

Written by Billy D. Spelchan for www.BlazingGames.com

Copyright © 2003-2005 Blazing Games Inc. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the file called fdl.txt

Chapter 16

Pent Up Anger

Contents

We start off by creating the board game Pent Up Anger. This version of the game has no computer opponents, just humans playing against each other.

- A Detailed look at Pent Up Anger - What the game is about.
- Building the board - creating the game board.
- Building the Player - Generic version of the basic player piece.
- Finishing the Piece - Finished version of the player's piece.
- Building the Die - Creating an eight sided die.
- Board Layout - Piece placements for all locations on the board.
- Preparing the Pieces - Setting up the pieces on the board.
- Turn Handling - rolling the die and looping through the players.
- Selecting the Piece - adding the ability to select a piece to move.
- Moving the Piece - Animating the piece movement.
- Winning the Game - determining the winner.

A Detailed look at Pent Up Anger

Pent up Anger is an original board game that I designed which is loosely based on other board games that I have seen. The game's board is pentagon shaped. Each point of the pentagon is a player's starting location. Each player is assigned a different color and has five playing pieces numbered one through five. Their starting gate also has the numbers one through five on it.

Turns revolve clockwise around the board. Players start their turn by rolling an eight sided dice. Before the player can move, they need to get one of their pieces out of the starting gate. This is done by rolling the number that is assigned to the piece. Players can only move or start once piece a turn. Any piece that is on the board can be moved as long as the move will not take the piece past the player's loading zone. If the player lands on an opponent's piece, that piece get's taken back to the starting gate.

The loading zone is a special four square line on the board that starts just before the players starting point. Once the player's piece has landed in the loading zone, the piece is able to leave the board. To leave the board, the number of the piece must be rolled.

Players win the game once all their pieces have been removed from the board.

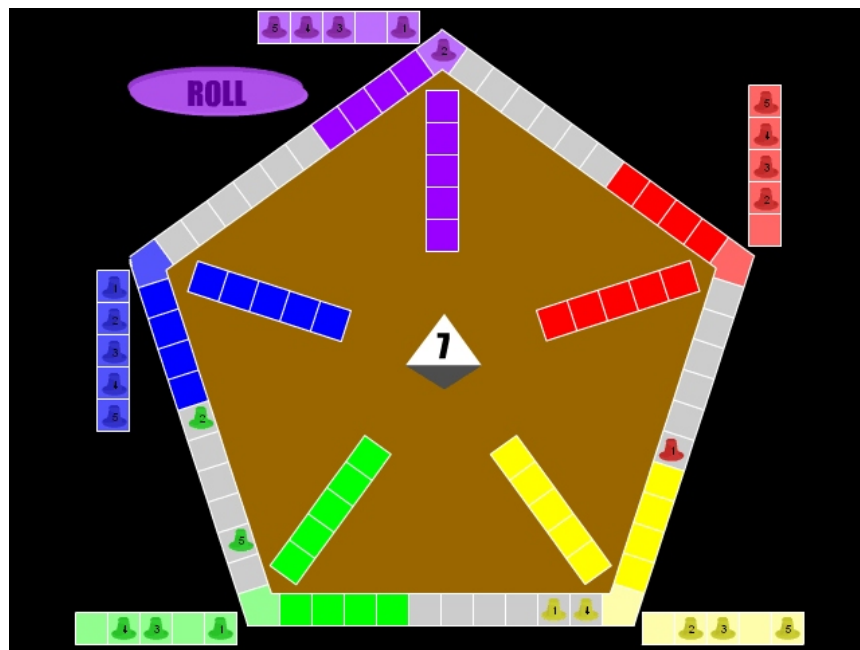


Figure 1: Pent Up Anger screen shot

Building the board

A hexagon board. Sounds complicated, but building it really isn't. First, we want 12 squares per side, with the corner square shared so we need to figure out how big to make a square. Whatever length we decide, needs to be divisible by 12. Making squares 25x25 would make a line that was 300 pixels long. If we draw a horizontal line 300 pixels long and then rotate it 72 degrees and rotate another 300 pixel horizontal line 144 degrees, we see that the combined lines go outside our 640x480 boundaries, but barely. If we use 288 pixel lines, the lines fit within the area of the screen. This means that 24x24 boxes will work.

I draw a 24x24 box. I then copy the box a couple of times and join them together. I then copy the three boxes and past the copy together. One final copy and past and we end up with a row of 12 boxes. Copy that row and rotate it 72 degrees. Join at the corner. Next past the copied boxes again and rotate 144 and place at the second corner. Again at 228 and 288 and you end up with a completed pentagon board.

Do a bit of colouring so the starting points and the exit zones are distinct. Then grab a 5 block chunk of boxes and create the inside exit zones by rotating the box as you did above.

Finally create starting zones. Turn the whole group into a movie clip.

