
18
19        // Рисуем контур блока
20        pDC->Rectangle(&rcBlock);
21    }
22 }
```

Вложенные циклы `for` перебирают строку за строкой, столбец за столбцом, получая цвет соответствующего пространства доски из `Document` с помощью функции `GetBoardSpace()`, вычисляя размер прямоугольника, который нужно закрасить, а затем выполняется сам процесс закрашивания блока. При отрисовке используются два метода:

метод `FillSolidRect()` — для заполнения цветной части блока;

метод `Rectangle()` — для рисования контура блока.

Последняя функция, которую мы вставили во `View`, изменяет размер окна в зависимости от размера игрового поля. На следующих уроках мы добавим дополнительный функционал, чтобы пользователь мог изменять количество и размер блоков. Получаем указатель на `Document`, а затем получаем размер текущей клиентской области и текущего окна.

```

1 // Получаем размер клиентской области и окна
2 CRect rcClient, rcWindow;
3 GetClientRect(&rcClient);
4 GetParentFrame()→GetWindowRect(&rcWindow);

```

```

1 // Вычисляем разницу
2 int nWidthDiff = rcWindow.Width() - rcClient.Width();
3 int nHeightDiff = rcWindow.Height() - rcClient.Height();
4
5 // Изменяем размер окна в соответствии с размером нашей доски
6 rcWindow.right = rcWindow.left +
7 pDoc→GetWidth() * pDoc→GetColumns() + nWidthDiff;
8 rcWindow.bottom = rcWindow.top +
9 pDoc→GetHeight() * pDoc→GetRows() + nHeightDiff;

```

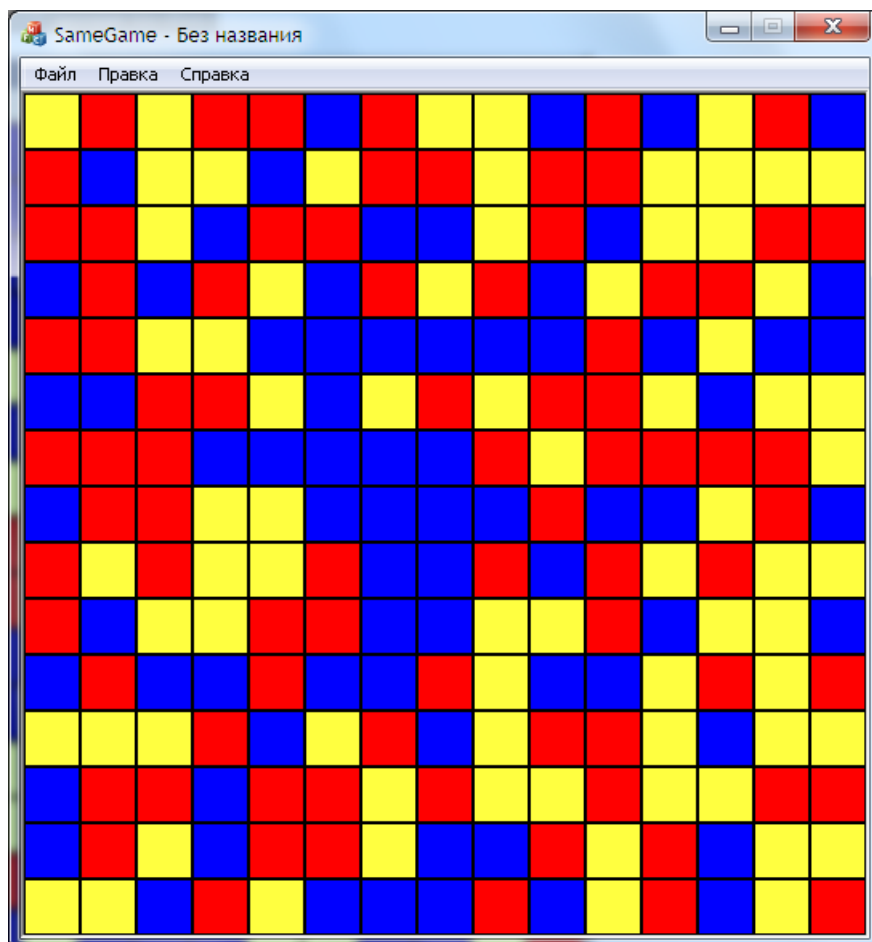
Функция `GetParentFrame()` возвращает указатель на класс `CMainFrame`, который является фактическим окном нашей игры, и мы изменяем размер окна, вызывая `MoveWindow()`.

```

1 GetParentFrame()→MoveWindow(&rcWindow);

```

В результате получаем такой вывод приложения:



5. Обработка событий в игре «SameGame» на C++/MFC

Событийно-ориентированное программирование

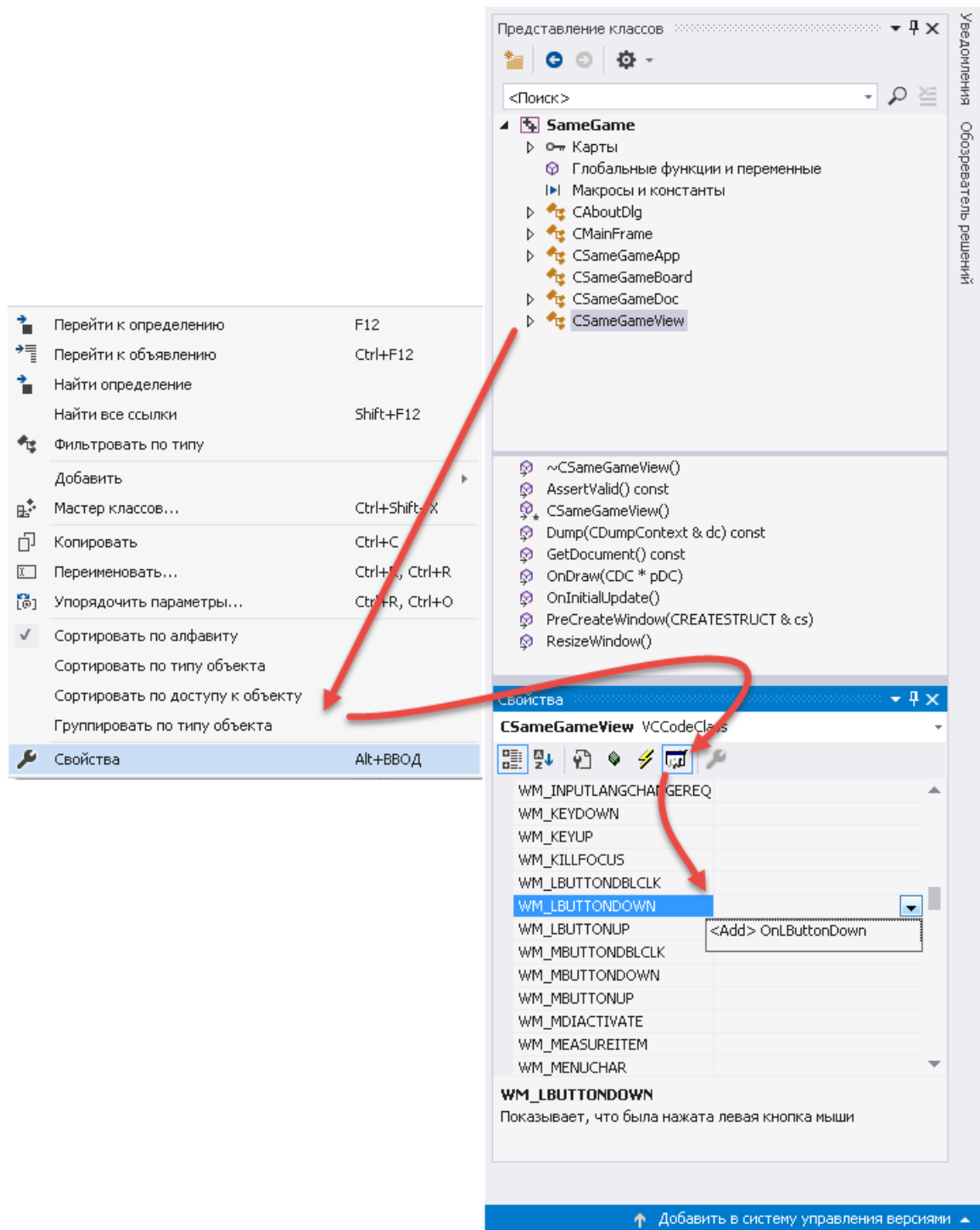
Контроль управления в событийно-ориентированном программировании определяется событиями (или сообщениями, посылаемыми программе). Пользователь выполняет какое-то действие, например, нажимает на клавишу (тем самым посылается сообщение «была нажата такая-то клавиша»), и программа реагирует на это событие, выполняя заранее определенный код. Основной цикл в программе, управляемой событиями, просто ожидает подобного сообщения, а затем вызывает соответствующий обработчик и возвращается к ожиданию другого события. В свою очередь, обработчик событий — это фрагмент кода, который вызывается каждый раз, когда происходит определенное событие.

Обработка событий

Библиотека MFC по своей сути основана на событиях/сообщениях и поэтому позволяет нам довольно легко создавать необходимые обработчики и реагировать на любое действие, которое мы хотим использовать в нашей программе. Чтобы настроить обработку событий в MFC, в Visual Studio есть возможность вывести и просмотреть список всех сообщений, на которые можно отреагировать. Все сообщения в Windows являются константами с приставкой WM_, за которой следует имя сообщения. Для реагирования на щелчки мыши в клиентской области нашей игры имеются сообщения для левой, правой и средней кнопок мыши.

Событие, которое мы будем при этом использовать — это WM_LBUTTONDOWN. Данное сообщение отправляется библиотекой MFC каждый раз, когда пользователь нажимает левую кнопку мыши. Всё, что нам нужно сделать, это настроить обработчик для этого события, а затем отреагировать на него, когда оно случится. Чтобы добавить обработчик событий, открываем окно "Свойства", выбрав класс CSameGameView, и нажимаем правую кнопку мыши (или Alt+Enter). Ниже приведено то, что вы увидите в окне свойств.

На следующем скриншоте мой курсор наведен на раздел «Сообщения», нажмите на него. Найдите опцию WM_LBUTTONDOWN, нажмите на нее, кликните на раскрывающийся список и выберите <Add> OnLButtonDown:



Это добавит к View обработчик события OnLButtonDown() с некоторым кодом по умолчанию для вызова реализации функции CView(). В этом месте поместим следующий код в тело функции.

Файл SameGameView.cpp:

```

1 void CSameGameView::OnLButtonDown(UINT nFlags, CPoint point)
2 {
3     // Вначале создаем указатель на Document
4     CSameGameDoc* pDoc = GetDocument();
5     ASSERT_VALID(pDoc);
6     if (!pDoc)

```

```

7         return;
8
9         // Получаем индекс строки и столбца элемента, по которому был
10        ↪ осуществлен клик мышкой
11        int row = point.y / pDoc→GetHeight();
12        int col = point.x / pDoc→GetWidth();
13
14        // Удаляем блоки из Document
15        int count = pDoc→DeleteBlocks(row, col);
16
17        // Проверяем, было ли удаление блоков
18        if (count > 0)
19        {
20            // Перерисовываем View
21            Invalidate();
22            UpdateWindow();
23
24            // Проверяем, закончилась ли игра
25            if (pDoc→IsGameOver())
26            {
27                // Получаем количество оставшихся блоков
28                int remaining = pDoc→GetRemainingCount();
29                CString message;
30                message.Format(_T
31                ("No more moves left\nBlocks remaining: %d"),
32                    remaining);
33
34                // Отображаем пользователю результат игры
35                MessageBox(message, _T("Game Over"), MB_OK |
36                    ↪ MB_ICONINFORMATION);
37            }
38        }
39        // OnLButtonDown по умолчанию
40        CView::OnLButtonDown(nFlags, point);
41    }

```

Объект типа CPoint содержит координаты (x, y) того места View, где был произведен клик мышкой. Это нужно для того, чтобы выяснить, на какой именно блок кликнули. Мы просто получаем корректный указатель на Document. Чтобы найти строку и столбец блока, по которому щелкнули, нужно координату x разделить на ширину блока, а координату y — на высоту:

