

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 13

Cribbage Square

Contents

Cribbage Square is a game where you have a grid where you place cards. You draw cards and try to place them in the appropriate grid location to maximize the hands as scored horizontally, vertically, and diagonally.

- Layout - Laying out the game's main screen
- Card Slots - Setting up areas where cards can be placed
- Initializing the game - Getting the game ready to run
- Dealing the Game - The main game loop
- Scoring Hands - A function for scoring cribbage hands
- Scoring the Game - Making the game use the scoring function
- Replay - The end of game options
- Title - finishing things up

Layout

To start laying out the game we need four layers. I have labelled these layers “Code”, “Totals”, “Cards”, and “Back”. Later on we will add additional layers.

The background layer consists of a bitmap fill, using a wood texture that was created in fireworks.

The cards layer consists of the card movies that we are using in the game for the cards in their final positions. In order to fit all the cards on the screen, we need to scale down the cards. This is done by simply setting the scaling property to 75%. The only card that isn’t scaled is the deck card. The card movies are labelled using the format, “cardRC_movie” where R is the number of the row and C is the letter of the column. The deck card is named “cardDeck_movie” and the nib card is named “cardStart_movie”.

The total layer holds all the totals of the columns, rows and diagonals. All the totals are made up of two thing. We have a box (made by using the box tool then changing the top and right side lines to be yellow while making the bottom and right side lines dark red). We also have a dynamic text object. The ten dynamic text boxes are labelled, “totalX_txt” with X being the description of the “Row”, “Col” or “Diag” that makes up the total.

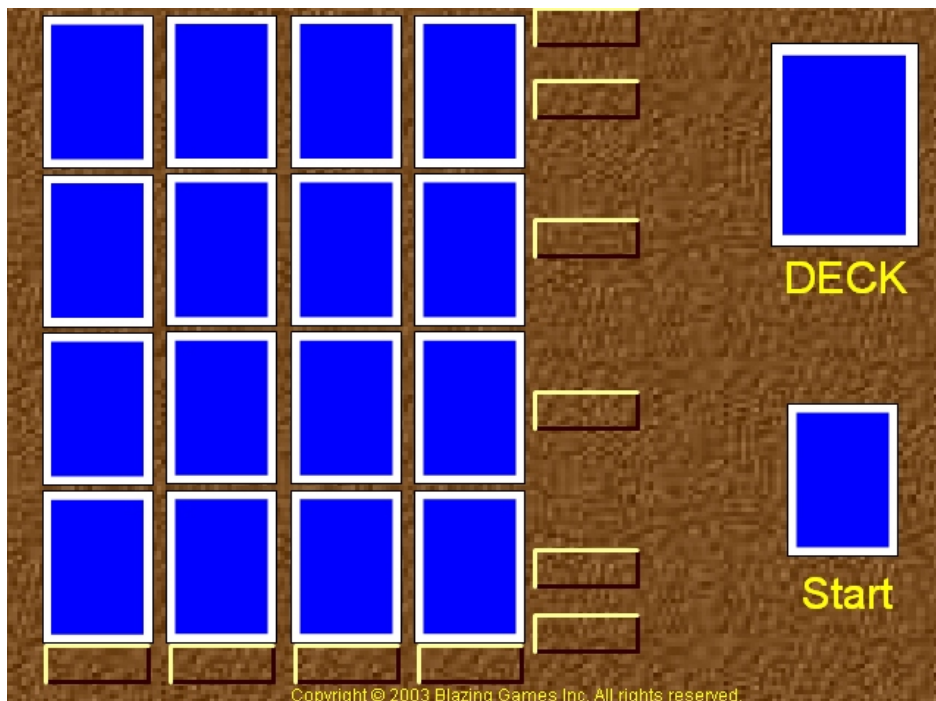


Figure 1: Layout

Card Slots

The purpose of a card slot is simply to show the player where cards can be placed and place the current card at the indicated location when the player clicks on the slot. A button is the ideal thing to use for a card slot. We design the card slot to be large enough to cover a card and draw on it some type of design so players know that clicking on the card slot will place the current card in that location. Figure 2 shows the card slot and it's three visible states.



