

## My Optional Project Implementation Steps

You are not in any way required to follow the steps below. I thought it might be useful, however, for you to know the sequence of steps I followed to complete the optional project. In most cases, I followed the steps in the order below, but in rare cases I actually decided I should have completed a particular step at a different point. That realization is reflected in the steps below.

As a final comment, you should note that I compiled and tested the game after every step listed below.

### ***Increment 1***

This increment displays the opening screen.

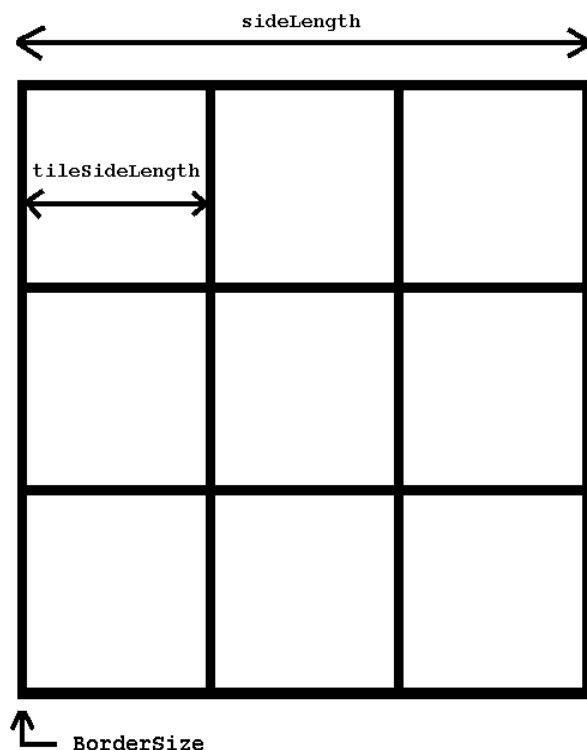
1. Run the project after you've added all the cs files and content to your game to make sure it runs
2. Add code to the `Game1` constructor to set the window width to 800 and height to 600 and to make the mouse visible (Chapter 5, Week 2)
3. Declare a `Texture2D` field in the `Game1` class to hold the opening screen image. As usual, we'll load content into this field then draw it as appropriate (Chapter 5, Week 2)
4. Declare a `Rectangle` field in the `Game1` class to hold the draw rectangle for the opening screen image. We'll use this field to draw the opening screen image as the appropriate size (Chapter 5, Week 2)
5. In the `Game1 LoadContent` method, load the opening screen texture into the field from Step 3 (Chapter 5, Week 2)
6. In the `Game1 LoadContent` method, create a new `Rectangle` object and put it in the field from Step 4. The rectangle should be created so the opening screen takes up the entire window (Chapter 5, Week 2)
7. In the `Game1 Draw` method, draw the opening screen texture in the opening screen draw rectangle (Chapter 5, Week 2)

### ***Increment 2***

When the player presses Enter, the opening screen disappears and the board (without any number tiles) is displayed.

8. You don't need to do anything for this step; it just provides information you need for the next step. I've declared a `GameState` field called `gameState` in the `Game1` class and I've set it to `GameState.Menu`. This field is used to keep track of the current game state

9. In the `Game1 Draw` method, add an if statement that checks if the current game state is `GameState.Menu` before drawing the opening screen texture in the opening screen draw rectangle (Chapter 7, Week 3)
10. Declare a `NumberBoard` field in the `Game1` class. This field will be used to hold the number board for the current game (Chapter 4, Week 2)
11. In the `Game1 LoadContent` method, create a new `NumberBoard` object and put it in the field from Step 10. Use the window width and height to calculate the board side length and the board center before calling the constructor. The board should be centered in the window and should be smaller than the window width and height. At this point, I just set the correct number to 8 (Chapter 4, Week 2)
12. You don't need to do anything for this step; it just provides information you need for the next step. In the `NumberBoard` class, I've declared `Texture2D` field called `boardTexture` to hold the board texture and I've declared a `Rectangle` field called `drawRectangle` to hold the draw rectangle for the board texture
13. Write the `NumberBoard LoadContent` method (Chapter 5, Week 2)
14. Write the `NumberBoard` constructor, which loads the content (by calling the `LoadContent` method you wrote in the previous step), creates a new draw rectangle object, and calculates the size of the number tiles on the board. Be sure to include borders around the tiles using the `BorderSize` constant; the picture below might help you generate the appropriate calculation. Don't create the number tiles yet (Chapter 5, Week 2)



