MonoGame Blocks Tutorial Part 3

Contents

Part 3 – Drawing Images on the Screen

Time required: 60 minutes

In the previous section, we loaded all of our image and sound assets into the content manager. Now that they are loaded, let's put them to use.

We'll start with the paddle. The paddle will be drawn near the bottom of the screen, and can move from the left to the right side of the screen. The player will try to hit the ball to keep it from falling off the bottom of the game screen.

The location where the ball contacts the paddle will determine the direction the ball will bounce. If it hits the left half of the paddle, it will be bounced toward the left. And if it hits the right half, it will bounce to the right. The farther the ball hits from the center of the paddle, the greater the angle of deflection.

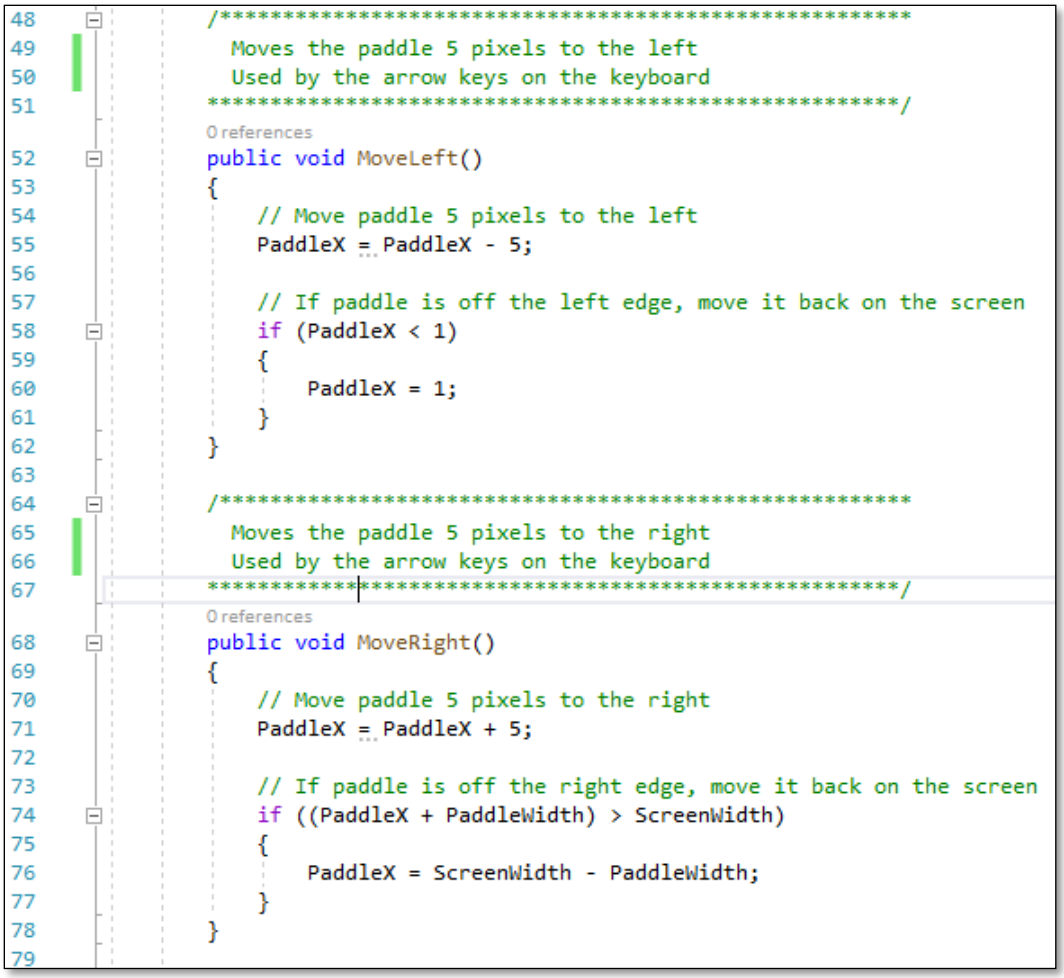
Paddle Class

Let's create a class called **Paddle** to represent the user's paddle.

