Block Breaker (Breakout)

What this tutorial covers

- The creation of a Breakout clone
- Loading level information from a file
- Building off of previously written code

Introduction

This tutorial will move faster than previous tutorials because you should already understand most of what's here from the Pong tutorial. Breakout really is just an extension of Pong. All we need to do is cut out the computer opponent and replace it with some blocks that will "break" when the ball hits them.

To make things a little more interesting, we'll be loading the locations of the blocks from .txt files. It's important to get used to loading data from external files because you'll be doing it like crazy for larger projects. Level data, model information, scripts, and generally all of the actual content in a game is loaded from external files. Not only does this allow you to load information from data files that were created by other programs (thus saving you from having to enter that information in by hand), this also saves you from having to recompile every time you need to change a value. If your program loads the data in at run-time, you just have to make the necessary changes to your files and re-run the program. No recompiling required.

Although we'll be building off of the code from the Pong tutorial, I think that it would get confusing if I just told you what changes to make. Instead, I'll be going through all of the code for the project and will let you know when we've already covered something. With that in mind, it's time to start a new project!

Getting started

Click Here if you need the files from the Introduction tutorial.

The first thing we need to do is start a new project called "Block Breaker" 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->Block Breaker 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.

The Code

Defines.h

We'll be storing our blocks in an array, so we need to specify how many blocks we want when we intialize the array. We might want to change this value later, so we'll define it here.

Since the only real strategy in Breakout is to get the ball above the blocks (so it'll keep rebounding off of the "roof"), it's a good idea to always have space between the sides of the screen and the blocks. It's tedious to have to break through a bunch of blocks before you can get to the roof. We'll define the amount of space here.

The player will be given a certain amount of lives and will have to pass a certain amount of levels. We'll define that information here too.

The rest should be self-explanatory, so here's what you should add after #define FRAME_RATE 1000/FRAMES_PER_SECOND:

```
// Location of images within bitmap
#define PADDLE_BITMAP_X  0
#define PADDLE_BITMAP_Y  0
#define BALL_BITMAP_X    100
#define BALL_BITMAP_Y    0
#define YELLOW_X         0
#define YELLOW_Y         20
#define RED_X            0
#define RED_Y            40
#define BLUE_X           80
#define BLUE_Y           20
#define GREEN_X          80
#define GREEN_Y          40

// Minimum distance from the side of the screen to a block
#define BLOCK_SCREEN_BUFFER 40

// Maximum number of blocks allowed
#define MAX_BLOCKS 80

// Number of rows and columns of blocks
#define NUM_ROWS 6
#define NUMCOLS    9

// Location of the player's paddle in the game
```

```
#define PLAYER_Y 550

// Dimensions of a paddle
#define PADDLE_WIDTH 100
#define PADDLE_HEIGHT 20

// Dimensions of a block
#define BLOCK_WIDTH 80
#define BLOCK_HEIGHT 20

// Diameter of the ball
#define BALL_DIAMETER 20

// Paddle speed
#define PLAYER_SPEED 10

// Ball speeds
#define BALL_SPEED_MODIFIER 5  // divide location on paddle by this
#define BALL_SPEED_Y        10 // max speed of ball along y axis

// Maximum number of times the player can miss the ball
#define NUM_LIVES 5

// Number of levels, increase this value to add new levels
#define NUM_LEVELS 3

// Locations of output text
#define LIVES_X 5
#define LIVES_Y 5
#define LEVEL_X 75
#define LEVEL_Y 5
```

Includes

We'll use the std::string for building output strings as well as for building file names (covered later). For file I/O, we'll use fstream. I'll show you how to perform file I/O when we get to initializing our blocks. For now, add this to "Main.cpp":

```
#include <string>  // We'll use the STL string for text output and
for our file names
#include <fstream> // We need to read in our levels from files
```

Global Data

Although the Entity structure we used in the Pong tutorial would work here for the paddle and the ball, I decided to split them into two different structures. The structure we'll be using for our blocks will be slighty different and I didn't want to have one general struct (Entity) and one specific one (Block). Thought it might get confusing...

You should have no problem with the Ball and Paddle structures, so I'll just post them.