15. Write the `NumberBoard` Draw method to draw the board texture in the board draw rectangle. Don't draw the number tiles yet. (Chapter 5, Week 2)

16. In the `Game1` Update method, add an if statement that changes the game state to `GameState.Play` if the current game state is `GameState.Menu` and the Enter key is pressed. The following code returns `true` if the Enter key is pressed and `false` otherwise:

```
Keyboard.GetState().IsKeyDown(Keys.Enter)
```

and the following code changes the game state to `GameState.Play`

```
gameState = GameState.Play; (Chapter 7, Week 3)
```

17. In the `Game1` Draw method, add an else clause to the if statement from Step 9 to have the board draw itself if the current game state is `GameState.Play` (Chapter 7, Week 3)

### **Increment 3**

This increment displays all the number tiles on the board.

21. You don't need to do anything for this step; it just provides information you need for the next step. In the `NumberTile` class, I've declared `Texture2D` field called `texture` to hold the tile texture and I've declared `Rectangle` fields called `drawRectangle` to hold the draw rectangle and `sourceRectangle` to hold the source rectangle. We need both draw and source rectangles because the tile textures we'll be using are sprite strips with multiple frames. The draw rectangle will tell us where to draw the tile and the source rectangle will tell us which texels from the `Texture2D` field we should draw (see Chapter 7, Week 3 for source rectangle idea from explosion animation example)

22. Write the `NumberTile` LoadContent method. I already provided code that uses the `ConvertIntToString` helper method to convert the integer to the string for the tile number (e.g., converting 8 to "eight"). We use this string in the LoadContent method so we can load the correct graphics content. You also need to create a new source rectangle object in this method; make it "cover" the left half of the texture you just loaded (Chapter 7, Week 3)

23. You don't need to do anything for this step; it just provides information you need for the next step. I've declared a `NUM_ROWS` by `NUM_COLUMNS` array field in the `NumberBoard` class to hold `NumberTile` objects and created the array object

24. In the `NumberBoard` constructor, add a nested for loop to initialize the `NumberTile` array. For now, I just create all the tiles with 8 for the tile number and correct number arguments and `null` for the sound bank argument. Note that we also have to calculate the center for each number tile to pass in to the `NumberTile` constructor. Use the `CalculateTileCenter` helper method I provided to help with this (Chapter 10, Week 7)

25. Write the `NumberTile` Draw method to draw the tile texture using the draw and source rectangles (Chapter 7, Week 3)

26. In the [NumberBoard](#) Draw method, add code to draw all the tiles. Using nested for loops is a reasonable way to do this (Chapter 10, Week 7)

27. At this point, when I run the game it draws the board with 9 "eight" tiles on it.

28. Change the calls to the [NumberTile](#) constructor to use the actual numbers for each of the tiles. The most intuitive way to do this is to use a counter variable that keeps track of the current tile number, but you can also do this by calculating the current tile number based on the loop control variables for the nested for loops. The board should now contain the correct number tiles when you run the game (Chapter 10, Week 7)

#### ***Increment 4***

This increment displays all the number tiles, which highlight when the mouse is over them. When incorrect tiles are clicked, they shrink until they disappear.

29. In the [Game1](#) Update method, if the current game state is [GameState.Play](#) get the current mouse state and call the [NumberBoard](#) Update method (Chapter 8, Week 5)

30. In the [NumberBoard](#) Update method, add code to call the Update method for each of the tiles. Using nested for loops is a reasonable way to do this (Chapter 10, Week 7)

31. In the [NumberTile](#) Update method, add an if statement at the beginning of the method to set the source rectangle X property appropriately based on whether or not the mouse is over the draw rectangle for the tile. At this point, each of the tiles highlights when the mouse is over the tile and unhighlights when the mouse isn't over the tile (Chapter 8, Week 5)

32. Add fields to the [NumberTile](#) class to tell if the tile is visible, if the tile is blinking, and if the tile is shrinking. Initialize those fields appropriately (Chapter 12, Week 9)