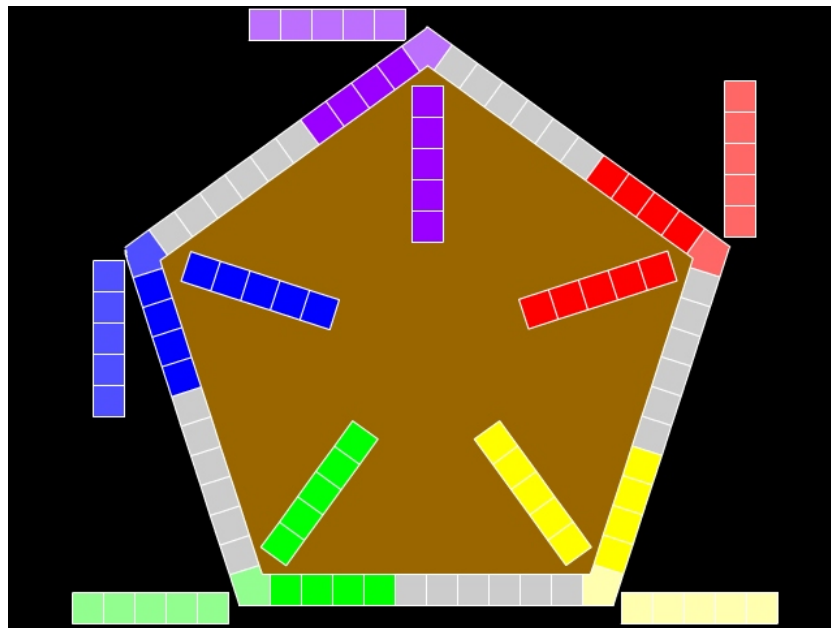


Figure 2: The Board

Building the player

Building a piece sounds simple, but there is a bit of a logistics problem. First, we need to know if the piece is movable and if it is the current piece selected. This means that a piece is going to have three states associated with it. Next, we are going to have to have a number on the piece indicating which of the five pieces the piece is. Three states with five pieces is 15. There are also five colors. fifteen times five is 75. Drawing 75 pieces is not a pleasant task so we are going to have to come up with a better solution.

First, lets have a generic piece that can be tinted. This piece will use grayscale gradients for it's color (the gradient is used to give the piece a bit more depth. Two ovals are drawn for the base and top of the piece. They have the gradient oriented horizontally. A rectangle is used to connect the two ovals. This rectangle is distorted by extending the bottom two points.

Once created, we copy the piece and remove the gradients, thereby creating a silhouette. One silhouette will be light gray, the other will be white. These silhouette images will be used for special highlight animation. Figure 3 shows the board piece and the two silhouettes.



Figure 3: Game piece and silhouettes

The player movie , which we are calling “player_image,” needs three layers. The “Code” layer is for our scripting. The “Pieces” layer is for the playing piece. Finally, the “Effects” layer is where we will place the animation.

The piece has three different states so labels for showing the states are going to be needed. I call the first state the normal state because this is what the piece will normally look like. This label is placed on the code layer on frame 2. The second state is called “Highlight” and is going to be used for indicating which of the player’s pieces can be moved. This label is placed on frame 5. Finally, we have the “Selected” label which will be placed on frame 35 and will be used for animating the piece that the player has selected while it is moving. The “Highlight” and “Selected” sequences use the animation layer beneath the piece to create a looping animation of the appropriate silhouette scaling to a larger size and then scaling back to it's original size.

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

As the movie has animation sequences controlled by some type of state variable, scripting is going to be needed in this movie clip. The initialization and main function calls for the movie are placed on frame 1.

We start by calling our initialization function, which is designed to only run once. As we will have multiple pieces, each piece will run the initialization so common state constants would be declared multiple times. To prevent this waist of time and memory, all the piece state constants are set up as global variables. To see if the global variables need to be defined, we simply check to see if one of the variables has already been defined.

```
init_piece();

function init_piece()
{
    if (piece_initialized != undefined)
        return;
    piece_initialized = true;
    piece_state = 0;
    if (_global.PIECE_NORMAL == undefined)
    {
        _global.PIECE_NORMAL = 0;
        _global.PIECE_HIGHLIGHT = 1;
        _global.PIECE_SELECTED = 2;
    }
}
```

The change state function simply changes the state of the piece and goes to the appropriate movie sequence.

```
function changeState(n)
{
    piece_state = n;
    switch (n)
    {
        case PIECE_NORMAL:
            gotoAndPlay("Normal");
            break;
        case PIECE_HIGHLIGHT:
            gotoAndPlay("Highlight");
            break;
        case PIECE_SELECTED:
            gotoAndPlay("Selected");
            break;
    }
}
```

Finally, we have a stop() on frame 4, a gotoAndPlay(“Highlight”) on frame 34 and a gotoAndPlay(“Selected”) on frame 64.

Finishing the Piece

The playing piece looks fine but there are still two things missing. The color and the numbers. To handle both of these aspects, we will create another movie which we will call “Piece.” This movie will have three layers. The “Code” layer, like always, is where we place labels and code. The “Number” layer will consist of a block of dynamic text which will be named “label_txt.” Finally, the “Piece” layer will contain the playing piece.