1. In Visual Studio, right-click on the **Blocks** project. Select **Add** → **Class** from the drop-down menus.
2. In the **Add New Item** Dialog, enter **Paddle.cs** for the class name, then click **Add**:
3. In the **Paddle.cs** file, enter the following code:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Blocks3
{
    3 references
    class Paddle
    {
        /*******************************************************
          Paddle class properties
        *******************************************************/
        13 references
        public float PaddleX { get; set; } // X position of paddle on screen
        2 references
        public float PaddleY { get; set; } // Y position of paddle on screen
        5 references
        public float PaddleWidth { get; set; } // Width of paddle
        1 reference
        public float PaddleHeight { get; set; } // Height of paddle

        // Width of game screen, used to keep paddle inside the game play boundary
        5 references
        public float ScreenWidth { get; set; }
        4 references
        private Texture2D imgPaddle { get; set; }  // Cached image of the paddle

        // Allows us to write on backbuffer when we need to draw self/this object
        private readonly SpriteBatch spriteBatch;
```

```
        /*******************************************************
          Paddle object constructor intitializes Paddle properties
        *******************************************************/
        1 reference
        public Paddle(float x, float y, float screenWidth,
                      SpriteBatch spriteBatch, GameContent gameContent)
        {
            PaddleX = x;
            PaddleY = y;
            imgPaddle = gameContent.imgPaddle;
            PaddleWidth = imgPaddle.Width;
            PaddleHeight = imgPaddle.Height;
            this.spriteBatch = spriteBatch;
            ScreenWidth = screenWidth;
        }

        /*******************************************************
          Draw the paddle on the screen
        *******************************************************/
        1 reference
        public void Draw()
        {
            spriteBatch.Draw(imgPaddle, new Vector2(PaddleX, PaddleY),
                             null, Color.White, 0, new Vector2(0, 0),
                             1.0f, SpriteEffects.None, 0);
        }
```



```
48          /*******************************************************
49            Moves the paddle 5 pixels to the left
50            Used by the arrow keys on the keyboard
51          *******************************************************/
            0 references
52          public void MoveLeft()
53          {
54              // Move paddle 5 pixels to the left
55              PaddleX = PaddleX - 5;
56
57              // If paddle is off the left edge, move it back on the screen
58              if (PaddleX < 1)
59              {
60                  PaddleX = 1;
61              }
62          }
63
64          /*******************************************************
65            Moves the paddle 5 pixels to the right
66            Used by the arrow keys on the keyboard
67          *******************************************************/
            0 references
68          public void MoveRight()
69          {
70              // Move paddle 5 pixels to the right
71              PaddleX = PaddleX + 5;
72
73              // If paddle is off the right edge, move it back on the screen
74              if ((PaddleX + PaddleWidth) > ScreenWidth)
75              {
76                  PaddleX = ScreenWidth - PaddleWidth;
77              }
78          }
79
```

```
80          /*******************************************************
81            Moves the paddle to a specified x location
82            Used by the mouse
83          *******************************************************/
            0 references
84          public void MoveTo(float x)
85          {
86              // Test to see if the paddle is inside the left side of the screen
87              if (x >= 0)
88              {
89                  // If the paddle is inside the right side of the screen
90                  // Move the paddle
91                  if (x < ScreenWidth - PaddleWidth)
92                  {
93                      PaddleX = x;
94                  }
95                  // If the paddle is off the right edge, move it back
96                  else
97                  {
98                      PaddleX = ScreenWidth - PaddleWidth;
99                  }
100             }
101             else
102             {
103                 // If the paddle is off the left edge, move it back
104                 if (x < 0)
105                 {
106                     PaddleX = 0;
107                 }
108             }
109         }
110     }
111 }
```

Let's look at the class properties.

1. **PaddleX** and **PaddleY:** Coordinates of the paddle on the screen (with 0, 0 being at the top left corner of the screen).
2. **PaddleWidth** and **PaddleHeight:** Size of the paddle in pixels.
3. **ScreenWidth:** Has the width of the game field in pixels. We'll use that to keep the paddle within the game play boundary.
4. **imgPaddle:** Contains the image of the paddle that we'll draw on the screen.
5. **spriteBatch:** Will be used to draw to an off screen buffer for eventual display to the screen. This is much faster than drawing directly to the screen.

All of the fields are initialized in the class constructor method. This create a Paddle object.

We will call the **Paddle Draw** method from the **Game1 Draw** method. We'll use the **spriteBatch Draw** method to write the image to the screen buffer for display:

