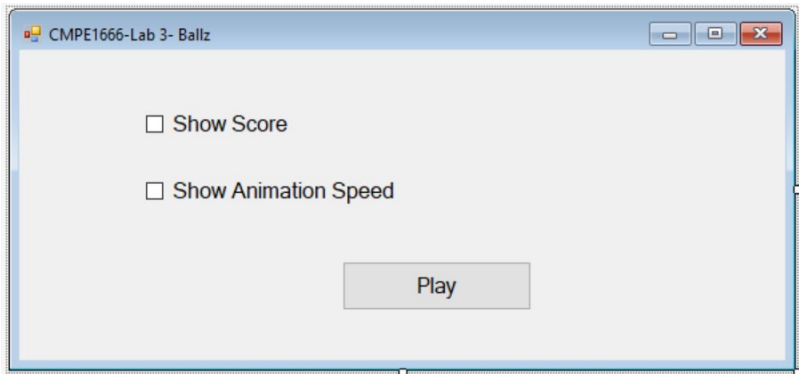
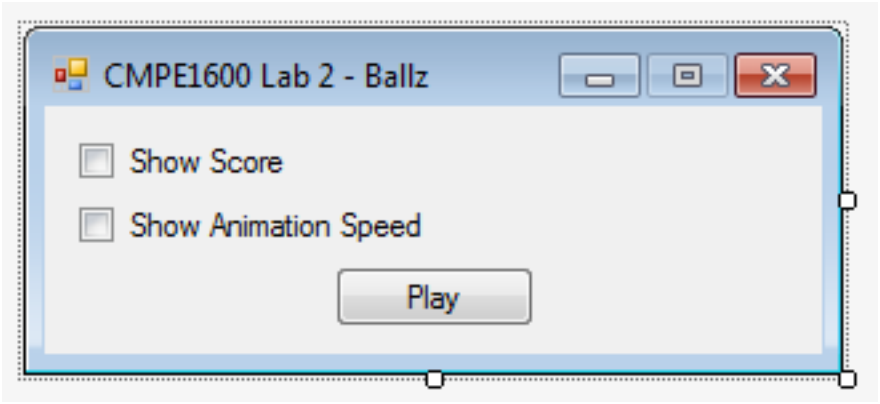
CMPE1666 – Lab 3 BallZ

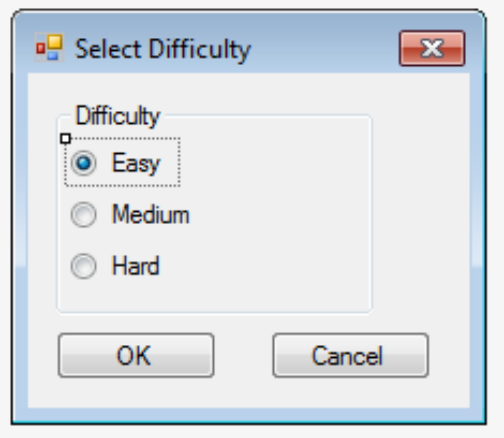
We shall utilize 2D arrays, structures and recursion to create a graphical game based roughly on the some the common “Dropping Object” games. The game will be played within the GDIDrawer window by using the mouse to select balls. Groups of balls selected that are the same color connected horizontally or vertically will “disappear” when the mouse is clicked. The more balls that are killed with a mouse click, the higher the score will be for that round.



Initially, the game will start with the main form shown below:



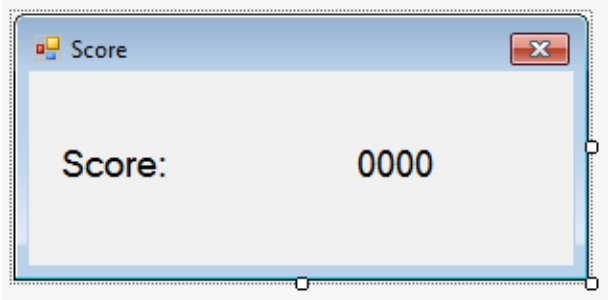
When the **Play** button is pressed, the Select Difficulty modal dialog will be displayed, as shown below. If the user presses the **Cancel** button, then playing the game will be cancelled. If the user presses the **OK** button, then the game will be created using the selected mode.



Once the Select Difficulty dialog has been dismissed, the random colored balls will be

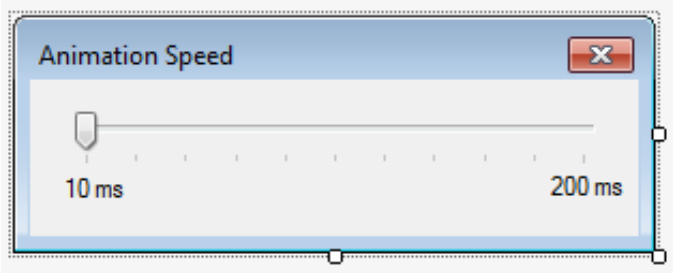
drawn to the screen with a ball size of 50 pixels. If the game mode is **Easy**, then 3 ball colors will be used. If the game mode is **Medium**, then 4 ball colors will be used. If the game mode is **Hard**, then 5 ball colors will be used. After the Select Difficulty dialog has been dismissed, a timer with an interval of 100 ms. will be started, and the **Play** button will be disabled until the game ends. The random balls will be drawn to the screen.