The movie will be broken into five sections. Each section will have a keyframe on the piece layer which will be a tinted version of the `player_image` movie. The piece will be labelled “piece_movie.” The labels used are “Purple,” “Red,” “Yellow,” “Green,” and “Blue.” On the last frame of each of these labels we place a `stop()` in the code layer.

Now, in the first frame of the movie clip, we have some functions that are used for manipulating this movie and for getting and setting some of the movies information. The first bit of code we are going to look at is the initialization code. This is designed to run once. Likewise, because we know every instance of this will run the initialization once, we have defined some global constants which are designed to only assign values if the global variables have not been defined yet.

The initialization also sets up the piece’s variables to default values. The information we need to know about a piece are the color, the number, and where on the game board the piece is currently at. By default, the piece has no defined color (white), is number X, and is located on location zero.

```
init_gamepiece()

function init_gamepiece()
{
    if (gamepiece_initialized != undefined)
        return;
    gamepiece_initialized = false;
    if (_global.PIECE_WHITE == undefined)
    {
        _global.PIECE_WHITE = -1;
        _global.PIECE_PURPLE = 0;
        _global.PIECE_RED = 1;
        _global.PIECE_YELLOW = 2;
        _global.PIECE_GREEN = 3;
        _global.PIECE_BLUE = 4;
    }
    piece_color = PIECE_WHITE;
    piece_label = "X";
    piece_location = 0;
}
```

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

The first function that we create, then, must obviously assign the piece a color and a number. To do this we simply assign the variables to the passed parameters, while also setting the label to the number of the piece. Once a color and number has been assigned to a piece, that color and number should never change, so the code for going to the appropriate color label can be placed here. This is simply a switch which goes to the appropriate color label.

```
function setPiece(c, n)
{
    piece_color = c;
    piece_label = n;
    label_txt.text = n;
    switch (c)
    {
        case PIECE_PURPLE:
            gotoAndPlay("Purple");
            break;
        case PIECE_RED:
            gotoAndPlay("Red");
            break;
        case PIECE_YELLOW:
            gotoAndPlay("Yellow");
            break;
        case PIECE_GREEN:
            gotoAndPlay("Green");
            break;
        case PIECE_BLUE:
            gotoAndPlay("Blue");
            break;
    }
}
```

This is a board game. As such pieces need to be moved. Likewise, the computer needs to know where pieces are located. For that reason we have the following functions.

```
function setLocation(id)
{
    piece_location = id;
}

function getLocation()
{
    return piece_location;
}
```

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

Now comes a function for handling the state of the piece. Remember that pieces have three states. The normal state is 0, the highlight state is 1 and the selected state is 2. When a piece is highlighted, it can be clicked on. To handle the changing of the state and the clicking of the piece, we have the following functions. Notice that the chosen function handles the click by passing the piece to a parent function (which we will be creating later).

```
function changeState(n)
{
    piece_movie.changeState(n);
    if (n == 1)
        onRelease = chosen;
    else
        onRelease = null;
}

function chosen()
{
    _parent.choosePiece(this);
}
```

Next we create a couple of functions for getting information about the color of the piece and the number of the piece as well as assigning a weight to the piece. These functions will not be needed until next chapter when we create the AI, but I figure it is best to get them out of the way while we are creating the playing piece. More information on how these functions are used will be given next chapter.

```
function getPlayer()
{
    return piece_color;
}

function getPieceNumber()
{
    return piece_label;
}

function setWeight(n)
{
    ai_weight = n;
}

function getWeight()
{
    return ai_weight;
}
```


Building the Die

This game uses an eight sided die. For those of you unfamiliar with such a die, the die looks sort of like a diamond. You use it like you would a normal six sided die, but it has eight possible results instead of six.

Building the die movie is fairly simple. First we start with an image of the die in a finished position. On a separate layer we have text for the eight different values (labelled r1 to r8). We also have mid-roll die images. Figure 4 shows the frames that make up the die.

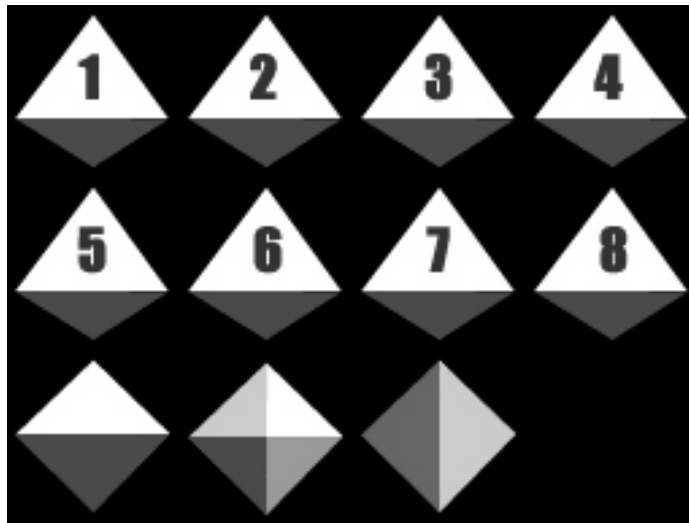


Figure 4: Die Frames

As with our other movies, we treat the die movie as a class. To handle the rolling of the die we will need some functions, but first we will initialize the die. This simply sets the value of the die to 8 and resets the listener and roll variables, which we will explain shortly.

```
init_die();

function init_die()
{
    if (die_initialized != undefined)
        return;
    die_initialized = true;
    value = 8;
    spins_remaining = 0;
    result_listener = null;
    gotoAndPlay("r8");
}
```

