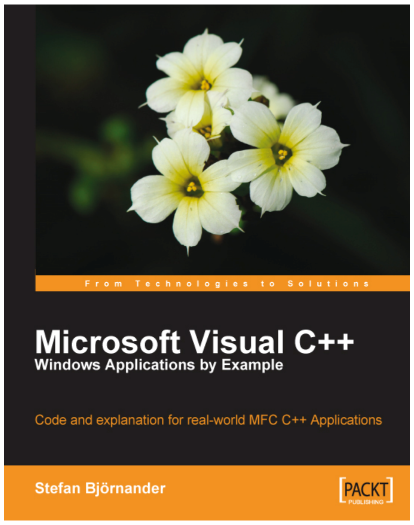
Microsoft Visual C++ Windows Applications by Example

Stefan Björnander


From Technologies to Solutions
Microsoft Visual C++
Windows Applications by Example
Code and explanation for real-world MFC C++ Applications
Stefan Björnander
PACKT PUBLISHING

Chapter No. 6

"The Tetris Application"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.6 "The Tetris Application"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Stefan Björnander is a Ph.D. candidate at Mälardalen University, Sweden. He has worked as a software developer and has taught as a senior lecturer at Umeå University, Sweden. He holds a master's degree in computer science and his research interests include compiler construction, mission-critical systems, and model-driven engineering. You can reach him at `stefan.bjornander@mdh.se`.

Microsoft Visual C++ Windows Applications by Example

This is a book about Windows application development in C++. It addresses some rather difficult problems that occur during the development of advanced applications. Most books in this genre have many short code examples. This one has only four main code examples, but rather extensive ones. They are presented in increasing complexity order. The simplest one is the *Tetris* application, which deals with graphics, timing, and message handling. The *Draw* application adds a generic coordinate system and introduces more complex applications states. The *Calc* application deals with formula interpretation and graph searching. Finally, in the *Word* application every character is allowed to hold its own font and size, resulting in a rather complex size and position calculation.

The book starts with an introduction to object-oriented programming in C++, followed by an overview of the Visual Studio environment with the *Ring* demonstration application as well as a presentation of some basic generic classes. Then the main applications are presented in one chapter each.

What This Book Covers

Chapter1. Introduction to C++—C++ is a language built on C. It is strongly typed; it has types for storing single as well as compound values. It supports dynamic memory management with pointers. It has a large set of operators to perform arithmetic, logical, and bitwise operations. The code can be organized into functions, and there is a pre-processor available, which can be used to define macros.

Chapter2. Object-oriented Programming in C++—C++ is an object-oriented language that fully supports the object-oriented model. The main feature of the language is the *class*, which can be instantiated into *objects*. A class can *inherit* another class. The inheritance can be *virtual*, which provides *dynamic binding*. A class can contain an object or have a *pointer* to another object. We can *overload operators* and we can throw *exceptions*. We can create generic classes by using *templates* and we can organize our classes into *namespaces*.

Chapter3. Windows Development—The development environment of this book is Microsoft Visual Studio, which holds several *Wizards* that generate skeleton code. With their help, we create a framework which we can add our own application specific code to. Microsoft Foundation Classes (MFC) is a powerful C++ class library built upon the Windows 32 bits Application Interface (Win32 API). It holds many classes to build and modify graphical Windows applications.

When an event occurs in Windows, a *message* is sent to the application in focus. When we want to paint or write in a window, we need a *device context*, which can be thought of both as painting toolbox and a connection to the painting canvas. When we develop an application such as a spreadsheet program, we want the users to be able to save their work. It can easily be obtained by *serialization*.

Chapter4. Ring: A Demonstration Example—As an introduction to the main applications of this book, we go through the step-by-step development process of a simple application that draws rings on the painting area of a window. The rings can be painted in different colors. We increase the painting area by using *scroll bars*. We increase the user-friendliness by introducing *menus*, *toolbars*, and *accelerators*. The RGB (Red, Green, Blue) standard can theoretically handle more than sixteen million colors. We use the *Color Dialog* to allow the user to handle them. Finally, we add serialization to our application.

Chapter5. Utility Classes—There are several generic classes available in MFC, we look into classes for handling points, sizes, and rectangles. However, some generic classes we have to write ourselves. We create classes to handle fonts, colors, and the caret. We also inherit MFC classes to handle lists and sets. Finally, we look into some appropriate error handling.

Chapter6. The Tetris Application—Tetris is a classic game. We have seven figures of different shapes and colors falling down. The player's task is to move and rotate them into appropriate positions in order to fill as many rows as possible. When a row is filled it disappears and the player gets credit. The game is over when it is not possible to add any more figures.

Chapter7. The Draw Application—In the Draw application, the users can draw lines, arrows, rectangles, and ellipses. They can move, resize, and change the color of the figures. They can cut and paste one or more figures, can fill the rectangles and ellipses, and can load and save a drawing. They can also write and modify text in different fonts.

Chapter8. The Calc Application—The Calc application is a spreadsheet program. The users can input text to the cells and they can change the text's font as well as its horizontal and vertical alignment. They can also load and save a spreadsheet and can cut and paste a block of cells. Furthermore, the user can input a formula into a cell. They can build expressions with the four arithmetic operators as well as parentheses.

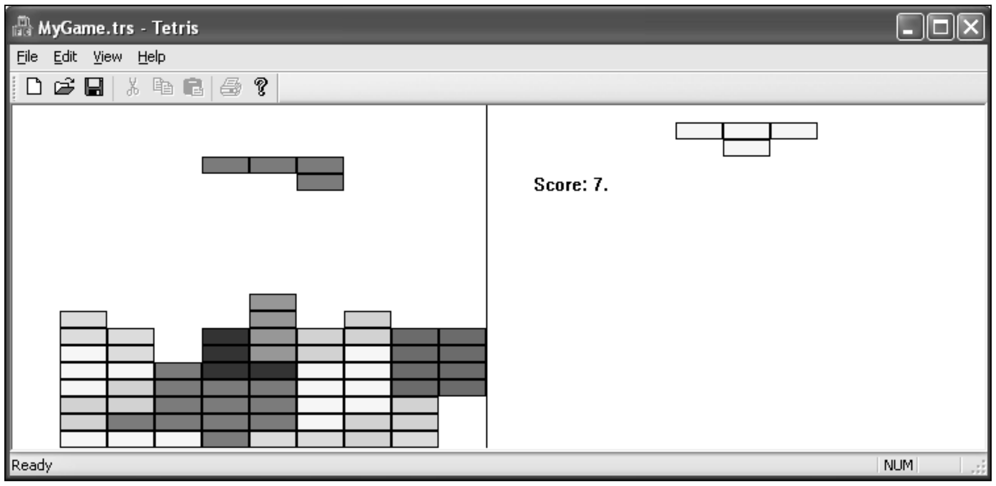
Chapter9. The Word Application—The Word application is a word processor program. The users can write and modify text in different fonts and with different horizontal alignment. The program has paragraph handling and a print preview function. The users can cut and paste blocks of text, they can also load and save a document.

6

The Tetris Application

Tetris is a classic game. In this chapter, we will develop a version very similar to the original version. Seven figures of different shapes and colors fall down and the player's job is to move and rotate them into positions so that as many rows as possible are completely filled. When a row is filled, it disappears. Every removed row gives one point.