Figure 2: Card Slot Buttons

With that done, we then need to actually place the card slots on the screen. After spending all the time we spent labelling all the cards on the grid, you may wonder why we would replace them with the slots. The truth is that we are not going to replace them, but instead, we are going to cover all the cards with the slot placement buttons. The slot buttons go on the button layer over top of the cards. The labelling scheme that I use is the same one as the one used for the cards. For those of you with a very short memory, the slot movies are labelled using the format, “slotRC_btn” where R is the number of the row and C is the letter of the column. The deck card is not covered up and the start card slot is named “slotStart_btn”.

Initializing the Game

Now we are ready to start coding the actual game. The first thing we need is a deck of cards. The Deck.as file that we created a couple of chapters ago would be ideal for our needs. Of course, we need to add this to our game. This is done by using the #include command. On the first frame of our game we only need to add the following line in order to use the deck class that we created.

```
#include "Deck.as"
```

With the deck class now usable by us, we can start initializing the game. I like placing all my code on frame 2. This way you know that all the objects have properly initialized. The first thing that we will do is a one-time initialization function. The Initial goals for initialization are threefold. First, we need a deck to play with. Second we need an easy way of accessing the cards and the slot buttons on the layout. Finally we need to activate all the slot buttons.

Creating a deck is simple enough. However, at some point we are going to want to make it possible to replay the same shuffle. While this is a feature we don't need at the moment, adding support for doing this wouldn't hurt. How to support this feature? Quite simply all we need to do is create two decks. The official deck will be the one that is used for shuffling the cards. It, however, will not be used for dealing the cards. For that we use a second deck, which I am calling myDeck, which will be a cloned copy of the official deck. By doing this we will always have a pristine deck to use if we want to replay the shuffle.

The second task can easily be handled by creating two arrays. The cardSet array holds the all of the cards the player can select. The slotSet holds all the slot buttons that cover the cards. The array is layed out in a left to right top to bottom order, with the nib being the last element of the array. You should remember that arrays start counting at 0, so card1a is at array index 0.

The third task is also very easy. I just assign the onRelease function of the buttons to a custom function that calls a placeCard function with the id of the card that is being placed. At the moment this function does nothing, but we will expand this function in a later section. This is an unfortunate necessity as the onRelease function doesn't have any parameters so we can't just write a generic button handler as there is no way of knowing what button was pressed.

This results in the following code on frame 2.

```
initializeCS();
newGame();

function initializeCS()
{
    myDeck = new Deck();
    officialDeck = new Deck();

    cardSet = new Array(card1a_movie, card1b_movie, card1c_movie,
card1d_movie, card2a_movie, card2b_movie, card2c_movie, card2d_movie,
card3a_movie, card3b_movie, card3c_movie, card3d_movie, card4a_movie,
card4b_movie, card4c_movie, card4d_movie, cardStart_movie);
    slotSet = new Array(slot1a_btn, slot1b_btn, slot1c_btn, slot1d_btn,
slot2a_btn, slot2b_btn, slot2c_btn, slot2d_btn, slot3a_btn, slot3b_btn,
slot3c_btn, slot3d_btn, slot4a_btn, slot4b_btn, slot4c_btn, slot4d_btn,
slotStart_btn);
    slot1a_btn.onRelease = function() { placeCard(0); };
    slot1b_btn.onRelease = function() { placeCard(1); };
    slot1c_btn.onRelease = function() { placeCard(2); };
    slot1d_btn.onRelease = function() { placeCard(3); };
    slot2a_btn.onRelease = function() { placeCard(4); };
    slot2b_btn.onRelease = function() { placeCard(5); };
    slot2c_btn.onRelease = function() { placeCard(6); };
    slot2d_btn.onRelease = function() { placeCard(7); };
    slot3a_btn.onRelease = function() { placeCard(8); };
    slot3b_btn.onRelease = function() { placeCard(9); };
    slot3c_btn.onRelease = function() { placeCard(10); };
    slot3d_btn.onRelease = function() { placeCard(11); };
    slot4a_btn.onRelease = function() { placeCard(12); };
    slot4b_btn.onRelease = function() { placeCard(13); };
    slot4c_btn.onRelease = function() { placeCard(14); };
    slot4d_btn.onRelease = function() { placeCard(15); };
    slotStart_btn.onRelease = function() { placeCard(16); };
}

function newGame()
{
    officialDeck.shuffle();
    myDeck.cloneDeck(officialDeck);
}

function placeCard(n)
{
    trace ("placeCard(" + n + ") not implemented yet!");
}
```