The **Score** modeless dialog will be displayed if the checkmark in the main form was checked. The Score dialog will display the user's current score while the game is being played. When the close button is pressed, the checkmark in the main form will be cleared.

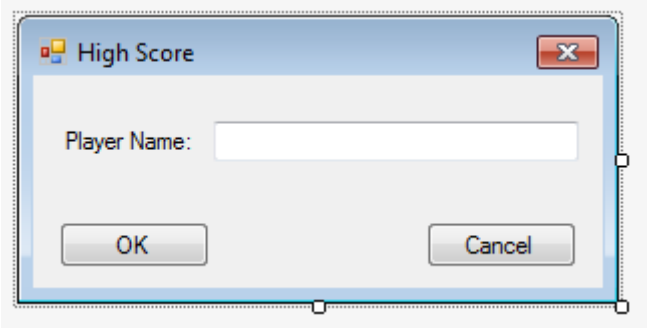


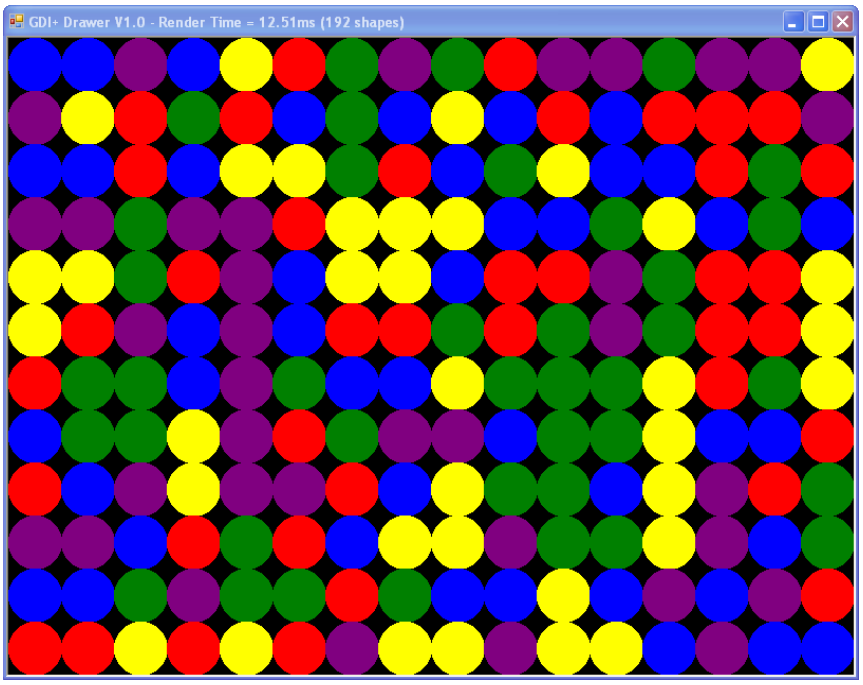
The timer tick event will be used to check the GDIDrawer for valid mouse clicks, update the current score, and check for the end of the game. The timer tick event will call method **Pick**, described below, to check for a mouse click. If **Pick** returns a non-zero score, then the balls will be displayed and the score updated on the form. If **BallsAlive**, described below, returns zero, then the game is over. Display "Game Over" in the GDIDrawer window, and enable the **Play** button.

The **Animation** modeless dialog will be displayed when the checkbox in the main form is checked. This dialog will determine the delay to be used when balls are dropping in the GDIDrawer window. The delay time will change in the game as the trackbar is manipulated. When the dialog is closed, the checkbox in the main form will be cleared.



When the game is over, if the player has achieved the highest score for the selected game mode, then the **High Score** modal dialog will be displayed to input the player's name. If the user presses the **OK** button, the player's name, game mode and score will be stored to a file. If the user presses the **Cancel** button, the high score information will not be stored.





Program Specification

1. This project will need some constant values, an enumerated type, and a structure:
    a. create the necessary constants to handle our array processing. You must have constants for screen width and height, ball size, row count, and column count.
        1. Row count is to be determined by Height and ball size
        2. Column count is to be determined by Width and ball size
    b. create an enumeration that will hold the state of our screen objects : alive or dead
    c. declare a structure for use as game elements within our 2D game grid :
        1. member of type Color – the color of our screen ball
        2. member of type state enumeration, as created above.
    d. The methods used for this project include :
        1. Randomize – mix up the grid state, preparing the beginning of the game
        2. Display – show the 2D array of balls in the graphics window

3. Pick – the main game processing function that picks the next mouse click
4. CheckBalls – Our recursive checking routine which clears currently selected balls
5. StepDown - "Drop" all the balls in our grid by one level
6. FallDown – Uses StepDown() to drop all our balls until they stop
7. BallsAlive – a worker method that counts balls that are alive

2. Create a class member which is an array of ball structures for the number of rows and columns to be used for the game. The ball size will be 50 pixels.

