Falling Blocks Part 1: Introduction

What this tutorial covers

- The creation of a Tetris clone
- Collision detection
- Victory and losing conditions
- Moving 2D objects
- Using user-defined objects in a game project
- Storing multiple images in a single file

Introduction

Welcome to this series' first actual game tutorial! By the end of this tutorial you'll know how to write a full featured **Tetris** clone. Before you start, you might want to familiarize yourself with the std::vector. It's essentially just an array with added features. I've written a very brief tutorial that can be found here that will teach you all you need to know right now about the std::vector.

Note that this tutorial, as well as the other tutorials in this series, all build off of the Introduction tutorial. If you haven't read it yet, you should do it now.

Setting Up

The first thing we need to do is start a new project called "Falling Blocks" and copy in "Main.cpp" and "Defines.h" from the Introduction tutorial.

Also remember to copy "SDL.dll", "SDL_ttf.dll", and "ARIAL.TTF" into you project directory and set **Project->Falling Blocks Properties->C/C++->Code Generation->Runtime Library** to **Multi-threaded DLL (/MD)**.

You'll also need the bitmap for this tutorial, which can be found in the downloadable source code. See the table of contents for a link to the downloadable source code.

The Code

Before we begin there's one thing I need to make clear. When I say "block", I mean the

actual objects in our game. These are usually called "Tetrinos" but that name's probably copyrighted. When I say "square", I mean the individual squares that make up our Tetrin...er...blocks.

Defines.h

There are some changes and additions we need to make to "Defines.h". First, our window dimensions are way to big. We also need to change our window caption. Make the following changes to "Defines.h":

```
#define WINDOW_WIDTH   300
#define WINDOW_HEIGHT  400
#define WINDOW_CAPTION "Falling Blocks"
```

Now we need to add some new values to "Defines.h". If you run the executable that comes with the downloadable source code, you'll see that the game area does not take up the entire screen. We'll need to know the location and dimensions of our game area within our window, so let's add the following to "Defines.h":

```
#define GAME_AREA_LEFT   53
#define GAME_AREA_RIGHT  251
#define GAME_AREA_BOTTOM 298
```

We want there to be a finite amount of levels in our game, so we'll define that here. We'll keep all of the values that are dependent on the number of levels in our game in "Defines.h". This way we only have to change a couple of values in "Defines.h" to add more levels.

In **Tetris**, everytime you clear a line of squares you get points. If you reach a certain number of points, you advance to the next level. Every time you advance to the next level, the blocks start moving down faster. We'll define the speed of the blocks in moves/frames so if the block is moving down every 60 frames, and we're running our game at 30 frames per second, our blocks will move down every two seconds. Add the following to "Defines.h":

```
#define NUM_LEVELS       5    // number of levels in the game
#define POINTS_PER_LINE  525  // points player receives for
completing a line
#define POINTS_PER_LEVEL 6300 // points player needs to advance a
level

#define INITIAL_SPEED 60  // initial interval at which focus block
moves down
#define SPEED_CHANGE  10  // the above interval is reduced by this
much each level
```

When the current block (we'll call it the "focus block") reaches the bottom of the game

area, the player should be given a brief moment in which to slide it left or right. This works great when the focus block hits another block and the player quickly slides it into a better place. We'll define the amount of time the player gets to slide the focus block here.

The main purpose of our game is to fill rows of squares in order to clear them and receive points. We'll need to define the number of squares that can fit in a row. We'll use this value later when we determine which rows are full of squares.

When we specify the locations of our squares within the game area, we'll be using our squares' centers. For this reason, we need to record the distance from the center of a square to its sides. This value will be used for getting the locations of the sides of our squares as well as for determining if two blocks are touching.

Add the following to "Defines.h":

```
#define SLIDE_TIME      15
#define SQUARES_PER_ROW 10  // number of squares that fit in a row
#define SQUARE_MEDIAN   10  // distance from the center of a square
to its sides
```

The rest of our defines involve locations within our game area and our bitmap. We need to define the starting point for our blocks, and where to display the current score, level, required score, and next block in line. Add the following to "Defines.h":

```
// Starting position of the focus block
#define BLOCK_START_X 151
#define BLOCK_START_Y  59

// Location on game screen for displaying...
#define LEVEL_RECT_X        42   // current level
#define LEVEL_RECT_Y        320
#define SCORE_RECT_X        42   // current score
#define SCORE_RECT_Y        340
#define NEEDED_SCORE_RECT_X 42   // score needed for next level
#define NEEDED_SCORE_RECT_Y 360
#define NEXT_BLOCK_CIRCLE_X 214  // next block in line to be focus
block
#define NEXT_BLOCK_CIRCLE_Y 347
```

In most games, each image is not actually stored in a separate file. Games generally cram as much information into as few files as possible. Because there isn't much to our game, we can easily store all of our images in a single file. To make this work, we need to define the locations within our bitmap of the game's background screens and squares. Add the following to "Defines.h":

```
// Locations within bitmap of background screens
#define LEVEL_ONE_X   0
#define LEVEL_ONE_Y   0
#define LEVEL_TWO_X   300
```

```
#define LEVEL_TWO_Y   0
#define LEVEL_THREE_X 300
#define LEVEL_THREE_Y 0
#define LEVEL_FOUR_X  0
#define LEVEL_FOUR_Y  396
#define LEVEL_FIVE_X  300
#define LEVEL_FIVE_Y  396

// Location within bitmap of colored squares
#define RED_SQUARE_X    600
#define RED_SQUARE_Y    400
#define PURPLE_SQUARE_X 620
#define PURPLE_SQUARE_Y 400
#define GREY_SQUARE_X   640
#define GREY_SQUARE_Y   400
#define BLUE_SQUARE_X   660
#define BLUE_SQUARE_Y   400
#define GREEN_SQUARE_X  680
#define GREEN_SQUARE_Y  400
#define BLACK_SQUARE_X  700
#define BLACK_SQUARE_Y  400
#define YELLOW_SQUARE_X 720
#define YELLOW_SQUARE_Y 400
```

Enums.h

A new file we'll be adding to our project is "Enums.h". This is a fairly simple file and could probably be fit into one of our other files but I like to keep things organized in separate files.

We only need two enumerations for this project. The first will be used for the types of blocks in our game. We enumerate our block types so we can generate random numbers to determine what block to add to our game next. The second will be used for directions. Blocks can be moved left, right, or down.

Add the following to "Enums.h":

```
#pragma once

enum BlockType
{
    SQUARE_BLOCK,
    T_BLOCK,
    L_BLOCK,
    BACKWARDS_L_BLOCK,
    STRAIGHT_BLOCK,
    S_BLOCK,
    BACKWARDS_S_BLOCK
};
```

```
enum Direction
{
    LEFT,
    RIGHT,
    DOWN
};
```

Falling Blocks Part 2: cSquare and cBlock

cSquare.h

We need a class to represent individual squares for a few reasons. For one, our blocks are composed of squares. Also, any part of our block can collide with something else so it's nice to be able to check each square for collisions separately. Most importantly though, when our focus block hits the bottom of the game area, it becomes a part of the square pile. Parts of this pile can be cleared and the squares above the cleared squares will move down. This completely destroys the shape of our blocks. For this reason, we can't just store everything as a block.