Dealing the Game

We are not quite finished with code for frame 2 yet. First, let's think about what is going to be happening. The cards are shuffled, which has been done. Now the game page should consist of all the slots covering all the cards. If this is the second game that the player is playing this may not be the case, so we are going to need to add code to set all the slot covers to visible. This code could be added to the newGame() function, but not all games that are started are going to be new games. Setting up a frame for starting the game which calls a start game function would be a good idea. Both types of game starting functions could then automatically go to the appropriate frame to get the game underway. We could put the code to handle the startup right in the start frame. Instead, I am going to write a function (placed in frame 2) that handles this activity. This way all the code is in one location which helps make the code easier to understand.

Frame 4 will be labelled "start". Frame 5 will contain code that calls the startGame function (which, as I said above, will be placed in frame 2) and will then stop the movie. Because all the buttons are active, and buttons react when clicked, the game will be able to function just fine and play out even though the movie is stopped. The code on frame 5 looks like this:

```
startGame();  
stop();
```

Frame 10 will be labelled "gameOver." At the moment, I just have a few seconds worth of delay and just let the movie restart. Later on we will add a menu that will allow the player to replay the shuffle, start a new game, or quit the game. This menu will also track the highest score for the current shuffle.

The startGame() function's first task is to reset the card count. The cardCount variable is used to count how many cards have been played. The next thing the function does is the task of resetting the slots and making sure that the deck card is not showing some value. It will also adjust the score by calling an updateScore() function that at the moment doesn't do anything, though will be doing all the scoring work when we get to that aspect of the game. Finally, it calls the nextCard() function which deals the player a card.

```
function startGame()  
{  
    cardCount = 0;  
    for (var cntr = 0; cntr < 17; ++cntr)  
    {  
        slotSet[cntr]._visible = true;  
    }  
    updateScore();  
    nextCard();  
}
```


The nextCard function is going to first increase the card count. Then it makes sure that there are still slots that need to be filled. New cards are grabbed from the deck and assigned to the cardDeck movie. If there are no slots left it will start playing the movie at the gameOver frame.

```
function nextCard()
{
    ++cardCount;
    if (cardCount > 17)
    {
        cardDeck_movie.hideFace();
        gotoAndPlay("gameOver");
    }
    else
    {
        cardDeck_movie.setCard(myDeck.draw());
        cardDeck_movie.showFace();
    }
}
```

We are ready to replace the placeCard function with code that actually does something. As the index value of the card is passed as a parameter, we simply need to use the cardSet and slotSet arrays using the passed to access the proper objects. This makes our function fairly straight forward. The first thing the function does is assigns the card underneath the selected button (remember that they share the same index) with the value that the deck card has showing. Next we make sure that the selected card is showing its face. We then make the button covering the card invisible, which has the added benefit of de-activating the button so that it can't be selected again. We call the nextCard function so the next card is dealt. Finally, we update the scores by calling the updateScore function.

```
function placeCard(n)
{
    cardSet[n].setCard(cardDeck_movie.getCard());
    cardSet[n].showFace();
    slotSet[n]._visible = false;
    nextCard();
    updateScore();
}
```

Scoring Hands

We have a functional game. At least as long as you are able to manually calculate the cribbage scores yourself. While a lot of people (myself included) have little problem doing this, what is the point of playing a cribbage square's computer game if at the end of the game you need to take out a piece of paper and add up your score? Especially when you consider that computers are better at doing calculations than humans. The trick then is to figure out how to teach the computer how to score the hands. We will do this in two sections. This section we will create the low-level scoring function that takes five raw card values and turns them into a score. Next section, we will create some glue functions that tie the scoring to the game.

