

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 17

Pent Up AI

Contents

Now we take a look at creating a computer opponent, or more commonly known as an AI. We apply this knowledge to Pent Up Anger to add the ability to play against a computer opponent.

- AI in Board Games - a brief look at the role of AI in board games.
- Pent Up AI - Designing the games AI.
- Tri-state Buttons - a three way selectable button.
- Title Menu - A nice title screen that lets players select ai opponents.
- Adding Skip - Adding the Ability to
- Making a Computer Opponent - Getting the computer to make moves.
- Fine Tuning - Adding finishing touches to the game.

AI in Board Games

AI stands for Artificial Intelligence. It is commonly used in computer games to describe a computer controlled opponent. Scientists also use the term AI to describe research into Artificial Intelligence. AI research has resulted in a variety of techniques, such as fuzzy logic, neural networks, genetic algorithms, expert systems, and many others. The techniques used in the scientific research of AI, however, are not always appropriate for use in computer games. What many beginning programmers seem to forget is that the goal of a computer game is to have a computer opponent capable of playing the game intelligently where the goal of AI research is to have the computer learn and understand.

I am not saying that you shouldn't look into the various AI research that has been done. What I am saying is to keep your goal (creating an enjoyable opponent for the player to play against) in mind. Another thing to keep in mind is that most people tend to play games to win. In other words, you want an AI that plays a challenging game, not one that will beat the champion.

There are two basic techniques that are used when it comes to AI for board games. They are Recursive Algorithms, and Decision Trees. Some more complicated board games (such as war games) may require other techniques, but for most classic board games these two techniques should work.

Recursive Algorithms are mostly associated with chess games. Even today's chess games use some variation of the recursive algorithm. This technique simply has the computer take a look at every possible move, with each move being examined by taking a look at the possible moves the opponent could make, with each of these moves then being examined for all the possible moves the computer could make which are then examined for ... well, you get the idea. Each level of examination is a recursion (with recursion being a mathematical term for when functions fall back on themselves) The theoretical ideal of this technique is that you would keep looking at possible moves until there are no more moves. Realistically, you can only look at so many moves, so a limit on the level of recursion is usually placed on the computer. By reducing the number of recursions the computer executes, you reduce the intelligence of the computer, so you can use recursion levels as a way of controlling difficulty.

A Decision Tree is a light weight expert system. The computer makes its decision about what to do by following a set of branching rules. As branching rules simply correlate to if.then blocks, such an algorithm is fairly easy to write once the tree has been worked out. This type of technique is really good for track based games and for games that have random elements to them.

Planning Pent Up AI

For Pent Up Anger we are going to use a decision tree to handle the computer moves. This is largely because of the random nature that the game has. As we don't know what the player's will roll, all decisions on the move will have to be based on the current state of the board. One problem with decision tree based AI's is that they tend to be predictable. In some games, having a predictable opponent gives the player a fairly large advantage. In our case, knowing the way the computer thinks doesn't help player much.

The best way to design a decision tree is to become familiar with the game you are designing the decision tree for. After playing through the game I came up with a basic strategy for the computer to use. All of this revolves around prioritizing the pieces that are able to move.

First, we assign a value to every piece and move the piece with the highest value. The weight value reflects the priority, so the higher the step is on the list of questions, the higher the weight.

1. Is the piece in the loading zone and the die roll that pieces number? This is our highest priority for two reasons. First, we want to win the game, and this is done by removing all our pieces. Second of all, getting to the loading zone is a time consuming process, so if an opponent takes us out at this point, we are losing a lot of time.
2. Is the piece in the starting location and the resulting move won't take out one of my own pieces? If it is, then it is blocking new pieces from coming out of the starting gate.
3. Is the piece in the starting gate, the roll is that pieces number and the starting location is empty? We want to have as many pieces as possible on the board, as having to wait for the number to be rolled is not very efficient. Therefore, we get the pieces out on the board as soon as possible.
4. Can the piece be moved without taking out one of my pieces? Obviously, knocking yourself back to the start isn't too bright.

The above is enough for our AI, and results in a decent computer opponent. There are other things we could add to the AI if we desired. For instance, we could make moves that take out an opponent piece a higher priority than moves that don't take out a piece. We could also add some type of distributed movement system so that the computer doesn't focus on just one piece.

Tri-state buttons

At this point in time we have a complete and playable game. I would like it possible to play against a computer player and also to leave some of the players out of the game. This means that we are going to need some type of menu system that lets the player choose who controls a particular color.

As every color is going to have one of three choices, and only one of the three choices is valid at a time, it makes sense to create a component that lets the player choose one of three states. As there is no such component built into flash, this tri-state button then needs to be created.