Rolling the die presents us with a bit of a problem. You see, we want the roll to be animated. If the roll is animated, then it can't return a value right away. To solve this problem, we have the main movie call the startRoll function to get the die rolling and then have the die caller movies roll_result function.

The actual die roll will consist of 5 to 20 spins (the number determined at random) with a spin consisting of an in between shape (the three shapes at the bottom of figure 4) followed by a number. To start the roll the number of spins is determined followed by a call to the roll_tween function.

```
function startRoll(listener)
{
    result_listener = listener;
    spins_remaining = Math.floor(Math.random() * 15) + 5;
    roll_tween();
}
```

The roll_tween function simply goes to a transition frame at random.

```
function roll_tween()
{
    var temp = Math.floor(Math.random() * 3);
    switch(temp)
    {
        case 0:
            gotoAndPlay("t1");
            break;

        case 1:
            gotoAndPlay("t2");
            break;

        case 2:
            gotoAndPlay("t3");
            break;
    }
}
```

The last frame of each transition frame contains the following line of code which simply goes to the number selection phase of the roll.

```
roll_number();
```

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

The number selection phase first reduces the number of spins remaining. If the spin is over it calls the listener function (the movie that called the roll). In either case it then goes to a number frame.

```
function roll_number()
{
    --spins_remaining;
    value = Math.floor(Math.random() * 8) + 1;
    if (spins_remaining <= 0)
    {
        result_listener.roll_result(value);
    }
    switch (value)
    {
        case 1:
            gotoAndPlay("r1");
            break;
        case 2:
            gotoAndPlay("r2");
            break;
        case 3:
            gotoAndPlay("r3");
            break;
        case 4:
            gotoAndPlay("r4");
            break;
        case 5:
            gotoAndPlay("r5");
            break;
        case 6:
            gotoAndPlay("r6");
            break;
        case 7:
            gotoAndPlay("r7");
            break;
        case 8:
            gotoAndPlay("r8");
            break;
    }
}
```

As with the transition frames, each number frame has code to continue the roll. Obviously, this needs to be done only if the spin is over, so the code is conditional.

```
if (spins_remaining > 0)
    rollTween();
else
    stop();
```

Board Layout

Pieces need locations. We need to know the coordinates of these locations so we can place the pieces on the proper screen locations. We could create a table by hand that contains all the coordinates of the locations. This is time consuming and very prone to errors. For board games that have complex layouts, this may be required. As our board is laid out in a fairly linear and consistent fashion, we can algorithmically create the table.

We still need some coordinates, which lead to a strange problem. Despite dragging an instance of the piece movie to the desired starting locations and using those coordinates, flash seems to shift the location. After a little investigation, I found out that the properties panel is actually giving you the top-left location of the object, not the location of the anchor.

All the pieces and code to handle the pieces are going to be placed in our board movie. To handle this, a couple additional layers are going to be needed. The layers are “code,” “pieces,” and our original layer which we will name “back.” On the pieces label we drag a copy of our piece movie which we will name “piece_movie.”

The board initialization will be done in a function called `init_board`. When you think about it, the board really can be broken into three sections for each of the five colors. Every color has a side of the board starting at their starting gate, they also have an ending block of squares where the pieces go when they have finished. Finally, there is the starting - or home - block. All of these things can have simple coordinates attached to them and then, if we know the angle, we can determine the location of the other locations within that group. For the game, we are going to also need the board id's of these locations as well as the starting and ending ids for the loading zone.

All of this information results in seventy different pieces of information. To handle all of this information without having too many variables to worry about, I have created an array of five arrays with the five arrays containing the fourteen items we are interested in. Each of the items is assigned an index value within the array, with a constant being created so in the future I don't need to remember the particular index value.

Once this array is created, two layout arrays are created to hold the x and y positions of all of the locations on the board. Some simple trigonometry is used to generate all the positions within that array. A lot of work, but all this initialization will make developing the rest of the game a bit easier.