The first thing we are going to need is some convenient functions for getting the face value and the suit. Now, if you look back a couple of chapters ago when we built the card deck you will recall that the card id is a value from 1 to 52 representing the deck. Cards are sorted from Ace through King. This is repeated four times. First, for Clubs, next for Diamonds, third for Hearts, and finally for Spades. Why this order? The suits are sorted alphabetically, of course.

With the knowledge of what the card ID values mean refreshed in our memory, we can see that getting the face value is simply a mathematical calculation. First, we take the id of the card and subtract 1 from it so that it is in the range of 0 and 51. We then take the modulus of this number and 13 (the remainder of the number divided by 13, as there are 13 cards in each suit). The reason we want the counting to start at 0 is that if we left the number alone, all the cards would return their correct value except for kings, would return a value of 0. By deducting one, all the values, including the king, are one lower than what they would be. After the modulus is done we add one to get the correct face value and then return that value.

```
function getCribFace(n)
{
    var val =Math.floor((n-1)%13)+1;
    return val;
}
```

Now, the suit function also is a simple mathematical calculation. As before, we deduct 1 from the card ID. This time, we are doing this to make sure that the floor value of the number when divided by thirteen will be all within the same group. That group will correspond to the suit that the card belongs to.

```
/* returns  0 for club
 *          1 for diamond
 *          2 for heart
 *          3 for spade
 */
function getSuit(n)
{
    return Math.floor((n - 1) / 13);
}
```

```
}
```

Now that we have the helper functions out of the way, we are going to write a monster of a function that calculates the score of the hand. While it is possible to break up this function into a bunch of smaller functions, the procedure is very linear in nature so I don't see any point in doing that other than making a printout look nicer. Still, because the function is so large, it is a wise idea to add comments throughout the function so it is clear what is happening at what point in the process. Instead of describing the whole function, I will have my explanation of what is going on scattered through the function.

As you can see, the function takes five values. The first four cards are the cards in the player's hand (a horizontal, vertical, or diagonal line on the square) while the fifth value holds the start card.

```
function ScoreHand(c1, c2, c3, c4, cs)
{
    var cntr, cntrB, low, temp;
    var score = 0;
    var cardFaces = new Array(5);
```

The first step in calculating the score is to build an array of the face value, as almost all of cribbage scoring is based on the face values of the cards. This array will simply hold five numbers, which will be valued between 1 and 13. At the same time we are building this array, we can see if there is a flush (4 or 5 cards). For the array, we simply put the return values from the `getSuit()` call for each of the five cards passed to this function. The flush starts out by setting a `haveFlush` flag to true. We then check the suit of all the cards (except for the start card) to see that they match. In other words, we assume that there is a flush until it is obvious that there isn't. If there is a four card flush we then check to see if the start card extends the flush to five cards. As this is the first score calculation, we can simply set the score to 4 or 5 at this point.

```
// check for flush while building face array
var nobSuit = getSuit(cs);
var flushSuit = getSuit(c1);
var haveFlush = true;
cardFaces[0] = getCribFace(c1);
if (flushSuit != getSuit(c2)) haveFlush = false;
cardFaces[1] = getCribFace(c2);
if (flushSuit != getSuit(c3)) haveFlush = false;
cardFaces[2] = getCribFace(c3);
if (flushSuit != getSuit(c4)) haveFlush = false;
cardFaces[3] = getCribFace(c4);
cardFaces[4] = getCribFace(cs);
if (haveFlush)
{
    if (flushSuit == nobSuit)
        score = 5;
    else
```

```
        score = 4;  
    }
```

If you look at the code above, you will notice that we the suit of the start card was stored in a variable called nobSuit. If you recall from our discussion of the cribbage scoring system in chapter 10, then you will know that if a hand contains a jack of the same suit as the start card, the hand gets a point. This point is known as the nob. How do we find this out? Quite simply, we take the nobSuit value and multiply it by 13 then add 11 to it. This gives us the actual card ID that we are looking for. We then set a variable called haveNob to false (this time we assume the hand does not have the desired card) and then check the four cards in the hand to see if they are the specific card that would result in the nob, setting the flag to true if the card is the nob card. Once all four cards have been checked, we simply test to see if the flag has been set and add a point to the score if it has.

```
// nob (Jack matching suit of nob card  
var nobCard = 13 * nobSuit + 11;  
var haveNob = false;  
if (c1 == nobCard) haveNob = true;  
if (c2 == nobCard) haveNob = true;  
if (c3 == nobCard) haveNob = true;  
if (c4 == nobCard) haveNob = true;  
if (haveNob)  
{  
    ++score;  
}
```

The next task we have to do is a fairly complex task. We want to find the number of runs and the number of pairs. It is fairly efficient to do both of these things at the same time. In fact, counting pairs is partially how we deal with multiple runs. This code is a fairly long process, so we are going to describe the task in three stages: Sorting, checking, and scoring.