```

1 // Получаем индекс строки и столбца элемента, по которому был произведен
2 ↪ клик мышкой
3 int row = point.y / pDoc→GetHeight();
4 int col = point.x / pDoc→GetWidth();

```

Используем целочисленное деление, результатом будут именно те строка и столбец, по которым пользователь кликнул. Когда есть строка и столбец, будем вызывать метод DeleteBlocks() в Document, чтобы удалить соседние блоки. Эта функция возвращает количество блоков, которые она удалила. Если ни один из блоков удалить не получилось, то функция просто завершает свое выполнение. Если блоки были удалены, то нам нужно заставить View «перерисовать себя», чтобы отразить изменения игрового поля. Вызов функции Invalidate() сигнализирует для View, что вся клиентская область должна быть перерисована, а вопросом перерисовки займется функция UpdateWindow():

```

1  int count = pDoc->DeleteBlocks(row, col);
2  // Проверяем, было ли удаление блоков
3  if(count > 0)
4  {
5      // Перерисовываем View
6      Invalidate();
7      UpdateWindow();
8      // ...
9  }
10 }
```

Когда игровое поле обновлено и перерисовано, проверяем, не закончилась ли игра.

```

1  if (pDoc->IsGameOver())
2  {
3      // Получаем количество оставшихся блоков
4      int remaining = pDoc->GetRemainingCount();
5      CString message;
6      message.Format(_T("Нет доступных ходов\
7      \Количество оставшихся блоков: %d"),
8
9      remaining);
10
11     // Отображаем пользователю результат игры
12     MessageBox(message, _T("Игра Закончена "), MB_OK |
13     ↪ MB_ICONINFORMATION);
14 }
```

Если игра завершилась, то получаем количество блоков, оставшихся на доске, и сообщаем об этом пользователю. Для этого создаем объект `CString`, который является строковым классом MFC, и вызываем его встроенный метод `format()`. Метод `format()` ведет себя так же, как и `sprintf()`. Здесь используем макрос `MFC_T()`, чтобы разрешить использовать различные типы строк (т.е. ASCII или Unicode). Вызываем функцию `MessageBox()`, которая отображает небольшое диалоговое окно с заголовком "Игра Закончена" и сообщением, которое создали с помощью метода `format()`. Диалоговое окно имеет кнопку ОК (`MB_OK`) и значок информации (`MB_ICONINFORMATION`).

Файл `SameGameDoc.h`:

```

1  bool IsGameOver() { return m_board.IsGameOver(); }
2  int DeleteBlocks(int row, int col)
3  {
4      return m_board.DeleteBlocks(row, col);
5  }
6  int GetRemainingCount()
7  {
8      return m_board.GetRemainingCount();
9  }
```

После того, как добавили эти функции в `Document`, меняем игровое поле, чтобы позаботиться об этих операциях. В заголовочном файле игрового поля добавьте следующие `public`-методы

Файл SameGameBoard.h:

```
1 // Мы закончили игру?
2 bool IsGameOver(void) const;
3
4 // Подсчет количества оставшихся блоков
5 int GetRemainingCount(void) const { return m_nRemaining; }
6
7 // Функция для удаления всех примыкающих блоков
8 int DeleteBlocks(int row, int col);
```

Функция `GetRemainingCount()` просто возвращает количество оставшихся блоков. Будем хранить это значение в переменной с именем `m_nRemaining`.private – .

```
1 // Количество оставшихся блоков
2 int m_nRemaining;
```

Добавляем еще одну переменную-член в наш класс, то нам нужно её инициализировать в конструкторе следующим образом.

Файл SameGameBoard.cpp:

```
1 CSameGameBoard::CSameGameBoard(void)
2     : m_arrBoard(NULL),
3       m_nColumns(15), m_nRows(15),
4       m_nHeight(35), m_nWidth(35), // ← Не забудьте поставить
5       ↪ запятую!
6       m_nRemaining(0)
7 {
8     m_arrColors[0] = RGB(0, 0, 0);
9     m_arrColors[1] = RGB(255, 0, 0);
10    m_arrColors[2] = RGB(255, 255, 64);
11    m_arrColors[3] = RGB(0, 0, 255);
12
13    // Создание и настройка параметров игровой доски
14    SetupBoard();
15 }
```

Обновим количество оставшихся блоков в методе `SetupBoard()`.

Файл SameGameBoard.cpp:

```
1 void CSameGameBoard::SetupBoard(void)
2 {
3     // При необходимости создаем доску
4     if (m_arrBoard == NULL)
5         CreateBoard();
6
7     // Устанавливаем каждому блоку случайный цвет
8     for (int row = 0; row < m_nRows; row++)
9         for (int col = 0; col < m_nColumns; col++)
10             m_arrBoard[row][col] = (rand() % 3) + 1;
11
12     // Устанавливаем количество оставшихся блоков
13     m_nRemaining = m_nRows * m_nColumns;
14 }
```

Удаление блоков с доски реализуем в виде двухэтапного процесса. Сначала заменяем все смежные блоки одного цвета на цвет фона (таким образом, удаляя их), а затем перемещаем верхние блоки вниз, а блоки, стоящие по бокам — вправо/влево (соответственно). Назовём это сжатием доски.

Осуществим удаление блоков, используя вспомогательную рекурсивную функцию

`DeleteNeighborBlocks()`, которую поместим в `private`-раздел класса. Она будет выполнять основную часть работы по удалению блоков. Для этого нужно в закрытом разделе класса сразу после функции `CreateBoard()` добавить следующее.

Файл `SameGameBoard.h`:

```
1  // Перечисление с вариантами направления (откуда мы пришли) потребуется
   ↪ для удаления блоков
2  enum Direction
3  {
4      DIRECTION_UP,
5      DIRECTION_DOWN,
6      DIRECTION_LEFT,
7      DIRECTION_RIGHT
8  };
9
10 // Вспомогательная рекурсивная функция для удаления примыкающих блоков
11 int DeleteNeighborBlocks(int row, int col, int color,
12     Direction direction);
13
14 // Функция для сжатия доски после того, как были удалены блоки
15 void CompactBoard(void);
```


6. Работа с алгоритмом в игре «SameGame» на C++/MFC

Алгоритм удаления блоков

Алгоритм удаления блоков: нужно начать с конкретного блока, а затем убедиться, что есть соседний блок с тем же цветом. Если это так, то изменяем значение цвета этого блока на цвет фона. Затем проходимся в каждом направлении и удаляем соседний блок, если он имеет тот же цвет. Этот процесс рекурсивно повторяется с каждым блоком. Ниже приведена функция DeleteBlocks() в полном объеме в файле SameGameBoard.cpp:

```
1  int CSameGameBoard::DeleteBlocks(int row, int col)
2  {
3      // Проверяем на корректность индексы ячейки и столбца
4      if (row < 0 || row ≥ m_nRows || col < 0 || col ≥ m_nColumns)
5          return -1;
6
7      // Если блок уже имеет цвет фона, то удалить его не получится
8      int nColor = m_arrBoard[row][col];
9      if (nColor == 0)
10         return -1;
11
12     // Сначала проверяем, есть ли примыкающие блоки с тем же цветом
13     int nCount = -1;
14     if ((row - 1 ≥ 0 && m_arrBoard[row - 1][col] == nColor) ||
15         (row + 1 < m_nRows && m_arrBoard[row + 1][col] == nColor)
16         ||
17         (col - 1 ≥ 0 && m_arrBoard[row][col - 1] == nColor) ||
18         (col + 1 < m_nColumns && m_arrBoard[row][col + 1] ==
19         nColor))
20     {
21         // Затем рекурсивно вызываем функцию для удаления
22         // примыкающих блоков одного цвета ...
23         m_arrBoard[row][col] = 0;
24         nCount = 1;
25
26         // ... сверху ...
27         nCount +=
28             DeleteNeighborBlocks(row - 1, col, nColor,
29             ↪ DIRECTION_DOWN);
30
31         // ... снизу ...
32         nCount +=
33             DeleteNeighborBlocks(row + 1, col, nColor,
34             ↪ DIRECTION_UP);
35
36         // ... слева ...
37         nCount +=
38             DeleteNeighborBlocks(row, col - 1, nColor,
39             ↪ DIRECTION_RIGHT);
40
41         // ... справа
42         nCount +=
43             DeleteNeighborBlocks(row, col + 1, nColor,
44             ↪ DIRECTION_LEFT);
45
46         // В конце выполняем «сжатие» нашей игровой доски
47         CompactBoard();
48     }
49 }
```

```

41         // Вычитаем число удаленных блоков из общего количества
42         ↪   блоков
43         m_nRemaining -= nCount;
44     }
45
46     // Возвращаем количество удаленных блоков
47     return nCount;
48 }

```

Сначала нужно проверить корректность индексов строки и столбца. Затем проверить, что выбранный блок не является частью фона. Далее следует проверить, есть ли хотя бы один соседний блок с таким же цветом, что у нас, но выше/ниже/слева/справа от выбранного блока. Если есть, то для выбранного блока устанавливается цвет фона (0), а для счетчика уничтоженных блоков устанавливается значение 1.

Это достигается несколькими вызовами функции `DeleteNeighborBlocks()`. При первом вызове функции мы поднимаемся выше по текущему столбцу.

С помощью `DIRECTION_DOWN` мы даем понять нашей рекурсивной функции, что пришли снизу (тем самым, пропуская это направление обхода, не выполняя лишних действий). После проверки всех 4 направлений наша игровая доска «уплотняется», и количество блоков, которые были удалены, вычитается из общего количества оставшихся блоков.

Функция `DeleteNeighborBlocks()` очень похожа на функцию `DeleteBlocks()`. Проверяем корректность значений строки и столбца и сравниваем цвет текущего блока с цветом исходного блока. После этого делаем три рекурсивных вызова функции для удаления соседних блоков. При этом используем аргумент-направление, чтобы не возвращаться туда, откуда мы пришли.

Файл `SameGameBoard.cpp`:

```

1  int CSameGameBoard::DeleteNeighborBlocks(int row, int col, int color,
2  ↪   Direction direction)
3  {
4      // Проверяем на корректность индексы ячейки и столбца
5      if (row < 0 || row ≥ m_nRows || col < 0 || col ≥ m_nColumns)
6          return 0;
7
8      // Проверка на то, что блок имеет тот же цвет
9      if (m_arrBoard[row][col] ≠ color)
10         return 0;
11     int nCount = 1;
12     m_arrBoard[row][col] = 0;
13
14     // Выполняем проверку направлений
15
16     if (direction ≠ DIRECTION_UP)
17         nCount += DeleteNeighborBlocks(row - 1, col, color,
18         ↪   DIRECTION_DOWN);
19     if (direction ≠ DIRECTION_DOWN)
20         nCount += DeleteNeighborBlocks(row + 1, col, color,
21         ↪   DIRECTION_UP);
22     if (direction ≠ DIRECTION_LEFT)
23         nCount += DeleteNeighborBlocks(row, col - 1, color,
24         ↪   DIRECTION_RIGHT);

```