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

```
init_board();
test_spot = 0;

function init_board()
{
    if (board_initialized != undefined)
        return;
    board_initialized = true;
    var cntr, cntrcol, nx, ny;

    var degree = Math.PI / 180;
    BOARD_START_INDEX = 0;
    BOARD_START_X_INDEX = 1;
    BOARD_START_Y_INDEX = 2;
    BOARD_START_ANGLE_INDEX = 3;
    BOARD_END_INDEX = 4;
    BOARD_END_X_INDEX = 5;
    BOARD_END_Y_INDEX = 6;
    BOARD_END_ANGLE_INDEX = 7;
    BOARD_HOME_INDEX = 8;
    BOARD_HOME_X_INDEX = 9;
    BOARD_HOME_Y_INDEX = 10;
    BOARD_HOME_ANGLE_INDEX = 11;
    BOARD_LOADING_START = 12;
    BOARD_LOADING_END = 13;

    BOARD_LAYOUT = new Array(
        //si,  sx,  sy, sa, ei,  ex,  ey, ea, hi,  hx,
        hy, ha, ls, le
        new Array( 0, -27,-224,180,  5,   1,-167,270, 50,
3,-207,324,101,104),
        new Array(10, 243, -72, 90, 15, 181, -36,198, 61, 217,
-46,252, 57, 60),
        new Array(20, 163, 224,  0, 25, 113, 174,126, 72, 133,
210,180, 68, 71),
        new Array(30,-163, 224,180, 35,-111, 173, 54, 83,-133,
204,108, 79, 82),
        new Array(40,-242, -30,270, 45,-173, -36,342, 94,-215, -52,
36, 90, 93) );

    layout_x = new Array(105);
    layout_y = new Array(105);
    layout_piece = new Array(105);

    for (cntrcol = 0; cntrcol < 5; ++cntrcol)
    {
        for (cntr = 0; cntr < 5; ++cntr)
        {
            layout_x[BOARD_LAYOUT[cntrcol][BOARD_START_INDEX]+cntr] =
                24 * cntr *
Math.cos(BOARD_LAYOUT[cntrcol][BOARD_START_ANGLE_INDEX] * degree) +
                BOARD_LAYOUT[cntrcol][BOARD_START_X_INDEX];
            layout_y[BOARD_LAYOUT[cntrcol][BOARD_START_INDEX]+cntr] =
                24 * cntr *
-Math.sin(BOARD_LAYOUT[cntrcol][BOARD_START_ANGLE_INDEX] * degree) +
                BOARD_LAYOUT[cntrcol][BOARD_START_Y_INDEX];
```

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

```
        layout_x[BOARD_LAYOUT[cntrcol][BOARD_END_INDEX]+cntr] =  
            24 * cntr *  
Math.cos(BOARD_LAYOUT[cntrcol][BOARD_END_ANGLE_INDEX] * degree) +  
            BOARD_LAYOUT[cntrcol][BOARD_END_X_INDEX];  
        layout_y[BOARD_LAYOUT[cntrcol][BOARD_END_INDEX]+cntr] =  
            24 * cntr *  
-Math.sin(BOARD_LAYOUT[cntrcol][BOARD_END_ANGLE_INDEX] * degree) +  
            BOARD_LAYOUT[cntrcol][BOARD_END_Y_INDEX];  
    }  
    for (cntrcol = 0; cntrcol < 5; ++cntrcol)  
    {  
        for (cntr = 0; cntr < 11; ++cntr)  
        {  
            layout_x[BOARD_LAYOUT[cntrcol][BOARD_HOME_INDEX]+cntr] =  
                24 * cntr *  
Math.cos(BOARD_LAYOUT[cntrcol][BOARD_HOME_ANGLE_INDEX] * degree) +  
                BOARD_LAYOUT[cntrcol][BOARD_HOME_X_INDEX];  
            layout_y[BOARD_LAYOUT[cntrcol][BOARD_HOME_INDEX]+cntr] =  
                24 * cntr *  
-Math.sin(BOARD_LAYOUT[cntrcol][BOARD_HOME_ANGLE_INDEX] * degree) +  
                BOARD_LAYOUT[cntrcol][BOARD_HOME_Y_INDEX];  
        }  
    }  
  
    pieces = new Array(5);  
    var temppiece, tempname, templayer;  
    for (cntrcol = 0; cntrcol < 5; ++cntrcol)  
    {  
        pieces[cntrcol] = new Array(5);  
        for (cntr = 0; cntr < 5; ++cntr)  
        {  
            templayer = cntrcol*5+cntr+1;  
            tempname = "p"+templayer;  
            piece_movie.duplicateMovieClip(tempname, templayer);  
            temppiece = eval(tempname);  
            pieces[cntrcol][cntr] = temppiece;  
        }  
    }  
    piece_movie._visible = false;  
    SPEED = 4;  
}
```

For testing, we create a loop where a piece goes through every location on the board. This is done in two frames. In the frame we label “Test” we place the following code:

```
test_movie._x = layout_x[test_spot];  
test_movie._y = layout_y[test_spot];
```

and ten to fifteen frames after the label (so we have time to see the position) we place this code:

```
++test_spot;  
if (test_spot >= 105)  
test_spot = 0;  
gotoAndPlay("Test");
```

Preparing the Pieces

At the end of initialization, the pieces used by the player exist in a two dimensional array named `pieces`. None of the pieces have been assigned their color, number or properly placed on the board. The reason for this is that the movie clips won't initialize until the first frame has been played, so you have to wait a frame before you can call the functions. In Flash MX 2004, you can get around this problem by creating a class for the movie clip and having the movie clip use this class.