The sorting phase is simply an insertion sort. This means that we we start off by going through all the cards in the hand and trying to find the smallest card. We place that card in the first slot and then go through the remaining cards for the smallest card, which goes in the second slot. This continues until all five cards have been sorted.

```
// pairs and runs - sorting  
for (cntr = 0; cntr < 4; ++cntr)  
{  
    low = cntr;  
    for (cntrB = cntr+1; cntrB < 5; ++cntrB)  
    {  
        if (cardFaces[low] > cardFaces[cntrB])  
            low = cntrB;  
    }  
    temp = cardFaces[low];  
    cardFaces[low] = cardFaces[cntr];  
    cardFaces[cntr] = temp;  
}
```

```
}
```

With the cards sorted, we are ready for the check. We want to find the longest run as with only 5 cards in the hand, the longest run will in fact be the only legitimate run in the hand. This is because a run requires 3 consecutive cards, so if there is a run of 3, there would only be two cards left. In addition to the longest run, we have to worry about multiple runs. In order for there to be a multiple run, there has to be at least one pair, but if there are two pairs, the multi-run value would be four not three, so we have to check this special case. Now, as it is possible to have a run as well as two out of sequence cards or two cards forming a non-scoring run of two in addition to a run of three, we need a separate variable to hold the actual longest run encountered in the check.

Pairs are a bit easier. We only need to know how many pairs there are. However, as tripples count as 3 pairs and quadruples count as 6 pairs, knowing the pair length is important. As it turns out, by adding previous pair length to the number of pairs solves that problem. Why? Because a card by itself has a pair length of 1 (a proper pair has a length of 2). When a second card is encountered, you would then add 1 to the number of pairs, which is the obvious thing to do. Now, if the next card is also matches the current pairing card, you would have a tripple. By adding the previous pair length (2) to the number of pairs, you end up with 3, which is the correct value. If you have a quadruple, then the next round you would add 3 to the number of pairs which results in 6.

The loop through the hand starts at array element 1 (not 0) as we compare the current card against the previous card, and element 0 has no previous card. Checking to see if a card is part of a pair sequence is simply checking that the current card has the same value as the previous card. Pairs do not break runs. They increase the number of pairs by the old pair length and then increase the pair length. Pairs also increase the multi-run potential, with a check to see if two pairs have occurred. If it is not a pair, we see if the cards are in sequence. For cards to be in sequence, the previous card has to be one less then the current card. If that is the case, we reset the pair length while increasing the run length. If the card is not a pair or part of a run, we see if the current run is larger then the largest run and if so record the new longest run and multi-run values and in all cases we reset the runLength and multiRun variables.

Once the loop is done, we need to see if the result of the last loop iteration resulted in a new longest run and adjust the run variables accordingly.

```
// pairs and runs - checking
var runLength = 1;
var longestRun = 0;
var numPairs = 0;
var pairLen = 1;
var multiRun = 1;
var lrMultiRun = 1;
trace ("Card 0 face is " + cardFaces[0]);
for (cntr = 1; cntr < 5; ++cntr)
{
    trace ("Card " + cntr + " face is " + cardFaces[cntr]);
    if (cardFaces[cntr-1] == cardFaces[cntr])
    {
        numPairs += pairLen;
        ++pairLen;
        ++multiRun;
        // special case, two pairs in run
        if (pairLen < multiRun)
            ++multiRun;
    }
    else if ((cardFaces[cntr-1]+1) == cardFaces[cntr])
    {
        ++runLength;
        pairLen = 1;
    }
    else
    {
        if (runLength > longestRun)
        {
            longestRun = runLength;
            lrMultiRun = multiRun;
        }
        runLength = 1;
        multiRun = 1;
        pairLen = 1;
    }
}
if (runLength > longestRun)
{
    longestRun = runLength;
    lrMultiRun = multiRun;
}
```