Add a file called "cSquare.h" to your project and then add the following code:

```
#pragma once

#include "Enums.h"
#include "Defines.h"

class cSquare
{
private:

public:

};
```

Note that we will be using values from both "Enums.h" and "Defines.h" in this class.

Now we need to determine the data that this class needs to store. As previously discussed, we will be storing the center of our square so we need variables to track its x and y locations. Each type of block in our game will be a different color, so we need to know what type of block our square belongs to. We'll also need a pointer to our bitmap surface so we can draw our square. Add the following to the private section of cSquare:

```
// Location of the center of the square
int m_CenterX;
int m_CenterY;
```

```
// Type of block. Needed to locate the correct square in our bitmap
BlockType m_BlockType;

// A pointer to our bitmap surface from "Main.cpp"
SDL_Surface* m_Bitmap;
```

The constructor for this class will be very simple. We'll just have it initialize our data members. We will also need to add a default constructor to our class because our compiler will expect it. If we declare pointers to cSquare objects and the compiler can't find a default constructor, it will issue error messages. Add the following to the public section of cSquare:

```
// Default constructor, your compiler will probably require this
cSquare()
{
}
// Main constructor takes location and type of block, and a pointer
to our bitmap surface.
cSquare(int x, int y, SDL_Surface* bitmap, BlockType type) :
m_CenterX(x), m_CenterY(y),
                m_Bitmap(bitmap), m_BlockType(type)
{
}
```

The only functionality we'll really need for our squares is the ability to draw and move them. Let's start with the Draw() method.

We'll be calling SDL_BlitSurface() which requires a pointer to our window surface so we should have our function take a pointer to our window surface so we can pass it in. The function will actually be quite simple. It will determine which type of block our square belongs to, find the appropriate colored square in our bitmap surface, and then draw the square at its current location. Add the following to the public section of cSquare:

```
// Draw() takes a pointer to the surface to draw to (our window)
void Draw(SDL_Surface* window)
{
    SDL_Rect source;
    // switch statement to determine the location of the square
within our bitmap
    switch (m_BlockType)
    {
        case SQUARE_BLOCK:
        {
            SDL_Rect temp = { RED_SQUARE_X, RED_SQUARE_Y,
SQUARE_MEDIAN * 2,
                SQUARE_MEDIAN * 2 };
            source = temp;
        } break;
```

```
        case T_BLOCK:
        {
            SDL_Rect temp = { PURPLE_SQUARE_X, PURPLE_SQUARE_Y,
SQUARE_MEDIAN * 2,
                SQUARE_MEDIAN * 2 };
            source = temp;
        } break;
        case L_BLOCK:
        {
            SDL_Rect temp = { GREY_SQUARE_X, GREY_SQUARE_Y,
SQUARE_MEDIAN * 2,
                SQUARE_MEDIAN * 2 };
            source = temp;
        } break;
        case BACKWARDS_L_BLOCK:
        {
            SDL_Rect temp = { BLUE_SQUARE_X, BLUE_SQUARE_Y,
SQUARE_MEDIAN * 2,
                SQUARE_MEDIAN * 2 };
            source = temp;
        } break;
        case STRAIGHT_BLOCK:
        {
            SDL_Rect temp = { GREEN_SQUARE_X, GREEN_SQUARE_Y,
SQUARE_MEDIAN * 2,
                SQUARE_MEDIAN * 2 };
            source = temp;
        } break;
        case S_BLOCK:
        {
            SDL_Rect temp = { BLACK_SQUARE_X, BLACK_SQUARE_Y,
SQUARE_MEDIAN * 2,
                SQUARE_MEDIAN * 2 };
            source = temp;
        } break;
        case BACKWARDS_S_BLOCK:
        {
            SDL_Rect temp = { YELLOW_SQUARE_X, YELLOW_SQUARE_Y,
SQUARE_MEDIAN * 2,
                SQUARE_MEDIAN * 2 };
            source = temp;
        } break;
    }
     // Draw at square's current location. Remember that m_X and m_Y
store the
     // center of the square.
    SDL_Rect destination = { m_CenterX - SQUARE_MEDIAN, m_CenterY -
SQUARE_MEDIAN,
    SQUARE_MEDIAN * 2, SQUARE_MEDIAN * 2 };
```

```
    SDL_BlitSurface(m_Bitmap, &source, window, &destination);
}
```

Move() is also very simple. We just determine which direction our square is moving and change the appropriate data member. Add the following to the public section of cSquare:

```
// Remember, SQUARE_MEDIAN represents the distance from the square's
center to
// its sides. SQUARE_MEDIAN*2 gives us the width and height of our
squares.
void Move(Direction dir)
{
    switch (dir)
    {
        case LEFT:
        {
            m_CenterX -= SQUARE_MEDIAN * 2;
        } break;
        case RIGHT:
        {
            m_CenterX += SQUARE_MEDIAN * 2;
        } break;
        case DOWN:
        {
            m_CenterY += SQUARE_MEDIAN*2;
        } break;
    }
}
```

To finish, let's add some accessor and mutator methods for our location variables:

```
// Accessors
int GetCenterX() { return m_CenterX; }
int GetCenterY() { return m_CenterY; }

// Mutators
void SetCenterX(int x) { m_CenterX = x; }
void SetCenterY(int y) { m_CenterY = y; }
```

cBlock.h

Our cBlock class will store its center, type, and the four squares that it's built from. Note that we'll consider a block's center to be the point about which we will rotate its squares. Our constructor will simply initialize the block's data members and call a function that sets up the squares. We'll get to this function in a moment. For now, add the following to "cBlock.h":

```
#pragma once
```

```
#include "cSquare.h"

class cBlock
{
private:
    // Location of the center of the block
    int m_CenterX;
    int m_CenterY;

    // Type of block
    BlockType m_Type;

    // Array of pointers to the squares that make up the block
    cSquare* m_Squares[4];

public:
    // The constructor just sets the block location and calls
SetupSquares
    cBlock(int x, int y, SDL_Surface* bitmap, BlockType type) :
m_CenterX(x), m_CenterY(y),
        m_Type(type)
    {
        // Set our square pointers to null
        for (int i=0; i<4; i++)
        {
            m_Squares[i] = NULL;
        }

        SetupSquares(x, y, bitmap);
    }
};
```

SetupSquares() initializes the locations of the block's squares with respect to its center. It takes the center point of the block so we can move it to somewhere else on the screen and reset its shape. It also takes a pointer to our bitmap, which we'll pass to cSquare's constructor.