As we have finished with testing, we will re-label the "Test" frame "startGame." We will remove the looping code and replace the code on the "startGame" frame with the following.

```
layout_pieces();  
stop();
```

We need to create the function for laying out the board, which we will place in frame 1. As the pieces already exist, all that needs to be done is setting the color, number, and position. If you remember from last section, the starting locations for all the pieces are stored in the `BOARD_LAYOUT` array that we created, so all we have to do is grab the value from that table using the appropriate indexes and adjusting the value based on the piece number.

```
function layout_pieces()  
{  
    var temp;  
  
    for (cntrcol = 0; cntrcol < 5; ++cntrcol)  
    {  
        for (cntr = 0; cntr < 5; ++cntr)  
        {  
            temp = BOARD_LAYOUT[cntrcol][BOARD_START_INDEX] + cntr;  
            pieces[cntrcol][cntr]._x = layout_x[temp];  
            pieces[cntrcol][cntr]._y = layout_y[temp];  
            pieces[cntrcol][cntr].setPiece(cntrcol, cntr+1);  
            pieces[cntrcol][cntr].setLocation(temp);  
        }  
    }  
  
    for (cntr = 0; cntr < 105; ++cntr)  
        layout_piece[cntr] = null;  
}
```

Turn Handling

At this point we are ready to get the game under way. The first thing we need is a way for the player to roll the die. This can be combined with our solution for determining which players' turn it is by simply having a roll button for each player. The first task then is to create a roll button. As we did with the pieces, instead of creating five buttons we only create one grayscale version and tint it.

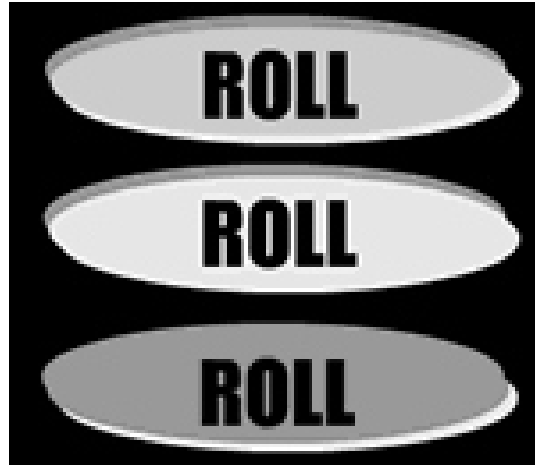


Figure 5: Roll Button states

To make use of this button, it actually has on the screen. Only the player who currently is playing can use this button, only the roll button for the current player needs to be on the screen. This allows us to set up the main movie into discrete frames for the flow of the game. Each color has a frame labelled “Color_row” (with Color being the player color) followed by a frame reserved for action script, then a frame labelled “Color_play” (with Color being the player color) finally followed by a frame reserved for action script.

On the “Color_row” frames we place the button close to the current player and tint it that players' color. The button is given the label “roll_btn.” On the next frame, which we reserved for Action Script, we add the following code:

```
roll_btn.onRelease = doRoll;  
stop();
```


Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

We currently don't have a doRoll function, so we are going to have to write one. This function would simply have to start the die rolling then go to the appropriate "Color_play" frame. The question is, how to determine the correct frame? To know this we need to know who the current player is. This can be solved by simply having a currentPlayer variable and a startGame function that resets this variable.

```
function startGame()
{
    currentPlayer = 0;
}

function doRoll()
{
    die_movie.startRoll(_root);

    switch (currentPlayer)
    {
        case 0:
            gotoAndPlay("Purple_play");
            break;
        case 1:
            gotoAndPlay("Red_play");
            break;
        case 2:
            gotoAndPlay("Yellow_play");
            break;
        case 3:
            gotoAndPlay("Green_play");
            break;
        case 4:
            gotoAndPlay("Blue_play");
            break;
    }
}

function roll_result(n)
{
    trace ("Result of roll was " + n);
    //test
    ++currentPlayer;
    if (currentPlayer > 4)
        currentPlayer = 0;
}
```

At the moment all this does is loop through the five players letting each player roll. Because we haven't done anything to stop the movie, it will start the next button before the roll is finished, so when testing please be sure to wait for the die to stop rolling.