The only change is that the Paddle struct no longer has a y_speed variable. The Ball struct is just the Entity struct with a new name. Add the following to "Main.cpp":

```
// The paddle only moves horizontally so there's no need for a
y_speed variable
struct Paddle
{
    SDL_Rect screen_location; // location on screen
    SDL_Rect bitmap_location; // location of image in bitmap

    int x_speed;
};

// The ball moves in any direction so we need to have two speed
variables
struct Ball
{
    SDL_Rect screen_location; // location on screen
    SDL_Rect bitmap_location; // location of image in bitmap

    int x_speed;
    int y_speed;
};
```

The Block structure doesn't need speed variables but it does need to keep track of the number of times the block has been hit. The blocks will change color when they get hit until their hit count reaches zero. Here's the Block struct:

```
// The block just stores it's location and the amount of times it has been hit
struct Block
{
    SDL_Rect screen_location; // location on screen
    SDL_Rect bitmap_location; // location of image in bitmap

    int num_hits; // "health"
};
```

For global data we need the player's paddle, the ball, the player's lives, the current level, the number of blocks, and an array to hold the blocks.

```
Paddle g_Player;             // The player's paddle
Ball g_Ball;                 // The game ball
int g_Lives;                 // Player's lives
int g_Level = 1;             // Current level
int g_NumBlocks = 0;         // Keep track of number of blocks
Block g_Blocks[MAX_BLOCKS];  // The blocks we're breaking
```

Notice that we keep track of the number of blocks. Since the number of blocks in our game will vary, the block array will almost never be full. When we loop through the array to draw the blocks or detect collisions, we'll need to know how far into the array to look. We don't want to loop into a part of the array that's empty and try to work with something that isn't there.

Function Prototypes

These functions should look pretty familar by now, add them right after the prototype for Exit():

```
void GameWon();
void GameLost();

// Helper functions for the main game state functions
...
void HandleWinLoseInput();
```

You can also delete the prototype and definition for DrawBackground(), unless you want a background for your game.

For collision detection, we need a function for checking collisions between the ball and the player's paddle, and one for checking for collisions between the ball and the blocks. We'll also have a function that handles the event of a block being hit by the ball. This function will take the index of the block that's been hit as a parameter.

You'll notice that we no longer pass parameters to CheckBallCollisions(). This is because there's only one paddle now. You'll also notice the I've included a function that checks to see if a point is inside a rectangle. Since our collision detection is based on checking for bounding rectangles, this function should save us a lot of time.

```
bool CheckBallCollisions();
void CheckBlockCollisions();
void HandleBlockCollision(int index);
bool CheckPointInRect(int x, int y, SDL_Rect rect);
```

The following prototypes are straight out of the Pong tutorial with the exception of ChangeLevel() which...changes the level.

```
void HandleBall();
void MoveBall();
void HandleLoss();
void HandleWin();
void ChangeLevel();
```

The last function to add is InitBlocks(). This function will load the locations and hit

counts of the blocks according to what level we're on. We'll call it at the beginning of the game, as well as when we change levels or restart the game.

void InitBlocks();

Init()