If you find the following code hard to follow, don't worry about it. I'm sure you could manage setting up the shapes of the blocks on your own. It's usually a bit trickier figuring out what someone else is doing. Anyways, add the following to the public section of cBlock:

```
// Setup our block according to its location and type. Note that the
squares
// are defined according to their distance from the block's center.
This
// function takes a surface that gets passed to cSquare's
constructor.
void SetupSquares(int x, int y, SDL_Surface* bitmap)
{
    // This function takes the center location of the block. We set
our data
```

```
    // members to these values to make sure our squares don't get
defined
    // around a new center without our block's center values changing
too.
    m_CenterX = x;
    m_CenterY = y;

    // Make sure that any current squares are deleted
    for (int i=0; i<4; i++)
    {
        if (m_Squares[i])
            delete m_Squares[i];
    }

    switch (m_Type)
    {
        case SQUARE_BLOCK:
        {
            // [0][2]
            // [1][3]
            m_Squares[0] = new cSquare(x - SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[1] = new cSquare(x - SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[2] = new cSquare(x + SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[3] = new cSquare(x + SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
        } break;
        case T_BLOCK:
        {
            //    [0]
            // [2][1][3]
            m_Squares[0] = new cSquare(x + SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[1] = new cSquare(x + SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[2] = new cSquare(x - SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[3] = new cSquare(x + (SQUARE_MEDIAN * 3), y +
SQUARE_MEDIAN,
                bitmap, m_Type);
        } break;
        case L_BLOCK:
```

```
        {
            // [0]
            // [1]
            // [2][3]
            m_Squares[0] = new cSquare(x - SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                 bitmap, m_Type);
            m_Squares[1] = new cSquare(x - SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                 bitmap, m_Type);
            m_Squares[2] = new cSquare(x - SQUARE_MEDIAN, y +
(SQUARE_MEDIAN * 3),
                 bitmap, m_Type);
            m_Squares[3] = new cSquare(x + SQUARE_MEDIAN, y +
(SQUARE_MEDIAN * 3),
                 bitmap, m_Type);
        } break;
        case BACKWARDS_L_BLOCK:
        {
            //    [0]
            //    [1]
            // [3][2]
            m_Squares[0] = new cSquare(x + SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[1] = new cSquare(x + SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[2] = new cSquare(x + SQUARE_MEDIAN, y +
(SQUARE_MEDIAN * 3),
                bitmap, m_Type);
            m_Squares[3] = new cSquare(x - SQUARE_MEDIAN, y +
(SQUARE_MEDIAN * 3),
                bitmap, m_Type);
        } break;
        case STRAIGHT_BLOCK:
        {
            // [0]
            // [1]
            // [2]
            // [3]
            m_Squares[0] = new cSquare(x + SQUARE_MEDIAN, y -
(SQUARE_MEDIAN * 3),
                bitmap, m_Type);
            m_Squares[1] = new cSquare(x + SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[2] = new cSquare(x + SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
```

```
            m_Squares[3] = new cSquare(x + SQUARE_MEDIAN, y +
(SQUARE_MEDIAN * 3),
                bitmap, m_Type);
        } break;
        case S_BLOCK:
        {
            //    [1][0]
            // [3][2]
            m_Squares[0] = new cSquare(x + (SQUARE_MEDIAN * 3), y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[1] = new cSquare(x + SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[2] = new cSquare(x + SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[3] = new cSquare(x - SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
        } break;
        case BACKWARDS_S_BLOCK:
        {
            // [0][1]
            //    [2][3]
            m_Squares[0] = new cSquare(x - SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[1] = new cSquare(x + SQUARE_MEDIAN, y -
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[2] = new cSquare(x + SQUARE_MEDIAN, y +
SQUARE_MEDIAN,
                bitmap, m_Type);
            m_Squares[3] = new cSquare(x + (SQUARE_MEDIAN * 3), y +
SQUARE_MEDIAN,
                bitmap, m_Type);
        } break;
    }
}
```

cBlock's Draw() method simply calls the Draw() methods of its squares. Its Move() method just changes its center variables and then calls its squares' Move() methods. Add the following to the public section of cBlock:

```
// Draw() simply iterates through the squares and calls their Draw()
functions.
void Draw(SDL_Surface* Window)
{
    for (int i=0; i<4; i++)
```

```
    {
        m_Squares[i]->Draw(Window);
    }
}
// Move() simply changes the block's center and calls the squares'
move functions.
void Move(Direction dir)
{
    switch (dir)
    {
        case LEFT:
        {
            m_CenterX -= SQUARE_MEDIAN * 2;
        } break;
        case RIGHT:
        {
            m_CenterX += SQUARE_MEDIAN * 2;
        } break;
        case DOWN:
        {
            m_CenterY += SQUARE_MEDIAN*2;
        } break;
    }
    for (int i=0; i<4; i++)
    {
        m_Squares[i]->Move(dir);
    }
}
```

cBlock's Rotate() function might be a little hard to explain without a bit of linear algebra, but I'll do my best. To rotate a point around the origin (0,0) by a given angle, we use the following formula:

```
x = x*cos(angle) - y*sin(angle);
y = x*sin(angle) + y*cos(angle);
```

We can simplify this formula however, because we'll always be rotating the block by 90 degrees. **cos(90)** is zero, and **sin(90)** is one. Our formula simplifies to:

```
x = -y;
y = x;
```

Quite a bit more manageable if you ask me. Don't get too excited though, if we plug this formula into our code right now, our blocks will wind up rotating around the top left of our window. This is because that is where the origin is; (0,0) refers to the top left of a window.

What we want is for our block to rotate around its center. We know where our center is because we store it in m_CenterX and m_CenterY, but how do we make our squares

rotate around that center?

First, we subtract the center of the square we want to rotate by the center of our block. This places the center of our block at (0,0). Because the center of our block is now at the origin, our rotation will work properly. After we rotate, we just add the center of our block to the center of our square, which puts it right back to where it was.

If you need to visualize this, image a block in the middle of the screen. If you could watch the entire rotation, you would see the block move to the top left of the window, rotate around the origin (it would rotate around itself in this case, because its center is at the origin), then move back to the middle of the screen.

Add the following to the public section of cBlock:

```
void Rotate()
{
    // We need two sets of temporary variables so we don't
incorrectly
    // alter one of them.If we set x to -y, then set y to x, we'd
actually
    // be setting y to -y because x is now -y
    int x1, y1, x2, y2;

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

        // Get the center of the current square
        x1 = m_Squares[i]->GetCenterX();
        y1 = m_Squares[i]->GetCenterY();

        // Move the square so it's positioned at the origin
        x1 -= m_CenterX;
        y1 -= m_CenterY;

        // Do the actual rotation
        x2 = - y1;
        y2 = x1;

        // Move the square back to its proper location
        x2 += m_CenterX;
        y2 += m_CenterY;

        // Set the square's location to our temporary variables
        m_Squares[i]->SetCenterX(x2);
        m_Squares[i]->SetCenterY(y2);
    }
}
```

Before we can rotate a block, we need to check to see if the block will collide with anything. We handle collision detection outside of cBlock, so we need an accesor method that returns the positions of our block's squares after rotation. Note that we wont bother returning the actual squares, just an array containing their locations. This function is

very similar to Rotate(). Add the following to the public section of cBlock:

```
// This function gets the locations of the squares after
// a rotation and returns an array of those values.
int* GetRotatedSquares()
{
    int* temp_array = new int[8];
    int x1, y1, x2, y2;

    for (int i=0; i<4; i++)
    {
        x1 = m_Squares[i]->GetCenterX();
        y1 = m_Squares[i]->GetCenterY();

        x1 -= m_CenterX;
        y1 -= m_CenterY;

        x2 = - y1;
        y2 = x1;

        x2 += m_CenterX;
        y2 += m_CenterY;

        // Instead of setting the squares, we just store the values
        temp_array[i*2]   = x2;
        temp_array[i*2+1] = y2;
    }

    return temp_array;
}
```