Selecting the Piece

The `roll_result` function that we wrote last chapter was for testing purposes. The first thing we want to do to make the pieces clickable is to make sure that we are calling the board's soon to be written routines for handling moving pieces. To do this we need to re-write the roll result function.

```
function roll_result(n)
{
    trace ("Result of roll was " + n);
    board_movie.prepareMove(currentPlayer, n, _root);
}
```

As you can see, the code now calls the movie function's `prepareMove` function. Before we get to that function, however, we are going to first create one of the board movie's support functions. The `clearActions` function quite simply resets the state for all pieces on the board. We want to do this to make sure that all highlighted and selected pieces from other players are reset. This function simply loops through all the pieces to accomplish this task.

```
function clearActions()
{
    for (cntrcol = 0; cntrcol < 5; ++cntrcol)
    {
        for (cntr = 0; cntr < 5; ++cntr)
        {
            pieces[cntrcol][cntr].onRelease = null;
            pieces[cntrcol][cntr].changeState(0);
        }
    }
}
```

The `prepareMove` function is a bit more complicated task. The first thing this function does is determine where the relevant board locations for that player are. This is a simple lookup in our `BOARD_LAYOUT` array.

```
function prepareMove(player, roll, listener)
{
    trace ("PrepareMove("+player+", "+roll+", this)");
    var moveCount, cntr, loc, zonedist;
    lastRoll = roll;
    clearActions();
    move_listener = listener;
    var home = BOARD_LAYOUT[player][BOARD_HOME_INDEX];
    var start = BOARD_LAYOUT[player][BOARD_START_INDEX];
    var end = BOARD_LAYOUT[player][BOARD_END_INDEX];
```

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

Next we loop through the five pieces that belong to the player. We then determine how far from home that particular piece is. The board is circular (sort of) which means that we have the extra complexity of dealing with the possibility that the target home location is actually less than the actual location. As this will result in a negative distance from home, we simply need to add the size of the board (55 locations) to this value to get the correct distance.

Knowing how far the piece has to travel to get to the home location is the way we determine if the piece is located in the loading zone. It is also going to prove to be a very useful bit of information when we get around to writing the AI next chapter.

```
movecount = 0;
for (cntr = 0; cntr < 5; ++cntr)
{
    loc = pieces[player][cntr].getLocation();
    trace ("piece " + cntr + "is at " + loc);
    zonedist = home - loc;
    if (zonedist <= 0)
        zonedist += 55;
    trace ("piece " + cntr + " is " + zonedist + " from reaching
home");
}
```

As you have probably already realized, not all pieces are going to be on the board. In fact, some of the pieces will be in the starting gates. It is important that those pieces be able to move, otherwise we won't have a game at all since there would never be any pieces on the board. To start a piece on the board, the roll must match that piece. Likewise, to remove the piece the roll must match the piece. This is simply enough to check. If the piece matches the roll, we then see if the piece is in the loading zone or in the starting gate. If the piece is in one of these area's then the piece is movable.

```
if ((cntr+1) == roll)
{
    if ( (loc == (start+cntr)) || (zonedist < 5) )
    {
        trace("Piece is launchable/exitable");
        pieces[player][cntr].changeState(1);
        ++movecount;
    }
}
```

We then see if the piece is on the board and can move without going past the loading zone. We know the board starts at location 50, so that check is simply a location check. The distance from home can be used to see if the roll will take you past home, as if the distance is less than the roll of the die, then the move will take you past the starting position!

```
if ( (loc >= 50) && (zonedist > roll) )
{
    trace("piece movable on board");
    pieces[player][cntr].changeState(1);
    ++movecount;
}
```

Finally, we see if there is actually a move available for the player. As all the moves are counted above, we simply see if there are moves. If not, we are done the move!

```
    if (movecount == 0)
        move_listener.doneMove();
}
```

The choosePiece function will start the animation so it will be covered next chapter. If you wanted to test what you were doing, however, you could have a placeholder function in its place, such as the following function.

```
function choosePiece()
{
    trace ("Piece was selected to be moved");
    move_listener.doneMove();
}
```

We are finished with the board movie for the moment, so the last thing we need to do is in the main movie. This task is simply a function to handle the game once the move is done! We will also place stop functions on the action script frame after every “Color_play” label.

```
function doneMove()
{
    trace ("Player finished moving");
    ++currentPlayer;
    if (currentPlayer > 4)
        currentPlayer = 0;
    switch (currentPlayer)
    {
        case 0:
            gotoAndPlay("Purple_roll");
            break;
        case 1:
            gotoAndPlay("Red_roll");
            break;
        case 2:
            gotoAndPlay("Yellow_roll");
            break;
        case 3:
            gotoAndPlay("Green_roll");
            break;
        case 4:
            gotoAndPlay("Blue_roll");
            break;
    }
}
```