```
spriteBatch.Draw(imgPaddle, new Vector2(PaddleX, PaddleY),
                 null, Color.White, 0, new Vector2(0, 0), 1.0f, SpriteEffects.None, 0);
```

Let's take a look at the arguments for the **Draw** method.

- **imgPaddle:** The first argument is the image we want to draw. We saved a reference to the image we loaded in the **GameContent** class so we can draw it when requested.
- **new Vector2(PaddleX, PaddleY):** This is the **X, Y** coordinates on the screen where the paddle will be drawn.
- **null:** This is the portion of the image we want to draw. It is a rectangle (`X`, `Y`, `Width`, `Height`). We want to draw the whole image, so we left this parameter "null". We could use this to clip the image if we wanted to.
- **Color.White:** This is the tint color. We specified **Color.White** which means we won't apply a tint. We'll use this parameter later to tint the game Blocks.
- **newVector2(0, 0)** The next two fields are used if we want to rotate the image. We don't want to, so we'll use a rotation of "0", and a rotation origin of "0,0".
- **1.0f:** This is the scale that we want to draw the image with. We are using a value of "1.0" which means we want it at 100%, or full size. If we used, say ".5", the image would be scaled to 50% of the size from the file.
- **SpriteEffects:** We aren't using SpriteEffects, we passed a value of "None". This field can be used to flip the image horizontally or vertically.
- **0:** The last field is the layer we want to draw the image on. If we are drawing multiple images at the same coordinates, we can use this to determine the order in which the images are drawn. We won't be using this for this tutorial.

We've also created three other methods.

- **MoveLeft**: This moves the paddle coordinates to the left by 5 pixels. It also ensure we don't move beyond the playing field.
- **MoveRight:** This method moves the paddle coordinates to the right by 5 pixels, It also ensures we don't move beyond the playing field.
- **MoveTo**: This method moves the paddle to the specified coordinate.

We won't use these methods yet. We will need them later when we add the logic to move the paddle to our game.

Blocks Class

Let's add the Blocks that we'll be smashing in our game. We'll add a new **Block** class for this.

1. Right-click on the project **Blocks** in the solution explorer. Select **Add** → **Class** from the drop-down menus.
2. In the **Add New Item** Dialog, enter **Block.cs** for the class name, then click **Add**:
3. Enter the following code into your **Block.cs** file.