To finish, we just need an accesor method that returns an array of pointers to our squares. Note that the notation ** is required because we are returning a pointer to an array of pointers. Add the following to the public section of cBlock:

```
// This returns a pointer to an array of pointers to the squares of
the block.
cSquare** GetSquares()
{
    return m_Squares;
}
```

Falling Blocks Part 3: Includes, Global Data, and Prototypes

Includes

Because this project is more advanced than the introduction tutorial, we're definately going to need to include some more files. The most obvious are "cBlock.h" and "Enums.h". Note that we don't really need to include "cSquare.h", because it is already included in

"cBlock.h". Add the following to "Main.cpp":

```
#include "Enums.h"  // Our enums header
#include "cBlock.h" // Contains the class that represents a game
block
```

When we start our game, we'll randomly select a block type to be the focus block, and once again randomly select a block type to be the next block in line. Every time our focus block changes, we need to select another block type to be the next block in line. We've already enumerated our block types, so we'll use rand() to get a random integer that we'll compare with our BlockType enumerations.

The one problem with rand() is that it doesn't really return a random number. It actually just returns a pattern of numbers that seem to be randomly generated. The function will actually always use the same pattern of number, which will be a very obvious problem to the player. Seeing the same pattern of blocks every game will really ruin the experience.

What we need to do is "seed" our random generator with another number. This basically means that we'll give rand() a different number to use when generating a pattern of numbers. The function that seeds rand() is called srand() and it takes a numerical value as a parameter. One of the most common numbers to seed rand() with is the current time, so we'll call the time() function to get the time. To use time(), we need to include "time.h".

Another file we need is "math.h". This file contains tons of math functions and you should generally assume that you'll be using it in your game projects. Add the following to "Main.cpp":

```
#include "time.h" // We use time(), located in "time.h", to seed our
random generator
#include "math.h" // We'll be using the abs() function located in
"math.h"
```

When the focus block can no longer move, we add its squares to our game area. We'll need to have instant access to the squares in the game area, so it seems like an array would be a good choice. If you want to access an element of an array, you just give the array the index you want to access and you get the element right away (array[4], for example).

One problem with arrays though is that you can't resize an array once you've declared it. Because there's no way to tell how many squares we'll have in our game area at any given time, we need a data structure that can hold a variable amount of data. I've decided to use the STL vector for this reason. The vector structure can change in size, and we can access its elements just like an array. We also should include the string class here. Add the following to "Main.cpp":

```
#include <vector> // An STL vector will store the game squares
#include <string> // Used for any string functionality we require
```

Note that the STL vector has no relation to the type of vector you would encounter in math or physics. If you'd like to learn more about the vector structure, see my STL Tutorial.

Global Data

We'll be using all of the global data from the previous tutorial, as well as the data required for this specific project. As already discussed, we'll have two blocks in our game. The focus block will be the block the player actually controls. The other block will be the block that is next in line to be the focus block. We want to have a pointer to this block so we can draw it in the bottom of the screen. Add the following to "Main.cpp":

```
cBlock* g_FocusBlock = NULL;  // The block the player is controlling
cBlock* g_NextBlock  = NULL;  // The next block to be the focus block
```

We just discussed the vector of squares in our game, so we can safely add that now. We'll also want data to keep track of the player's current score, as well as the current level. Each time the level changes, the focus block will speed up. We'll have a a global integer to keep track of the focus block's speed. Add the following to "Main.cpp":

```
vector<cSquare*> g_OldSquares;         // The squares that no longer
form the focus block
int g_Score = 0;                       // Players current score
int g_Level = 1;                       // Current level player is on
int g_FocusBlockSpeed = INITIAL_SPEED; // Speed of the focus block
```

That's it for our global variables!

Function Prototypes

In the introduction tutorial, we had three functions to represent the three states of our program. For the rest of our projects, we'll need to add two more states. One state will display a victory message and the other will display a game over message. Both states will give the player the option to either quit the program or go back to the main menu. Add the following right below void Exit(); in "Main.cpp":

```
void Win();
void Lose();
```

We'll also need a function to handle the input for these two states. We could have two different functions, but all do is check to see if the player has pressed 'n' or 'y' so there wouldn't be much point. Add the following right below the other input functions in "Main.cpp":

```
void HandleWinLoseInput();
```

Now we need some functions to handle our game. Collision detection is always necessary in games, so we'll start with that. The two things we can do with our objects are move them around and rotate them. Our objects can either collide with each other, or with the sides of the game area.

Because we'll be moving individual squares as well as entire blocks, we need functions that take both types of objects. Instead of writing functions with different names, we'll take advantage of function overloading. One set of functions will take pointers to squares and the other set will take pointers to blocks.

Only blocks can rotate so we only need one function to handle collisions when the player rotates a block.

Add the following to "Main.cpp":

```
bool CheckEntityCollisions(cSquare* square, Direction dir);
bool CheckWallCollisions(cSquare* square, Direction dir);
bool CheckEntityCollisions(cBlock* block, Direction dir);
bool CheckWallCollisions(cBlock* block, Direction dir);
bool CheckRotationCollisions(cBlock* block);
```

Each time the player advances to the next level, we need to check to see if the game has been won.

Each time the focus block is changed and reset to the top of the game area, we need to see if it can still move. If it can't, we know that the game is over because the squares have built up to high.

We'll have functions that check for and handle these two events. Add the following to "Main.cpp":

```
void CheckWin();
void CheckLoss();
```

When the focus block either reaches the bottom of the game area or is blocked by another square, we need to change the focus block and check to see if a line has been completed. We'll have three functions handle this process. Add the following to "Main.cpp":

```
void HandleBottomCollision();
void ChangeFocusBlock();
int CheckCompletedLines();
```

We are now ready to begin writing our game specific code!

Falling Blocks Part 4: Init(), Shutdown(), and State Functions

Init()

Changes to Init() will actually be fairly minor. We first need to tell SDL_LoadBMP() the name of the file we'll be using. We also need to make a call to srand() to seed our random number generator. After that, all we have to do is initialize our two main blocks. Add the following to Init() in "Main.cpp" and be sure to remove the old call to SDL_LoadBMP():

```
// Fill our bitmap structure with information
g_Bitmap = SDL_LoadBMP("data/FallingBlocks.bmp");

// Seed our random number generator
srand( time(0) );

// Initialize blocks and set them to their proper locations.
g_FocusBlock = new cBlock( BLOCK_START_X, BLOCK_START_Y, g_Bitmap,
    (BlockType)(rand()%7) );
g_NextBlock  = new cBlock( NEXT_BLOCK_CIRCLE_X, NEXT_BLOCK_CIRCLE_Y,
    g_Bitmap, (BlockType)(rand()%7) );
```

Notice the statement (BlockType)(rand()%7). We first call rand() to get a random number. We then use the modulo operator (%) to reduce that number to a number between 0 and 6. The modulo operator returns the remainder after dividing a number by another number. If you divide 7 by 7, you get a remainder of 0. If you divide 8 by 7, you get a remainder of 1. Note that you can never get a remainder of 7, so 6 will be the limit. Finally, we cast the number to be of type BlockType because that's what cBlock's constructor takes.