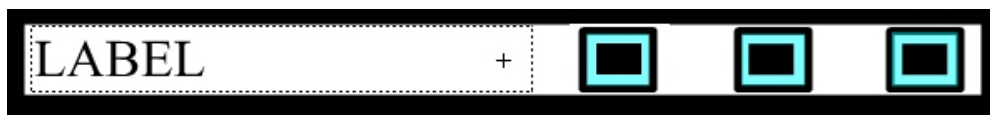


Figure 1: Tri-State button

First, we create a movie object the size of the desired object. Then we assign a dynamic text object to the left side of the tri-state button. This is named `label_txt` and is going to be used for setting the label. Next we draw the three boxes. We create a fill object to represent the check state of the object. This is converted into a movie named `select_box`. Three instances of this movie are used, one for each box. The idea here is that, as we are going to have to control the state anyway, one of the three box movies will be visible at a time.

Now we need a way of actually making this component do it's work. We create an invisible button the size of a box. The over frame will be a visible solid color as will the highlight. Instances of these buttons go over each of the three boxes. Now we are ready to write the code for this component. First, some of the user interface code, which will be placed on frame 1.

We start of with the initialization code. We always want one of the three boxes to be selected, so initialization simply makes the first box the selected box.

```
initTriState();

function initTriState()
{
    if (tristateInitialized != undefined)
        return;
    tristateInitialized = true;
    current_select = 1;
}
```

Next, we have the `refreshComponent` function. This function is what determines which of the three buttons to show as selected and which to hide. This is done by assigning the box movie's visibility property to the result of a conditional statement that checks to see if the currently selected box is the box in question.

```
function refreshComponent()
{
    box1_movie._visible = (current_select == 1);
    box2_movie._visible = (current_select == 2);
    box3_movie._visible = (current_select == 3);
}
```

Next, we need to be able to select a button (otherwise, how are we going to change the state). Note that this function automatically calls the `refreshComponent` function to make sure that the changed state is drawn. This is paired with a `getSelected` function which simply returns the currently selected state (1 for the first button, 2 for the second, and 3 for the third).

```
function setSelected(n)
{
    current_select = n;
    refreshComponent();
}

function getSelected()
{
    return current_select;
}

function setName(s)
{
    label_txt.text = s;
}
```

To finish off the first frame, we have button handlers. Unfortunately, Flash does not have that great of button handling, so instead of having one button handler that could handle all three buttons, we need 3 separate button handlers which just so happen to call the same function with a distinguishing id value.

```
function box1select()
{
    setSelected(1);
}

function box2select()
{
    setSelected(2);
}

function box3select()
{
    setSelected(3);
}
```

```
}
```

With frame 1 finished, we are now ready to add the code for frame 2. This code quite simply activates the three buttons by calling the appropriate function forwarding function. Then, to make sure that it is displayed properly, it calls the refreshComponent function. Finally, it stops the playhead.

```
box1_btn.onRelease = box1select;  
box2_btn.onRelease = box2select;  
box3_btn.onRelease = box3select;  
refreshComponent();  
stop();
```

This is one area where Java programmers have it much nicer. In Java you are able to assign a distinguishing ID to buttons so that all buttons can be routed to the same button handling function. More importantly, you are able to actually specify objects that are interested in being notified if the button is clicked. You can even have multiple listeners! While it is theoretically possible to do all of this with Flash, you would have to do a lot of the work yourself.

Title Menu

Now we can assemble the title screen. First, a simple logo. This is simply a nice cursive font for the “Pent up” text and a harsh, bold font for the “Anger” portion. Certainly, if you have the time you could design a fancy title. The key part of the title screen, however, is speed in loading.

Next we put together the menu of options and the “Start Game” button. The menu is made up of five of the tri-state buttons that we created in the last section. Each of the tri-state buttons is tinted the appropriate color. They are each named using the following template: label#_movie. The number of the player replaces the crosshatch symbol. The start button is given the label “start_btn.” To make the three states of the tri-state button understandable, the labels “No player,” “Human,” and “Computer” are used. Figure 2 shows the resulting title screen.