```
1   using Microsoft.Xna.Framework;
2   using Microsoft.Xna.Framework.Graphics;
3
4   namespace Blocks3
5   {
        5 references
6       class Block
7       {
8           /*********************************************************
9             Block class properties
10          *********************************************************/
            2 references
11          public float BlockX { get; set; } // X position of block on screen
            2 references
12          public float BlockY { get; set; } // Y position of block on screen
            1 reference
13          public float BlockWidth { get; set; } // Width of block
            1 reference
14          public float BlockHeight { get; set; } // Height of block
            2 references
15          public bool IsBlockVisible { get; set; } // Does block still exist?
16          private Color color;
            4 references
17          private Texture2D imgBlock { get; set; }  // Cached image of the block
18
19          // Allows us to write on backbuffer when we need to draw self/object
20          private SpriteBatch spriteBatch;
21
22          /*********************************************************
23            Block object constructor intitializes Block properties
24          *********************************************************/
            1 reference
25          public Block(float x, float y, Color color,
26              SpriteBatch spriteBatch, GameContent gameContent)
27          {
28              BlockX = x;
29              BlockY = y;
30              imgBlock = gameContent.imgBlock;
31              BlockWidth = imgBlock.Width;
32              BlockHeight = imgBlock.Height;
33              this.spriteBatch = spriteBatch;
34              IsBlockVisible = true;
35              this.color = color;
36          }
37
            1 reference
38          public void Draw()
39          {
40              if (IsBlockVisible)
41              {
42                  spriteBatch.Draw(imgBlock, new Vector2(BlockX, BlockY), null,
43                                   color, 0, new Vector2(0, 0), 1.0f, SpriteEffects.None, 0);
44              }
45          }
46      }
47  }
```

This is the code we'll need to manage a single block. Let's start with the properties.

- **BlockX, BlockY:** These properties are the X and Y coordinates of the block on the screen.
- **BlockWidth, BlockHeight:** These properties give us the size of the block.
- **IsBlockVisible:** We'll use the **Visible** property to determine if the block has been destroyed or not. We'll set this property to **false** after the ball destroys it.
- **Color**: This property will be used to determine what color the block should draw itself as.
- **imgBlock:** We'll store a reference to the block image in this property
- **spriteBatch:** We use this to draw the block when requested.

All of the properties will be set in the constructor methods from arguments that are passed.

The **Draw** method will be called when the block needs to be drawn. It will first check to see if the block is visible. If it is, it will draw the block using the **spriteBatch.Draw** method call.

All of the parameters are the same as we've previously described in the paddle class, with one exception. In this call, we are making use of the *color* argument to tint the block to different colors. If you took a look at the "block.png" file, you would notice that it is black and white.

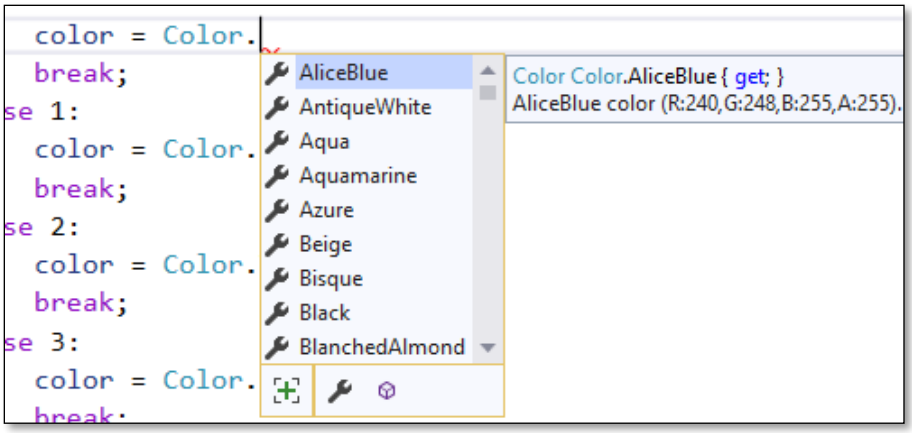
As you will see when we draw them in the game, we show all of the colors of the rainbow. We do this by using a feature of MonoGame. The *color* argument will apply a tint to the image. A value of `Color.White` will tells MonoGame to apply no tint. Any other color value will cause that tint color to be applied to the image when drawn.

Wall Class

Our game won't be terribly impressive with only one block. Let's build a wall of Blocks. We'll create a new class called **Wall** that will have seven rows of Blocks that we can destroy.

Block is a class, we can create as many Block objects as we need for our game.

1. Right-click on the project **Blocks** in the solution explorer. Select **Add** → **Class** from the drop-down menus.
2. In the **Add New Item** Dialog, enter **Wall.cs** for the class name, then click **Add**.
3. Enter the following code into your **Wall.cs** file.
4. For the colors, you can choose any built-in system colors you wish. When you type in Color. You will see a big list of colors available.


color = Color.
break;
se 1:
color = Color.
break;
se 2:
color = Color.
break;
se 3:
color = Color.
break;
AliceBlue
AntiqueWhite
Aqua
Aquamarine
Azure
Beige
Bisque
Black
BlanchedAlmond
Color Color.AliceBlue { get; }
AliceBlue color (R:240,G:248,B:255,A:255).

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Blocks3
{
    3 references
    class Wall
    {
        /*************************************************************
          Wall class properties
          7 rows, each with its own color
          11 bricks per row, 3 blank rows at top
          Each brick is 50 x 16
          A 7 x 11 2 dimensional array will hold the bricks
        *************************************************************/
        3 references
        public Block[,] BlockWall { get; set; }
        const int ROWS = 7;
        const int COLUMNS = 11;

        1 reference
        public Wall(float x, float y, SpriteBatch spriteBatch, GameContent gameContent)
        {
            // Create a 2 dimensional array of empty Block objects
            BlockWall = new Block[ROWS, COLUMNS];
            float BlockX = x;
            float BlockY = y;
            Color color = Color.White;

            // Use a different color for each row
            for (int i = 0; i < ROWS; i++)
            {
                switch (i)
                {
                    case 0:
                        color = Color.Red;
                        break;
                    case 1:
                        color = Color.Orange;
                        break;
                    case 2:
                        color = Color.Yellow;
                        break;
                    case 3:
                        color = Color.Green;
                        break;
                    case 4:
                        color = Color.Blue;
                        break;
                    case 5:
                        color = Color.IndianRed;
                        break;
                    case 6:
                        color = Color.Violet;
                        break;
                }