```

21     if (direction != DIRECTION_RIGHT)
22         nCount += DeleteNeighborBlocks(row, col + 1, color,
23             ↪ DIRECTION_LEFT);
24
25     // Возвращаем общее количество удаленных блоков
26     return nCount;
27 }

```

К этому моменту все смежные блоки одного цвета были удалены и заменены на цвет фона, так что всё, что нам осталось сделать — это выполнить «сжатие» доски, сдвинув все вышестоящие блоки вниз, а столбцы — влево.

Файл SameGameBoard.cpp:

```

1  void CSameGameBoard::CompactBoard(void)
2  {
3      // Сначала мы всё сдвигаем вниз
4      for (int col = 0; col < m_nColumns; col++)
5      {
6          int nNextEmptyRow = m_nRows - 1;
7          int nNextOccupiedRow = nNextEmptyRow;
8          while (nNextOccupiedRow ≥ 0 && nNextEmptyRow ≥ 0)
9          {
10             // Находим пустую строку
11             while (nNextEmptyRow ≥ 0 &&
12                 m_arrBoard[nNextEmptyRow][col] != 0)
13                 nNextEmptyRow--;
14             if (nNextEmptyRow ≥ 0)
15             {
16                 // Затем находим занятую строку,
17                 ↪ расположенную следом за пустой
18                 nNextOccupiedRow = nNextEmptyRow - 1;
19                 while (nNextOccupiedRow ≥ 0 &&
20                     m_arrBoard[nNextOccupiedRow][col]
21                     ↪ = 0)
22                     nNextOccupiedRow--;
23                 if (nNextOccupiedRow ≥ 0)
24                 {
25                     // Теперь перемещаем блоки с
26                     ↪ занятой строки на пустую
27                     m_arrBoard[nNextEmptyRow][col] =
28                     m_arrBoard[nNextOccupiedRow][col];
29                     m_arrBoard[nNextOccupiedRow][col]
30                     ↪ = 0;
31                 }
32             }
33         }
34     }
35     // Затем всё, что находится справа, смещаем влево
36     int nNextEmptyCol = 0;
37     int nNextOccupiedCol = nNextEmptyCol;
38     while (nNextEmptyCol < m_nColumns && nNextOccupiedCol <
39         ↪ m_nColumns)
40     {
41         // Находим пустой столбец
42         while (nNextEmptyCol < m_nColumns &&
43             m_arrBoard[m_nRows - 1][nNextEmptyCol] != 0)
44             nNextEmptyCol++;
45     }
46 }

```

```

40         if (nNextEmptyCol < m_nColumns)
41         {
42             // Затем находим занятый столбец, расположенный
43             // ↳ следом за пустым
44             nNextOccupiedCol = nNextEmptyCol + 1;
45             while (nNextOccupiedCol < m_nColumns &&
46                 m_arrBoard[m_nRows - 1][nNextOccupiedCol]
47                 ↳ = 0)
48                 nNextOccupiedCol++;
49             if (nNextOccupiedCol < m_nColumns)
50             {
51                 // Сдвигаем весь столбец влево
52                 for (int row = 0; row < m_nRows; row++)
53                 {
54                     m_arrBoard[row][nNextEmptyCol] =
55                     ↳ m_arrBoard[row][nNextOccupiedCol];
56                     m_arrBoard[row][nNextOccupiedCol] = 0;
57                 }
58             }
59         }
60     }
61 }

```

Сначала проходим столбец за столбцом, перемещая блоки вниз. Затем ищем пустую строку. Как только её нашли, то запускается другой цикл, который ищет занятую строку. После этого пустая строка заполняется занятой строкой. Этот процесс повторяется до тех пор, пока не останется блоков, которые мы должны переместить вниз.

Вторая часть функции почти идентична первой, за исключением цикла `for`. Причина, по которой можем убрать внешний цикл, заключается в том, что нужно смотреть всего лишь на нижнюю строку в каждом столбце, и, если она будет пустой, то и весь столбец пуст, и мы можем переместить на его место что-то другое.

Реализуем функцию `IsGameOver()`. Данная функция проверяет, возможно ли вообще переместить блок, т.е. пересматривает каждый блок, чтобы определить, есть ли у него соседи с тем же цветом или нет. Когда первый из этих блоков с тем же цветом найден, то функция прекращает свое выполнение, возвращая `false`. Дальше проверку можно не продолжать. Единственный способ определить, что игра на самом деле закончена — это сделать проход по всем блокам и убедиться, что не осталось никаких возможных ходов.

Файл `SameGameBoard.cpp`:

```

1  bool CSameGameBoard::IsGameOver(void) const
2  {
3      // Проверяем столбец за столбцом (слева-направо)
4      for (int col = 0; col < m_nColumns; col++)
5      {
6          // Строку за строкой (снизу-вверх)
7          for (int row = m_nRows - 1; row ≥ 0; row--)
8          {
9              int nColor = m_arrBoard[row][col];
10
11              // Если мы попали на ячейку с цветом фона, то это
12              ↳ значит, что столбец уже уничтожен
13              if (nColor == 0)
14                  break;
15              else

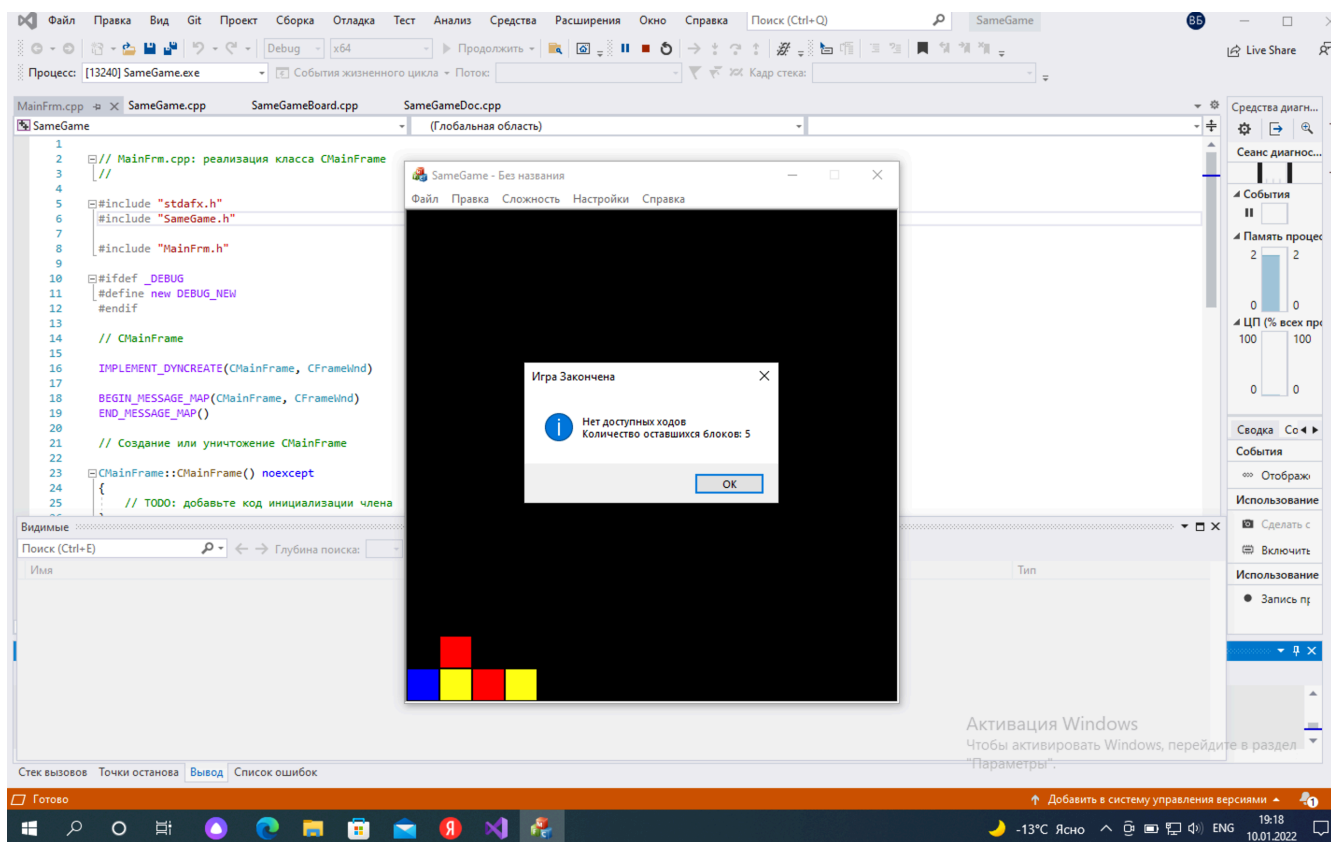
```

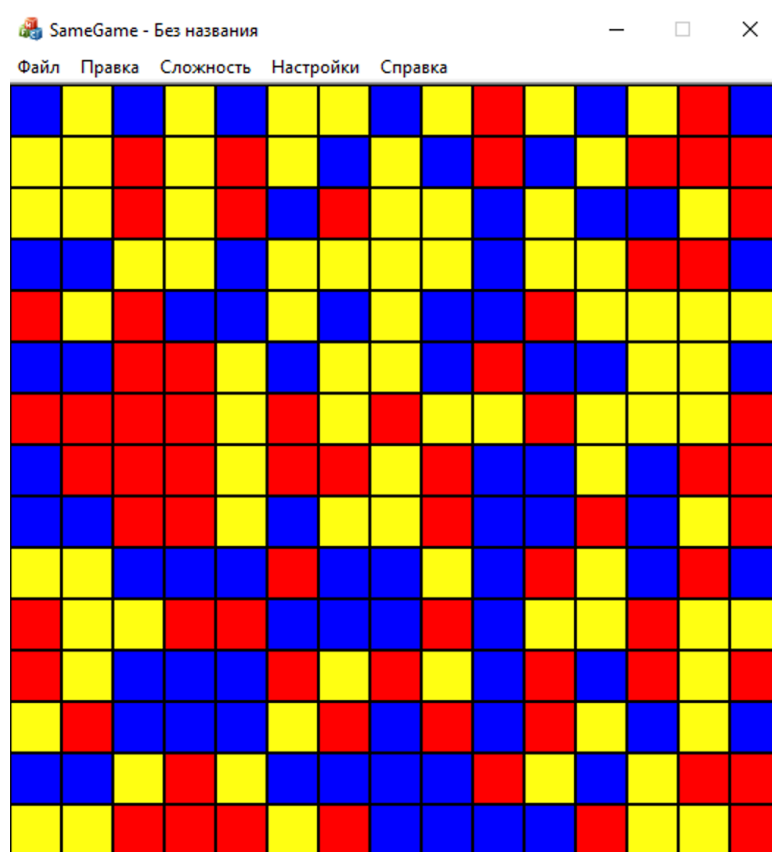
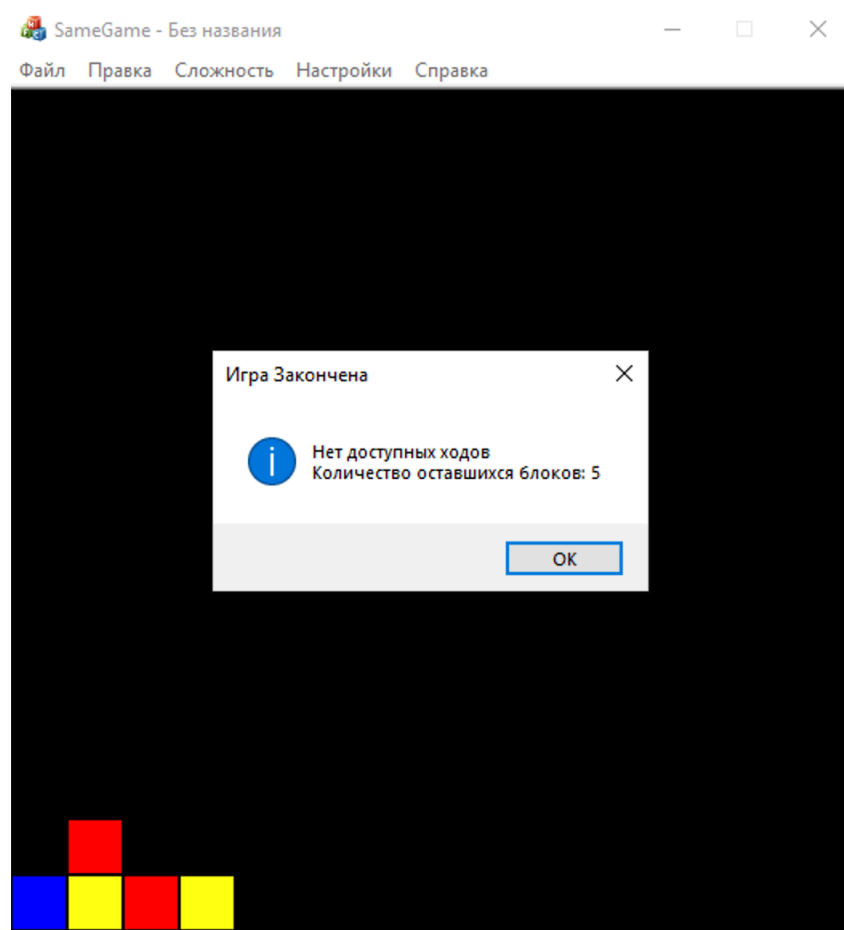
```