This application is the only one in this book that supports the single document interface, which implies that we have one document class object and one view class object. The other applications support the multiple document interface, they have one document class object and zero or more view class objects. The following screenshot depicts a classic example of the Tetris Application:



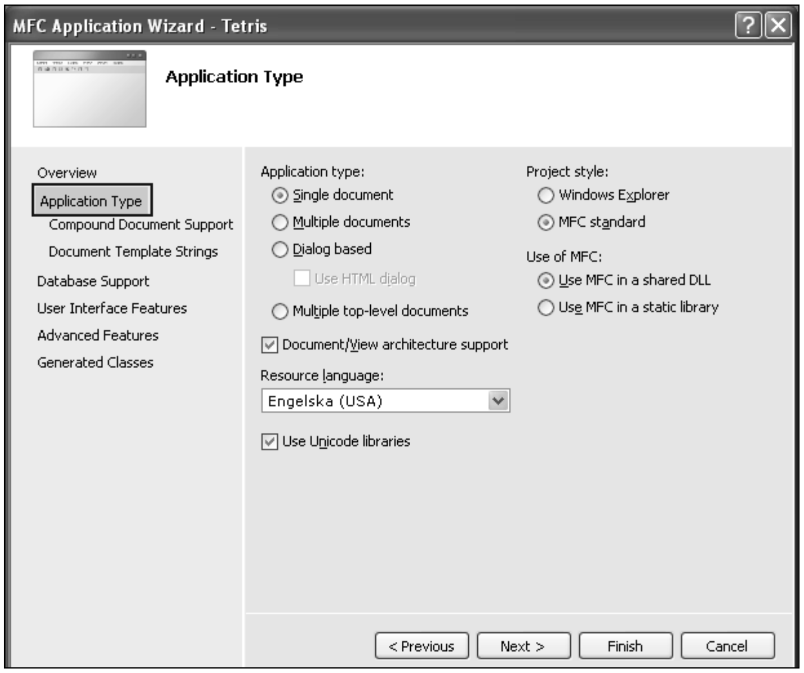
- We start by generating the application's skeleton code with The Application Wizard. The process is similar to the Ring application code.
- There is a small class `Square` holding the position of one square and a class `ColorGrid` managing the game grid.

- The `document` class manages the data of the game and handles the active (falling down) figure and the next (shown to the right of the game grid) figure.
- The `view` class accepts input from the keyboard and draws the figures and the game grid.
- The `Figure` class manages a single figure. It is responsible for movements and rotations.
- There are seven kinds of figures. The *Figure Info* files store information pertaining to their colors and shapes.

The Tetris Files

We start by creating a MFC application with the name Tetris and follow the steps of the Ring application. The classes `CTetrisApp`, `CMainFrame`, `CTetrisDoc`, `CTetrisView`, and `CAboutDlg` are then created and added to the project.

There are only two differences. We need to state that we are dealing with a "Single Document Application Type", that the file extension is "Trs" and that the file type long name is "A Game of Tetris". Otherwise, we just accept the default settings. Note that in this application we accept the `CView` base class instead of the `CScrollView` like we did in the Ring application.



Application Type

Application type:
[✓ Single document]
Multiple documents
Dialog based
Use HTML dialog
Multiple top-level documents

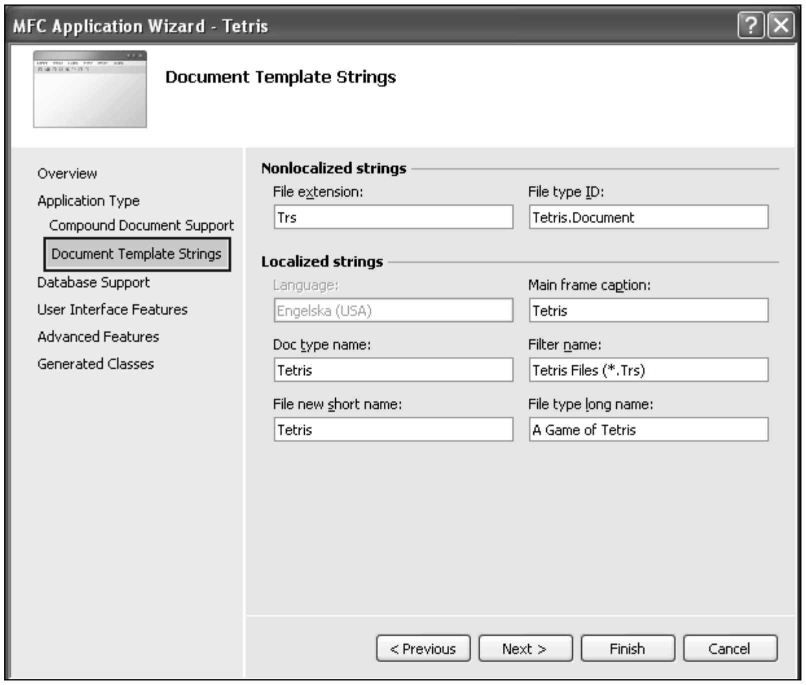
[✓] Document/View architecture support

Resource language: Engelska (USA)

[✓] Use Unicode libraries

Project style:
Windows Explorer
[✓ MFC standard]

Use of MFC:
[✓ Use MFC in a shared DLL]
Use MFC in a static library



Document Template Strings

Nonlocalized strings

File extension: Trs

File type ID: Tetris.Document

Localized strings

Language: Engelska (USA)

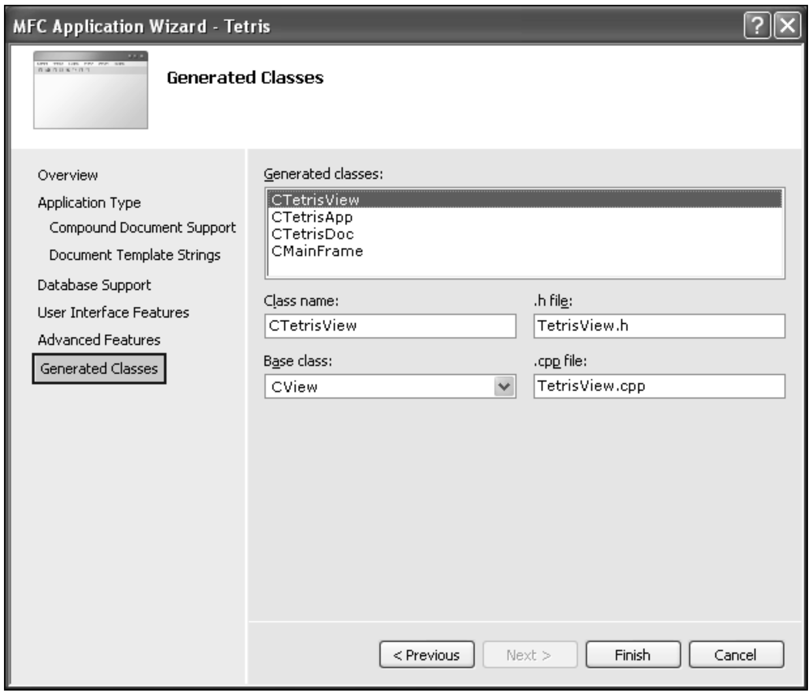
Main frame caption: Tetris

Doc type name: Tetris

Filter name: Tetris Files (*.Trs)

File new short name: Tetris

File type long name: A Game of Tetris



Generated Classes

Generated classes:

CTetrisView
CTetrisApp
CTetrisDoc
CMainFrame

Class name: CTetrisView

.h file: TetrisView.h

Base class: CView

.cpp file: TetrisView.cpp

We add the marked lines below. In all other respects, we leave the file unmodified. We will not need to modify the files `Tetris.h`, `MainFrm.h`, `MainFrm.cpp`, `StdAfx.h`, `StdAfx.cpp`, `Resource.h`, and `Tetris.rc`.

```
#include "stdafx.h"

#include "Square.h"
#include "Figure.h"
#include "ColorGrid.h"

#include "Tetris.h"
#include "MainFrm.h"
#include "TetrisDoc.h"
#include "TetrisView.h"

// ...
```

The Square Class

`Square` is a small class holding a row and column position. It is used by the `figureInfo` class in this application.

Square.h

```
class Square
{
  public:
    Square();
    Square(int iRow, int iCol);

    int Row() const {return m_iRow;}
    int Col() const {return m_iCol;}

  private:
    int m_iRow, m_iCol;
};
```

The Color Grid Class

The `ColorGrid` handles the background game grid of twenty rows and twenty columns. Each square can have a color. At the beginning, every square is initialized to the default color white. The `Index` method is overloaded with a constant version that returns the color of the given square, and a non-constant version that returns a reference to the color. The latter version makes it possible to change the color of a square.