```

```
55                  BlockY = y + i * (gameContent.imgBlock.Height + 1);
56
57                  for (int j = 0; j < COLUMNS; j++)
58                  {
59                      BlockX = x + j * (gameContent.imgBlock.Width);
60                      // Create a new Block object
61                      Block block = new Block(BlockX, BlockY, color, spriteBatch, gameContent);
62                      // Place Block object into array
63                      BlockWall[i, j] = block;
64                  }
65              }
66          }
67
68          // Iterate through the array to draw the block wall
            1 reference
69          public void Draw()
70          {
71              for (int i = 0; i < ROWS; i++)
72              {
73                  for (int j = 0; j < COLUMNS; j++)
74                  {
75                      BlockWall[i, j].Draw();
76                  }
77              }
78          }
79      }
80  }
```

Our wall will have 7 rows of 11 Blocks. We create a 2 dimensional array of `Block` objects called **BlockWall** to hold our Blocks. Our constructor will pass the X and Y coordinates of the top left corner of the wall. Since we know the height and width of a single block, our constructor will just increment the coordinates to lay out the Blocks in a 7 x 11 grid. For each of the seven rows we assign a different drawing color.

In our **Draw** method we iterate through the **block** array, and tell each block to draw itself.

Game1 Class

We need to create an instance of the **Paddle** and **Wall** class in **Game1.cs**. We'll create two private variable to hold the screen height and width. The **Game1** class will create an instance of the wall and paddle , and have its **Draw** method call the **Wall** and **Paddle** class **Draw** method.

In "Game1.cs", add new **wall, paddle, screenWidth** and **screenHeight** properties.

```
public class Game1 : Game
{
    /*************************************************************
      Game class properties
    *************************************************************/
    private readonly GraphicsDeviceManager _graphics;
    private SpriteBatch _spriteBatch;
    private GameContent _gameContent;

    private Paddle paddle;
    private Wall wall;

    // Variables to store the screen size
    private int screenWidth = 0;
    private int screenHeight = 0;
```

We'll update our **LoadContent** method in **Game1.cs** to create our **Paddle** and Wall object, and also set the game screen size. Add the new lines below to **Game1 LoadContent**.

```
/*****************************************************
  Load game content resources into memory
*****************************************************/
7 references
protected override void LoadContent()
{
    // Create a SpriteBatch instance/object to draw our sprites on the screen
    _spriteBatch = new SpriteBatch(GraphicsDevice);

    // Create a GameContent object that references the content assets
    _gameContent = new GameContent(Content);

    // Get the current screensize
    screenWidth = GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Width;
    screenHeight = GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Height;

    // Set game to 502x700 or screen max if smaller
    if (screenWidth >= 552)
    {
        screenWidth = 552;
    }
    if (screenHeight >= 700)
    {
        screenHeight = 700;
    }

    // Set the program window size
    _graphics.PreferredBackBufferWidth = screenWidth;
    _graphics.PreferredBackBufferHeight = screenHeight;
    _graphics.ApplyChanges();

    // Center the paddle horizontally on the screen to start
    int paddleX = (screenWidth - _gameContent.imgPaddle.Width) / 2;

    // Set paddle 100 pixels from the bottom of the screen
    int paddleY = screenHeight - 100;

    // Create game objects
    paddle = new Paddle(paddleX, paddleY, screenWidth, _spriteBatch, _gameContent);
    wall = new Wall(1, 50, _spriteBatch, _gameContent);
}
```

This method will resize the game board and window to the size we want, and create our **Paddle** and **Wall** object instance. We are using the **GraphicsAdapter** Class to get the current screen size, and then changing it to 502×700 pixels. We use the **graphics** object to change the game window size, and call the **ApplyChanges** method to have MonoGame update the window's size. The next couple of lines will set the *X*,*Y* coordinates for the paddle to the middle of the screen, 100 pixels from the bottom.