Figure 2: Title Screen

While the title screen looks nice, it doesn't do anything at the moment. To get the title screen to actually do something, we are going to need to add some code. In the first frame we have the initialization code that sets the default player to human, and the startGame function which is called by the start button when it is selected. The startGame function simply sets the global player_type variables to the values in the tri-state fuctions. To prevent an infinite loop from occurring, it also makes sure that there is at least one human or computer player.

```
init_title();

function init_title()
{
    if (title_initiated != undefined)
        return;
    title_initiated = true;
    player_type = new Array(5);
    for (var cntr = 0; cntr < 5; ++cntr)
        player_type[cntr] = 2;
}

function startGame()
{
    player_type[0] = label1_movie.getSelected();
    player_type[1] = label2_movie.getSelected();
    player_type[2] = label3_movie.getSelected();
    player_type[3] = label4_movie.getSelected();
    player_type[4] = label5_movie.getSelected();
    var valid = false;
    for (var cntr = 0; cntr < 5; ++cntr)
        if (player_type[cntr] > 1)
            valid = true;
    if (valid)
        gotoAndPlay("Game", 1);
}
```

The second frame sets all the tri-state labels to the appropriate color names and activates all the buttons.

```
label1_movie.setName("Purple");
label1_movie.setSelected(player_type[0]);
label2_movie.setName("Red");
label2_movie.setSelected(player_type[1]);
label3_movie.setName("Yellow");
label3_movie.setSelected(player_type[2]);
label4_movie.setName("Green");
label4_movie.setSelected(player_type[3]);
label5_movie.setName("Blue");
label5_movie.setSelected(player_type[4]);
start_btn.onRelease = startGame;
stop();
```


Adding skipping

One of the options in the tri-state menu's is "No player". This is to allow a small number of players play against each other without having any computer opponents. After playing through the game I noticed that it would be nice to be able to skip a turn. Especially in cases where your only move is to move your piece onto a player's launching area when that launching area is nearly full! Having no opponent is the same as skipping a turn so it would only be a marginal amount of effort to add a skip feature.

First, we create a button. This was done extra easily by cloning the roll button and changing the text layer to read Skip. Next, we simply copy the keyframe for the color_play frames and swap out the roll button with the skip button. We change the label of the button to skip_button. We add the following code on the frame after each of the five play frames.

```
skip_btn.onRelease = skip;
stop();
```

The actual skip function, which we place on frame 1, is as follows.

```
function skip()
{
    board_movie.clearActions();
    doneMove();
}
```

Next we change the code on the frames after the roll frames to:

```
stop(); // must be here otherwise can stall as it would be interpreted after
the gotoAndPlay
if (player_type[0] == 1)
    skip();
else if (player_type[0] == 2)
    roll_btn.onRelease = doRoll;
else
    doRoll();
```

changing the number in the player_type array to one less then the number of the player (remember that arrays start counting at zero instead of one).

Making a Computer Opponent

As computers can't think, we need to give the computer a set of rules for playing. There are many ways of handling the rules. The most common approach is to look at all possible moves and assign the moves a weight. This is the approach we will be using, and the piece class already has support functions for setting and getting a weight (see chapter 16 if you want to review the functions).

To determine the weight, we are going to make changes to the prepareMove function that is located in the board symbol. Quite simply, we are assigning a weight to each move. The weight is assigned using the rules we outlined earlier in the chapter.

```
function prepareMove(player, roll, listener)
{
    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];

    movecount = 0;
    for (cntr = 0; cntr < 5; ++cntr)
    {
        pieces[player][cntr].setWeight(0);
        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");
        // see if piece can be launched or finished
        if ((cntr+1) == roll)
        {
            if ( (loc == (start+cntr)) || (zonedist < 5) )
            {
                trace("Piece is launchable/exitable");
                pieces[player][cntr].changeState(1);
                if (zonedist < 5)
                    pieces[player][cntr].setWeight(100);
                else if (layout_piece[home].getPlayer() != player)
                    pieces[player][cntr].setWeight(90);

                pieces[player][cntr].onRelease = choosePiece;
                ++movecount;
            }
        }
        if ( (loc >= 50) && (zonedist > roll) )
```

```
        {
            trace("piece movable on board");
            pieces[player][cntr].changeState(1);
            var targetloc = loc + roll;
            if (targetloc > 104)
                targetloc -= 55;
            if (layout_piece[targetloc].getPlayer() != player)
                if (loc == home)
                    pieces[player][cntr].setWeight(60);
                else
                    pieces[player][cntr].setWeight(60-zonedist);
        }
//        pieces[player][cntr].onRelease = choosePiece;
        ++movecount;
    }
}
if (_parent.player_type[player] == 3)
    ai_move(player);
else if (movecount == 0)
    move_listener.doneMove();
}
```

