```
1  using System; // add to allow Windows message box
2  using System.Runtime.InteropServices; // add to allow Windows message box
3
4  using Microsoft.Xna.Framework;
5  using Microsoft.Xna.Framework.Graphics;
6  using Microsoft.Xna.Framework.Input;
7  using Microsoft.Xna.Framework.Audio;
8  using System.Collections.Generic;
9
10 namespace Demo_MG_MazeGame
11 {
12     public enum GameAction
13     {
14         None,
15         PlayerRight,
16         PlayerLeft,
17         PlayerUp,
18         PlayerDown,
19         Quit
20     }
21
22     /// <summary>
23     /// This is the main type for your game.
24     /// </summary>
25     public class MazeGame : Game
26     {
27         // add code to allow Windows message boxes when running in a Windows enviro
           nment
28         [DllImport("user32.dll", CharSet = CharSet.Auto)]
29         public static extern uint MessageBox(IntPtr hWnd, String text, String
           caption, uint type);
30
31         // set the cell size in pixels
32         private const int CELL_WIDTH = 64;
33         private const int CELL_HEIGHT = 64;
34
35         // set the map size in cells
36         private const int MAP_CELL_ROW_COUNT = 9;
37         private const int MAP_CELL_COLUMN_COUNT = 9;
38
39         // set the window size
40         private const int WINDOW_WIDTH = MAP_CELL_COLUMN_COUNT * CELL_WIDTH;
41         private const int WINDOW_HEIGHT = MAP_CELL_ROW_COUNT * CELL_HEIGHT;
42
43         // wall objects
44         private List<Wall> walls;
45         private Wall wall01;
46         private Wall wall02;
47
48         // player object
49         private Player player;
50
51         // variable to hold the player's current game action
52         GameAction playerGameAction;
53
54         // keyboard state objects to track a single keyboard press
55         KeyboardState newState;
56         KeyboardState oldState;
57
58         // declare a GraphicsDeviceManager object
59         GraphicsDeviceManager graphics;
60
61         // declare a SpriteBatch object
62         SpriteBatch spriteBatch;
63
64         public MazeGame()
65         {
66             graphics = new GraphicsDeviceManager(this);
```

```
67
68              // set the window size
69              graphics.PreferredBackBufferWidth = MAP_CELL_COLUMN_COUNT * CELL_WIDTH
                ;
70              graphics.PreferredBackBufferHeight = MAP_CELL_ROW_COUNT * CELL_HEIGHT;
71
72              Content.RootDirectory = "Content";
73          }
74
75          /// <summary>
76          /// Allows the game to perform any initialization it needs to before starti
                ng to run.
77          /// This is where it can query for any required services and load any non-g
                raphic
78          /// related content.  Calling base.Initialize will enumerate through any co
                mponents
79          /// and initialize them as well.
80          /// </summary>
81          protected override void Initialize()
82          {
83              // add floors, walls, and ceilings
84              walls = new List<Wall>();
85
86              wall01 = new Wall(Content, "wall", new Vector2(4 * CELL_WIDTH, 4 *
                CELL_HEIGHT));
87              wall01.Active = true;
88              walls.Add(wall01);
89
90              // add the player
91              player = new Player(Content, new Vector2(2 * CELL_WIDTH, 2 *
                CELL_HEIGHT));
92              player.Active = true;
93
94              // set the player's initial speed
95              player.SpeedHorizontal = 10;
96              player.SpeedVertical = 10;
97
98              base.Initialize();
99          }
100
101         /// <summary>
102         /// LoadContent will be called once per game and is the place to load
103         /// all of your content.
104         /// </summary>
105         protected override void LoadContent()
106         {
107             // Create a new SpriteBatch, which can be used to draw textures.
108             spriteBatch = new SpriteBatch(GraphicsDevice);
109
110             // Note: wall and player sprites loaded when instantiated
111         }
112
113         /// <summary>
114         /// UnloadContent will be called once per game and is the place to unload
115         /// game-specific content.
116         /// </summary>
117         protected override void UnloadContent()
118         {
119             // Unload any non ContentManager content here
120         }
121
122         /// <summary>
123         /// Allows the game to run logic such as updating the world,
124         /// checking for collisions, gathering input, and playing audio.
125         /// </summary>
126         /// <param name="gameTime">Provides a snapshot of timing values.</param>
127         protected override void Update(GameTime gameTime)
```

```
128             {
129                 // get the player's current action based on a keyboard event
130                 playerGameAction = GetKeyboardEvents();
131 
132                 switch (playerGameAction)
133                 {
134                 case GameAction.None:
135                         break;
136 
137                     // move player right
138                 case GameAction.PlayerRight:
139                         player.PlayerDirection = Player.Direction.Right;
140 
141                         // only move player if allowed
142                         if (CanMove())
143                         {
144                             player.Position = new Vector2(player.Position.X + player
                        .SpeedHorizontal, player.Position.Y);
145                         }
146                         break;
147 
148                     //move player left
149                 case GameAction.PlayerLeft:
150                         player.PlayerDirection = Player.Direction.Left;
151 
152                         // only move player if allowed
153                         if (CanMove())
154                         {
155                             player.Position = new Vector2(player.Position.X - player
                        .SpeedHorizontal, player.Position.Y);
156                         }
157 
158                         break;
159 
160                     // move player up
161                 case GameAction.PlayerUp:
162                         player.PlayerDirection = Player.Direction.Up;
163 
164                         // only move player if allowed
165                         if (CanMove())
166                         {
167                             player.Position = new Vector2(player.Position.X, player.