Everything in Init() has already been covered, so I'll just give you the code. The only real change here is that we call InitBlocks().

```
void Init()
{
    // Initiliaze SDL video and our timer.
    SDL_Init( SDL_INIT_VIDEO | SDL_INIT_TIMER);
    // Setup our window's dimensions, bits-per-pixel (0 tells SDL to
choose for us),
    // and video format (SDL_ANYFORMAT leaves the decision to SDL).
This function
    // returns a pointer to our window which we assign to g_Window.
    g_Window = SDL_SetVideoMode(WINDOW_WIDTH, WINDOW_HEIGHT, 0,
SDL_ANYFORMAT);
    // Set the title of our window.
    SDL_WM_SetCaption(WINDOW_CAPTION, 0);
    // Get the number of ticks since SDL was initialized.
    g_Timer = SDL_GetTicks();
    // Initialize the player's data

    // screen locations
    g_Player.screen_location.x = (WINDOW_WIDTH / 2) - (PADDLE_WIDTH /
2); // center screen
    g_Player.screen_location.y = PLAYER_Y;
    g_Player.screen_location.w = PADDLE_WIDTH;
    g_Player.screen_location.h = PADDLE_HEIGHT;
    // image location
    g_Player.bitmap_location.x = PADDLE_BITMAP_X;
    g_Player.bitmap_location.y = PADDLE_BITMAP_Y;
    g_Player.bitmap_location.w = PADDLE_WIDTH;
    g_Player.bitmap_location.h = PADDLE_HEIGHT;
    // player speed
    g_Player.x_speed = PLAYER_SPEED;
    // lives
    g_Lives = NUM_LIVES;
    // Initialize the ball's data //

    // screen location
    g_Ball.screen_location.x = (WINDOW_WIDTH / 2) - (BALL_DIAMETER /
2); // center screen
```

```
    g_Ball.screen_location.y = (WINDOW_HEIGHT / 2) - (BALL_DIAMETER /
2); // center screen
    g_Ball.screen_location.w = BALL_DIAMETER;
    g_Ball.screen_location.h = BALL_DIAMETER;
    // image location
    g_Ball.bitmap_location.x = BALL_BITMAP_X;
    g_Ball.bitmap_location.y = BALL_BITMAP_Y;
    g_Ball.bitmap_location.w = BALL_DIAMETER;
    g_Ball.bitmap_location.h = BALL_DIAMETER;
    // speeds
    g_Ball.x_speed = 0;
    g_Ball.y_speed = 0;

    // We'll need to initialize our blocks for each level, so we have
a
    // separate function handle it
    InitBlocks();

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

    // Set our transparent color (magenta)
    SDL_SetColorKey( g_Bitmap, SDL_SRCCOLORKEY, SDL_MapRGB(g_Bitmap-
>format, 255, 0, 255) );

    // We start by adding a pointer to our exit state, this way
    // it will be the last thing the player sees of the game.
    StateStruct state;
    state.StatePointer = Exit;
    g_StateStack.push(state);

    // Then we add a pointer to our menu state, this will
    // be the first thing the player sees of our game.
    state.StatePointer = Menu;
    g_StateStack.push(state);

    // Initialize the true type font library.
    TTF_Init();
}
```

InitBlocks()

Loading information from external files can be frustrating at times. If you try to perform the file I/O before you even know that your program works properly, it will be very hard to tell whether the bug is in your game code, your I/O code, or if there's something wrong with the file itself. When I first wrote the code for this tutorial, I used a loop to intialize the blocks to dummy values and made sure that the code worked. I then went ahead and started loading level information from a file.

So what information are we going to load from a file? For this project, we'll only be

loading the number of hits our blocks can take (i.e. their health). We'll store this information in .txt files. Here's what "level1.txt", the file for the first level, looks like:

0 3 3 3 3 3 3 3 0
2 0 3 3 3 3 3 0 2
2 2 0 3 3 3 0 2 2
2 2 2 0 3 0 2 2 2
2 2 2 2 0 2 2 2 2
2 2 2 0 4 0 2 2 2

Our game area will consist of 6 rows and 9 columns of blocks. Each block will be a certain color that depends on how many hits it has left. When we read in these values, we'll skip any blocks that have been given a zero for their hit count.

You'll notice that there are spaces between each number in "level1.txt". When we read in a value, we'll use the overloaded >> operator, which will read a string of characters from our file until it reaches a space or new line character. We only want it to read one character at a time, so we put spaces between each number.

The first thing we need to do in InitBlocks() is declare a file stream object. This is the object that contains the functions we need to carry out our I/O. We then construct a string according to what level the player is on. This code looks a lot like what we did for outputting the player and computer scores in the Pong tutorial, only now we're constructing a string for the name of a file. After that, a call to fstream's open() function will allow us to begin reading information from our file. Add the following code to "Main.cpp":

```
void InitBlocks()
{
    fstream inFile;

    // The following code creates a string storing the proper file
name. If
    // g_Level = 1, we get: "data\\level" + "1" + ".txt" =
"data\\level1.txt"
    char level_num[256];              // for itoa
    string file_name = "data\\level"; // the file will always start
with "level"
    itoa(g_Level, level_num, 10);     // convert g_Level to a string
    file_name.append(level_num);      // append the level number
    file_name.append(".txt");         // we'll just use txt's for our
levels
```

```
    // Open the file for input. Note that this function takes a
    // char* so we need to use the std::string's c_str() function.
    // ios::in specifies that we want to read from this file.
    inFile.open(file_name.c_str(), ios::in);
```