Shutdown()

All we have to add to Shutdown() is some code that deletes our two main blocks. We have to be careful how we do this though. In cBlock, we store an array of pointers to cSquare objects. We don't delete these objects in cBlock's constructor because we usually still need them after deleting our focus block. This is because we add the squares of our focus block to the game area.

In order to delete these objects, we need to get pointers to our blocks' arrays and delete them separately. Add the following code to Shutdown() in "Main.cpp":

```
// Get pointers to the squares in our focus and next-in-line block so
we
// can delete them. We must do this before we delete our blocks so we
// don't lose references to the squares. Note that these are pointers
```

```
to
// arrays of pointers.
cSquare** temp_array_1 = g_FocusBlock->GetSquares();
cSquare** temp_array_2 = g_NextBlock->GetSquares();

// Delete our blocks
delete g_FocusBlock;
delete g_NextBlock;

// Delete the temporary arrays of squares
for (int i=0; i<4; i++)
{
    delete temp_array_1[i];
    delete temp_array_2[i];
}
```

One last thing we need to do in Shutdown() is delete the squares in our global vector. Add the following just below the previous code:

```
// Delete the squares that are in the game area
for (int i=0; i<g_OldSquares.size(); i++)
{
    delete g_OldSquares[i];
}
```

Menu() and Exit()

When we drew our text in the introduction tutorial, we drew it in the middle of the window. Since we've changed the size of the window, we need to change the location of our text. Replace the previous calls to DisplayText() in Menu() with the following:

```
DisplayText("Start (G)ame", 120, 120, 12, 255, 255, 255, 0, 0, 0);
DisplayText("(Q)uit Game", 120, 150, 12, 255, 255, 255, 0, 0, 0);
```

Now do the same thing in Exit():

```
DisplayText("Quit Game (Y or N)?", 100, 150, 12, 255, 255, 255, 0, 0,
0);
```

Game()

Our Game() function will need a lot more changes than the other states. We need to move the focus block down at the previously specified interval, give the player time to slide the focus block if it hits something, draw our blocks and squares, and display the current level, score, and the score needed to advance to the next level.

To begin, we need two counter variables. One will count the amount of time that has passed since we moved the focus block down, and the other will keep track of how much

time the player has been allowed to slide the focus block. We'll make these variable static because we need to keep their values between frames. Add the following to the top of Game() in "Main.cpp":

```
static int force_down_counter = 0;
static int slide_counter = SLIDE_TIME;
```

In every frame we begin by incrementing force_down_counter. We then check to see if it has exceeded our focus block's movement rate (g_FocusBlockSpeed). If it has, we make sure there won't be a collision and move the focus block down. Finally, we reset force_down_counter to zero. Add the following to Game() just below the call to HandleGameInput():

```
force_down_counter++;

if (force_down_counter >= g_FocusBlockSpeed)
{
    // Always check for collisions before moving anything
    if ( !CheckWallCollisions(g_FocusBlock, DOWN) &&
         !CheckEntityCollisions(g_FocusBlock, DOWN) )
    {
        g_FocusBlock->Move(DOWN); // move the focus block
        force_down_counter = 0; // reset our counter
    }
}
```

Once we've moved the focus block down, we should check to see if it has reached the bottom of the game area or if it is blocked by any squares. If it is, we start decrementing our slide counter. If there isn't a collision detected, we reset slide_counter. This is because the player might have initially hit something, thus causing our slide counter to start ticking, but then may have moved the focus block out of the way. If we don't reset our slide counter, it will stay at whatever value it was decremented to.

If our slide counter reaches zero, we call HandleBottomCollision() which will change the focus block and check for any full rows of squares. Add the following to Game():

```
// Check to see if focus block's bottom has hit something.
// If it has, we decrement our counter.
if ( CheckWallCollisions(g_FocusBlock, DOWN) ||
     CheckEntityCollisions(g_FocusBlock, DOWN) )
{
    slide_counter--;
}
// If there isn't a collision, we reset our counter.
// This is in case the player moves out of a collision.
else
{
    slide_counter = SLIDE_TIME;
}
```

```
// If the counter hits zero, we reset it and call our
// function that handles changing the focus block.
if (slide_counter == 0)
{
    slide_counter = SLIDE_TIME;
    HandleBottomCollision();
}
```

Because we already wrote drawing functions for our blocks and squares, we can just call them to draw our objects. Add the following to Game(), just below DrawBackground():

```
// Draw the focus block and next block.
g_FocusBlock->Draw(g_Window);
g_NextBlock->Draw(g_Window);

// Draw the old squares.
for (int i=0; i < g_OldSquares.size(); i++)
{
    g_OldSquares[i]->Draw(g_Window);
}
```

All we have to do now is display the current level, score, and needed score. DisplayText() takes a string so we need to build three strings to pass to it. To build these strings, we first need to assign them some text ("Score: ", "Needed Score: ", and "Level: "). We then need to add the appropriate values to each string. To do this, we'll use the append() function, which takes a char array and attaches it to the end of a string. We'll convert our numerical values to char arrays with the itoa() function. This function just converts an integer into a char array. Add the following to Game():

```
// This will be passed to itoa()
char temp[256];

string score = "Score: ";
itoa(g_Score, temp, 10); // the 10 just tells itoa to use decimal
notation
score.append( temp );

string nextscore = "Needed Score: ";
itoa(g_Level*POINTS_PER_LEVEL, temp, 10);
nextscore.append(temp);

string level = "Level: ";
itoa(g_Level, temp, 10);
level.append(temp);

DisplayText(score, SCORE_RECT_X, SCORE_RECT_Y, 8, 0, 0, 0, 255, 255,
255);
DisplayText(nextscore, NEEDED_SCORE_RECT_X, NEEDED_SCORE_RECT_Y, 8,
0, 0, 0,
    255, 255, 255);
DisplayText(level, LEVEL_RECT_X, LEVEL_RECT_Y, 8, 0, 0, 0, 255, 255,
```

```
255);
```

GameWon() and GameLost()

Everything in GameWon() and GameLost() has already been covered, so just add the following to "Main.cpp":

```
// Display a victory message.
void GameWon()
{
    if ( (SDL_GetTicks() - g_Timer) >= FRAME_RATE )
    {
        HandleWinLoseInput();
        ClearScreen();
        DisplayText("You Win!!!", 100, 120, 12, 255, 255, 255, 0, 0,
0);
        DisplayText("Quit Game (Y or N)?", 100, 140, 12, 255, 255,
255, 0, 0, 0);
        SDL_UpdateRect(g_Window, 0, 0, 0, 0);
        g_Timer = SDL_GetTicks();
    }
}
// Display a game over message.
void GameLost()
{
    if ( (SDL_GetTicks() - g_Timer) >= FRAME_RATE )
    {
        HandleWinLoseInput();
        ClearScreen();
        DisplayText("You Lose.", 100, 120, 12, 255, 255, 255, 0, 0,
0);
        DisplayText("Quit Game (Y or N)?", 100, 140, 12, 255, 255,
255, 0, 0, 0);
        SDL_UpdateRect(g_Window, 0, 0, 0, 0);
        g_Timer = SDL_GetTicks();
    }
}
```