Now that we have the pair and run information, we are ready to adjust the score. As you know, a run only scores if it is longer than three and it's score is the length of the run. If there are pairs in the run, then each pair forms it's own run, so the final score from runs is the number of runs times the length of the run. Pairs, of course, count as two times the number of pairs.

```
if (longestRun > 2)
{
    temp = longestRun * lrMultiRun;
    score += temp;
}
if (numPairs > 0)
{
    temp = numPairs * 2;
    score += temp;
}
```

Finally, we finish of scoring by seeing how many combinations add up to 15. This is done by using bits to represent cards. There are 5 cards, meaning five bits. If you can't remember how binary works, look back at chapter 7. Five bits give you 32 combination. Loop through the combinations and add up the face values (making sure that face cards only count as 10). Any combination that adds up to 15 earns two points.

```
// count combinations that add up to 15
for (cntr = 0; cntr < 32; ++cntr)
{
    temp = 0;
    if ((cntr & 1) == 1) temp += Math.min(cardFaces[0], 10);
    if ((cntr & 2) == 2) temp += Math.min(cardFaces[1], 10);
    if ((cntr & 4) == 4) temp += Math.min(cardFaces[2], 10);
    if ((cntr & 8) == 8) temp += Math.min(cardFaces[3], 10);
    if ((cntr & 16) == 16) temp += Math.min(cardFaces[4], 10);
    if (temp == 15)
    {
        score += 2;
    }
}
return score;
}
```

And that's it for scoring. Quite a bit of work, but as you can see, when broken down into smaller chunks, it is not that complicated. At the moment, the game doesn't use this function so if you want to test it, you will simply have to call the function with appropriate values, as with the following test line.

```
trace("Test hand value is: " + ScoreHand(11,18,31,44,5));
```

Scoring the Game

The scoring routine that we wrote is designed for five cards, so in order to score any line in the square all five cards that make up that line must be available. To make sure the user isn't confused about the lack of a score, we should display some type of message. I have chosen to use the letters NA, which stands for Not Available, as the value to assign any unfinished hand. A second task that has to be done is grabbing the actual card ID values that the score system needs to find the score.

To accomplish the above tasks I have written a function, called checkScore, that takes four index values and returns the score text to show. Only four index numbers are needed as the start card index is always going to be used.

The first step that this function needs to do is make sure that there are five cards available. This is done by simply logically ORing the visibility flag of the five card slot buttons together. If one of the slots is visible then obviously one or more of the cards is not selected. The only time a hand is valid is if all five slots are false, as an OR only requires one true value to become true. In other words, the hand is a valid hand if the result of the OR checks returns false. We can use this technique because the index values of the cards match the index values of the card slots.

If the hand is not valid for scoring we return the "NA" message, otherwise we make a call to the scoreHand function with the id's of the five cards that make up the hand.

```
function checkScore(ci1, ci2, ci3, ci4)
{
    var notvalid = slotSet[16]._visible; // always need the nib
    notvalid |= slotSet[ci1]._visible;
    notvalid |= slotSet[ci2]._visible;
    notvalid |= slotSet[ci3]._visible;
    notvalid |= slotSet[ci4]._visible;

    if (notvalid)
        return "NA";
    else
        return scoreHand(cardSet[ci1].getCard(), cardSet[ci2].getCard(),
                        cardSet[ci3].getCard(),
cardSet[ci4].getCard(),
                        cardSet[16].getCard());
}
```