We now loop through each block in our game and read in its hit count from the file. We'll be using nested for loops for this. The outside loop will be for the rows of blocks, the inside loop will be for the columns. We'll set the location of the current block being intialized according to where we are in in the loop. Note that we'll also need a variable to keep track of what block we're at in the array. We'll call this variable index. Here's the code for the start of the loop:

```
      int index = 0; // used to index blocks in g_Blocks array
    // Temporary variable to hold the number we read in from our file

    int temp_hits;
    // Iterate through each row and column of our blocks
    for (int row=1; row<=NUM_ROWS; row++)
    {
        for (int col=1; col<=NUMCOLS; col++)
        {
```

We now read in the next value from our file. We'll store this value in temp_hits so we can make sure it's not zero. If it is, we'll just skip the current block. There's no need to store blocks with zero hits left. If the hit count isn't zero, we intialize the block's num_hits variable as well as it's location. Notice that we apply BLOCK_SCREEN_BUFFER to make sure our blocks are always a specific distance from the sides of the screen.

```
          // Read the next value into temp_hits
            inFile >> temp_hits;
            // If temp_hits is zero, we go on to the next block
            if (temp_hits != 0)
            {
                g_Blocks[index].num_hits = temp_hits;

                // We set the location of the block according to what
row and column
                // we're on in our loop. Notice that we use
BLOCK_SCREEN_BUFFER to set
                // the blocks away from the sides of the screen.
                g_Blocks[index].screen_location.x = col*BLOCK_WIDTH -
BLOCK_SCREEN_BUFFER;
                g_Blocks[index].screen_location.y = row*BLOCK_HEIGHT
+ BLOCK_SCREEN_BUFFER;
                g_Blocks[index].screen_location.w = BLOCK_WIDTH;
                g_Blocks[index].screen_location.h = BLOCK_HEIGHT;
```

```
                g_Blocks[index].bitmap_location.w = BLOCK_WIDTH;
                g_Blocks[index].bitmap_location.h = BLOCK_HEIGHT;
```

Now we set the color of the block according to its hit count. A switch statement handles this process nicely. With that done, we increment the index variable as well as g_NumBlocks. Remember that every time we add a block we have to increment g_Numblocks and we have to decrement it every time we remove a block.

```
            // Now we set the bitmap location rect according to
num_hits
                switch (g_Blocks[index].num_hits)
                {
                    case 1:
                    {
                        g_Blocks[index].bitmap_location.x = YELLOW_X;
                        g_Blocks[index].bitmap_location.y = YELLOW_Y;
                    } break;
                    case 2:
                    {
                        g_Blocks[index].bitmap_location.x = RED_X;
                        g_Blocks[index].bitmap_location.y = RED_Y;
                    } break;
                    case 3:
                    {
                        g_Blocks[index].bitmap_location.x = GREEN_X;
                        g_Blocks[index].bitmap_location.y = GREEN_Y;
                    } break;
                    case 4:
                    {
                        g_Blocks[index].bitmap_location.x = BLUE_X;
                        g_Blocks[index].bitmap_location.y = BLUE_Y;
                    } break;
                }
                // For future use, keep track of how many blocks we
have.
                g_NumBlocks++;
                index++; // move to next block
            }
        }
    }
```

With our loop completed, we just have to call close() to tell fstream that we are done reading from the file.

```
   inFile.close();
}
```

Game()

We actually don't need to make that many changes to Game(). As with the Pong tutorial, we need to make a call to HandleBall(). Add the line

HandleBall();

just below the call to HandleGameInput().