ColorGrid.h

```
const int ROWS = 20;
const int COLS = 10;

class ColorGrid
{
  public:
    ColorGrid();
    void Clear();

    COLORREF& Index(int iRow, int iCol);
    const COLORREF Index(int iRow, int iCol) const;

    void Serialize(CArchive& archive);

  private:
    COLORREF m_buffer[ROWS * COLS];
};
```

There are two `Index` methods, the second one is intended to be called on a constant object. Both methods check that the given row and position have valid values. The checks are, however, for debugging purposes only. The methods are always called with valid values. Do not forget to include the file `StdAfx.h`.

ColorGrid.cpp

```
#include "StdAfx.h"

COLORREF& ColorGrid::Index(int iRow, int iCol)
{
  check((iRow >= 0) && (iRow < ROWS));
  check((iCol >= 0) && (iCol < COLS));
  return m_buffer[iRow * COLS + iCol];
}

const COLORREF ColorGrid::Index(int iRow, int iCol) const
{
  check((iRow >= 0) && (iRow < ROWS));
  check((iCol >= 0) && (iCol < COLS));
  return m_buffer[iRow * COLS + iCol];
}
```

The Document Class

`CTetrisDoc` is the document class of this application. When created, it overrides `OnNewDocument` and `Serialize` from its base class `CDocument`.

We add to the `CTetrisDoc` class a number of fields and methods. The field `m_activeFigure` is active figure, that is the one falling down during the game. The field `m_nextFigure` is the next figure, that is the one showed in the right part of the game view. They both are copies of the objects in the `m_figureArray`, which is an array figure object. There is one figure object of each kind (one figure of each color). The integer list `m_scoreList` holds the ten top list of the game. It is loaded from the file `ScoreList.txt` by the constructor and saved by the destructor. The integer field `m_iScore` holds the score of the current game. `GetScore`, `GetScoreList`, `GetActiveFigure`, `GetNextFigure`, and `GetGrid` are called by the `view` class in order to draw the game grid. They simply return the values of the corresponding fields.

The field `m_colorGrid` is an object of the class `ColorGrid`, which we defined in the previous section. It is actually just a matrix holding the colors of the squares of the game grid. Each square is intialized to the color white and a square is considered to be empty as long as it is white.

When the application starts, the constructor calls the C standard library function `srand`. The name is an abbreviation for sowing a random seed. By calling `srand` with an integer seed, it will generate a series of random number. In order to find a new seed every time the application starts, the C standard library function `time` is called, which returns the number of seconds elapsed since January 1, 1970. In order to obtain the actual random number, we call `rand` that returns a number in the interval from zero to the predefined constant `RAND_MAX`. The prototypes for these functions are defined in `time.h` (`time`) and `stdlib.h` (`rand` and `srand`), respectively.

```
#include <time.h>
#include <stdlib.h>
time_t time(time_t *pTimer);
void srand(unsigned int uSeed);
int rand();
```

When the user presses the space key and the active figure falls down or when a row is filled and is flashed, we have to slow down the process in order for the user to apprehand the event. There is a Win32 API function `Sleep` that pauses the application for the given amount of milliseconds.

```
void Sleep(int iMilliSeconds);
```

The user can control the horizontal movement and rotation of the falling figures by pressing the arrow keys. Left and right arrow keys move the figure to the left or right. The up and down arrow key rotates the figure clockwise or counterclockwise, respectively. Every time the user presses one of those keys, a message is sent to the view class object and caught by the method `OnKeyDown`, which in turn calls one of the methods `LeftArrowKey`, `RightArrowKey`, `UpArrowKey`, `DownArrowKey` to deal with the message. They all work in a similar fashion. They try to execute the movement or rotation in question. If it works, both the old and new area of the figure is repainted by making calls to `UpdateAllViews`.

The `view` class also handles a timer that sends a message every second the view is in focus. The message is caught by the view class method `OnTimer` that in turn calls `Timer`. It tries to move the active figure one step downwards. If that is possible, the area of the figure is repainted in the same way as in the methods above. However, if it is not possible, the squares of the figure are added to the game grid. The active figure is assigned to the next figure, and the next figure is assigned a copy of a randomly selected figure in `m_figureArray`. We also check whether any row has been filled. In that case, it will be removed and we will check to see if the game is over.

The user can speed up the game by pressing the space key. The message is caught and sent to `SpaceKey`. It simply calls `OnTimer` as many times as possible at intervals of twenty milliseconds in order to make the movement visible to the user.

When a figure has reached its end position and any full rows have been removed, the figure must be valid. That is, its squares are not allowed to occupy any already colored position. If it does, the game is over and `GameOver` is called. It starts by making the game grid gray and asks the users whether they want to play another game. If they do, the game grid is cleared and set back to colored mode and a new game starts. If they do not, the application exits.

`NewGame` informs the players whether they made to the top ten list and inquires about another game by displaying a message box. `AddToScore` examines whether the player has made to the ten top list. If so, the score is added to the list and the ranking is returned, if not, zero is returned.

`DeleteFullRows` traverses the game grid from top to bottom flashing and removing every full row. `IsRowFull` traverses the given row and returns true if no square has the default color (white). `FlashRow` flashes the row by showing it three times in grayscale and color at intervals of twenty milliseconds. `DeleteRow` removes the row by moving all rows above one step downwards and inserting an empty row (all white squares) at top.

The next figure and the current high score are painted at specific positions on the client area, the rectangle constants `NEXT_AREA` and `SCORE_AREA` keep track of those positions.

TetrisDoc.h

```
typedef CList<int> IntList;
const int FIGURE_ARRAY_SIZE = 7;
class CTetrisDoc : public CDocument
{
  protected:
    CTetrisDoc();
  public:
    virtual ~CTetrisDoc();
    void SaveScoreList();
  protected:
    DECLARE_MESSAGE_MAP()
    DECLARE_DYNCREATE(CTetrisDoc)
  public:
    virtual void Serialize(CArchive& archive);
    int GetScore() const {return m_iScore;}
    const IntList* GetScoreList() {return &m_scoreList;}
    const ColorGrid* GetGrid() {return &m_colorGrid;}
    const Figure& GetActiveFigure() const {return
                                          m_activeFigure;}
    const Figure& GetNextFigure() const {return m_nextFigure;}
  public:
    void LeftArrowKey();
    void RightArrowKey();
    void UpArrowKey();
    void DownArrowKey();
    BOOL Timer();
    void SpaceKey();
  private:
    void GameOver();
    BOOL NewGame();
    int AddScoreToList();
    void DeleteFullRows();
    BOOL IsRowFull(int iRow);
    void FlashRow(int iFlashRow);
    void DeleteRow(int iDeleteRow);
  private:
    ColorGrid m_colorGrid;
    Figure m_activeFigure, m_nextFigure;
    int m_iScore;
    IntList m_scoreList;
```

```
        const CRect NEXT_AREA, SCORE_AREA;
        static Figure m_figureArray[FIGURE_ARRAY_SIZE];
    };
```

The field `m_figureArray` holds seven figure objects, one of each color. When we need a new figure, we just randomly copy one of them.

TetrisDoc.cpp

```
    Figure redFigure(NORTH, RED, RedInfo);
    Figure brownFigure(EAST, BROWN, BrownInfo);
    Figure turquoiseFigure(EAST, TURQUOISE, TurquoiseInfo);
    Figure greenFigure(EAST, GREEN, GreenInfo);
    Figure blueFigure(SOUTH, BLUE, BlueInfo);
    Figure purpleFigure(SOUTH, PURPLE, PurpleInfo);
    Figure yellowFigure(SOUTH, YELLOW, YellowInfo);
    Figure CTetrisDoc::m_figureArray[] = {redFigure, brownFigure,
                       turquoiseFigure, greenFigure, yellowFigure,
                       blueFigure, purpleFigure};
```

When the user presses the left arrow key, the view class object catches the message and calls `LeftArrowKey` in the document class object. We try to move the active figure one step to the left. It is not for sure that we succeed. The figure may already be located at the left part of the game grid. However, if the movement succeeds, the figure's position is repainted and true is returned. In that case, we repaint the figure's old and new graphic areas in order to repaint the figure. Finally, we set the modified flag since the figure has been moved. The method `RightArrowKey` works in a similar way.

```
    void CTetrisDoc::LeftArrowKey()
    {
      CRect rcOldArea = m_activeFigure.GetArea();
      if (m_activeFigure.MoveLeft())
      {
        CRect rcNewArea = m_activeFigure.GetArea();
        UpdateAllViews(NULL, COLOR, (CObject*) &rcOldArea);
        UpdateAllViews(NULL, COLOR, (CObject*) &rcNewArea);
        SetModifiedFlag();
      }
    }
```

Timer is called every time the active figure is to moved one step downwards. That is, each second when the application has focus. If the downwards movement succeeds, then the figure is repainted in a way similar to LeftArrowKey above. However, if the movement does not succeed, the movement of the active figure has come to an end. We call AddToGrid to color the squares of the figure. Then we copy the next figure to the active figure and randomly copy a new next figure. The next figure is the one shown to the right of the game grid.