3. Create a class member which is a CDrawer object.

4. **Randomize**()
    a. returns nothing
    b. declare an array of 5 Color values, initializing to something pleasant
    c. iterate through the 2D array and set the ball state to alive, and the color to a randomly chosen color from your array. The number of colors to be used is determined by the game mode (easy, medium, hard).

5. **Display**()
    a. returns nothing
    b. clear the Drawer window.
    c. iterate through the array, if the structure state member indicates alive, use the color value to add an ellipse at the proper screen location (ie. X = column * ball_size)

6. **BallsAlive**()
    a. returns an integer representing the number of "alive" elements

7. **CheckBalls**()
    a. accepts the row, column to check
    b. accepts the Color to compare against
    c. This is a recursive routine which, given an initial row/column start it will set all adjacent balls that are the same color to dead. This spreads to all adjacent like colored ball elements.
        - There are many escape conditions to test for :
            i. is the row, column valid ? If either is not valid, return 0
            ii. is the element already dead ? May have already been checked, return 0
            iii. is the element a different color ? Return 0
            iv. if you make it past all these tests, set the state to dead then set the number of balls killed to 1. Invoke a recursive call to all immediately adjacent elements (ie. above, below, left, right) and then sum all the return values from the recursive calls. Return the sum of all the balls killed.

8. **StepDown**()
    a. iterate through the 2D array from top to bottom
    b. if the element is “dead”, then a ball could fall into that spot, so check the element above the current one, if it is “alive”, then copy the element to the current location, set the element above to “dead”, thereby “dropping” the ball
    c. display the game using the Display method with a time delay determined by the Animation dialog
    d. loop through all elements once.
    e. return the number of balls that dropped in this function

9. **FallDown**()
    a. invoke StepDown() repeatedly until the return value indicates no balls dropped
    b. keeping a running total count for multiple StepDown() calls, return this value when complete

10. **Pick**()
    a. returns an integer representing the score obtained for that pick. The score is a value that increases exponentially based on the number of balls killed in that pick.
    b. Processing is as follows :
        - **if** the GetLastClickPos() method returns false, return. If the method returned true, a new mouse position is reported.
        - Convert the X,Y pixel coordinates into the respective row/column equivalents. Then check the game array if the ball at that location is dead, then beep the speaker **briefly** - ie. if X = 341 pixels, then column = X / width_per_column = X/BallSize = 6
        - else continue on, invoking CheckBalls() to “kill” all the adjacent balls, CheckBalls() will return the number of balls killed which can be used to calculate the score for that pick. I used 50 points per ball killed plus an escalating bonus for killing groups of balls beyond one.
        - invoke FallDown() to “drop all balls”
        - return the score obtained for that pick.

Play Button

When the play button is clicked, the drawer window will be cleared, total score zeroed, Randomize will create the balls in the array, and Display will show the balls on the screen. A timer with interval of 100 ms. will be started, and the play button will be disabled.

Timer Tick

When the timer ticks, your program will call BallsAlive and check for a zero return value. If BallsAlive returns zero, turn off the timer tick, and display “Game Over” in the drawer window. The final score for the game will be displayed in the form, and the Play button enabled. If BallsAlive returns a non-zero value, call Pick and add the score returned to the total score. Display the updated total score on the modeless dialog.

Programming Assumptions, Hints and Gotchas

Tackle this lab incrementally, creating the enum, structure and game array first. Attempt the Randomize() and Display() methods next. Create the event handlers for the Play button and the timer tick. Then develop Pick() to allow ball selection. Next develop DropDown() and FallDown(), these can be tested in conjunction with Pick(). Finally complete the CheckBalls recursive routine, this fits nicely into the framework you have created already. It may seem like a lot but it will go smoothly if you write and debug along the way.

Rubric

This application will require visual inspection of functionality and code.
Mark loss is at your instructor's discretion but will be applied consistently across all students.

Item	Marks
UI Design (25)	

• All UI as Directed
  ○ Main Form
  ○ Difficulty-Level Dialog
  ○ Score Dialog
  ○ Animation-Speed Dialog
  ○ High-Speed Dialog
• Ok and Cancel button working as required
• Values being properly read/displayed on all Forms/Dialogs as required
• Control names are consistent and appropriate
	10
	
	5
	5
	5

Code Design and implementation(75)	
Documentation • Code is well documented, where applicable. • Code contains a programmer's block. • Variable names are appropriate.	15
Form-Load Event causes GDI window to appear	5
Program Design follow modular approach as specified	10
Generate button causes balls to be displayed as required	5
No. of Colors generated depends on difficulty-level chosen	5
Recursive algorithm for killing balls working as required	20
Input/Output information communicated properly between GDI Drawer and Forms/Dialogs	15