DrawBackground()

Before we get to our new input function, there's a quick change we need to make to DrawBackground(). Previously, we always drew the same background, but in this game our background will change according to the current level. All we need to do to handle

this is check to see what the current level is and set our source rectangle to the appropriate location within our bitmap. Replace the current DrawBackground() function with the following:

```
void DrawBackground()
{
    SDL_Rect source;

    // Set our source rectangle to the current level's background
    switch (g_Level)
    {
        case 1:
        {
            SDL_Rect temp = { LEVEL_ONE_X, LEVEL_ONE_Y, WINDOW_WIDTH,

                WINDOW_HEIGHT };
            source = temp;
        } break;
        case 2:
        {
            SDL_Rect temp = { LEVEL_TWO_X, LEVEL_TWO_Y, WINDOW_WIDTH,

                WINDOW_HEIGHT };
            source = temp;
        } break;
        case 3:
        {
            SDL_Rect temp = { LEVEL_THREE_X, LEVEL_THREE_Y,
WINDOW_WIDTH,
                WINDOW_HEIGHT };
            source = temp;
        } break;
        case 4:
        {
            SDL_Rect temp = { LEVEL_FOUR_X, LEVEL_FOUR_Y,
WINDOW_WIDTH,
                WINDOW_HEIGHT };
            source = temp;
        } break;
        case 5:
        {
            SDL_Rect temp = { LEVEL_FIVE_X, LEVEL_FIVE_Y,
WINDOW_WIDTH,
                WINDOW_HEIGHT };
            source = temp;
        } break;
    }

    SDL_Rect destination = { 0, 0, WINDOW_WIDTH, WINDOW_HEIGHT };
```

```
    SDL_BlitSurface(g_Bitmap, &source, g_Window, &destination);
}
```

Notice that we used a temporary variable when creating our source rectangle. This is just so we can use initializer lists. We can't do this:

```
int source;
source = { LEVEL_FOUR_X, LEVEL_FOUR_Y, WINDOW_WIDTH, WINDOW_HEIGHT };
```

because source has already been defined. You can only just initalizer lists when initializing something.

HandleGameInput()

When playing a game like **Tetris**, it can become very tedious if you have to press an arrow key every time you want the block to move. A better approach is to allow the player to hold down the arrow keys. To do this, we'll declare three boolean variables at the top of our function to represent the left, right, and down arrow keys. As far as detecting input goes, all we'll do is set the appropriate variable to true when the user presses a key, and back to false when the user releases the key. After that, all we need to do is check the boolean variables each time our input function gets called and move the focus block in the appropriate direction. The rest is easy, so here's the code:

```
void HandleGameInput()
{
    // These variables allow the user to hold the arrow keys down
    static bool down_pressed = false;
    static bool left_pressed = false;
    static bool right_pressed = false;

    // Fill our event structure with event information.
    if ( SDL_PollEvent(&g_Event) )
    {
        // Handle user manually closing game window
        if (g_Event.type == SDL_QUIT)
        {
            // While state stack isn't empty, pop
            while (!g_StateStack.empty())
            {
                g_StateStack.pop();
            }

            return; // game is over, exit the function
        }

        // Handle keyboard input here
        if (g_Event.type == SDL_KEYDOWN)
        {
            if (g_Event.key.keysym.sym == SDLK_ESCAPE)
```

```
            {
                g_StateStack.pop();

                return; // this state is done, exit the function
            }

            if (g_Event.key.keysym.sym == SDLK_UP)
            {
                // Check collisions before rotating
                if (!CheckRotationCollisions(g_FocusBlock))
                {
                    g_FocusBlock->Rotate();
                }
            }
            // For the left, right, and down arrow keys, we just set
a bool variable
            if (g_Event.key.keysym.sym == SDLK_LEFT)
            {
                left_pressed = true;
            }
            if (g_Event.key.keysym.sym == SDLK_RIGHT)
            {
                right_pressed = true;
            }
            if (g_Event.key.keysym.sym == SDLK_DOWN)
            {
                down_pressed = true;
            }
        }
        // If player lifts key, set bool variable to false
        if (g_Event.type == SDL_KEYUP)
        {
            if (g_Event.key.keysym.sym == SDLK_LEFT)
            {
                left_pressed = false;
            }
            if (g_Event.key.keysym.sym == SDLK_RIGHT)
            {
                right_pressed = false;
            }
            if (g_Event.key.keysym.sym == SDLK_DOWN)
            {
                down_pressed = false;
            }
        }
    }
    // Now we handle the arrow keys, making sure to check for
collisions
```

```
    if (down_pressed)
    {
        if ( !CheckWallCollisions(g_FocusBlock, DOWN) &&
            !CheckEntityCollisions(g_FocusBlock, DOWN) )
        {
            g_FocusBlock->Move(DOWN);
        }
    }
    if (left_pressed)
    {
        if ( !CheckWallCollisions(g_FocusBlock, LEFT) &&
            !CheckEntityCollisions(g_FocusBlock, LEFT) )
        {
            g_FocusBlock->Move(LEFT);
        }
    }
    if (right_pressed)
    {
        if ( !CheckWallCollisions(g_FocusBlock, RIGHT) &&
            !CheckEntityCollisions(g_FocusBlock, RIGHT) )
        {
            g_FocusBlock->Move(RIGHT);
        }
    }
}
```

HandleWinLoseInput()

Before pushing our win or lose states onto our stack, we'll first clear the stack (covered more when we get to HandleBottomCollision()). This is so the player can press 'escape' to quit the game while in the win/lose state. If the player chooses to continue the game though, we need to push our exit and menu states back onto the stack. Aside from that, this function is just like our other input functions. Add the following to "Main.cpp":

```
// Input handling for win/lose screens.
void HandleWinLoseInput()
{
    if ( SDL_PollEvent(&g_Event) )
    {
        // Handle user manually closing game window
        if (g_Event.type == SDL_QUIT)
        {
            // While state stack isn't empty, pop
            while (!g_StateStack.empty())
            {
                g_StateStack.pop();
            }
```

```
            return;
        }
        // Handle keyboard input here
        if (g_Event.type == SDL_KEYDOWN)
        {
            if (g_Event.key.keysym.sym == SDLK_ESCAPE)
            {
                g_StateStack.pop();

                return;
            }
            if (g_Event.key.keysym.sym == SDLK_y)
            {
                g_StateStack.pop();
                return;
            }
            // If player chooses to continue playing, we pop off
            // current state and push exit and menu states back on.
            if (g_Event.key.keysym.sym == SDLK_n)
            {
                g_StateStack.pop();
                StateStruct temp;
                temp.StatePointer = Exit;
                g_StateStack.push(temp);
                temp.StatePointer = Menu;
                g_StateStack.push(temp);
                return;
            }
        }
    }
}
```

Falling Blocks Part 5: Collision Detection

CheckEntityCollisions()

Detecting collisions between two squares in 2D is about as easy as it gets. All we have to do is compare the distance between two squares and see if it's equal to the distance between two touching squares. We know that the distance between the centers of two touching squares will always be SQUARE_MEDIAN*2. This is just the distance from one square's center to its side plus the distance between another square's center and its side.

Note that we can also use SQUARE_MEDIAN*2 as the distance a square will move.