Now we need to draw the ball, paddle, and blocks. For the blocks, we iterate through g_Blocks, drawing each block. Add the following after the call to ClearScreen():

```
// Draw the paddle and the ball
SDL_BlitSurface(g_Bitmap, &g_Player.bitmap_location, g_Window,
     &g_Player.screen_location);
SDL_BlitSurface(g_Bitmap, &g_Ball.bitmap_location, g_Window,
     &g_Ball.screen_location);

// Iterate through the blocks array, drawing each block
for (int i=0; i<g_NumBlocks; i++)
{
    SDL_BlitSurface(g_Bitmap, &g_Blocks[i].bitmap_location, g_Window,
        &g_Blocks[i].screen_location);
}
```

All that's left is to display the current level and the number of lives the player has left. This code should look very familar to you:

```
// Output the number of lives the player has left and the current
level
char buffer[256];

string lives = "Lives: ";
itoa(g_Lives, buffer, 10);
lives.append(buffer);

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

DisplayText(lives, LIVES_X, LIVES_Y, 12, 66, 239, 16, 0, 0, 0);
DisplayText(level, LEVEL_X, LEVEL_Y, 12, 66, 239, 16, 0, 0, 0);
```

GameWon() and GameLost()

Since we're using the code from the Introduction tutorial, we need to add GameWon(), GameLost(), and HandleWinLoseInput(). Instead of making you copy them out of the Pong tutorial, I'll just give you the code here:

```
// Display a victory message.
```

```
void GameWon()
{
    if ( (SDL_GetTicks() - g_Timer) >= FRAME_RATE )
    {
        HandleWinLoseInput();
        ClearScreen();
        DisplayText("You Win!!!", 350, 250, 12, 255, 255, 255, 0, 0,
0);
        DisplayText("Quit Game (Y or N)?", 350, 270, 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.", 350, 250, 12, 255, 255, 255, 0, 0,
0);
        DisplayText("Quit Game (Y or N)?", 350, 270, 12, 255, 255,
255, 0, 0, 0);
        SDL_UpdateRect(g_Window, 0, 0, 0, 0);
        g_Timer = SDL_GetTicks();
    }
}
// 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;
            }
        }
    }
}
```

HandleGameInput()

HandleGameInput() is going to look just like the one from the Pong tutorial, only we now want to check for collisions with the sides of the walls here. Remember that we got rid of HandleWallCollisions() because it took an Entity as a parameter but we split the ball and paddle into different structures. We could write two functions, one for the ball and one for the paddle, but it really wouldn't save us any coding. There's nothing new here, so I have to dump some more code on you. Don't worry, we're almost at the new stuff.

```
void HandleGameInput()
{
    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_SPACE)
            {
                // Player can hit 'space' to make the ball move at
start
                if (g_Ball.y_speed == 0)
                    g_Ball.y_speed = BALL_SPEED_Y;
            }
            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.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;
            }
        }
    }
    // This is where we actually move the paddle
```

```
    if (left_pressed)
    {
        // Notice that we do this here now instead of in a separate
function
        if ( (g_Player.screen_location.x - PLAYER_SPEED) >= 0 )
        {
            g_Player.screen_location.x -= PLAYER_SPEED;
        }
    }
    if (right_pressed)
    {
        if ( (g_Player.screen_location.x + PLAYER_SPEED) <=
WINDOW_WIDTH )
        {
            g_Player.screen_location.x += PLAYER_SPEED;
        }
    }
}
```

CheckBallCollisions(), HandleBall(), and MoveBall()

You're probably sick of copy-pasting by now, so let's get the rest over with. CheckBallCollisions() just handles collisions with the player's paddle now. There's no need to pass it any parameters because there's only one paddle. HandleBall() is the exact same except for a call to HandleBlockCollisions() at the end.