15         {
16             // Проверяем сверху и справа
17             if (row - 1 ≥ 0 &&
18                 m_arrBoard[row - 1][col] ==
19                 ↪ nColor)
20                 return false;
21             else if (col + 1 < m_nColumns &&
22                 m_arrBoard[row][col + 1] ==
23                 ↪ nColor)
24                 return false;
25         }
26     }
27     // Если примыкающих блоков не обнаружено, то
28     return true;

```

Два цикла for позволяют проходить столбец за столбцом и строку за строкой в поисках допустимых ходов. Поскольку выполняем проход слева-направо, то не нужно слева проверять смежные блоки схожего цвета. Поиск снизу-вверх исключает необходимость проверять возможные ходы по направлению вниз. Этот порядок поиска также позволяет немного оптимизировать функцию IsGameOver(). Как только цвет блока станет фоновым, то сможем пропустить остальную часть столбца, потому что всё, что выше него, — также будет пустым (благодаря функции CompactBoard()).





7. Работа с меню в игре «SameGame» на C++/MFC

Меню выбора уровня сложности

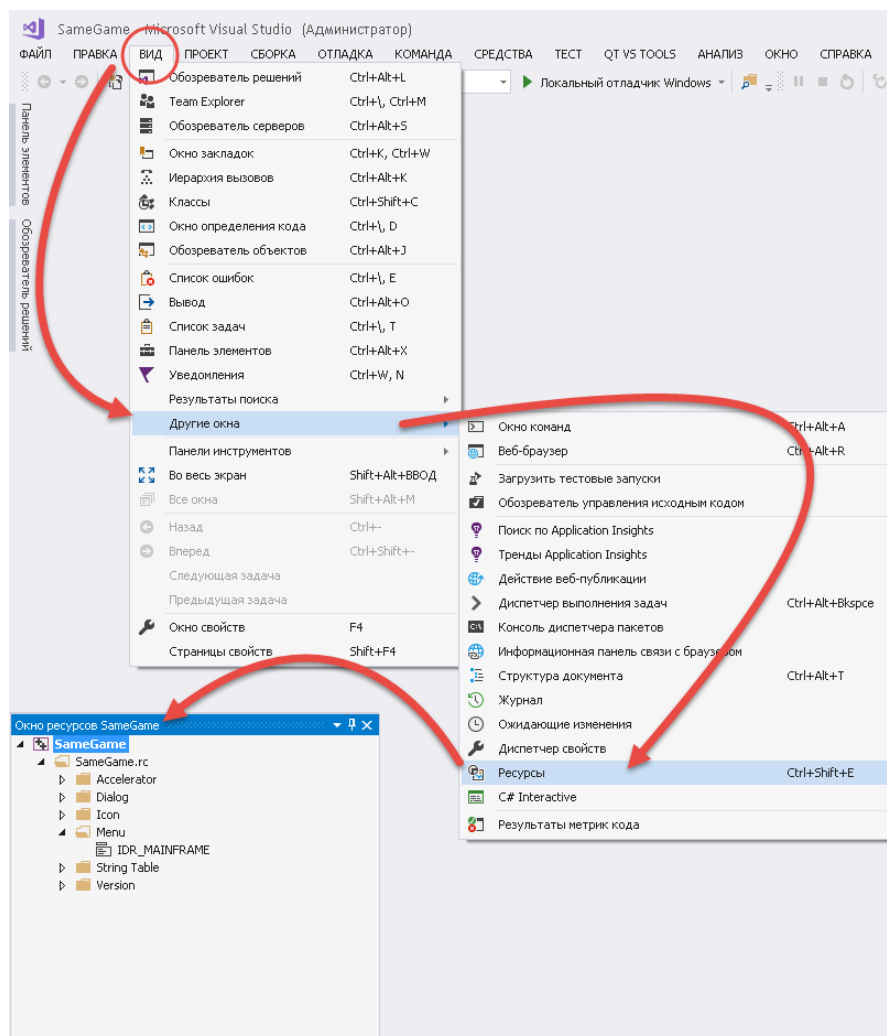
Теперь у нас есть рабочий прототип игры «SameGame». Добавим возможность выбора уровня сложности. Для этого нужно:

- обновить стандартное меню, которое было добавлено мастером приложений MFC;
- добавить новые пункты в меню;
- написать соответствующие обработчики событий.

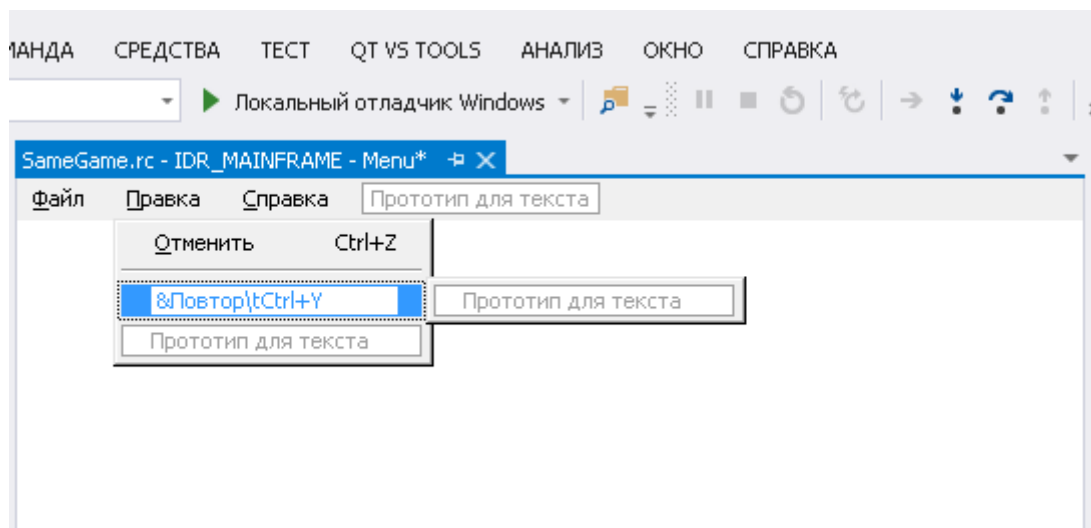
Сделаем добавление уровня сложности, через настройку количества цветов для игровых блоков.

Пункты меню

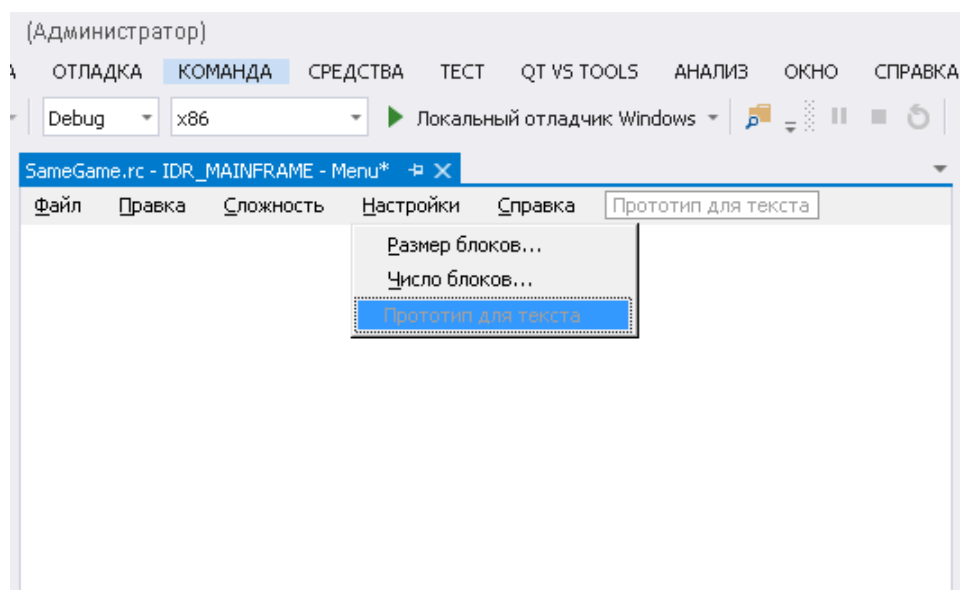
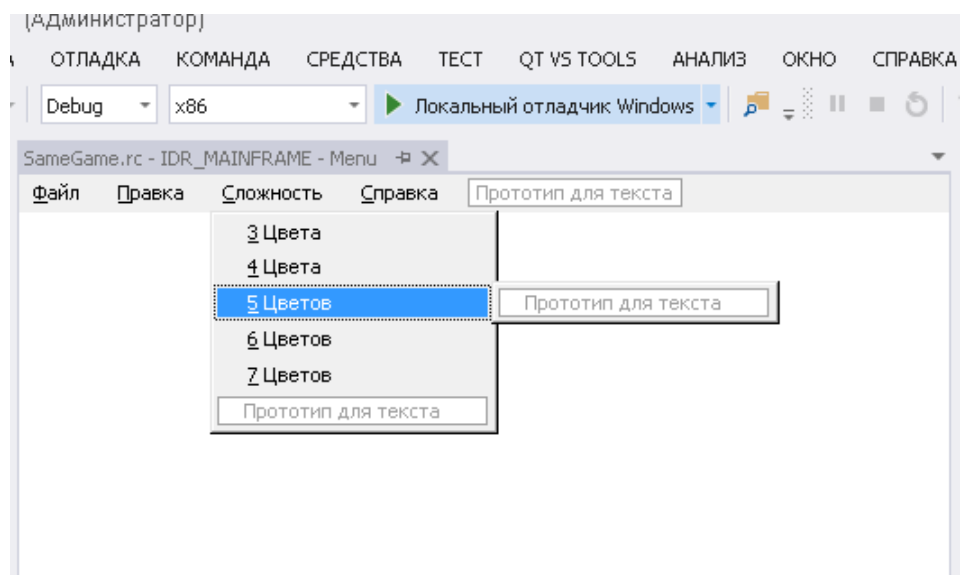
Добавление нового пункта меню осуществляется через "Окно Ресурсов". Вы можете открыть окно ресурсов из меню "Вид" > "Другие Окна" > "Ресурсы" или воспользоваться сочетанием клавиш Ctrl+Shift+E:



Открываем наше меню в "Редакторе меню", дважды щелкнув по опции `IDR_MAINFRAME`.



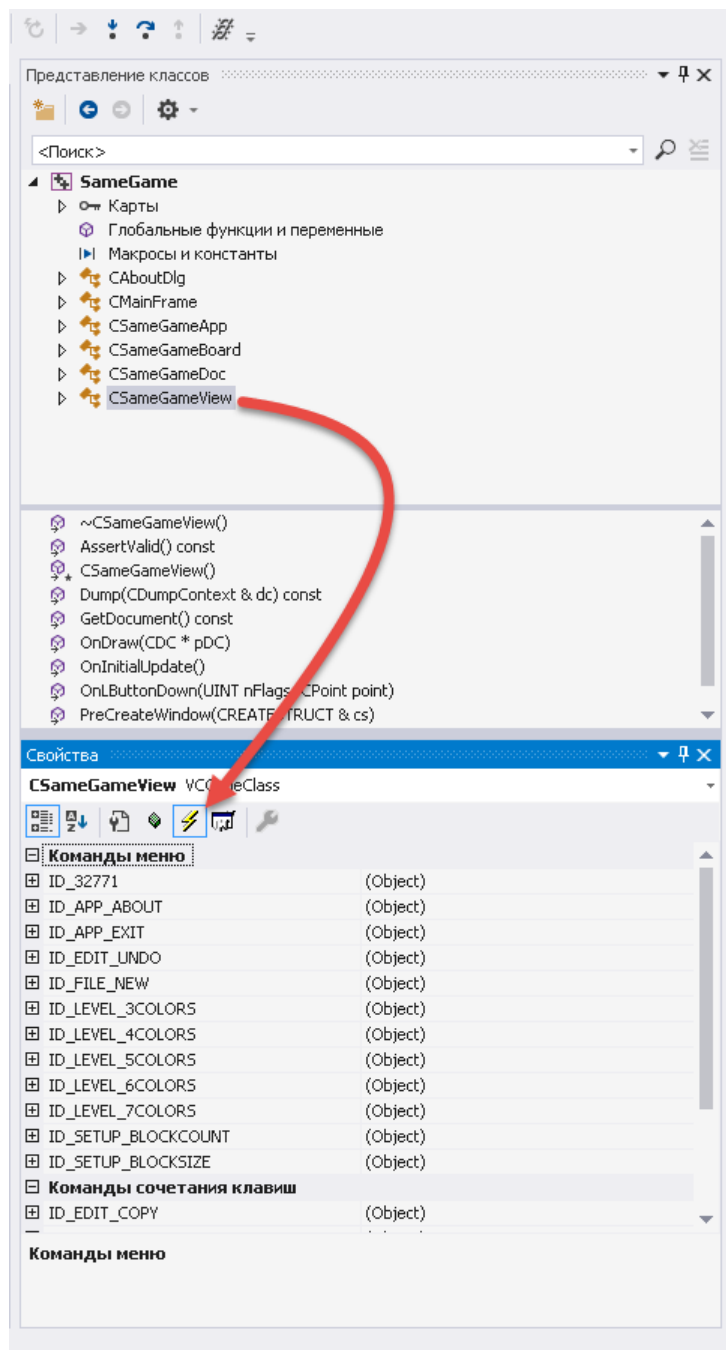
Добавляем последовательно пункты меню как на скринах



8. Добавление уровней сложности в игре «SameGame» на C++/MFC

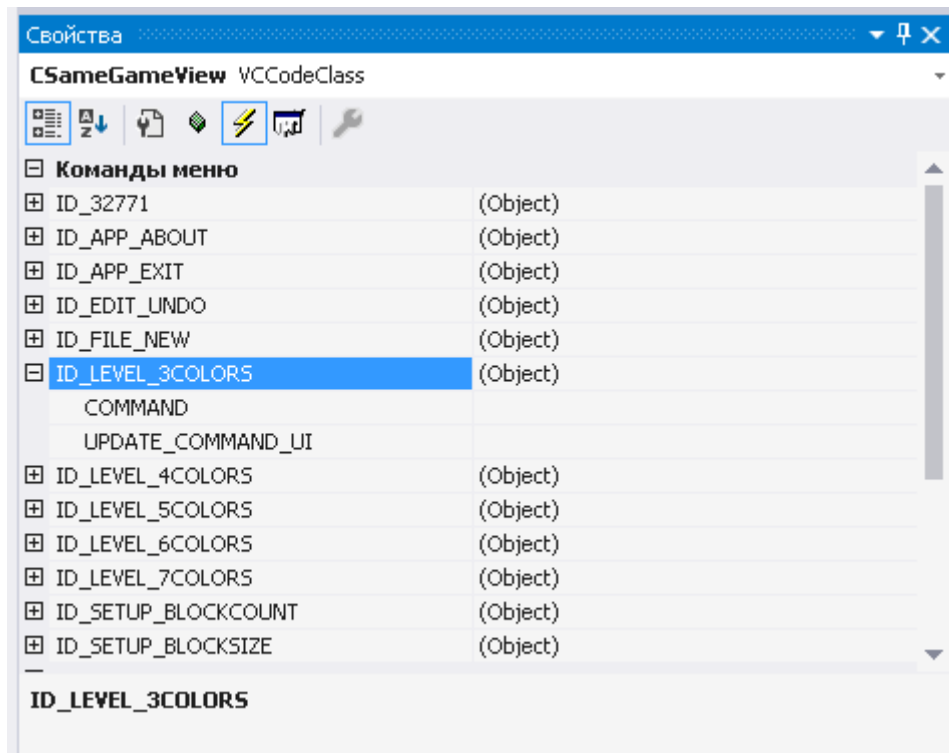
Уровни сложности

Нужно запустить игру и поймать событие с помощью обработчика. Для этого в окне "Представление классов" выберите CSameGameView, а затем нажмите на кнопку "События" (иконка молнии).



"Команды меню". Нужно нажать на + напротив ID_LEVEL_3COLORS (он соответствует пункту меню "Сложность" > "3 Цвета"). Параметр COMMAND — это обработчик события для выбора пункта меню. Параметр UPDATE_COMMAND_UI — это обработчик события, который позволяет изменить состояние опции меню. Под состоянием

подразумевается активно/неактивно или установить/снять флаг выбора пункта меню. В нашем случае мы собираемся поставить галочку напротив выбранного уровня сложности.



Нажмем на стрелочку, которая раскрывает список, соответствующий пункту COMMAND и выберите Add. Повторим то же самое и для UPDATE_COMMAND_UI. Прделаем эти действия для всех ID_LEVEL_3COLORS, ID_LEVEL_4COLORS и вплоть до ID_LEVEL_7COL. Начнем с редактирования игровой доски, затем перейдем к Document и закончим на View. В заголовочном файле SameGameBoard.h после $m_nRemaining$ — :