However, the case may occur that the game grid is full. That is the case if the new active figure is not valid, that is, the squares occupied by the figure are not free. If so, the game is over, and the user is asked whether he wants a new game.

```
BOOL CTetrisDoc::Timer()
{
  SetModifiedFlag();
  CRect rcOldArea = m_activeFigure.GetArea();

  if (m_activeFigure.MoveDown())
  {
    CRect rcNewArea = m_activeFigure.GetArea();

    UpdateAllViews(NULL, COLOR, (CObject*) &rcOldArea);
    UpdateAllViews(NULL, COLOR, (CObject*) &rcNewArea);

    return TRUE;
  }
  else
  {
    m_activeFigure.AddToGrid();
    m_activeFigure = m_nextFigure;
    CRect rcActiveArea = m_activeFigure.GetArea();
    UpdateAllViews(NULL, COLOR, (CObject*) &rcActiveArea);
    m_nextFigure = m_figureArray[rand() % FIGURE_ARRAY_SIZE];
    UpdateAllViews(NULL, COLOR, (CObject*) &NEXT_AREA);
    DeleteFullRows();

    if (!m_activeFigure.IsFigureValid())
    {
      GameOver();
    }
    return FALSE;
  }
}
```

If the user presses the space key, the active figure falling will fall faster. The `Timer` method is called every 20 milisseconds.

```
void CTetrisDoc::SpaceKey()
{
  while (Timer())
  {
    Sleep(20);
  }
}
```

When the game is over, the users are asked whether they want a new game. If so, we clear the grid, randomly select the the next active and next figure, and repaint the whole client area.

```
void CTetrisDoc::GameOver()
{
  UpdateAllViews(NULL, GRAY);
  if (NewGame())
  {
    m_colorGrid.Clear();
    m_activeFigure = m_figureArray[rand() %FIGURE_ARRAY_SIZE];
    m_nextFigure = m_figureArray[rand() % FIGURE_ARRAY_SIZE];
    UpdateAllViews(NULL, COLOR);
  }
  else
  {
    SaveScoreList();
    exit(0);
  }
}
```

Each time a figure is moved, one or more rows may be filled. We start by checking the top row and then go through the rows downwards. For each full row, we first flash it and then remove it.

```
void CTetrisDoc::DeleteFullRows()
{
  int iRow = ROWS - 1;
  while (iRow >= 0)
  {
    if (IsRowFull(iRow))
    {
      FlashRow(iRow);
      DeleteRow(iRow);
      ++m_iScore;
      UpdateAllViews(NULL, COLOR, (CObject*) &SCORE_AREA);
```

```
        }
        else
        {
          --iRow;
        }
      }
    }
```

When a row is completely filled, it will flash before it is removed. The flash effect is executed by redrawing the row in color and in grayscale three times with an interval of 50 milliseconds.

```
void CTetrisDoc::FlashRow(int iRow)
{
  for (int iCount = 0; iCount < 3; ++iCount)
  {
    CRect rcRowArea(0, iRow, COLS, iRow + 1);
    UpdateAllViews(NULL, GRAY, (CObject*) &rcRowArea);
    Sleep(50);
    CRect rcRowArea2(0, iRow, COLS, iRow + 1);
    UpdateAllViews(NULL, COLOR, (CObject*) &rcRowArea2);
    Sleep(50);
  }
}
```

When a row is removed, we do not really remove it. If we did, the game grid would shrink. Instead, we copy the squares above it and clear the top row.

```
void CTetrisDoc::DeleteRow(int iMarkedRow)
{
  for (int iRow = iMarkedRow; iRow > 0; --iRow)
  {
    for (int iCol = 0; iCol < COLS; ++iCol)
    {
      m_colorGrid.Index(iRow, iCol) =
      m_colorGrid.Index(iRow - 1, iCol);
    }
  }
  for (int iCol = 0; iCol < COLS; ++iCol)
  {
    m_colorGrid.Index(0, iCol) = WHITE;
  }
  CRect rcArea(0, 0, COLS, iMarkedRow + 1);
  UpdateAllViews(NULL, COLOR, (CObject*) &rcArea);
}
```

The View Class

CTetrisView is the view class of the application. It receives system messages and (completely or partly) redraws the client area.

The field m_iColorStatus holds the painting status of the view. Its status can be either color or grayscale. The color status is the normal mode, m_iColorStatus is initialized to color in the constructor. The grayscale is used to flash rows and to set the game grid in grayscale while asking the user for another game.

OnCreate is called after the view has been created but before it is shown. The field m_pTetrisDoc is set to point at the document class object. It is also confirmed to be valid. OnSize is called each time the size of the view is changed. It sets the global variables g_iRowHeight and g_iColWidth (defined in Figure.h), which are used by method of the Figure and ColorGrid classes to paint the squares of the figures and the grid.

OnSetFocus and OnKillFocus are called when the view receives and loses the input focus. Its task is to handle the timer. The idea is that the timer shall continue to send timer messages every second as long as the view has the input focus. Therefore, OnSetFocus sets the timer and OnKillFocus kills it. This arrangement implies that OnTimer is called each second the view has input focus.

In Windows, the timer cannot be turned off temporarily; instead, we have to set and kill it. The base class of the view, CWnd, has two methods: SetTimer that initializes a timer and KillTimer that stops the timer. The first parameter is a unique identifier to distinguish this particular timer from any other one. The second parameter gives the time interval of the timer, in milliseconds. When we send a null pointer as the third parameter, the timer message will be sent to the view and caught by OnTimer. KillTimer simply takes the identity of the timer to finish.

```
UINT_PTR SetTimer(UINT_PTR iIDEvent, UINT iElapse,
                  void (CALLBACK* lpfnTimer)
                       (HWND, UINT, UINT_PTR, DWORD));
BOOL KillTimer(UINT_PTR nIDEvent);
```

OnKeyDown is called every time the user presses a key on the keyboard. It analizes the pressed key and calls suitable methods in the document class if the left, right, up, or down arrow key or the space key is pressed.

When a method of the document class calls `UpdateAllViews`, `OnUpdate` of the view class object connected to the document object is called. As this is a single view application, the application has only one view object on which `OnUpdate` is called. `UpdateAllViews` takes two extra parameters, hints, which are sent to `OnUpdate`. The first hint tells us whether the next repainting shall be done in color or in grayscale, the second hint is a pointer to a rectangle holding the area that is to be repainted. If the pointer is not null, we calculate the area and repaint it. If it is null, the whole client area is repainted.

`OnUpdate` is also called by `OnInitialUpdate` of the base class `CView` with both hints set to zero. That is not a problem because the `COLOR` constant is set to zero. The effect of this call is that the whole view is painted in color.

`OnUpdate` calls `UpdateWindow` in `CView` that in turn calls `OnPaint` and `OnDraw` with a device context. `OnPaint` is also called by the system when the view (partly or completely) needs to be repainted. `OnDraw` loads the device context with a black pen and then draws the grid, the score list, and´the active and next figures.