33. Add `clickStarted` and `buttonReleased` fields to the [NumberTile](#) class. In the [NumberTile](#) Update method, change the if statement from Step 31 to detect mouse clicks on the number tile. You can use the [MenuButton](#) Update method code at the very end of Chapter 8 to help you with this (I also provided the code for that method in the zip file for this project). Replace the code (from Chapter 8) that changes the game state when a click is finished on the menu button with code that does the following instead: If the player just clicked on the tile and the tile corresponds to the correct number, set the tile is blinking field to `true`, otherwise set the tile is shrinking field to `true`. You need to detect clicks rather than left mouse button presses so the player can't just "sweep" the mouse over the tiles to guess numbers (Chapter 12, Week 9 for fields. Chapter 8, Week 5 for using Chapter 8 code)

34. You don't have to do anything for this step; it's just explaining our plan for moving forward. We now need to modify our [NumberTile](#) Draw method and continue to refine our [NumberTile](#) Update method to handle this new information. Specifically:

a. Once the user has clicked the mouse button on the tile (it's blinking or shrinking), we don't want to use the if statement from the previous step to set any flags. That way, the user can only click the mouse button on a tile once

- b. Once a tile is blinking or shrinking, we need to update the blinking animation or the (shrinking) draw rectangle size on each update
- c. Once the tile is no longer visible (it shrunk away to nothing or finished blinking), we don't want to draw it at all any more

35. In the `NumberTile` Draw method, only draw the tile if it's visible (Chapter 7, Week 3)

36. Declare and initialize fields in the `NumberTile` class to keep track of the total milliseconds it takes the tile to shrink (this should be a constant) and the elapsed shrink milliseconds so far (this should be a variable) (Chapter 12, Week 9 for fields. Chapter 7, Week 3 for timers)

37. In the `NumberTile` Update method, add an if statement that checks if the tile is shrinking. If it is, update the elapsed shrink milliseconds field by adding `gameTime.ElapsedGameTime.Milliseconds` to it. Calculate the new side length for the tile using the original side length and the ratio between (total shrink milliseconds - elapsed shrink milliseconds) and total shrink milliseconds. **CAUTION: Force the ratio to get calculated as a float, otherwise your tile will just disappear instead of gradually shrinking.** Doing it this way makes the shrinking tile start at (almost) full size, then shrink down to 0. If the new side length is  $> 0$ , set the width and height of the draw rectangle to the new side length; otherwise, set the visible flag for the tile to `false` (Chapter 7, Week 3 for timers)

38. At this point, clicking on any tile except the 8 tile should make the tile gradually shrink until it's disappeared. Unfortunately, if you move the mouse off the shrinking tile it's no longer highlighted, which doesn't look right. In the `NumberTile` Update method, move the entire if statement you wrote in Step 33 into an else clause for the if statement you wrote in Step 37 so the source rectangle is only changed if the tile isn't shrinking. There are other ways to make sure the tile stays highlighted, but doing it this way is required for later steps (Chapter 7, Week 3)

## ***Increment 5***

This increment finishes the game. When the correct tile is clicked, it blinks for a while, then a new game is started. The game also includes sound effects.

Note that this increment has a lot more steps than the previous increments.

39. Next we want to make it so that when the correct tile is clicked, that tile starts blinking.

40. Declare a `Texture2D` field in the `NumberTile` class to hold the blinking tile texture (Chapter 12, Week 9 for fields)

41. In the `NumberTile` LoadContent method, add code to load the blinking tile texture into the field from the previous step. Note that you can use the string for the tile number to help build the name of the texture (all the blinking textures start with "blinking" followed by the number string) (Chapter 6, Week 3)

42. Now we have two textures we need to use for drawing: one for normal and shrinking tiles and the other for blinking tiles. That means we'll need to keep track of the current texture we're using so we draw the appropriate one. Declare a `Texture2D` field in the `NumberTile`

class to hold the current texture we'll be drawing (Chapter 12, Week 9)