MoveBall() has changed a bit more. Since we got rid of HandleWallCollissions(), we check for wall collisions here. We also now check for collisions with the top of the screen so the ball doesn't disappear into the abyss. If the ball passes the player, we reset it as before and then decrement the player's lives. If the player has zero lives left, we call HandleLoss().

```
// Check to see if the ball is going to hit the paddle
bool CheckBallCollisions()
{
    // Temporary values to keep things tidy
    int ball_x = g_Ball.screen_location.x;
    int ball_y = g_Ball.screen_location.y;
    int ball_width = g_Ball.screen_location.w;
    int ball_height = g_Ball.screen_location.h;
    int ball_speed = g_Ball.y_speed;

    int paddle_x = g_Player.screen_location.x;
    int paddle_y = g_Player.screen_location.y;
    int paddle_width = g_Player.screen_location.w;
    int paddle_height = g_Player.screen_location.h;

    // Check to see if ball is in Y range of the player's paddle.
```

```
    // We check its speed to see if it's even moving towards the
player's paddle.
    if ( (ball_speed > 0) && (ball_y + ball_height >= paddle_y) &&
        (ball_y + ball_height <= paddle_y + paddle_height) ) // side
hit
    {
        // If ball is in the X range of the paddle, return true.
        if ( (ball_x <= paddle_x + paddle_width) && (ball_x +
ball_width >= paddle_x) )
        {
            return true;
        }
    }

    return false;
}

void HandleBall()
{
    // Start by moving the ball
    MoveBall();

    if ( CheckBallCollisions() )
    {
        // Get center location of paddle
        int paddle_center = g_Player.screen_location.x +
g_Player.screen_location.w / 2;
        int ball_center = g_Ball.screen_location.x +
g_Ball.screen_location.w / 2;

        // Find the location on the paddle that the ball hit
        int paddle_location = ball_center - paddle_center;

        // Increase X speed according to distance from center of
paddle.
        g_Ball.x_speed = paddle_location / BALL_SPEED_MODIFIER;
        g_Ball.y_speed = -g_Ball.y_speed;
    }

    // Check for collisions with blocks
    CheckBlockCollisions();
}

void MoveBall()
{
    g_Ball.screen_location.x += g_Ball.x_speed;
    g_Ball.screen_location.y += g_Ball.y_speed;

    // If the ball is moving left, we see if it hits the wall. If
does,
    // we change its direction. We do the same thing if it's moving
right.
    if ( ( (g_Ball.x_speed < 0) && (g_Ball.screen_location.x <= 0) )
||
```

```
        ( (g_Ball.x_speed > 0) && (g_Ball.screen_location.x >=
WINDOW_WIDTH) ) )
    {
        g_Ball.x_speed = -g_Ball.x_speed;
    }

    // If the ball is moving up, we should check to see if it hits
the 'roof'
    if ( (g_Ball.y_speed < 0) && (g_Ball.screen_location.y <= 0) )
    {
        g_Ball.y_speed = -g_Ball.y_speed;
    }

    // Check to see if ball has passed the player
    if ( g_Ball.screen_location.y >= WINDOW_HEIGHT )
    {
        g_Lives--;

        g_Ball.x_speed = 0;
        g_Ball.y_speed = 0;

        g_Ball.screen_location.x = WINDOW_WIDTH/2 -
g_Ball.screen_location.w/2;
        g_Ball.screen_location.y = WINDOW_HEIGHT/2 -
g_Ball.screen_location.h/2;

        if (g_Lives == 0)
        {
            HandleLoss();
        }
    }
}
```

CheckBlockCollisions()

When we checked for collisions between the ball and the player's paddle, we checked to see if any of the four corners of the ball had made contact with the paddle. This worked fine since we knew that the ball would always hit the top of the paddle.

Things are slightly different with the blocks though. We want the ball to hit the top and bottom of the blocks, and we also want it to hit the sides. More importantly, we need to be able to tell whether the ball has hit a side or the top/bottom of a block. Obviously if the ball hits the top/bottom of a block we want to change its vertical direction, and if it hits a side of a block we want to change its horizontal direction.

If we handle collisions by checking the corners of the ball, we'll have no idea which direction we need to deflect the ball. Think about the top-right corner of the ball hitting a block. How can we tell if it hit the block from the side, the top, or both?