Finally, we can put together the final version of the updateScore function. This function just calls the checkScore function for all the rows, columns, and diagonals assigning the result to the appropriate dynamic text block.

```
function updateScore()
{
    totalRow1_txt.text = checkScore(0, 1, 2, 3);
    totalRow2_txt.text = checkScore(4, 5, 6, 7);
    totalRow3_txt.text = checkScore(8, 9, 10, 11);
    totalRow4_txt.text = checkScore(12, 13, 14, 15);
    totalCol1_txt.text = checkScore(0, 4, 8, 12);
    totalCol2_txt.text = checkScore(1, 5, 9, 13);
    totalCol3_txt.text = checkScore(2, 6, 10, 14);
    totalCol4_txt.text = checkScore(3, 7, 11, 15);
    totalDiag1_txt.text = checkScore(3, 6, 9, 12);
    totalDiag2_txt.text = checkScore(0, 5, 10, 15);
}
```

Replay

When people finish playing a round of cribbage squares, there are three things that most people will do. Grab up the dealt cards and play through that particular shuffle again, shuffle the deck and deal a new hand, or quit playing. Right now the game just stops, so the only choice is to quit playing.

Creating a popup menu that gives the player the above choices is not that difficult of a task. The problem is that many people want to spend a bit of time looking at their results. For that reason, we most certainly do not want a popup menu appearing in the center of the screen as that would obscure the cards that the player is looking at. Doing a menu that appears over top the now useless deck, on the other hand, would not detract from the player. Another thing that we are going to want is the final score and best score so far using that particular shuffle.



Figure 3: End game menu

The menu itself is a movie which uses a purple gradient as its background. The total and best scores are dynamic variables named “total_txt” and “best_txt”. We will write a function called setScores() that will assign values to the variables. In addition the three buttons need to do something. As the action should be controlled by the parents, we will simply have these buttons call a function in the parent movie.

```
replay_btn.onRelease = function()
{
    _parent.replayGame();
}

new_btn.onRelease = function()
{
    _parent.newGame();
}

quit_btn.onRelease = function()
{
    _parent.quitGame();
}

function setScores(score, best)
{
    total_txt.text = score;
    best_txt.text = best;
}
```

The newGame function already exists, so all that is needed to make the make this movie usable to the main program is to write a replayGame and quitGame function. The replayGame function simply clones the official deck without shuffling the deck first. The quitGame function simply returns to the title screen.

```
function replayGame()
{
    myDeck.cloneDeck(officialDeck);
    gotoAndPlay("start");
}

function quitGame()
{
    gotoAndPlay("Title", 1);
}
```

Now that the menu will work, we need the animation that reveals this menu. As we already have a gameOver label, we only need to worry about adding a layer for the menu animation. The animation itself is a simple motion tween from the outside corner to the final resting location over the deck. The ending menu, however, doesn't have any scoring information. This will be done by adding a keyframe in the code layer right after the animation starts to appear. This keyframe will contain a single line.

```
endMenu_movie.setScores(finalTotals(), shuffleBestScore);
```

You will notice that this line calls a function called finalTotals. This function will be placed in frame 2. It simply calls the checkScore method for all of the rows, columns, and diagonals. This time, however, instead of placing the numbers into dynamic text variables, it adds the scores together to form a final score. As we are going to want to keep the best score, we are going to have to also see if the new final score is better than the current best.

```
function finalTotals()
{
    var total = 0;
    total = checkScore(0, 1, 2, 3);
    total += checkScore(4, 5, 6, 7);
    total += checkScore(8, 9, 10, 11);
    total += checkScore(12, 13, 14, 15);
    total += checkScore(0, 4, 8, 12);
    total += checkScore(1, 5, 9, 13);
    total += checkScore(2, 6, 10, 14);
    total += checkScore(3, 7, 11, 15);
    total += checkScore(3, 6, 9, 12);
    total += checkScore(0, 5, 10, 15);

    if (total > shuffleBestScore)
    {
        shuffleBestScore = total;
    }
    return total;
}
```

In order for there to be a current best, we need to reset the best score every time a new shuffle occurs. This is done by adding the following line to the newGame function.

```
shuffleBestScore = 0;
```

Title

The final thing that we have to do is the title sequence. For this sequence, I simply wrote up the title text. I then did a motion tween which consisted of the text scaled down and located off screen to the text in the final position in the final size. Adding a few rotations to the motion finishes of this title sequence.

The buttons are made up of the texture that was used for the background of the game with text over top. Each of the buttons are animated into the scene on their own layer. The final screen is below.



Figure 4: Title Screen