To get the distance between two squares, we subtract the location of one square's center from the other's. We then take the absolute value of the number we get because the distance between two squares should never be negative. Note that we could easily get a negative number depending on the order of subtraction.

Our function takes a pointer to the square that is moving and its direction. We need to get the location of the square after it moves, which is accomplished using a simple switch statement. Add the following to "Main.cpp":

```
// Check collisions between a given square and the squares in
g_OldSquares

bool CheckEntityCollisions(cSquare* square, Direction dir)
{
    // Width/height of a square. Also the distance
    // between two squares if they've collided.
    int distance = SQUARE_MEDIAN * 2;

    // Center of the given square
    int centerX = square->GetCenterX();
    int centerY = square->GetCenterY();

    // Determine the location of the square after moving
    switch (dir)
    {
        case DOWN:
        {
            centerY += distance;
        } break;

        case LEFT:
        {
            centerX -= distance;
        } break;

        case RIGHT:
        {
            centerX += distance;
        } break;
    }

    // Iterate through the old squares vector, checking for
collisions
    for (int i=0; i<g_OldSquares.size(); i++)
    {
        if ( ( abs(centerX - g_OldSquares[i]->GetCenterX() ) <
distance ) &&
            ( abs(centerY - g_OldSquares[i]->GetCenterY() ) <
```

```
distance ) )
        {
            return true;
        }
    }
    return false;
}
```

Our other CheckEntityCollisions() function takes a pointer to a block. Because a block is made up of four squares, we can get an array of pointers to these squares and just call our previous function on them. Add the following to "Main.cpp":

```
// Check collisions between a given block and the squares in
g_OldSquares
bool CheckEntityCollisions(cBlock* block, Direction dir)
{
    // Get an array of the squares that make up the given block
    cSquare** temp_array = block->GetSquares();

    // Now just call the other CheckEntityCollisions() on each square

    for (int i=0; i<4; i++)
    {
        if ( CheckEntityCollisions(temp_array[i], dir) )
            return true;
    }
    return false;
}
```

CheckWallCollisions()

To check to see if a square is going to collide with the walls of the game area, we get the location of the square after it moves and compare that with the dimensions of the game area. If the square is out of bounds, then we know that there's been a collision. Add the following to "Main.cpp":

```
// Check collisions between a given square and the sides of the game
area
bool CheckWallCollisions(cSquare* square, Direction dir)
{
    // Get the center of the square
    int x = square->GetCenterX();
    int y = square->GetCenterY();

    // Get the location of the square after moving and see if its out
of bounds
    switch (dir)
    {
```

```
        case DOWN:
        {
            if ( (y + (SQUARE_MEDIAN*2)) > GAME_AREA_BOTTOM )
            {
                return true;
            }
            else
            {
                return false;
            }
        } break;

        case LEFT:
        {
            if ( (x - (SQUARE_MEDIAN*2)) < GAME_AREA_LEFT )
            {
                return true;
            }
            else
            {
                return false;
            }
        } break;

        case RIGHT:
        {
            if ( (x + (SQUARE_MEDIAN*2)) > GAME_AREA_RIGHT )
            {
                return true;
            }
            else
            {
                return false;
            }
        } break;
    }

    return false;
}
```

For blocks, we do the exact same thing with CheckWallCollisions() as we did with CheckEntityCollisions(). Add the following to "Main.cpp":

```
// Check for collisions between a given block a the sides of the game
area
bool CheckWallCollisions(cBlock* block, Direction dir)
{
    // Get an array of squares that make up the given block
    cSquare** temp_array = block->GetSquares();

    // Call other CheckWallCollisions() on each square
    for (int i=0; i<4; i++)
```

```
    {
        if ( CheckWallCollisions(temp_array[i], dir) )
        return true;
    }
    return false;
}
```

CheckRotationCollisions()

Thanks to cBlock's GetRotatedSquares() function, checking for rotation collisions is really simple. The same two events that can occur when we move a block can occur when we rotate it. It can either hit the walls of the game area or it can hit a square. The process is the exact same as it was for the last two collision detection functions. Note that we always delete the array before returning anything.

Add the following to "Main.cpp":

```
bool CheckRotationCollisions(cBlock* block)
{
    // Get an array of values for the locations of the rotated
block's squares
    int* temp_array = block->GetRotatedSquares();

    // Distance between two touching squares
    int distance = SQUARE_MEDIAN * 2;

    for (int i=0; i<4; i++)
    {
        // Check to see if the block will go out of bounds
        if ( (temp_array[i*2] < GAME_AREA_LEFT) ||
             (temp_array[i*2] > GAME_AREA_RIGHT) )
        {
            delete temp_array;
            return true;
        }

        if ( temp_array[i*2+1] > GAME_AREA_BOTTOM )
        {
            delete temp_array;
            return true;
        }

        // Check to see if the block will collide with any squares
        for (int index=0; index<g_OldSquares.size(); index++)
        {
            if ( ( abs(temp_array[i*2] - g_OldSquares[index]-
>GetCenterX()) < distance ) &&
                 ( abs(temp_array[i*2+1] - g_OldSquares[index]-
>GetCenterY()) < distance ) )
            {
```

```
                delete temp_array;
                return true;
            }
        }
    }
    delete temp_array;
    return false;
}
```

Falling Blocks Part 6: Changing the Focus Block

When the focus block can no longer move, we need to add its squares to the game area, check for any completed rows, increase the player's score and level if necessary, increase the focus block's speed if the level changes, change the focus block, and check to see if the player has won or lost. We'll handle these processes in five serparate functions.

HandleBottomCollision()

When the focus block can't move anymore, this is the function that gets called. It will handle all of the functionality we just discussed. Some of the processes will be delegated to other functions, but we'll call them from here.

We'll start with a call to a function that checks for completed lines. If any lines are completed, we'll increase the player's score and check to see if it's time to change levels. If the level changes, we'll increase the focus block's speed and check to see if the player has finished the last level.

When we change the focus block, we set it back to the top of the game area. This is a good time to see if the player has lost because if the focus block can't move now, we know that the squares have built up to the top of the game area.

Add the following to "Main.cpp":

```
void HandleBottomCollision()
{
    ChangeFocusBlock();
    // Check for completed lines and store the number of lines
completed
    int num_lines = CheckCompletedLines();
    if ( num_lines > 0 )
    {
        // Increase player's score according to number of lines
```

```
completed
        g_Score += POINTS_PER_LINE * num_lines;

        // Check to see if it's time for a new level
        if (g_Score >= g_Level * POINTS_PER_LEVEL)
        {
            g_Level++;
            CheckWin(); // check for a win after increasing the level

            g_FocusBlockSpeed -= SPEED_CHANGE; // shorten the focus
blocks movement interval
        }
    }

    // Now would be a good time to check to see if the player has
lost
    CheckLoss();
}
```

ChangeFocusBlock()

Before we change the focus block, we need to get a pointer to its array of squares and add them to the squares in the game area. We then need to delete the current focus block and assign the next block in line as the focus block. Because the next block in line is in the bottom right of the screen, we need to call SetupSquares() to move it to the top of the game area.

After that, we just assign our next block in line to a new, randomly choosen block.