The simplest solution to this problem is to check the middle points of each side of the ball. If the middle point of the left side of the ball hits something, we can be pretty confident that we need to deflect the ball to the right. To save on typing, we'll store these values in temporary variables. Here's what we have so far:

```
void CheckBlockCollisions()
{
    // Temporary values to save on typing
    int left_x = g_Ball.screen_location.x;
    int left_y = g_Ball.screen_location.y +
g_Ball.screen_location.h/2;
    int right_x = g_Ball.screen_location.x +
g_Ball.screen_location.w;
    int right_y = g_Ball.screen_location.y +
g_Ball.screen_location.h/2;
    int top_x = g_Ball.screen_location.x +
g_Ball.screen_location.w/2;
    int top_y = g_Ball.screen_location.y;
    int bottom_x = g_Ball.screen_location.x +
g_Ball.screen_location.w/2;
    int bottom_y = g_Ball.screen_location.y +
g_Ball.screen_location.h;
```

Now we need to loop through the blocks and check for collisions. One thing to note is that the ball might hit more than one block. If the top/bottom of the ball hits a block and the left/right side of the ball hits another block, we'll need to deflect it vertically and horizontally. Because of this, we'll loop through the entire array and keep track of any sides of the ball that have hit something. When we've checked every block, we'll make the appropriate changes to the ball's direction. Here's our ball-to-block collision detection loop:

```
    for (int block=0; block<g_NumBlocks; block++)
    {
        // top
        if ( CheckPointInRect(top_x, top_y,
g_Blocks[block].screen_location) )
        {
            top = true;
            HandleBlockCollision(block);
        }
        // bottom
        if ( CheckPointInRect(bottom_x, bottom_y,
g_Blocks[block].screen_location) )
        {
            bottom = true;
            HandleBlockCollision(block);
        }
        // left
```

```
        if ( CheckPointInRect(left_x, left_y,
g_Blocks[block].screen_location) )
        {
            left = true;
            HandleBlockCollision(block);
        }
        // right
        if ( CheckPointInRect(right_x, right_y,
g_Blocks[block].screen_location) )
        {
            right = true;
            HandleBlockCollision(block);
        }
    }
```

Note the use of CheckPointInRect(). We'll get to its implementation soon. All it does is return true if the given point is inside the given rect. HandleBlockCollision() takes care of removing the block. We'll go through its implementation next.

Now it's time to handle deflecting the ball. This should be pretty self-explanatory. You'll notice that we also move the ball by BALL_SPEED in the direction that we're deflecting it. This is to make sure that the ball isn't inside the block after the function finishes.

```
        if (top)
    {
        g_Ball.y_speed = -g_Ball.y_speed;
        g_Ball.screen_location.y += BALL_DIAMETER;
    }
    if (bottom)
    {
        g_Ball.y_speed = -g_Ball.y_speed;
        g_Ball.screen_location.y -= BALL_DIAMETER;
    }
    if (left)
    {
        g_Ball.x_speed = -g_Ball.x_speed;
        g_Ball.screen_location.x += BALL_DIAMETER;
    }
    if (right)
    {
        g_Ball.x_speed = -g_Ball.x_speed;
        g_Ball.screen_location.x -= BALL_DIAMETER;
    }
}
```

HandleBlockCollision()

HandleBlockCollision() takes the index to the block that has been hit and handles

changing the block's hit count. If the block's hit count reaches zero, we remove that block from the array.

Since we don't need to worry about the order of blocks in the array, we can just copy the last valid block in the array (located at g_Blocks[g_NumBlocks-1]) over top of the block at index. This effectively removes the block from the array while saving us from having to shift everything around. Note that we have to decrement g_NumBlocks so we don't access any blocks twice.

```
void HandleBlockCollision(int index)
{
    g_Blocks[index].num_hits--;
    // If num_hits is 0, the block needs to be erased
    if (g_Blocks[index].num_hits == 0)
    {
        g_Blocks[index] = g_Blocks[g_NumBlocks-1];
        g_NumBlocks--;
```

Everytime we remove a block, we need to see if it was the last block in the level. If it is, we change levels with a call to ChangeLevel().

```
       // Check to see if it's time to change the level
        if (g_NumBlocks == 0)
        {
            ChangeLevel();
        }
    }
```