TetrisView.h

```
const int TIMER_ID = 0;
enum {COLOR = 0, GRAY = 1};

class CTetrisDoc;
COLORREF GrayScale(COLORREF rfColor);
class CTetrisView : public CView
{
  protected:
    CTetrisView();

    DECLARE_DYNCREATE(CTetrisView)
    DECLARE_MESSAGE_MAP()

  public:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnSize(UINT nType, int iClientWidth,
                        int iClientHeight);

    afx_msg void OnSetFocus(CWnd* pOldWnd);
    afx_msg void OnKillFocus(CWnd* pNewWnd);
    afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt,
                           UINT nFlags);
    afx_msg void OnTimer(UINT nIDEvent);

    void OnUpdate(CView* /* pSender */, LPARAM lHint,
                  CObject* pHint);
    void OnDraw(CDC* pDC);
  private:
    void DrawGrid(CDC* pDC);
    void DrawScoreAndScoreList(CDC* pDC);
    void DrawActiveAndNextFigure(CDC* pDC);
  private:
    CTetrisDoc* m_pTetrisDoc;
    int m_iColorStatus;
};
```

TetrisView.cpp

This application catches the messsages `WM_CREATE`, `WM_SIZE`, `WM_SETFOCUS`, `WM_KILLFOCUS`, `WM_TIMER`, and `WM_KEYDOWN`.

```
BEGIN_MESSAGE_MAP(CTetrisView, CView)
  ON_WM_CREATE()
  ON_WM_SIZE()
  ON_WM_SETFOCUS()
  ON_WM_KILLFOCUS()
  ON_WM_TIMER()
  ON_WM_KEYDOWN()
END_MESSAGE_MAP()
```

When the view object is created, is connected to the document object by the pointer `m_pTetrisDoc`.

```
int CTetrisView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  // We check that the view has been correctly created.
  if (CView::OnCreate(lpCreateStruct) == -1)
  {
    return -1;
  }
  m_pTetrisDoc = (CTetrisDoc*) m_pDocument;
  check(m_pTetrisDoc != NULL);
  ASSERT_VALID(m_pTetrisDoc);
  return 0;
}
```

The game grid is dimensioned by the constants `ROWS` and `COLS`. Each time the user changes the size of the application window, the global variables `g_iRowHeight` and `g_iColWidth`, which are defined in `Figure.h`, store the height and width of one square in pixels.

```
void CTetrisView::OnSize(UINT /* uType */,int iClientWidth,
                                          int iClientHeight)
{
  g_iRowHeight = iClientHeight / ROWS;
  g_iColWidth = (iClientWidth / 2) / COLS;
}
```

`OnUpdate` is called by the system when the window needs to be (partly or completely) repainted. In that case, the parameter `pHint` is zero and the whole client area is repainted. However, this method is also indirectly called when the document class calls `UpdateAllView`. In that case, `lHint` has the value color or gray, depending on whether the client area shall be repainted in color or in a grayscale.

If `pHint` is non-zero, it stores the coordinates of the area to be repainted. The coordinates are given in grid coordinates that have to be translated into pixel coordinates before the area is invalidated.

The method first calls `Invalidate` or `InvalidateRect` to define the area to be repainted, then the call to `UpdateWindow` does the actual repainting by calling `OnPaint` in `CView`, which in turn calls `OnDraw` below.

```
void CTetrisView::OnUpdate(CView* /* pSender */, LPARAM lHint,
                           CObject* pHint)
{
  m_iColorStatus = (int) lHint;
  if (pHint != NULL)
  {
    CRect rcArea = *(CRect*) pHint;
    rcArea.left *= g_iColWidth;
    rcArea.right *= g_iColWidth;
    rcArea.top *= g_iRowHeight;
    rcArea.bottom *= g_iRowHeight;
    InvalidateRect(&rcArea);
  }
  else
  {
    Invalidate();
  }
  UpdateWindow();
}
```

`OnDraw` is called when the client area needs to be repainted, by the system or by `UpdateWindow` in `OnUpdate`. It draws a vertical line in the middle of the client area, and then draws the game grid, the high score list, and the current figures.

```
void CTetrisView::OnDraw(CDC* pDC)
{
  CPen pen(PS_SOLID, 0, BLACK);
  CPen* pOldPen = pDC->SelectObject(&pen);
  pDC->MoveTo(COLS * g_iColWidth, 0);
  pDC->LineTo(COLS * g_iColWidth, ROWS * g_iRowHeight);
  DrawGrid(pDC);
  DrawScoreAndScoreList(pDC);
  DrawActiveAndNextFigure(pDC);
  pDC->SelectObject(&pOldPen);
}
```

`DrawGrid` traverses through the game grid and paints each non-white square. If a square is not occupied, it has the color white and it not painted. The field `m_iColorStatus` decides whether the game grid shall be painted in color or in grayscale.

```
void CTetrisView::DrawGrid(CDC* pDC)
{
  const ColorGrid* pGrid = m_pTetrisDoc->GetGrid();
  for (int iRow = 0; iRow < ROWS; ++iRow)
  {
    for (int iCol = 0; iCol < COLS; ++iCol)
    {
      COLORREF rfColor = pGrid->Index(iRow, iCol);
      if (rfColor != WHITE)
      {
        CBrush brush((m_iColorStatus == COLOR)
               ? rfColor :GrayScale(rfColor));
        CBrush* pOldBrush = pDC->SelectObject(&brush);
        DrawSquare(iRow, iCol, pDC);
        pDC->SelectObject(pOldBrush);
      }
    }
  }
}
```

`GrayScale` returns the grayscale of the given color, which is obtained by mixing the average of the red, blue, and green component of the color.

```
COLORREF GrayScale(COLORREF rfColor)
{
  int iRed = GetRValue(rfColor);
  int iGreen = GetGValue(rfColor);
  int iBlue = GetBValue(rfColor);
  int iAverage = (iRed + iGreen + iBlue) / 3;
  return RGB(iAverage, iAverage, iAverage);
}
```

The active figure (`m_activeFigure`) is the figure falling down on the game grid. The next figure (`m_nextFigure`) is the figure announced at the right side of the client area. In order for it to be painted at the right-hand side, we alter the origin to the middle of the client area, and one row under the upper border by calling `SetWindowOrg`.

```
void CTetrisView::DrawActiveAndNextFigure(CDC* pDC)
{
  const Figure activeFigure = m_pTetrisDoc->GetActiveFigure();
  activeFigure.Draw(m_iColorStatus, pDC);

  const Figure nextFigure = m_pTetrisDoc->GetNextFigure();
  CPoint ptOrigin(-COLS * g_iColWidth, -g_iRowHeight);
  pDC->SetWindowOrg(ptOrigin);
  nextFigure.Draw(m_iColorStatus, pDC);
}
```

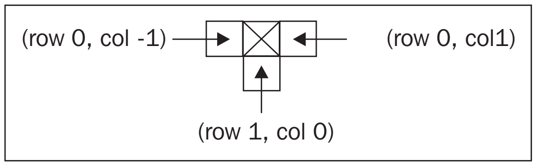
The Figure Class

All figures can be moved to the left or the right as well as be rotated clockwise or counterclockwise as a response to the user's requests. They can also be moved downwards as a response to the timer. The crossed square in the figures of this section marks the center of the figure, that is, the position the fields `m_iRow` and `m_iCol` of the `Figure` class refer to.

All kinds of figures are in fact objects of the `Figure` class. What differs between the figures are their colors and their shapes. The files `FigureInfo.h` and `FigureInfo.cpp` holds the information specific for each kind of figure, see the next section.

The field `m_rfColor` holds the color of the figure, `m_pColorGrid` is a pointer to the color grid of the game grid, `m_iRow`, `m_iCol`, and `m_iDirection` are the positions and the directions of the figure, respectively. The figure can be rotated into the directions north, east, south, and west. However, the red figure is a square, so it cannot be rotated at all. Moreover, the brown, turquoise, and green figures can only be rotated into vertical and horizontal directions, which implies that the north and south directions are the same for these figures, as are the east and west directions.

The second constructor takes a parameter of the type `FigureInfo`, which holds the shape of the figure in all four directions. They hold the position of the squares of the figure relative to the middle squares referred to by `m_iRow` and `m_iCol` for each of the four directions. The `FigureInfo` type consists of four arrays, one for each direction. The arrays in turn hold four positions, one for each square of the figure. The first position is always zero since it refers to the center square. For instance, let us look at the yellow figure in south direction.