Moving the piece

The choosePiece function within the board movie is called when a piece is clicked. The first thing the function has to do is store information about the piece that called it. It then figures out where the piece is going to be moving to. Finally, it sets up the onEnterFrame variable to point to the updateMove function. This results in the updateMove function being called every frame.

```
function choosePiece(piece)
{
    trace ("Piece was selected to be moved");
    var loc = piece.getLocation();
    trace ("Piece was on " + loc);
    clearActions();
    piece.changeState(2);

    var player = piece.getPlayer();
    var nmber = piece.getPieceNumber();
    if (loc == (BOARD_LAYOUT[player][BOARD_START_INDEX]+nmber-1) )
        movingTargetLoc = BOARD_LAYOUT[player][BOARD_HOME_INDEX];
    else if ( (loc >= BOARD_LAYOUT[player][BOARD_LOADING_START]) &&
        (loc <= BOARD_LAYOUT[player][BOARD_LOADING_END]) &&
        (lastRoll == nmber) )
        movingTargetLoc = BOARD_LAYOUT[player][BOARD_END_INDEX]+nmber-1;
    else
    {
        movingTargetLoc = loc + lastRoll;
        if (movingTargetLoc > 104)
            movingTargetLoc -= 55;
    }
    movingPiece = piece;
    movingTargetX = layout_x[movingTargetLoc];
    movingTargetY = layout_y[movingTargetLoc];
    layout_piece[loc] = null;
    trace("Moving from " + loc + " to " + movingTargetLoc);
    onEnterFrame = updateMove;
}
```

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

The `updateMove` function should seem familiar, as it is based on the movement code that we wrote back in chapter 8. The first thing the function does is figures out if the current frame of motion will be the last frame of motion. This is simply a check of the distance from the current position to the ending position. If the distance remaining to move is less than the speed then we obviously are on the last frame of animation. Or at least the last frame of that pieces' animation.

```
function updateMove()
{
    var curX = movingPiece._x;
    var curY = movingPiece._y;
    var deltaX = movingTargetX - curX;
    var deltaY = movingTargetY - curY;
    var distance = Math.abs(deltaX) + Math.abs(deltaY);

    if (distance <= SPEED)
    {
        movingPiece._x = movingTargetX;
        movingPiece._y = movingTargetY;
        movingPiece.changeState(0);
        movingPiece.setLocation(movingTargetLoc);
    }
}
```

This is the point in the function where we veer away from the code we had earlier. What we are doing at this point is simply checking to see if there is a piece already occupying our target location. If there is, we prepare to animate that piece returning to it's home location. If no piece is there, we tell the layout array that the animated piece is now located at that location and then end the animation by clearing out the `onEnterFrame` variable and then calling the listener (root movie) to let it know that the animation is done.

```
        if (layout_piece[movingTargetLoc] != null)
        {
            var temp = layout_piece[movingTargetLoc];
            layout_piece[movingTargetLoc] = movingPiece;
            movingTargetLoc =
BOARD_LAYOUT[temp.getPlayer()][BOARD_START_INDEX]+temp.getPieceNumber()-1;
            movingTargetX = layout_x[movingTargetLoc];
            movingTargetY = layout_y[movingTargetLoc];
            movingPiece = temp;
        }
        else
        {
            layout_piece[movingTargetLoc] = movingPiece;
            onEnterFrame = null;
            move_listener.doneMove();
        }
    }
}
```

Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

The final bit of code returns us to our movement method. This code is executed when the movement is not finished, so it simply continues to move the piece towards the destination.

```
else
{
    var moveX = deltaX * SPEED / distance;
    var moveY = deltaY * SPEED / distance;
    movingPiece._x += moveX;
    movingPiece._y += moveY;
}
}
```

Winning the Game

To see if the current player has won, we simply need to check to see if all the pieces are in their end positions. A simple function in the board movie can handle this.

```
function checkWin(player)
{
    var cntr, result = true;
    var end = BOARD_LAYOUT[player][BOARD_END_INDEX];

    for (cntr = 0; cntr < 5; ++cntr)
    {
        if (pieces[player][cntr].getLocation() != (end + cntr))
        {
            result = false;
            break;
        }
    }

    return result;
}
```

Now we simply need to create five win pages with a generic (so the color can be tinted) continue button that sends the movie to the title screen. The code to do this is fairly simple

```
continue_btn.onRelease = function() { gotoAndPlay("Title", 1); };
stop();
```


Blazing Games Guide to Flash Game Development Chapter 16: Pent Up Anger

With the winning pages created, we need a way to reach these pages. To do this we modify the doneMove function.

```
function doneMove()
{
    trace ("Player finished moving");
    if (board_movie.checkWin(currentPlayer) == true)
    {
        switch (currentPlayer)
        {
            case 0:
                gotoAndPlay("Purple_wins");
                break;
            case 1:
                gotoAndPlay("Red_wins");
                break;
            case 2:
                gotoAndPlay("Yellow_wins");
                break;
            case 3:
                gotoAndPlay("Green_wins");
                break;
            case 4:
                gotoAndPlay("Blue_wins");
                break;
        }
    }
    else
    {
        ++currentPlayer;
        if (currentPlayer > 4)
            currentPlayer = 0;
        switch (currentPlayer)
        {
            case 0:
                gotoAndPlay("Purple_roll");
                break;
            case 1:
                gotoAndPlay("Red_roll");
                break;
            case 2:
                gotoAndPlay("Yellow_roll");
                break;
            case 3:
                gotoAndPlay("Green_roll");
                break;
            case 4:
                gotoAndPlay("Blue_roll");
                break;
        }
    }
}
```