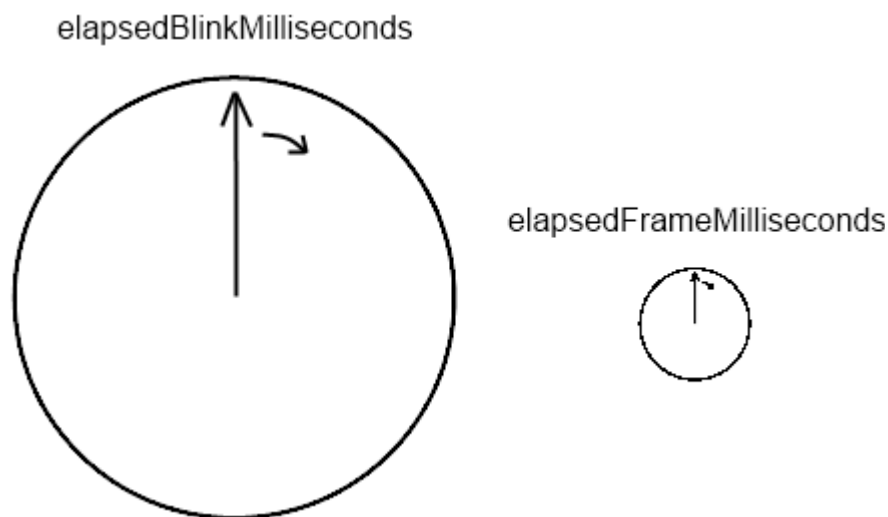
43. In the [NumberTile](#) LoadContent method, add code to set the current texture to the non-blinking texture (Chapter 3, Week 1)

44. In the [NumberTile](#) Draw method, change the code to draw the current texture (Chapter 5, Week 2)

45. In the [NumberTile](#) Update method, add two lines of code just after the line where you set the tile is blinking field to [true](#). Your first new line of code should set the current texture to the blinking texture. Your second new line of code should set the X property for your source rectangle to 0 (Chapter 5, Week 2)

46. In the [NumberTile](#) Update method, change the original if clause (that checked if the tile is shrinking) for the if statement from Step 37 to an else if instead. Now add an if clause before that else if clause to check if the tile is blinking. You'll now have an if statement with an if clause that checks if the tile is blinking, an else if clause that checks if the tile is shrinking, and an else clause that highlights/unhighlights the tile and lets the player click the mouse button on the tile. Basically, we're doing different update processing on the tile based on the current tile state (blinking, shrinking, or normal). (Chapter 7, Week 3)

47. You don't have to do anything in this step, it's just explaining how the next two steps will work. For our blinking animation, we need to have two timers. We'll use one timer, `elapsedBlinkMilliseconds`, to tell us when the blinking animation should stop. We'll use the other (shorter) timer, `elapsedFrameMilliseconds`, to tell us when it's time to change frames in the animation. See the picture below.



48. Inside the if clause you added in Step 46, add an if statement to determine when to make the tile invisible (after it's done blinking). Note that I've already included `elapsedBlinkMilliseconds` and `TotalBlinkMilliseconds` fields for you to use for this. You'll need to update the elapsed blink milliseconds field by adding `gameTime.ElapsedGameTime.Milliseconds` to it before deciding whether or not to make the tile invisible (Chapter 7, Week 3 for timers)

49. Add an else clause to the if statement you added in the previous step to update the animation frame as appropriate. I've already included `elapsedFrameMilliseconds` and `TotalFrameMilliseconds` fields for you to use for this. Add `gameTime.ElapsedGameTime.Milliseconds` to the elapsed frame milliseconds field, then use the new value to decide when to change to a new animation frame. To change frames in the blinking animation, you need to move the source rectangle left or right (depending on where the source rectangle currently is) after each frame ends to change the texels being displayed. Make sure you reset `elapsedFrameMilliseconds` back to 0 as well. (Chapter 7, Week 3)

50. You don't have to do anything for this step, it's just a check on your progress. At this point, clicking the 8 tile should make it blink between yellow and green for a few seconds, then the tile should disappear. Clicking on all the other tiles should make them shrink down until they're no longer visible

51. You don't have to do anything for this step either; it's just explaining what you'll be doing next. Now we're faced with a difficult problem. When the correct number is guessed, we're supposed to reset the board and start a new game. We allocated the responsibility of determining whether or not the correct number was guessed to each individual number tile (this was the correct design decision), but how does the number tile let the game know that the correct number was guessed? The `NumberTile` class doesn't know anything about the `Game1` or `NumberBoard` classes, so how can it communicate this information to them? There's a very slick way to do this in C# using something called *event handlers*, but that's not introductory-level material. Instead, I've made the `NumberBoard` Update method and the `NumberTile` Update method return a `bool` instead of the typical `void`. A `false` will mean the correct number hasn't been guessed yet and a `true` will mean it has. This isn't the best general solution (event handlers are), but it's the best solution using what we know at this point