```
1 // Количество цветов
2 int m_nColors;
```

Добавим в public-раздел 2 функции. С помощью первой функции мы сможем устанавливать значение количества цветов, а с помощью второй — сможем это значение считывать.



Файл SameGameBoard.h:

```
1 // Гетеры и сеттеры для количества цветов
2 int GetNumColors(void) { return m_nColors; }
3 void SetNumColors(int nColors)
4 { m_nColors = (nColors ≥ 3 && nColors ≤ 7) ? nColors : m_nColors; }
```

Функция SetNumColors() ограничивает набор значений числом от 3 до 7 в соответствии с параметрами нашего меню. Так как добавляем дополнительные цвета, то нужно увеличить массив `m_arrColors`.

Файл SameGameBoard.h:

```

1 // Список цветов, 0  это цвет фона, а 1-7  это цвета блоков
2 static COLORREF m_arrColors[8];

```

В исходном файле для игрового поля обновим несколько дополнительных функций и массив цветов. Обновим конструктор.

Файл SameGameBoard.cpp:

```

1 COLORREF CSameGameBoard::m_arrColors[8];
2
3 CSameGameBoard::CSameGameBoard(void)
4 : m_arrBoard(NULL),
5   m_nColumns(15), m_nRows(15),
6   m_nHeight(35), m_nWidth(35),
7   m_nRemaining(0), m_nColors(3)
8 {
9     m_arrColors[0] = RGB(0,0,0);
10    m_arrColors[1] = RGB(255,0,0);
11    m_arrColors[2] = RGB(255,255,64);
12    m_arrColors[3] = RGB(0,0,255);
13
14    m_arrColors[4] = RGB(0,255,0);
15    m_arrColors[5] = RGB(0,255,255);
16    m_arrColors[6] = RGB(255,0,128);
17    m_arrColors[7] = RGB(0,64,0);
18
19    // Создаем и настраиваем игровую доску
20    SetupBoard();
21 }

```

SetupBoard(). В предыдущие разы зафиксировали количество цветов числом 3. Теперь нужно изменить это значение с 3 на m_nColors, чтобы потом, при помощи функции rand(), получать случайное количество цветов для каждой отдельной партии в игре.

Файл SameGameBoard.cpp:

```

1 void CSameGameBoard::SetupBoard(void)
2 {
3     // При необходимости создаем доску
4     if(m_arrBoard == NULL)
5         CreateBoard();
6
7     // Устанавливаем каждому блоку случайный цвет
8     for(int row = 0; row < m_nRows; row++)
9         for(int col = 0; col < m_nColumns; col++)
10            m_arrBoard[row][col] = (rand() % m_nColors) + 1;
11
12     // Устанавливаем количество оставшегося пространства
13     m_nRemaining = m_nRows * m_nColumns;
14 }

```

Document. Добавим функции, чтобы View мог изменить количество цветов. Функцию GetNumColors() следует поместить в public-раздел Document заголовочного файла SameGameDoc.h. Также необходимо добавить реализацию функции SetNumColors().

Файл SameGameDoc.cpp:

```
1 void CSameGameDoc::SetNumColors(int nColors)
2 {
3     // Сначала задаем количество цветов ...
4     m_board.SetNumColors(nColors);
5
6     // ... затем устанавливаем параметры игровой доски
7     m_board.SetupBoard();
8 }
```

Отредактируем View. При добавлении обработчиков событий в заголовочный файл SameGameView. автоматически должны были прописаться следующие прототипы функций.

Файл SameGameView.h:

```
1 // Функции для изменения уровня сложности
2 afx_msg void OnLevel3colors();
3 afx_msg void OnLevel4colors();
4 afx_msg void OnLevel5colors();
5 afx_msg void OnLevel6colors();
6 afx_msg void OnLevel7colors();
7
8 // Функции для обновления меню
9 afx_msg void OnUpdateLevel3colors(CCmdUI *pCmdUI);
10 afx_msg void OnUpdateLevel4colors(CCmdUI *pCmdUI);
11 afx_msg void OnUpdateLevel5colors(CCmdUI *pCmdUI);
12 afx_msg void OnUpdateLevel6colors(CCmdUI *pCmdUI);
13 afx_msg void OnUpdateLevel7colors(CCmdUI *pCmdUI);
```

Обозначение `afx_msg` указывает на то, что функция является обработчиком событий. Функции, начинающиеся с `OnUpdateLevel...()`, используют указатель на объект `CCmdUI`. Изменяя его состояние на активное/неактивное и устанавливая метку выбрано/не выбрано для подпунктов меню. В исходный файл `SameGameView.cpp` нужно добавить некоторое количество дополнительного кода:

```
1 BEGIN_MESSAGE_MAP(CSameGameView, CView)
2     ON_WM_LBUTTONDOWN()
3     ON_WM_ERASEBKGD()
4
5     ON_COMMAND(ID_LEVEL_3COLORS, &CSameGameView::OnLevel3colors)
6     ON_COMMAND(ID_LEVEL_4COLORS, &CSameGameView::OnLevel4colors)
7     ON_COMMAND(ID_LEVEL_5COLORS, &CSameGameView::OnLevel5colors)
8     ON_COMMAND(ID_LEVEL_6COLORS, &CSameGameView::OnLevel6colors)
9     ON_COMMAND(ID_LEVEL_7COLORS, &CSameGameView::OnLevel7colors)
10    ON_UPDATE_COMMAND_UI(ID_LEVEL_3COLORS,
11        &CSameGameView::OnUpdateLevel3colors)
12    ON_UPDATE_COMMAND_UI(ID_LEVEL_4COLORS,
13        &CSameGameView::OnUpdateLevel4colors)
14    ON_UPDATE_COMMAND_UI(ID_LEVEL_5COLORS,
15        &CSameGameView::OnUpdateLevel5colors)
16    ON_UPDATE_COMMAND_UI(ID_LEVEL_6COLORS,
17        &CSameGameView::OnUpdateLevel6colors)
18    ON_UPDATE_COMMAND_UI(ID_LEVEL_7COLORS,
19        &CSameGameView::OnUpdateLevel7colors)
```

```
20
21 END_MESSAGE_MAP()
```

Карта сообщений (MESSAGE_MAP) — это список макросов в языке C++, которые связывают событие с соответствующим обработчиком. Данный список генерируется автоматически, вам не нужно будет его редактировать. Функции OnLevel*colors() (вместо * пишется номер от 3 до 7) практически одинаковые и имеют следующий вид.

Файл SameGameView.cpp:

```
1 void CSameGameView::OnLevel3colors()
2 {
3     // Получаем указатель на Document
4     CSameGameDoc* pDoc = GetDocument();
5     ASSERT_VALID(pDoc);
6     if(!pDoc)
7         return;
8
9     // Устанавливаем количество цветов
10    pDoc->SetNumColors(3);
11
12    // Перерисовываем View
13    Invalidate();
14    UpdateWindow();
15 }
```

Во всех функциях представления View нужно сначала получить указатель на Document. Затем устанавливаем количество используемых цветов, которое указано в имени соответствующей функции, т.е. OnLevel3colors() вызывает SetNumColors(3) и так далее. В конце перерисовываем View. Эти действия необходимо повторить для всех обработчиков событий пунктов меню. После того, как все эти шаги будут выполнены, можно скомпилировать и запустить проект. Видно что количество цветов изменилось от 3 до 4 и так далее. Как вариант, можно попробовать рассмотреть возможность создания вспомогательной функции, которая выполнит за нас всю эту работу, принимая количество цветов в качестве аргумента. Например, в заголовочном файле SameGameView.h добавляем следующий прототип функции:

```
1 void setColorCount(int numColors);
```

А в SameGameView.cpp добавим следующий код:

```
1 void CSameGameView::setColorCount(int numColors)
2 {
3     // Сначала получаем указатель на документ
4     CSameGameDoc* pDoc = GetDocument();
5     ASSERT_VALID(pDoc);
6     if(!pDoc)
7         return;
8     // Устанавливаем количество цветов
9     pDoc->SetNumColors(numColors);
10    // Перерисовываем View
11    Invalidate();
```

```

12     UpdateWindow();
13 }
14
15 void CSameGameView::OnLevel3colors()
16 {
17     setColorCount(3);
18 }

```

Обработчики событий ON_UPDATE_COMMAND_UI вызываются при раскрытии списка меню по команде пользователя (по одной функции для каждого пункта меню). Затем используется функция SetCheck() объекта CCmdUI для установки/снятия флажка выбора уровня сложности. Как всегда, начинаем с получения указателя на Document, а затем проверяем количество выставленных на доске цветов.

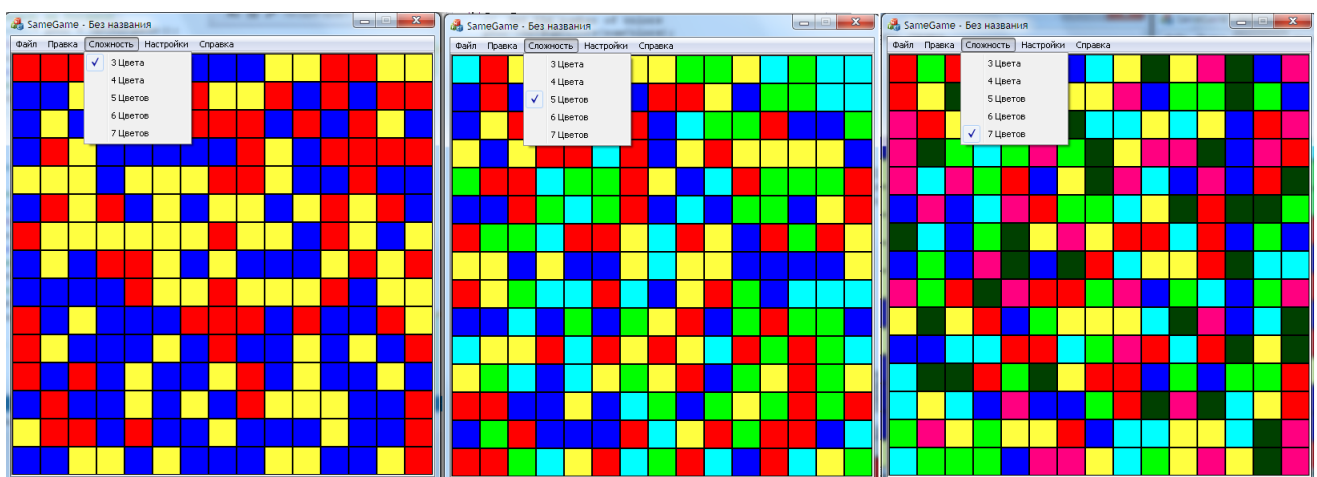
Файл SameGameView.cpp:

```

1 void CSameGameView::OnUpdateLevel3colors(CCmdUI *pCmdUI)
2 {
3     // Сначала получаем указатель на Document
4     CSameGameDoc* pDoc = GetDocument();
5     ASSERT_VALID(pDoc);
6     if(!pDoc)
7         return;
8
9     // Проверка установленного уровня сложности
10    pCmdUI->SetCheck(pDoc->GetNumColors() == 3);
11 }

```

Все функции OnUpdateLevel*colors() практически идентичны друг другу, за исключением числа, с которым сравниваем GetNumColors(). Все эти пять функций вызываются при отображении меню уровня сложности для установки или снятия флажка выбора пользователя. Теперь игра должна выглядеть примерно следующим образом:



9. Реализация функционала «Отмена/Повтор» игры «SameGame»

Это похоже на функцию стек «Отмена/Повтор» из-за того, что в её основе лежит принцип «Абстрактных типов данных», к которым относится и стек. Стек — это набор объектов, который похож на стопку тарелок на столе. Единственный способ добраться до нижних тарелок — снять верхние. Чтобы добавить тарелку в стопку, нужно поместить её поверх других. Таким образом реализуется принцип LIFO (сокр. от «Last In, First Out» = «Последним Пришёл, Первым Ушёл»). Когда выполняете любое действие в игре, оно сразу же помещается на вершину стека, откуда его можно потом как отменить, так и восстановить (повторить). Способ, которым мы собираемся это всё реализовать заключается в том, чтобы сохранить копию старого объекта игрового поля в стеке «Отмена». Когда мы отменяем ход, текущая игровая доска помещается в стек «Повтор», а верхний объект из стека «Отмена» становится текущим.

Нужно создать конструктор глубокого копирования. Для этого добавим прототип функции конструктора копирования прямо между конструктором по умолчанию и деструктором в SameGameBoard.h:

```
1 // Конструктор по умолчанию
2 CSameGameBoard(void);
3
4 // Конструктор глубокого копирования
5 CSameGameBoard(const CSameGameBoard& board);
6
7 // Деструктор
8 ~CSameGameBoard(void);
```

Конструктор глубокого копирования используется, потому что у нас есть указатель на некоторую область динамически выделенной памяти. Это означает, что не можем просто скопировать указатель, нужно динамически выделить новый блок памяти, а затем скопировать содержимое прошлого блока памяти в новый. Ниже описывается добавление конструктора копирования в исходный файл для игрового поля SameGameBoard.cpp:

```
1 CSameGameBoard::CSameGameBoard(const CSameGameBoard& board)
2 {
3     // Копирование всех элементов класса
4     m_nColumns = board.m_nColumns;
5     m_nRows = board.m_nRows;
6     m_nHeight = board.m_nHeight;
7     m_nWidth = board.m_nWidth;
8     m_nRemaining = board.m_nRemaining;
9     m_nColors = board.m_nColors;
10
11     // Копирование цветовых элементов
12     for ( int i = 0; i < 8; i++ )
13         m_arrColors[i] = board.m_arrColors[i];
14     m_arrBoard = NULL;
15
16     // Создание нового игрового поля
17     CreateBoard();
18
19     // Копирование содержимого игрового поля
```

```

20     for(int row = 0; row < m_nRows; row++)
21         for(int col = 0; col < m_nColumns; col++)
22             m_arrBoard[row][col] = board.m_arrBoard[row][col];
23     }

```

Код конструктора копирования:

сначала мы копируем переменные-члены класса;

затем указателю на доску присваиваем значение NULL;

после этого вызываем функцию CreateBoard(), которая создает новый двумерный массив игрового поля того же размера, что и исходный (т.к. мы уже установили количество строк и столбцов перед вызовом функции).

последними идут 2 цикла for, которые копируют все цвета блоков со старой доски на новую.

Большая часть работы будет выполняться самим Document, т.к. именно он будет создавать цепочку действий «Отмена/Повтор» и хранить оба типа стека. Для этого мы воспользуемся библиотекой STL. Она содержит класс стека, который очень прост в использовании.

Передаем классу тип переменной (указатель на SameGameBoard), а он предоставляет нам несколько простых функций для работы со стеком:

функция push() — добавляет новый элемент в стек;

функция pop() — удаляет самый последний элемент из стека;

функция top() — возвращает элемент, находящийся на вершине стека;

функция empty() — определяет, является ли стек пустым.

Ниже расположен полный исходный код заголовочного файла SameGameDoc.h со всеми изменениями:

```

1  #pragma once
2  #include "SameGameBoard.h"
3  #include <stack>
4
5  class CSameGameDoc : public CDocument
6  {
7  protected:
8      CSameGameDoc() noexcept;
9      DECLARE_DYNCREATE(CSameGameDoc)
10
11  // Операции
12  public:
13      // Функции доступа к игровой доске
14      COLORREF GetBoardSpace(int row, int col) { return
        ↳ m_board.GetBoardSpace(row, col); }
15      void SetupBoard(void) { m_board.SetupBoard(); }
16
17      int GetWidth(void) { return m_board.GetWidth(); }
18      void SetWidth(int nWidth) { m_board.SetWidth(nWidth); }
19
20      int GetHeight(void) { return m_board.GetHeight(); }
21      void SetHeight(int nHeight) { m_board.SetHeight(nHeight); }
22
23      int GetColumns(void) { return m_board.GetColumns(); }

```