The actual routine that uses the weight is also placed in the board symbol. It simply goes through the pieces and moves the piece with the highest weight. Ties go to the earlier piece. This allows us to know if none of the pieces are able to move (as all the pieces will have a weight of 0 in this case). The biggest problem with this AI is the predictability of the computer. Still, I have played against many human opponents who are just as predictable.

```
function ai_move(player)
{
    var cntr, choice = -1;
    var best = 0;
    for (cntr = 0; cntr < 5; ++cntr)
    {
        if (pieces[player][cntr].getWeight() > best)
        {
            choice = cntr;
            best = pieces[player][cntr].getWeight();
        }
    }
    if (choice == -1)
        move_listener.doneMove();
    else
        choosePiece(pieces[player][choice]);
}
```

Fine Tuning the code

The first thing I am going to add to the game to give it more of a finished feel is a sound when the dice is rolled. To do this we start by importing a sound file called “dice.wav.” We then go to the “roll” button symbol and add a layer. On the down frame, we add a keyframe to this new layer. In the keyframe we select the dice sound to start playing.

More sound for when a piece moves should now be added. This is a bit trickier as the sound for handling this is going to have to be handled entirely with Action Script. As before the sounds need to be imported. This time, however, you are going to have to go to the sound’s linkage settings (on windows, you right click the sound and select “linkage” from the drop down menu). Check the “Export for Action Script” box so that the sound will be available to Action Script. The sounds we will be importing for movement are “XCLICK.WAV” and “explosion.mp3.”

As the board symbol is where the sound will be used, we will add the code to control the sound there. First, at the end of the `init_board` function add the following lines. These lines simply create instances of the Sound object and attach the desired sounds to those instances. To actually play the sounds, the start function needs to be called.

```
clickSound = new Sound(this);  
clickSound.attachSound("Click");  
explodeSound = new Sound(this);  
explodeSound.attachSound("Explosion");
```

In the `updateMove` function, we are going to have to add the code to play our two sounds. The explosion sound is added at the beginning of the code block after the “if (`layout_piece[movingTargetLoc] != null`)” line and simply consists of a call to the start function.

```
explodeSound.start();
```

The click sound is placed in the code block of the following else statement.

```
clickSound.start();
```

The final sound that we are going to add to the game is my over-used win sound which will be played when one of the players wins. This is done by adding another layer to the main movie time line which we will call sound. Over the start of each of the win keyframes, we place a keyframe in the sound layer. This keyframe simply plays the win sound.

The game is now fairly good. One thing still bothers me, however. The skip button is visible when the result of the dice hasn’t even been revealed as well as when the player has started to move.

This can be handled entirely with Action Script. First, we create a variable that will hold the current skip button (I say current, because the skip button wavers in and out of frames, so to be sure that the proper instance is being used, I place the current skip button into this variable). I initialize this variable in the main time line's startGame function by adding the following line to the end of the function.

```
currentSkip = null;
```

If you look carefully at the movie, you will notice that the skip button is created just after the dice starts rolling. Obviously it will be visible during the duration of the roll. The solution is to make the skip button initially invisible. We do that by replacing all the code after the five Color_play frames with the following code.

```
skip_btn.onRelease = skip;  
currentSkip = skip_btn;  
skip_btn._visible = false;  
stop();
```

In addition to activating the button, it also stores the button into our global currentSkip variable and makes the button invisible. Now we need to make the button visible when the roll is done. This is a very easy problem, as the roll procedure was designed with a calling mechanism which calls the roll_result function once the roll has been completed. All that we need to do to get the skip button to appear once the roll is finished is add the following line to the roll_result function.

```
currentSkip._visible = (player_type[currentPlayer] == 2);
```

We are still not quite finished. If you were going to run the movie at this moment you would notice that the skip button would appear right after the roll stopped, just like we wanted. The skip button, however, would not disappear once the player selected a piece to move. This is a bit trickier of a problem to handle. First we will create a function that hides the skip button. This function is placed on the first frame of the main time line.

```
function hideSkip()  
{  
    currentSkip._visible = false;  
}
```

If you remember when we were making the pieces selectable, all the pieces end up calling a function in the board movie called choosePiece. This happens to be the ideal spot to hide the skip button, so we simply make a call to the hideSkip function using the following line of code.

```
_parent.hideSkip();
```

Blazing Games Guide to Flash Game Development Chapter 17: Pent Up AI

Now the only remaining tasks are minor housekeeping tasks. These include adding the Blazing Games logo and setting up the publish settings to desired values. Testing is also an important task. The best way of testing is by letting the computer play and watch.

What are you looking for? Quite simply, you want to make sure it is possible for all the colors to win (the code looks fine, but it never hurts to be sure) and that there are no stupid surprises. If after a dozen or so games everything seems fine, then you can be fairly sure that the game is working properly. Of course, you can never be 100% sure.