52. In the `NumberBoard` Update method, we need to return `true` if one of the `NumberTile` Update calls returns `true` (indicating that the correct number was guessed), otherwise we need to return `false`. Change the call to the `NumberTile` Update method to put the result into a `bool` variable; the syntax is similar to when you called the `Deck` TakeTopCard method and put the returned value into a `Card` variable (in Lab 4). If this `bool` variable is `true` after the call to the method, immediately return `true` from the `NumberBoard` Update method. Doing it this way ensures the method immediately returns `true` if the correct number is guessed and also lets us skip the updates for the rest of the number tiles. There are more complicated ways to do this to make sure all the tiles get updated (to make sure we don't skip updating shrinking animations for 1/60 of a second, for example, for tiles that are currently shrinking), but this approach works fine here (Chapter 13, Week 9 for method return)

53. You don't have to do anything for this step, but you might be wondering if the `NumberBoard` Update method returns `false` correctly. The answer is yes, because the only way we get to the last line of code in the method is if none of the `NumberTile` Update method calls return `true`, which means the correct number wasn't guessed on this update. Slick, huh?

54. In the `Game1` Update method, change the call to the `NumberBoard` Update method to declare a variable to tell whether or not the correct number has been guessed and put the returned value from the method call into that variable (Chapter 4, Week 2)



55. Cut the code you added to the `Game1` `LoadContent` method to calculate the board size and actually create the board and paste that code into the `StartGame` method I provided at the end of the `Game1` class. Call the `StartGame` method from the `Game1` `LoadContent` method where you used to have that code. You'll see why we need to do this soon (Chapter 4, Week 2)
56. In the `Game1` `Update` method, add an if statement right after the call to the `NumberBoard` `Update` method to check if the correct number has been guessed. If it has, call the `StartGame` method (Chapter 7, Week 3)
57. At this point, we never actually end up restarting a game. That's because the `NumberTile` `Update` method always returns `false`. In the code in that method that sets visible for a blinking tile to `false` (see step 48), change the code that sets visible to `false` to return `true`; instead. We don't want to return `true` from this method right when the user picks the correct number because then they won't get to see any blinking, so we return `true` when the blinking is done (Chapter 13, Week 9 for return)
58. At this point, our game always has 8 as the correct number. Declare a `Random` field in the `Game1` class and use `new Random()` to instantiate the object for the field (Chapter 12, Week 9)
59. In the `StartGame` method, declare a variable to hold the correct number for the current game and set it to a random number (from 1 to 9 inclusive) you generate using the field from the previous step. Pass that number as the correct number argument when you call the `NumberBoard` constructor in the `StartGame` method (Chapter 4, Week 2)
60. In the `NumberBoard` constructor, change the call to the `NumberTile` constructor to pass the `correctNumber` parameter as the correct number argument (instead of 8 or whatever hard-coded number you used) (Chapter 4, Week 2)
61. Holy smokes! The game is – finally – almost done. It should all be working properly for you by this point, with the sound effects the only thing that are missing.
62. Declare a field in the `NumberTile` class to hold a `SoundEffect` object; you'll load the appropriate sound effect in the `NumberTile` constructor based on the `isCorrectNumber` flag you set at the end of the constructor. (Chapter 12, Week 9)
63. In the `NumberTile` constructor, use the constructor's `contentManager` parameter to load the appropriate sound effect (explosion for correct guess, loser for incorrect guess) into your sound effect field based on the value of `isCorrectNumber`. (Lecture 12.3, Week 11)
64. Run your code to make sure the sound effects are loaded without crashing the program.
65. In the `NumberTile` `Update` method, add code to play the sound effect when the player clicks on the tile (Lecture 12.3, Week 11)
66. In the fields for the `Game1` class, add a field to hold a `SoundEffect` object for the new game sound effect



67. In the [Game1](#) LoadContent method, use Content to load the applause asset into your new game sound effect field. (Lecture 12.3, Week 11)

68. In the [Game1](#) Update method, add code to play the new game sound effect before calling the StartGame method. We do this here rather than in the StartGame method because we don't want this sound effect to play for the first game (Lecture 12.3, Week 11)

69. That's it, you're done!