If the block's hit count hasn't reached zero, we just have to change its color. This is done with almost the same switch statement as in InitBlocks(). The only difference is that we don't have to handle the block having 4 hits left.

```
      // If the hit count hasn't reached zero, we need to change the
block's color
    else
    {
        switch (g_Blocks[index].num_hits)
        {
            case 1:
            {
                g_Blocks[index].bitmap_location.x = YELLOW_X;
                g_Blocks[index].bitmap_location.y = YELLOW_Y;
            } break;
            case 2:
            {
                g_Blocks[index].bitmap_location.x = RED_X;
                g_Blocks[index].bitmap_location.y = RED_Y;
            } break;
            case 3:
```

```
            {
                g_Blocks[index].bitmap_location.x = GREEN_X;
                g_Blocks[index].bitmap_location.y = GREEN_Y;
            } break;
        }
    }
}
```

CheckPointInRect()

This function returns true if the given point is within the given rect. The algorithm is the same as we've been using in all of the tutorials. We're just putting it into its own function so we don't have to type long if statements anymore.

```
bool CheckPointInRect(int x, int y, SDL_Rect rect)
{
    if ( (x >= rect.x) && (x <= rect.x + rect.w) &&
         (y >= rect.y) && (y <= rect.y + rect.h) )
    {
        return true;
    }
    return false;
}
```

ChangeLevel()

To change the level, we first need to increment g_Level. We then check to see if the player just finished the last level, in which case we call HandleWin(). Otherwise, we reset the ball, set g_NumBlocks to zero, and call InitBlocks(). InitBlocks() will load the necessary data from the next level file.

```
void ChangeLevel()
{
    g_Level++;
    // Check to see if the player has won
    if (g_Level > NUM_LEVELS)
    {
        HandleWin();
        return;
    }
    // Reset the ball
    g_Ball.x_speed = 0;
    g_Ball.y_speed = 0;
    g_Ball.screen_location.x = WINDOW_WIDTH/2 -
g_Ball.screen_location.w/2;
```

```
    g_Ball.screen_location.y = WINDOW_HEIGHT/2 -
g_Ball.screen_location.h/2;

    g_NumBlocks = 0; // Set this to zero before calling InitBlocks()
    InitBlocks();             // InitBlocks() will load the proper
level
}
```

HandleLoss() and HandleWin()

These two functions are almost identical. They first pop all of the states off of the state stack. They then reset the ball, player's lives, level, block count, and blocks in case the player chooses to restart the game. Finally, they push the appropriate state onto the stack.

```
void HandleLoss()
{
    while ( !g_StateStack.empty() )
    {
        g_StateStack.pop();
    }

    g_Ball.x_speed = 0;
    g_Ball.y_speed = 0;

    g_Ball.screen_location.x = WINDOW_WIDTH/2 -
g_Ball.screen_location.w/2;
    g_Ball.screen_location.y = WINDOW_HEIGHT/2 -
g_Ball.screen_location.h/2;

    g_Lives = NUM_LIVES;
    g_NumBlocks = 0;
    g_Level = 1;
    InitBlocks();

    StateStruct temp;
    temp.StatePointer = GameLost;
    g_StateStack.push(temp);

}

void HandleWin()
{
    while ( !g_StateStack.empty() )
    {
        g_StateStack.pop();
    }

    g_Ball.x_speed = 0;
    g_Ball.y_speed = 0;

    g_Ball.screen_location.x = WINDOW_WIDTH/2 -
g_Ball.screen_location.w/2;
```

```
    g_Ball.screen_location.y = WINDOW_HEIGHT/2 -
g_Ball.screen_location.h/2;

    g_Lives = NUM_LIVES;
    g_NumBlocks = 0;
    g_Level = 1;
    InitBlocks();

    StateStruct temp;
    temp.StatePointer = GameWon;
    g_StateStack.push(temp);
}
```

Conclusion

That's all for this tutorial. Breakout and Pong are very similar, so there really wasn't a lot of new stuff to cover. The most important thing we did here was set up our game to read level information from external files. If you want to change the levels, you can just load the .txt files in notepad and change them around. If you want to add new levels, just create more .txt files and change NUM_LEVELS.

Click here to download the source code for this tutorial.