The crossed square above is the one referred to by `m_iRow` and `m_iCol`. The south array for the yellow figure is initialized as follows.

```
SquareArray YellowSouth = {Square(0, 0), Square(0, -1),
                           Square(1, 0), Square(0, 1)};
```

The first square object refers to the center square, so it always holds zero. The other square objects holds the position of one square each relative to the center square. The second square object refers to the square to the left of the center square in the figure. Note that the row numbers increase downward and the column numbers increase to the right. Therefore, the relative column is negative. The third square object refers to the square below the crossed one, one row down and the same column, and the fourth square object refers to the square to the right, the same row and one column to the right.

The methods `RotateClockwiseOneQuarter` and `RotateCounterclockwiseOneQuarter` move the direction 90 degrees. `MoveLeft`, `MoveRight`, `RotateClockwise`, `RotateCounterclockwise`, and `MoveDown` all works in the same way. They execute the operation in question, test whether the figure is still valid (its squares are not already occupied), and return true if it is. Otherwise, they undo the operation and return false. Again, note that row numbers increase downwards and column numbers increase to the right.

IsSquareValid tests whether the given position is on the game grid and not occupied by a color other then white. IsFigureValid tests whether the four squares of the whole figure are valid at their current position and in their current direction.

GetArea returns the area currently occupied by the figure. Note that the area is returned in color grid coordinates (rows and columns). The coordinates are translated into pixel coordinates by OnUpdate in the view class before the figure is repainted.

When a figure is done falling, its squares shall be added to the grid. AddToGrid takes care of that, it sets the color of this figure to the squares currently occupied of the figure in the color grid.

Draw is called by the view class when the figure needs to be redrawn. It draws the four squares of the figure in color or grayscale. DrawSquare is called by Draw and does the actual drawing of each square. It is a global function because it is also called by the ColorGrid class to draw the squares of the grid. The global variables g_iRowHeight and g_iColWidth are set by the view class method OnSize every time the user changes the size of the view. They are used to calculate the positions and dimensions of the squares in DrawSquare.

Serialize stores and loads the current row, column, and direction of the figure as well as its color. It also writes and reads the four direction arrays.

The two global C standard library methods memset and memcpy come in handy when we want to copy a memory block or turn it to zero. They are used by the constructors to copy the directions arrays and turn them to zero.

```
void *memset(void* pDestination, int iValue, size_t iSize);
void *memcpy(void* pDestination, const void* pSource,
             size_t iSize);
```

Figure.h

```
const COLORREF BLACK = RGB(0, 0, 0);
const COLORREF WHITE = RGB(255, 255, 255);
const COLORREF DEFAULT_COLOR = WHITE;

class ColorGrid;
extern int g_iRowHeight, g_iColWidth;
enum {NORTH = 0, EAST = 1, SOUTH = 2, WEST = 3};

const int SQUARE_ARRAY_SIZE = 4;
const int SQUARE_INFO_SIZE = 4;

typedef Square SquareArray[SQUARE_ARRAY_SIZE];
typedef SquareArray SquareInfo[SQUARE_INFO_SIZE];
```

```
class Figure
{
  public:
    Figure();
    Figure(int iDirection, COLORREF rfColor,
           const SquareInfo& squareInfo);
    Figure operator=(const Figure& figure);
    void SetColorGrid(ColorGrid* pColorGrid) {m_pColorGrid =
                                              pColorGrid;};
  private:
    BOOL IsSquareValid(int iRow, int iCol) const;
  public:
    BOOL IsFigureValid() const;
    BOOL MoveLeft();
    BOOL MoveRight();
  private:
    void RotateClockwiseOneQuarter();
    void RotateCounterclockwiseOneQuarter();
  public:
    BOOL RotateClockwise();
    BOOL RotateCounterclockwise();
    BOOL MoveDown();
    void AddToGrid();
    CRect GetArea() const;
  public:
    void Draw(int iColorStatus, CDC* pDC) const;
    friend void DrawSquare(int iRow, int iCol, CDC* pDC);
  public:
    void Serialize(CArchive& archive);
  private:
    COLORREF m_rfColor;
    ColorGrid* m_pColorGrid;
    int m_iRow, m_iCol, m_iDirection;
    SquareInfo m_squareInfo;
};
typedef CArray<const Figure> FigurePtrArray;
```

Figure.cpp

`Figure.cpp` the main constructor. It initializes the direction (north, east, south, or west), the color (red, brown, turquoise, green, yellow, blue, or purple), the pointer to `ColorGrid`, and the specific figure information. The red figure sub class will initialize all four direction arrays with the same values because it cannot be rotated. The brown, turquoise, and green figure sub classes will initialize both the north and south arrays to its vertical direction as well as the east and west directions to its horizontal direction. Finally, the yellow, blue, and purple figure sub classes will initialize all four arrays with different values because they can be rotated in all four directions.

The C standard funtion `memcpy` is used to copy the figure specific information.

```
Figure::Figure(int iDirection, COLORREF rfColor,
               const SquareInfo & squareInfo)
 :m_iRow(0),
  m_iCol(COLS / 2),
  m_iDirection(iDirection),
  m_rfColor(rfColor),
  m_pColorGrid(NULL)
{
  ::memcpy(&m_squareInfo, &squareInfo, sizeof m_squareInfo);
}
```

`IsSquareValid` is called by `IsFigureValid` below. It checks whether the given square is on the grid and that it is not already occupied by another color.

```
BOOL Figure::IsSquareValid(int iRow, int iCol) const
{
  return (iRow >= 0) && (iRow < ROWS) &&
         (iCol >= 0) && (iCol < COLS) &&
         (m_pColorGrid->Index(iRow, iCol) == DEFAULT_COLOR);
}
```

`IsFigureValid` checks whether the figure is at a valid position by examining the four squares of the figure. It is called by `MoveLeft`, `MoveRight`, `Rotate`, and `MoveDown` below:

```
BOOL Figure::IsFigureValid() const
{
  SquareArray* pSquareArray = m_squareInfo[m_iDirection];
  for (int iIndex = 0; iIndex < SQUARE_ARRAY_SIZE; ++iIndex)
  {
    Square& square = (*pSquareArray)[iIndex];
    if (!IsSquareValid(m_iRow + square.Row(), m_iCol + square.Col()))
```

```
        {
          return FALSE;
        }
      }
    return TRUE;
  }
```

`RotateClockwiseOneQuarter` rotates the direction clockwise one quarter of a complete turn. `RotateCounterclockwiseOneQuarter` works in a similar way.

```
void Figure::RotateClockwiseOneQuarter()
{
    switch (m_iDirection)
    {
      case NORTH:
        m_iDirection = EAST;
        break;
      case EAST:
        m_iDirection = SOUTH;
        break;
      case SOUTH:
        m_iDirection = WEST;
        break;
      case WEST:
        m_iDirection = NORTH;
        break;
    }
}
```

`MoveLeft` moves the figure one step to the left. If the figure then is valid it returns true. If it is not, it puts the figure to back in origional position and returns false. `MoveRight`, `RotateClockwise`, `RotateCounterclockwise`, and `MoveDown` work in a similar way. Remember that the rows increase downwards and the columns increase to the right.

```
BOOL Figure::MoveLeft()
{
    --m_iCol;
    if (IsFigureValid())
    {
      return TRUE;
    }
    else
    {
      ++m_iCol;
      return FALSE;
    }
}
```

`AddToGrid` is called by the document class when the figure cannot be moved another step downwards. In that case, a new figure is introduced and the squares of the figure are added to the grid, that is, the squares currently occupied by the figure are the to the figure's color.

```
void Figure::AddToGrid()
{
  SquareArray* pSquareArray = m_squareInfo[m_iDirection];
  for (int iIndex = 0; iIndex < SQUARE_ARRAY_SIZE; ++iIndex)
  {
    Square& square = (*pSquareArray)[iIndex];
    m_pColorGrid->Index(m_iRow + square.Row(),
                        m_iCol + square.Col()) = m_rfColor;
  }
}
```