Add the following to "Main.cpp":

```
// Add the squares of the focus block to g_OldSquares
// and set the next block as the focus block.
void ChangeFocusBlock()
{
    // Get an array of pointers to the focus block squares
    cSquare** square_array = g_FocusBlock->GetSquares();

    // Add focus block squares to g_OldSquares
    for (int i=0; i<4; i++)
    {
        g_OldSquares.push_back(square_array[i]);
    }

    delete g_FocusBlock; // delete the current focus block
    g_FocusBlock = g_NextBlock; // set the focus block to the next
block
    g_FocusBlock->SetupSquares(BLOCK_START_X, BLOCK_START_Y,
g_Bitmap);
```

```
    // Set the next block to a new block of random type
    g_NextBlock = new cBlock(NEXT_BLOCK_CIRCLE_X,
NEXT_BLOCK_CIRCLE_Y,
                             g_Bitmap, (BlockType)(rand()%7));
}
```

CheckCompletedLines()

To check for completed lines, we'll create an array that stores the number of squares in each row. There are 13 rows so we make our array that size. We then iterate through our vector of squares and increment the appropriate index in our array. Note that when we intialize our array, we want all of its indices set to zero. We have to do this manually because the compiler will just fill it with junk values if we don't.

Determining what row a certain square is in takes a bit of math. First, we need to know the size of one row. This is the same as the size of one block, SQUARE_MEDIAN*2. We then need to know where the bottom row is. Remember that we store our squares by their centers so we should find the location of the center of the bottom row. This is accomplished by subtracting SQUARE_MEDIAN from the location of the bottom of the game area (GAME_AREA_BOTTOM).

To understand how we determine which row a square is in, we need an example. Let's assume that our rows are 20 units in height and that the center of the first row is located at 10 units from the top of the screen. If a square's Y value is 10 and we divide it by 20 we get zero. This is because we are using integers so the remainder gets chopped off. Zero is the index of our first row because we always start counting at zero. If the square's Y value is 30 and we divide it by 20, we get 1. If the square's Y value is 50 and we divide it by 20 we get 2.

This seems to be working perfectly but there is one problem. Our top row doesn't start at 10 pixels from the top of the window. The game area actually starts a bit further down. To get the location of the top row, we subtract the location of the bottom row by one less than the number of rows times the size of each row. What this does is move us up by 12 rows, which takes us to the top.

Now, to figure out which row a given square is in, we first subtract the location of the top of the game area from the location of the square. This effectively eliminates the part of the background that comes before the game area. We then just divide by the size of a single row.

Our function returns the number of lines that have been completed so we'll have a variable to keep track of that. We first iterate through the entire vector of squares,

incrementing the indices of our counter array that correspond to the rows in which our squares are in.

We then search through our counter array and see if any rows have been completed. When we find a completed row, we increment our line counter and erase the squares in the completed row.

After erasing any lines, we have to move all of the squares down that were above those lines.

Add the following to "Main.cpp":

```
// Return amount of lines cleared or zero if no lines were cleared
int CheckCompletedLines()
{
    // Store the amount of squares in each row in an array
    int squares_per_row[13];

    // The compiler will fill the array with junk values if we don't
do this
    for (int index=0; index<13; index++)
        squares_per_row[index] = 0;

    int row_size = SQUARE_MEDIAN * 2; // pixel size of one row
    int bottom = GAME_AREA_BOTTOM - SQUARE_MEDIAN; // center of
bottom row
    int top = bottom - (12 * row_size); // center of top row

    int num_lines = 0; // number of lines cleared
    int row; // multipurpose variable

    // Check for full lines
    for (int i=0; i<g_OldSquares.size(); i++)
    {
        // Get the row the current square is in
        row = (g_OldSquares[i]->GetCenterY() - top) / row_size;

        // Increment the appropriate row counter
        squares_per_row[row]++;
    }

    // Erase any full lines
    for (int line=0; line<13; line++)
    {
        // Check for completed lines
        if (squares_per_row[line] == SQUARES_PER_ROW)
        {
            // Keep track of how many lines have been completed
            num_lines++;

            // Find any squares in current row and remove them
            for (int index=0; index<g_OldSquares.size(); index++)
            {
```

```
                if ( ( (g_OldSquares[index]->GetCenterY() - top) /
row_size ) == line )
                {
                    // delete the square
                    delete g_OldSquares[index];
                    // remove it from the vector
                    g_OldSquares.erase(g_OldSquares.begin() + index);
                    // When we delete a square, the next square in
the vector takes
                    // its place. We have to be sure to stay at the
current index so
                    // we don't skip any squares. For example, if we
delete the first
                    // square, the second square now becomes the
first. We have to
                    // stay at the current (first) index so we can
check the second
                    // square (now the first).
                    index--;
                }
            }
        }
    }

    // Move squares above cleared line down
    for (int index=0; index<g_OldSquares.size(); index++)
    {
        for (int line=0; line<13; line++)
        {
            // Determine if this row was filled
            if (squares_per_row[line] == SQUARES_PER_ROW)
            {
                // If it was, get the location of it within the game
area
                row = (g_OldSquares[index]->GetCenterY() - top) /
row_size;
                // Now move any squares above that row down one
                if ( row < line )
                {
                    g_OldSquares[index]->Move(DOWN);
                }
            }
        }
    }

    return num_lines;
}
```

CheckWin()

To check to see if the player has beat the game, we just see if the current level is beyond the maximum number of levels. If it is, we pop all of the states off our stack and push the game won state on. We also need to clear the old squares array in case the user decides to have another game. Add the following to "Main.cpp":

```
// Check to see if player has won. Handle winning condition if
needed.
void CheckWin()
{
    // If current level is greater than number of levels, player has
won
    if (g_Level > NUM_LEVELS)
    {
        // Pop all states
        while (!g_StateStack.empty())
        {
            g_StateStack.pop();
        }
        // Push the victory state onto the stack
        StateStruct win;
        win.StatePointer = GameWon;
        g_StateStack.push(win);
    }
}
```

CheckLoss()

CheckLoss() will only ever be called when the focus block is at the top of the game area. We just have to check to see if the focus block can move down. If it can't, we know that the squares have built up too high and the game is over. We then do the same thing that we did for CheckWin(). Add the following to "Main.cpp":

```
void CheckLoss()
{
    // We call this function when the focus block is at the top of
that
    // game area. If the focus block is stuck now, the game is over.
    if ( CheckEntityCollisions(g_FocusBlock, DOWN) )
    {
        // Clear the old squares vector
        for (int i=0; i<g_OldSquares.size(); i++)
        {
            delete g_OldSquares[i];
        }
```

```
        g_OldSquares.clear();

        // Pop all states
        while (!g_StateStack.empty())
        {
            g_StateStack.pop();
        }

        // Push the losing state onto the stack
        StateStruct lose;
        lose.StatePointer = GameLost;

        g_StateStack.push(lose);
    }
}
```

Conclusion

I hope you've enjoyed this tutorial and that you're now comfortable with how a game project works. The next two tutorials will be much shorter and easier. I strongly suggest you read through them and try some similarly simple game projects on your own. There's no way you can go on to more complex projects until you master this stuff. Trust me, I tried skipping this stuff and ended up wasting months and months on projects I wasn't ready for.