Now that the paddle and the wall has logic to draw itself, we need to tell the paddle to draw itself whenever we get a draw event from MonoGame. We need to add the lines to **Game1.cs** to call our new **Draw** Method. We also need to change the background color of the game board to Black. We'll do that by changing the

GraphicsDevice.Clear(Color.CornflowerBlue)” statement to

GraphicsDevice.Clear(Color.Black)”.

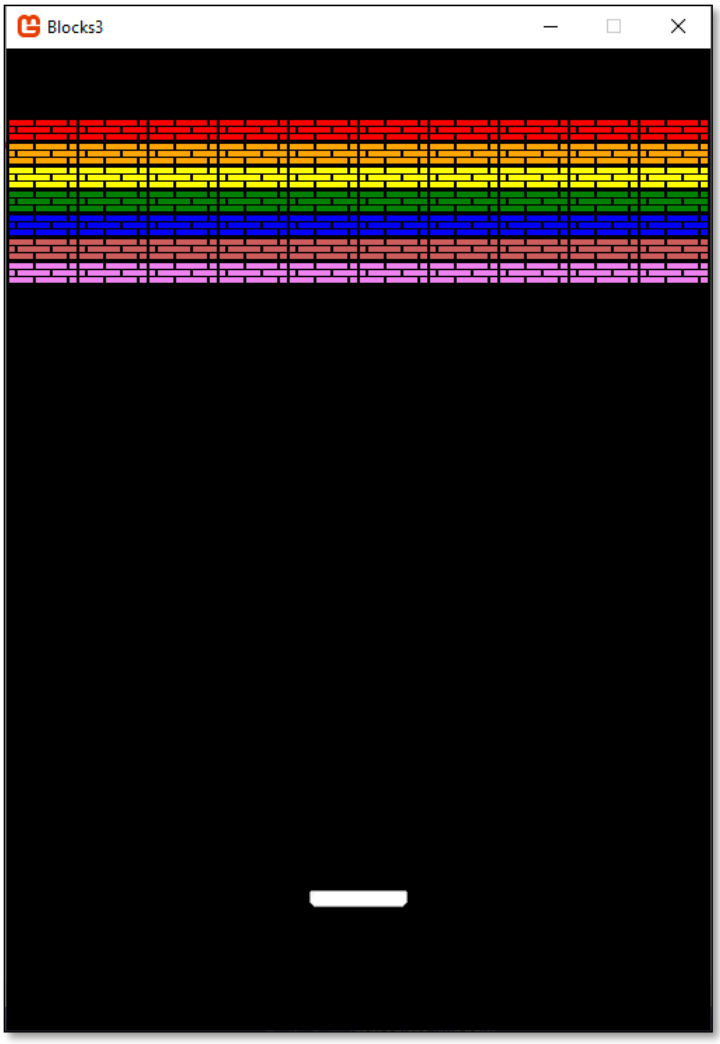
We also add the `spriteBatch.Begin()`, `paddle.Draw()`, and `spriteBatch.End()` method calls:

```
/*****************************************************
  Draw sprites to the screen
*****************************************************/
7 references
protected override void Draw(GameTime gameTime)
{
    // Draw the program window as black
    GraphicsDevice.Clear(Color.Black);

    // Begin drawing to buffer
    _spriteBatch.Begin();

    /// Call the game objects Draw methods
    paddle.Draw();
    wall.Draw();

    // Write buffer to screen
    _spriteBatch.End();

    base.Draw(gameTime);
}
```

Let’s talk about the `spriteBatch` object. We will be using this object to draw all of the objects to the screen. All of the `spriteBatch.Draw` method calls between the `spriteBatch.Begin` call and `spriteBatch.End` call, will cause the images to be written to an off screen memory buffer. The `End` call will cause that memory to be written to the screen. All of our `Draw` calls must be done between the pair of `Begin` and `End` calls.

Run the Game

Run the game by pressing **F5**. The game should now display both the paddle and the wall of Blocks we just added.



Our playing field is nearly complete. We are nearly ready to start adding some action to the game, but first, we need to add a game border.

Assignment Submission

Zip up the pseudocode and the project folder. Submit in Blackboard.