When a figure has been moved and rotated, it needs to be repainted. In order to do so without having to repaint the whole game grid we need the figures area. We calculate it by comparing the values of the squares of the figure in its current direction. The rectangle returned holds the coordinates of the squares, not pixel coordinates. The translation is done by `OnUpdate` in the `view` class.

```
CRect Figure::GetArea() const
{
  int iMinRow = 0, iMaxRow = 0, iMinCol = 0, iMaxCol = 0;
  SquareArray* pSquareArray = m_squareInfo[m_iDirection];
  for (int iIndex = 0; iIndex < SQUARE_ARRAY_SIZE; ++iIndex)
  {
    Square& square = (*pSquareArray)[iIndex];
    int iRow = square.Row();
    iMinRow = (iRow < iMinRow) ? iRow : iMinRow;
    iMaxRow = (iRow > iMaxRow) ? iRow : iMaxRow;
    int iCol = square.Col();
    iMinCol = (iCol < iMinCol) ? iCol : iMinCol;
    iMaxCol = (iCol > iMaxCol) ? iCol : iMaxCol;
  }
  return CRect(m_iCol + iMinCol, m_iRow + iMinRow,
               m_iCol + iMaxCol + 1, m_iRow + iMaxRow + 1);
}
```

`Draw` is called when the figure needs to be repainted. It selects a black pen and a brush with the figure's color. Then it draws the four squares of the figure. The `iColorStatus` parameter makes the figure appear in color or in grayscale.

```
void Figure::Draw(int iColorStatus, CDC* pDC) const
{
  CPen pen(PS_SOLID, 0, BLACK);
  CPen* pOldPen = pDC->SelectObject(&pen);
  CBrush brush((iColorStatus == COLOR) ? m_rfColor : GrayScale(
                                                  m_rfColor));
  CBrush* pOldBrush = pDC->SelectObject(&brush);
  SquareArray* pSquareArray = m_squareInfo[m_iDirection];
  for (int iIndex = 0; iIndex < SQUARE_ARRAY_SIZE; ++iIndex)
  {
    Square& square = (*pSquareArray)[iIndex];
    DrawSquare(m_iRow + square.Row(), m_iCol + square.Col(), pDC);
  }
  pDC->SelectObject(&pOldBrush);
  pDC->SelectObject(&pOldPen);
}
```

The Figure Information

There are seven figures, each of them has their own color: red, brown, turquoise, green, yellow, blue, and purple. Each of them also has a unique shape. However, they all consist of four squares. They can further be divided into three groups based on the ability to rotate. The red figure is the simplest one, as it does not rotate at all. The brown, turquoise, and green figures can be rotated in vertical and horizontal directions while the yellow, blue, and purple figures can be rotated in north, east, south, and west directions.

As seen above, the document class creates one object of each figure. When doing so, it uses the information stored in `FigureInfo.h` and `FigureInfo.cpp`.

In this section, we visualize every figure with a sketch like the one in the previous section. The crossed square is the center position referred to by the fields `m_iRow` and `m_iCol` in `Figure`. The positions of the other squares relative to the crossed one are given by the integer pairs in the directions arrays.

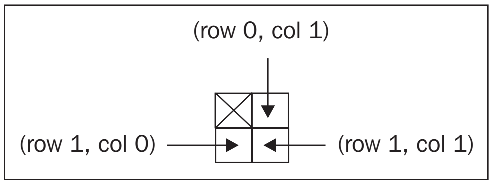
First of all, we need do define the color of each figure. We do so by using the `COLORREF` type.

FigureInfo.cpp

```
const COLORREF RED = RGB(255, 0, 0);
const COLORREF BROWN = RGB(255, 128, 0);
const COLORREF TURQUOISE = RGB(0, 255, 255);
const COLORREF GREEN = RGB(0, 255, 0);
const COLORREF BLUE = RGB(0, 0, 255);
const COLORREF PURPLE = RGB(255, 0, 255);
const COLORREF YELLOW = RGB(255, 255, 0);
```

The Red Figure

The red figure is one large square, built up by four regular squares. It is the simplest figure of the game since it does not change shape when rotating. This implies that we just need to look at one figure.

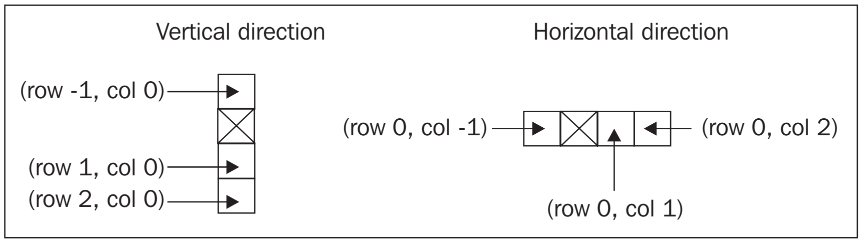


In this case, it is enough to define the squares for one direction and use it to define the shape of the figure in all four directions.

```
SquareArray RedGeneric = {Square(0, 0), Square(0, 1),
                          Square(1, 1), Square(1, 0)};
SquareInfo RedInfo = {&RedGeneric, &RedGeneric,
                      &RedGeneric, &RedGeneric};
```

The Brown Figure

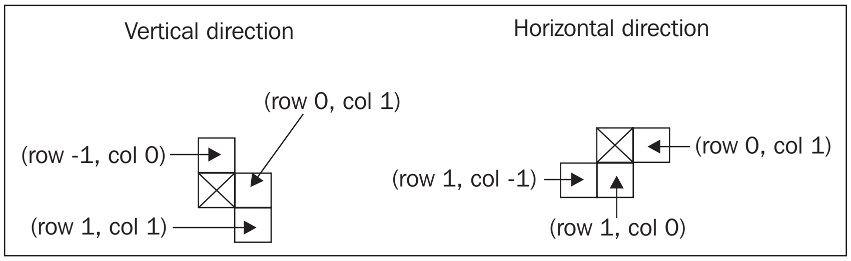
The brown figure can be oriented in horizontal and vertical directions. It is initialized by the constructor to a vertical direction. As it can only be rotated into two directions, the north and south array will be initialized with the vertical array and the east and west array will be initialized with the horizontal array.



```
SquareArray BrownVertical = {Square(0, 0), Square(-1, 0),
                             Square(1, 0), Square(2, 0)};
SquareArray BrownHorizontal = {Square(0, 0), Square(0, -1),
                               Square(0, 1), Square(0, 2)};
SquareInfo BrownInfo = {&BrownVertical, &BrownHorizontal,
                        &BrownVertical, &BrownHorizontal};
```

The Turquoise Figure

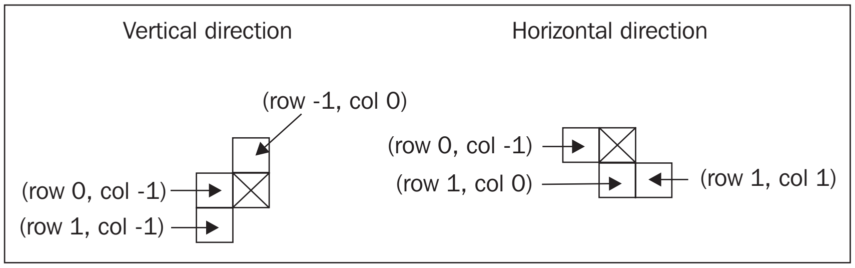
Similar to the brown figure, the turquoise figure can be rotated in the vertical and horizontal directions.



```
SquareArray TurquoiseVertical = {Square(0, 0), Square(-1, 0),
                                 Square(0, 1), Square(1, 1)};
SquareArray TurquoiseHorizontal = {Square(0, 0), Square(1, -1),
                                   Square(1, 0), Square(0, 1)};
SquareInfo TurquoiseInfo = {&TurquoiseVertical, &TurquoiseHorizontal,
                            &TurquoiseVertical,&TurquoiseHorizontal};
```

The Green Figure

The green figure is a mirror image of the turquoise figure.