```

24     void SetColumns(int nColumns) { m_board.SetColumns(nColumns); }
25
26     int GetRows(void) { return m_board.GetRows(); }
27     void SetRows(int nRows) { m_board.SetRows(nRows); }
28
29     void DeleteBoard(void) { m_board.DeleteBoard(); }
30     bool IsGameOver() { return m_board.IsGameOver(); }
31
32     int DeleteBlocks(int row, int col) { return
        ↪ m_board.DeleteBlocks(row, col); }
33     int GetRemainingCount() { return m_board.GetRemainingCount(); }
34     int GetNumColors() { return m_board.GetNumColors(); }
35     void SetNumColors(int nColors);
36
37     // Переопределения
38     public:
39         virtual BOOL OnNewDocument();
40         virtual void Serialize(CArchive& ar);
41     #ifdef SHARED_HANDLERS
42         virtual void InitializeSearchContent();
43         virtual void OnDrawThumbnail(CDC& dc, LPRECT lprcBounds);
44     #endif
45
46     // Реализация
47     public:
48         virtual ~CSameGameDoc();
49     #ifdef _DEBUG
50         virtual void AssertValid() const;
51         virtual void Dump(CDumpContext& dc) const;
52     #endif
53
54     protected:
55     // Функции очистки стеков «Отмена/Повтор»
56         void ClearUndo();
57         void ClearRedo();
58
59     // Экземпляр класса игровой доски. Теперь мы сделали его указателем на
        ↪ класс
60     CSameGameBoard* m_board;
61
62     // Стек "Отмена"
63     std::stack<CSameGameBoard*> m_undo;
64
65     // Стек "Повтор"
66     std::stack<CSameGameBoard*> m_redo;
67
68     // Генерация функции сообщений
69     protected:
70         DECLARE_MESSAGE_MAP()
71
72     #ifdef SHARED_HANDLERS
73
74     // Вспомогательная функция, задающая содержимое поиска для обработчика
        ↪ поиска
75     void SetSearchContent(const CString& value);
76     #endif
77     };

```

Прежде всего нужно подключить заголовочный файл стека. Поскольку мы собираемся

изменить переменную `m_board` на указатель, то нам придется перейти от использования оператора прямой принадлежности «точка» к оператору не прямой принадлежности «стрелка» (или к разыменованию указателя в каждой функции `Document`). Далее помещаем реализацию функции `DeleteBlocks()` в исходный файл.

Затем мы добавляем шесть новых функций, четыре из которых будут располагаться в `public`-разделе класса, а две другие - в `protected`-разделе. `public`-функции разделены на две группы: функции `UndoLast()` и `RedoLast()` выполняют «Отмена/Повтор», в то время как функции `CanUndo()` и `CanRedo()` являются обычными тестами, которые будем использовать для включения и отключения параметров меню, когда они недоступны. `protected`-функции являются простыми вспомогательными функциями для очистки и освобождения памяти, связанной с обоими типами стеков. В конце добавляем два объявления стеков «Отмена/Повтор».

Файл `SameGameDoc.cpp`:

```
1  #include "stdafx.h"
2  #ifndef SHARED_HANDLERS
3  #include "SameGame.h"
4  #endif
5
6  #include "SameGameDoc.h"
7
8  #include <propkey.h>
9
10 #ifdef _DEBUG
11 #define new DEBUG_NEW
12 #endif
13
14 // CSameGameDoc
15 IMPLEMENT_DYNCREATE(CSameGameDoc, CDocument)
16
17 BEGIN_MESSAGE_MAP(CSameGameDoc, CDocument)
18 END_MESSAGE_MAP()
19
20
21 // Создание/уничтожение CSameGameDoc
22 CSameGameDoc::CSameGameDoc() noexcept
23 {
24     // Здесь всегда должна быть игровая доска
25     m_board = new CSameGameBoard();
26 }
27
28 CSameGameDoc::~CSameGameDoc()
29 {
30     // Удаляем текущую игровую доску
31     delete m_board;
32
33     // Удаляем всё из стека «Отмена»
34     ClearUndo();
35
36     // Удаляем всё из стека «Повтор»
37     ClearRedo();
38 }
39
40 BOOL CSameGameDoc::OnNewDocument()
41 {
42     if (!CDocument::OnNewDocument())
```

```

43     return FALSE;
44
45     // Устанавливаем (или сбрасываем) игровую доску
46     m_board→SetupBoard();
47
48     // Очистка стеков «Отмена/Повтор»
49     ClearUndo();
50     ClearRedo();
51
52     return TRUE;
53 }
54
55 void CSameGameDoc::SetNumColors(int nColors)
56 {
57     // Сначала задаем количество цветов
58     m_board→SetNumColors(nColors);
59
60     // А затем устанавливаем параметры игровой доски
61     m_board→SetupBoard();
62 }
63
64 int CSameGameDoc::DeleteBlocks(int row, int col)
65 {
66     // Сохранение текущего состояния доски в стеке «Отмена»
67     m_undo.push(new CSameGameBoard(*m_board));
68
69     // Очищаем стек «Повтор»
70     ClearRedo();
71
72     // Затем удаляем блоки
73     int blocks = m_board→DeleteBlocks(row, col);
74
75     // Очищаем стек «Отмена» в конце игры
76     if(m_board→IsGameOver())
77         ClearUndo();
78
79     // Возвращаем количество блоков
80     return blocks;
81 }
82
83 void CSameGameDoc::UndoLast()
84 {
85     // Смотрим, есть ли у нас что-нибудь в стеке «Отмена»
86     if(m_undo.empty())
87         return;
88
89     // Помещаем текущую игровую доску в стек «Повтор»
90     m_redo.push(m_board);
91
92     // Назначаем верхний элемент стека «Отмена» текущим
93     m_board = m_undo.top();
94     m_undo.pop();
95 }
96
97 bool CSameGameDoc::CanUndo()
98 {
99     // Убеждаемся, что у нас есть возможность выполнить отмену действия
100     return !m_undo.empty();
101 }
102

```

```

103 void CSameGameDoc::RedoLast()
104 {
105     // Смотрим, есть ли у нас что-нибудь в стеке «Повтор»
106     if(m_redo.empty())
107         return;
108
109     // Помещаем текущую игровую доску в стек «Отмена»
110     m_undo.push(m_board);
111
112     // Назначаем верхний элемент стека «Повтор» текущим
113     m_board = m_redo.top();
114     m_redo.pop();
115 }
116
117 bool CSameGameDoc::CanRedo()
118 {
119     // Убеждаемся, сможем ли мы выполнить повтор действия (не пуст ли стек)
120     return !m_redo.empty();
121 }
122
123 void CSameGameDoc::ClearUndo()
124 {
125     // Очищаем стек «Отмена»
126     while(!m_undo.empty())
127     {
128         delete m_undo.top();
129         m_undo.pop();
130     }
131 }
132
133 void CSameGameDoc::ClearRedo()
134 {
135     // Очищаем стек «Повтор»
136     while(!m_redo.empty())
137     {
138         delete m_redo.top();
139         m_redo.pop();
140     }
141 }

```

Мы предполагаем, что всегда будет действительная игровая доска, на которую ссылается указатель `m_board`, поэтому она должна создаваться в конструкторе, а удаляться в деструкторе. Как только объект будет удален в деструкторе, то мы также должны удалить все другие игровые доски, которые были сохранены, вызвав функции `Clear...`() для очистки стеков «Отмена/Повтор».

Затем мы добавили изменения в функцию `OnNewDocument()`, которые очищают стеки «Отмена/Повтор», чтобы новая игра начиналась со свежего набора стеков. Последнее изменение в этом файле — это перемещение функции `DeleteBlocks()` из заголовочного файла в исходный файл `.cpp`.

Прежде чем мы удалим какие-либо блоки или изменим расположение элементов на игровом поле, нам нужно сохранить копию текущей игровой доски в стеке «Отмена». Выполняется это с помощью конструктора копирования, который мы написали. Как только мы выполним какое-либо действие, нам нужно сразу же очистить стек «Повтор», потому что все данные, которые были в нем, больше не действительны. После того, как эти два стека будут обновлены, мы готовы приступить к фактическому удалению блоков.

Далее, как только игра закончится, мы должны очистить стек «Отмена», потому что игра в своей фазе достигла своего окончательного состояния. Очистка стека ставит точку в текущей игровой партии, т.к. не позволяет игроку вернуться на несколько шагов назад и по-другому «сыграть прошлые ходы». В самом конце мы возвращаем количество блоков, которые были удалены.

Функции UndoLast() и RedoLast() похожи друг на друга. Они выполняют действия в обратном порядке. Сначала мы должны убедиться, что у нас есть возможность отменить или повторить действие. Для этого можно было бы использовать функцию CanUndo() или CanRedo(), но из соображений эффективности мы используем STL-функцию empty().

Поэтому, если у нас есть действие, для которого мы можем выполнить «Отмена/Повтор», мы берем текущую игровую доску и помещаем её в соответствующий противоположный стек: в стек «Повтор», если отменяем действие, и в стек «Отмена» — если повторяем действие. Затем указатель текущей игровой доски мы устанавливаем на доску, расположенную на вершине стека «Отмена» или стека «Повтор», и достаём её оттуда. Функции CanUndo() и CanRedo() нужны нам для того, чтобы определить, можем ли мы отменить/повторить действие.

Последние две функции, добавленные в класс Document, нужны для очистки и освобождения памяти, используемой различными стеками. Мы просто перебираем все указатели в стеке, удаляя объект, а затем выталкиваем указатель из стека. Это гарантирует то, что вся память будет освобождена.

Сейчас нужно внести изменения во View. Эти изменения являются добавлением простых обработчиков событий для параметров меню «Отмена» и «Повтор». Сначала мы создаём их с помощью кнопки "События" (молния) на панели «Свойства» файла CSameGameView.h. Мы хотим добавить обработчики ON_COMMAND и ON_UPDATE_COMMAND_UI. Обработчик ON_UPDATE_COMMAND_UI позволяет нам отключить параметры меню, когда нет действий для выполнения «Отмена/Повтор». После добавления всех 4 обработчиков событий следующий код будет автоматически добавлен в заголовочный файл SameGameView.h:

```
1 // Функции для меню «Отмена/Повтор»
2 afx_msg void OnEditUndo();
3 afx_msg void On32771();
4
5 // Функции для обновления меню «Отмена/Повтор»
6 afx_msg void OnUpdateEditUndo(CCmdUI *pCmdUI);
7 afx_msg void OnUpdate32771 (CCmdUI *pCmdUI);
```

```
1 ON_COMMAND(ID_32771, &CSameGameView::On32771)
2 ON_COMMAND(ID_EDIT_UNDO, &CSameGameView::OnEditUndo)
3 ON_UPDATE_COMMAND_UI(ID_EDIT_UNDO, &CSameGameView::OnUpdateEditUndo)
4 ON_UPDATE_COMMAND_UI(ID_32771, &CSameGameView::OnUpdate32771)
```

Получаем указатель на Document, вызываем функцию в Document и, наконец, перерисовываем View. Сделаем это для реализации действий «Отмена/Повтор»:

```
1 void CSameGameView::OnEditUndo()
2 {
```

```

3      // Получаем указатель на Document
4      CSameGameDoc* pDoc = GetDocument();
5      ASSERT_VALID(pDoc);
6      if(!pDoc)
7          return;
8      pDoc->UndoLast();
9
10     // Перерисовываем View
11     Invalidate();
12     UpdateWindow();
13 }
14
15 void CSameGameView::On32771()
16 {
17     // Получаем указатель на Document
18     CSameGameDoc* pDoc = GetDocument();
19     ASSERT_VALID(pDoc);
20     if(!pDoc)
21         return;
22     pDoc->RedoLast();
23
24     // Перерисовываем View
25     Invalidate();
26     UpdateWindow();
27 }

```

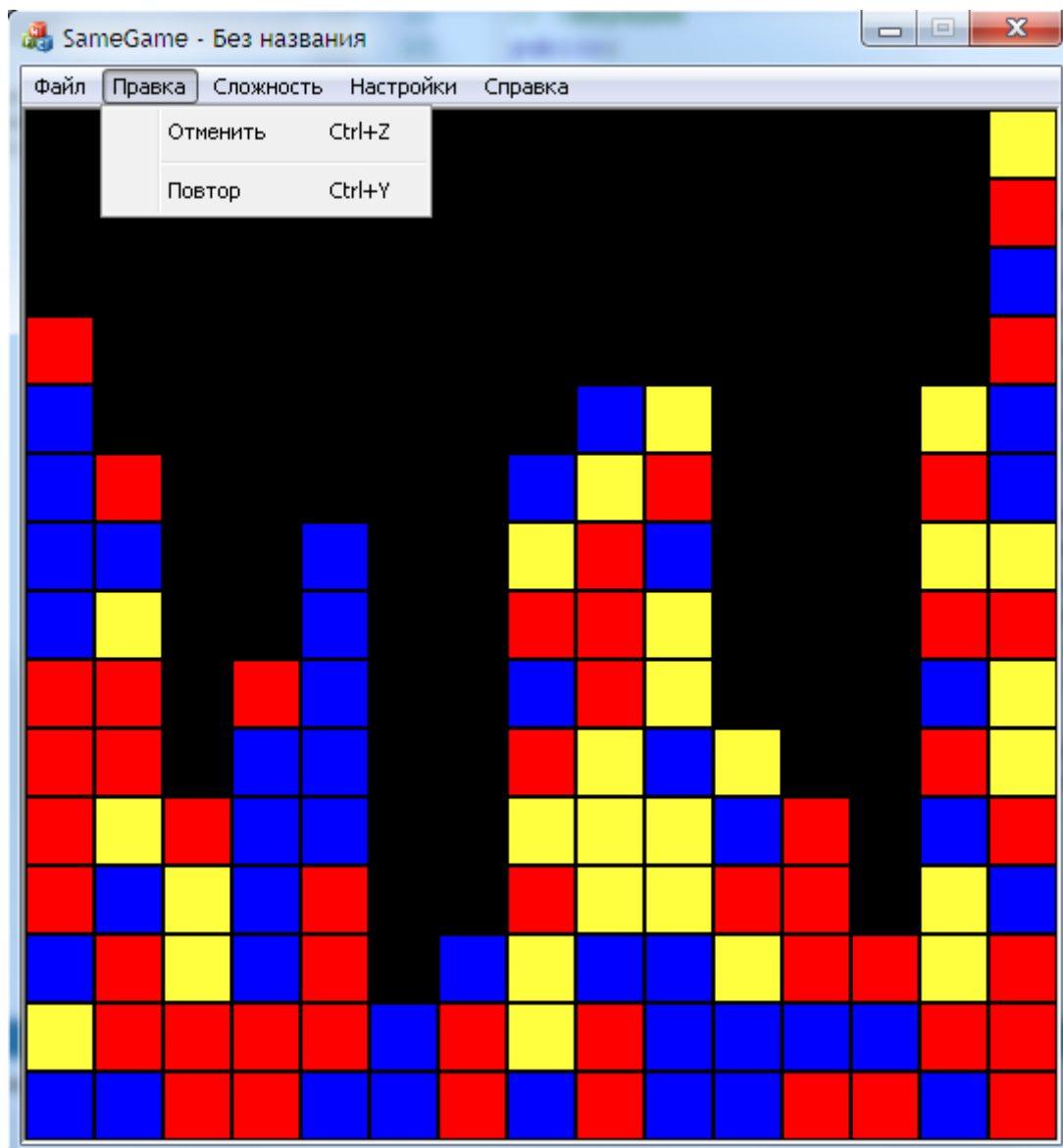
Теперь очередь дошла и до обработчиков событий для ON_UPDATE_COMMAND_UI. Новой здесь является функция Enable(), которая указывает на то, следует ли включить или отключить опцию меню на основе результата, который возвращает функция CanUndo() или CanRedo():

```

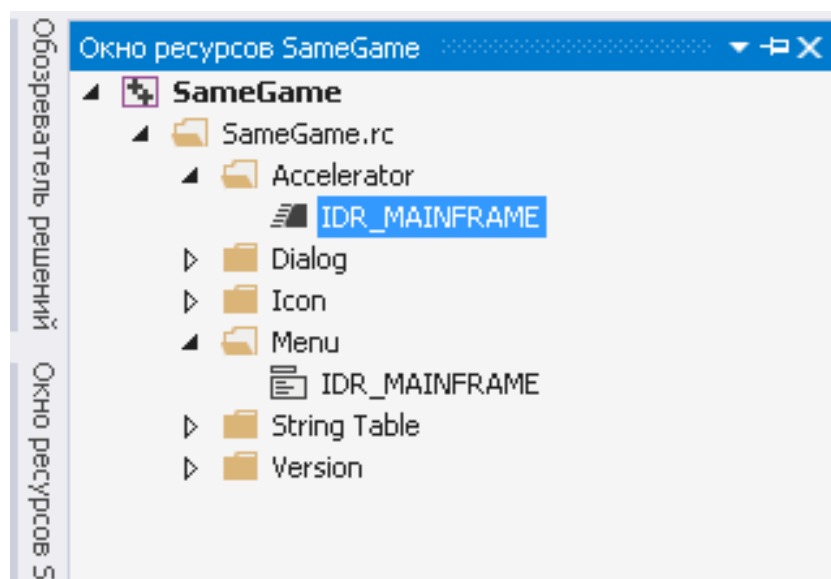
1 void CSameGameView::OnUpdateEditUndo(CCmndUI *pCmdUI)
2 {
3     // Сначала получаем указатель на Document
4     CSameGameDoc* pDoc = GetDocument();
5     ASSERT_VALID(pDoc);
6     if(!pDoc)
7         return;
8
9     // Включаем опцию, если она доступна
10    pCmdUI->Enable(pDoc->CanUndo());
11 }
12
13 void CSameGameView::OnUpdate32771 (CCmndUI *pCmdUI)
14 {
15     // Сначала получаем указатель на Document
16     CSameGameDoc* pDoc = GetDocument();
17     ASSERT_VALID(pDoc);
18     if(!pDoc)
19         return;
20
21     // Включаем опцию, если она доступна
22     pCmdUI->Enable(pDoc->CanRedo());
23 }

```

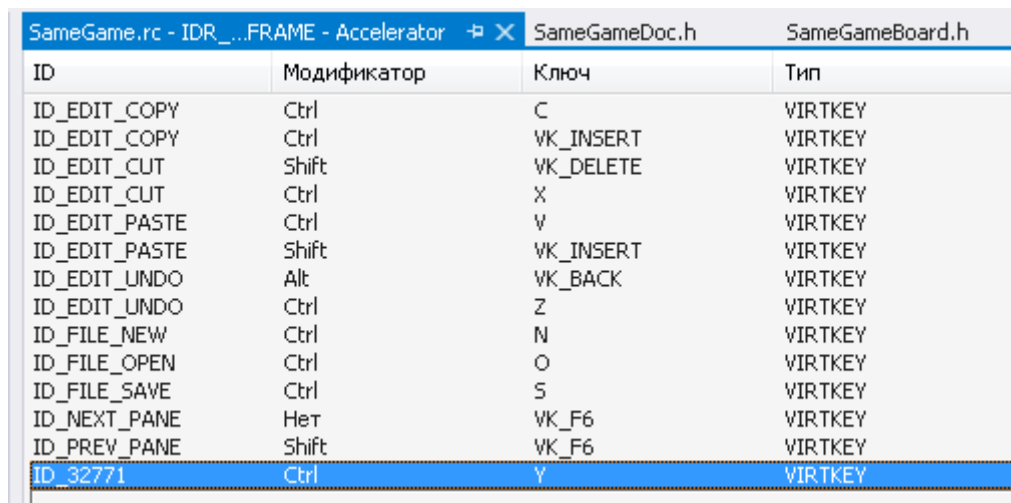
Стеки «Отмена/Повтор» теперь полностью функционируют. Вот примерно следующим образом должна выглядеть игра сейчас:



Акселераторы



Чтобы получить доступ к акселераторам, откройте «Редактор ресурсов» из меню "Вид" > "Другие окна" или нажать сочетание клавиш Ctrl+Shift+E. Затем открыть опцию акселератора в SameGame.rc и дважды щелкнуть по IDR_MAINFRAME, чтобы вызвать редактор акселератора. На следующем скриншоте мы добавили акселератор для команды «Повтор»:



ID	Модификатор	Ключ	Тип
ID_EDIT_COPY	Ctrl	C	VIRTKEY
ID_EDIT_COPY	Ctrl	VK_INSERT	VIRTKEY
ID_EDIT_CUT	Shift	VK_DELETE	VIRTKEY
ID_EDIT_CUT	Ctrl	X	VIRTKEY
ID_EDIT_PASTE	Ctrl	V	VIRTKEY
ID_EDIT_PASTE	Shift	VK_INSERT	VIRTKEY
ID_EDIT_UNDO	Alt	VK_BACK	VIRTKEY
ID_EDIT_UNDO	Ctrl	Z	VIRTKEY
ID_FILE_NEW	Ctrl	N	VIRTKEY
ID_FILE_OPEN	Ctrl	O	VIRTKEY
ID_FILE_SAVE	Ctrl	S	VIRTKEY
ID_NEXT_PANE	Нет	VK_F6	VIRTKEY
ID_PREV_PANE	Shift	VK_F6	VIRTKEY
ID_32771	Ctrl	Y	VIRTKEY

Чтобы добавить свой собственный акселератор, нажмите на пустую строку после последнего акселератора в столбце ID — это вызовет выпадающее меню, которое позволит выбрать ID_EDIT_32771 (идентификатор опции меню для команды повтора). Присвойте ему ключ Y и модификатор Ctrl (Ctrl+Y). Теперь скомпилируйте свою игру и запустите её. Мы только что добавили комбинацию клавиш, которая теперь отправляет ON_COMMAND в ID_EDIT_32771.

10. Заключение

Средствами библиотек MFC и STL C++ сделали полноценную игру, затронули множество тем, связанных с разработкой игр и работой с платформой Windows. С помощью библиотеки STL, был реализован функционал «Отмена/Повтор» в игре, позволяющий отменять или вернуть ход. Каждый ход (изменение данных доски) помещается в стек, и если мы хотим вернуть ход (Отмена/ STL+Z), то из стека выбирается последнее помещенное в стек состояние доски. Благодаря глубокому копированию, мы можем хранить большое количество ходов в стеке. Подробно о реализации функционала описано в 9 главе. Понимание работы с объектами в ООП помогает лучше структурировать и управлять объектами в коде и дает безграничные возможности.