                        Position.Y - player.SpeedVertical);
168                         }
169                         break;
170 
171                 case GameAction.PlayerDown:
172                         player.PlayerDirection = Player.Direction.Down;
173 
174                         // only move player if allowed
175                         if (CanMove())
176                         {
177                             player.Position = new Vector2(player.Position.X, player.
                        Position.Y + player.SpeedVertical);
178                         }
179                         break;
180 
181                     // quit game
182                 case GameAction.Quit:
183                         Exit();
184                         break;
185 
186                 default:
187                         break;
188                 }
189 
190                 base.Update(gameTime);
191             }
```

```
192
193         /// <summary>
194         /// This is called when the game should draw itself.
195         /// </summary>
196         /// <param name="gameTime">Provides a snapshot of timing values.</param>
197         protected override void Draw(GameTime gameTime)
198         {
199             GraphicsDevice.Clear(Color.CornflowerBlue);
200
201             spriteBatch.Begin();
202
203             wall01.Draw(spriteBatch);
204
205             player.Draw(spriteBatch);
206
207             spriteBatch.End();
208
209             base.Draw(gameTime);
210         }
211
212         /// <summary>
213         /// get keyboard events
214         /// </summary>
215         /// <returns>GameAction</returns>
216         private GameAction GetKeyboardEvents()
217         {
218             GameAction playerGameAction = GameAction.None;
219
220             newState = Keyboard.GetState();
221
222             if (CheckKey(Keys.Right) == true)
223             {
224                 playerGameAction = GameAction.PlayerRight;
225             }
226             else if (CheckKey(Keys.Left) == true)
227             {
228                 playerGameAction = GameAction.PlayerLeft;
229             }
230             else if (CheckKey(Keys.Up) == true)
231             {
232                 playerGameAction = GameAction.PlayerUp;
233             }
234             else if (CheckKey(Keys.Down) == true)
235             {
236                 playerGameAction = GameAction.PlayerDown;
237             }
238             else if (CheckKey(Keys.Escape) == true)
239             {
240                 playerGameAction = GameAction.Quit;
241             }
242
243             oldState = newState;
244
245             return playerGameAction;
246         }
247
248         /// <summary>
249         /// check the current state of the keyboard against the previous state
250         /// </summary>
251         /// <param name="theKey">bool new key press</param>
252         /// <returns></returns>
253         private bool CheckKey(Keys theKey)
254         {
255             // allows the key to be held down
256             return newState.IsKeyDown(theKey);
257
258             // player must continue to tap the key
259             //return oldState.IsKeyDown(theKey) && newState.IsKeyUp(theKey);
```

```
260             }
261
262             /// <summary>
263             /// check to confirm that player movement is allowed
264             /// </summary>
265             /// <returns></returns>
266             private bool CanMove()
267             {
268                 bool canMove = true;
269
270                 // do not allow movement into wall
271                 if (WallCollision(wall01))
272                 {
273                     canMove = false;
274                 }
275
276                 return canMove;
277             }
278
279             /// <summary>
280             /// test for player collision with a wall object
281             /// </summary>
282             /// <param name="wall">wall object to test</param>
283             /// <returns>true if collision</returns>
284             private bool WallCollision(Wall wall)
285             {
286                 bool wallCollision = false;
287
288                 // create a Rectangle object for the new move's position
289                 Rectangle newPlayerPosition = player.BoundingRectangle;
290
291                 // test the new move's position for a collision with the wall
292                 switch (player.PlayerDirection)
293                 {
294                 case Player.Direction.Left:
295                         // set the position of the new move's rectangle
296                         newPlayerPosition.Offset(-player.SpeedHorizontal, 0);
297
298                         // test for a collision with the new move and the wall
299                         if (newPlayerPosition.Intersects(wall.BoundingRectangle))
300                         {
301                             wallCollision = true;
302
303                             // move player next to wall
304                             player.Position = new Vector2(wall.BoundingRectangle.
                         Right, player.Position.Y);
305                         }
306                         break;
307
308                 case Player.Direction.Right:
309                         // set the position of the new move's rectangle
310                         newPlayerPosition.Offset(player.SpeedHorizontal, 0);
311
312                         // test for a collision with the new move and the wall
313                         if (newPlayerPosition.Intersects(wall.BoundingRectangle))
314                         {
315                             wallCollision = true;
316
317                             // move player next to wall
318                             player.Position = new Vector2(wall.BoundingRectangle.
                         Left - player.BoundingRectangle.Width, player.Position.Y);
319                         }
320                         break;
321
322                 case Player.Direction.Up:
323                         // set the position of the new move's rectangle
324                         newPlayerPosition.Offset(0, -player.SpeedVertical);
325
```

```
326                         // test for a collision with the new move and the wall
327                         if (newPlayerPosition.Intersects(wall.BoundingRectangle))
328                         {
329                             wallCollision = true;
330
331                             // move player next to wall
332                             player.Position = new Vector2(player.Position.X, wall.
                            BoundingRectangle.Bottom);
333                         }
334                         break;
335
336                 case Player.Direction.Down:
337                         // set the position of the new move's rectangle
338                         newPlayerPosition.Offset(0, player.SpeedVertical);
339
340                         // test for a collision with the new move and the wall
341                         if (newPlayerPosition.Intersects(wall.BoundingRectangle))
342                         {
343                             wallCollision = true;
344
345                             // move player next to wall
346                             player.Position = new Vector2(player.Position.X, wall.
                            BoundingRectangle.Top - player.BoundingRectangle.Height);
347                         }
348                         break;
349
350                 default:
351                         break;
352                 }
353
354                 return wallCollision;
355             }
356         }
357 }
```