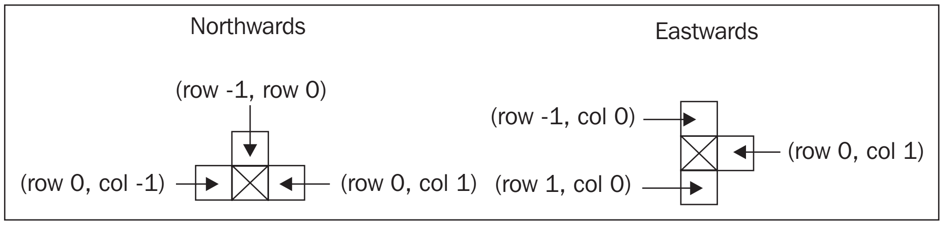


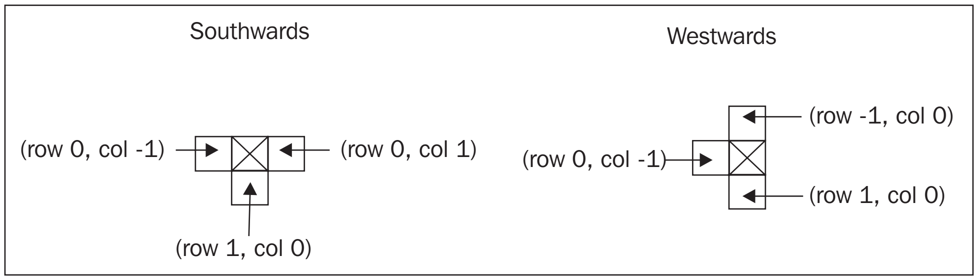
```
SquareArray GreenVertical = {Square(0, 0), Square(1, -1),
                             Square(0, -1), Square(-1, 0)};
SquareArray GreenHorizontal = {Square(0, 0), Square(0, -1),
                               Square(1, 0), Square(1, 1)};

SquareInfo GreenInfo = {&GreenVertical, &GreenHorizontal,
                        &GreenVertical, &GreenHorizontal};
```

The Yellow Figure

The yellow figure can be rotated in the north, east, south, and west directions. It is initialized by the `Figure` class constructor to the south direction.

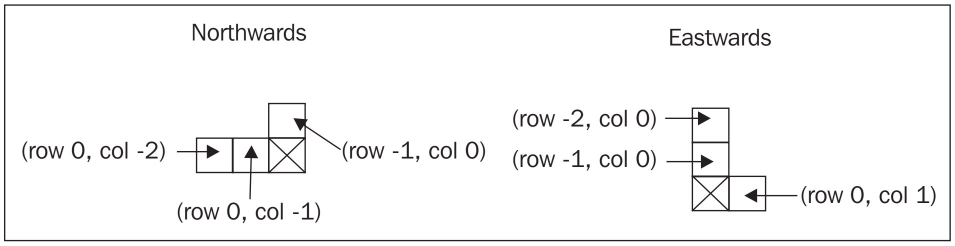


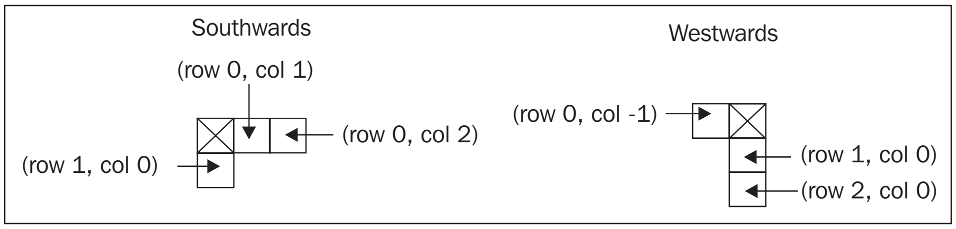


```
SquareArray YellowNorth = {Square(0, 0), Square(0, -1),
                           Square(-1, 0), Square(0, 1)};
SquareArray YellowEast = {Square(0, 0), Square(-1, 0),
                          Square(0, 1), Square(1, 0)};
SquareArray YellowSouth = {Square(0, 0), Square(0, -1),
                           Square(1, 0), Square(0, 1)};
SquareArray YellowWest = {Square(0, 0), Square(-1, 0),
                          Square(0, -1), Square(1, 0)};
SquareInfo YellowInfo = {&YellowNorth, &YellowEast,
                         &YellowSouth, &YellowWest};
```

The Blue Figure

The blue figure can also be in all four directions. It is initialized to the south direction.

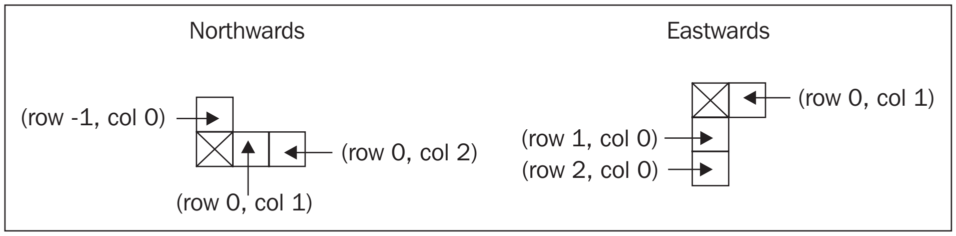


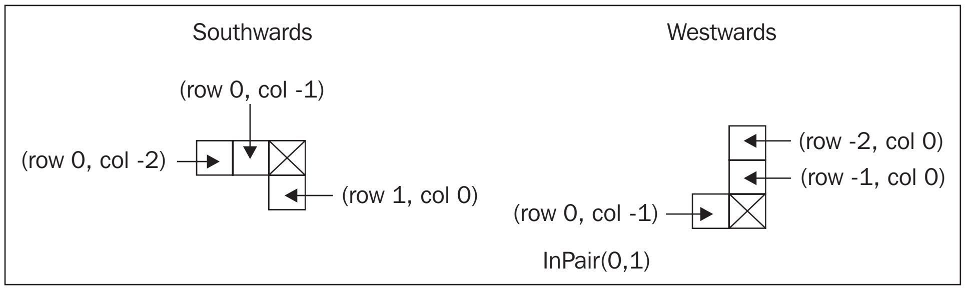


```
SquareArray BlueNorth = {Square(0,  0), Square(0, -2),
                         Square(0, -1),Square(-1, 0)};
SquareArray BlueEast = {Square(0,  0), Square(-2, 0),
                        Square(-1, 0), Square(0, 1)};
SquareArray BlueSouth = {Square(0,  0), Square(1, 0),
                         Square(0, 1), Square(0, 2)};
SquareArray BlueWest = {Square(0,  0), Square(0, -1),
                        Square(1, 0), Square(2, 0)};
SquareInfo BlueInfo = {&BlueNorth, &BlueEast,
                       &BlueSouth, &BlueWest};
```

The Purple Figure

The purple figure, finally, is a mirror image of the blue figure, it is also initialized into the south direction.





```
SquareArray PurpleNorth = {Square(0, 0), Square(-1, 0),
                           Square(0, 1), Square(0, 2)};
SquareArray PurpleEast = {Square(0, 0), Square(1, 0),
                          Square(2, 0), Square(0, 1)};
SquareArray PurpleSouth = {Square(0, 0), Square(0, -2),
                           Square(0, -1), Square(1, 0)};
SquareArray PurpleWest = {Square(0, 0), Square(0, -1),
                          Square(-2, 0), Square(-1, 0)};
SquareInfo PurpleInfo = {&PurpleNorth, &PurpleEast,
                         &PurpleSouth, &PurpleWest};
```

Summary

- We have generated a framework for the application with the Application Wizard.
- We added the classes Square and ColorGrid that keep track of the game grid.
- We defined the document class. It holds the data of the game and keeps track of when the game is over.
- We defined the view class, it accepts keyboard input and draws the figures and the game grid.
- The `Figure` class manages a single figure, it keeps track of its position and decides whether it is valid to move it into another position.
- The `Figure` info files store information of the seven kinds of figures.