```
1  using System;
2  using System.Collections.Generic;
3  using TMPro;
4  using UnityEngine;
5  using UnityEngine.EventSystems;
6
7  /// <summary>
8  /// BehaviorManager primarily handles the bulk of all dynamic and scripted
      non-UI events triggerable by the user.
9  /// </summary>
10 public class BehaviorManager : MonoBehaviour
11 {
12     // Singleton state reference
13     private static BehaviorManager instance;
14
15     /// <summary>
16     /// The current state that the editor scene is in.
17     /// </summary>
18     public enum GameState { GRID_HOVER, CIRCUIT_HOVER, CIRCUIT_MOVEMENT,
         CIRCUIT_PLACEMENT, IO_HOVER, IO_PRESS, USER_INTERFACE, WIRE_HOVER,
         WIRE_PRESS }
19
20     /// <summary>
21     /// Utilized alongside a <seealso cref="GameState"/> to determine the
         consequences of each game state.<br/><br/>
22     /// <seealso cref="UNRESTRICTED"/>: nothing occurs.<br/>
23     /// <seealso cref="LOCKED"/>: most/all other non-UI elements are
         locked; UI is still enabled.<br/>
24     /// <seealso cref="PAUSED"/>: all non-UI elements are locked.
25     /// </summary>
26     public enum StateType { UNRESTRICTED, LOCKED, PAUSED }
27
28     /// <summary>
29     /// Cancels most non-UI and UI related events when pressed.<br/><br/>
30     /// More often than not, an alternate option to cancel such events
         also occur with the right mouse button.
31     /// </summary>
32     [SerializeField]
33     KeyCode cancelKey;
34
35     /// <summary>
36     /// Displays the name of any hovered inputs/outputs belonging to a
         custom circuit, if any.
37     /// </summary>
38     [SerializeField]
39     TextMeshProUGUI ioText;
40
41     /// <summary>
42     /// Reserved for some UI events that should only occur once.
```

```
43      /// </summary>
44      private bool doOnce;
45
46      /// <summary>
47      /// Whether the left mouse button was pressed when hovered on an input or output node.
48      /// </summary>
49      private bool ioLMB;
50
51      /// <summary>
52      /// Whether all non-UI elements should remain cancelled no matter what.<br/><br/>.
53      /// If this value is enabled, then control must be restored to the user through external means.
54      /// </summary>
55      private bool lockUI;
56
57      /// <summary>
58      /// Utilized internally for placing, moving, and deleting circuits and their physical GameObjects.
59      /// </summary>
60      private Circuit currentCircuit;
61
62      /// <summary>
63      /// The current input that the user is attempting to connect.
64      /// </summary>
65      private Circuit.Input currentInput;
66
67      /// <summary>
68      /// The current output that the user is attempting to connect.
69      /// </summary>
70      private Circuit.Output currentOutput;
71
72      /// <summary>
73      /// The current preview pin corresponding to an input node that the user is hovered on.
74      /// </summary>
75      private GameObject currentPreviewPin;
76
77      /// <summary>
78      /// Keeps track of the current game state as well as the previous game state if the game is currently paused.
79      /// </summary>
80      private GameState gameState, unpausedGameState;
81
82      /// <summary>
83      /// The opposite layer of the first input or output the user has pressed in a new connection attempt.<br/><br/>
84      /// This helps determine whether the next element to press should be
```

```
          an input or output.<br/>
85        /// If an input was first pressed, then the second valid press must be for an output, and vice-versa.
86        /// </summary>
87        private int ioLayerCheck;
88
89        /// <summary>
90        /// Utilized for raycasting all in-scene GameObjects to determine the current game state and state type.
91        /// </summary>
92        private Ray ray;
93
94        /// <summary>
95        /// Keeps track of the current state type as well as the previous state type if the game is currently paused.
96        /// </summary>
97        private StateType stateType, unpausedStateType;
98
99        /// <summary>
100       /// Utilized for moving circuits around the editor scene.
101       /// </summary>
102       private Vector3 deltaPos, endingOffset, prevDeltaPos, startingOffset, startingPos;
103
104       // Enforces a singleton state pattern
105       private void Awake()
106       {
107           if (instance != null)
108           {
109               Destroy(this);
110               throw new Exception("BehaviorManager instance already established; terminating.");
111           }
112
113           instance = this;
114       }
115
116       // Listens to and acts on additional UI-based events.
117       private void Update()
118       {
119           // If the scene is currently listening to UI, return and disable set values
120           if (EventSystem.current.IsPointerOverGameObject() || lockUI)
121           {
122               if (currentPreviewPin != null)
123               {
124                   currentPreviewPin.SetActive(false);
125                   currentPreviewPin = null;
126               }
```

```
127
128             // Disables the currently hovered display pin, if any
129             // This specifically occurs once as ioText is externally used
                  and can have non-empty text.
130             if (doOnce && ioText.text != "") ioText.text = "";
131
132             doOnce = false;
133             return;
134         }
135
136         // Otherwise, checks for all relevant events by beginning to
              raycast.
137         doOnce = true;
138         ray = CameraMovement.Instance.PlayerCamera.ScreenPointToRay
              (Input.mousePosition);
139
140         // If nothing is raycasted, also return and disable set values.
141         if (!Physics.Raycast(ray, out RaycastHit hitInfo))
142         {
143             if (currentPreviewPin != null)
144             {
145                 currentPreviewPin.SetActive(false);
146                 currentPreviewPin = null;
147             }
148
149             if (ioText.text != "") ioText.text = "";
150
151             return;
152         }
153
154         GameObject hitObj = hitInfo.transform.gameObject;
155
156         // If hovered on an input or output belonging to a custom circuit,
               obtain its label.
157         if ((hitObj.layer == 9 || hitObj.layer == 10) &&
              hitObj.GetComponentInParent<CircuitReference>().Circuit.GetType
              () == typeof(CustomCircuit))
158         {
159             CustomCircuit customCircuit = (CustomCircuit)
                  hitObj.GetComponentInParent<CircuitReference>().Circuit;
160
161             int index;
162             string label;
163
164             // Is an input, therefore looks in relevant input variables.
165             if (hitObj.layer == 9)
166             {
167                 index = Array.IndexOf(customCircuit.Inputs,
                      hitObj.GetComponent<CircuitVisualizer.InputReference>
```

```
                    ().Input);
168                 label = customCircuit.PreviewStructure.InputLabels[index];
169             }
170
171             // Is an output, therefore looks in relevant output variables.
172             else
173             {
174                 index = Array.IndexOf(customCircuit.Outputs,
                      hitObj.GetComponent<CircuitVisualizer.OutputReference>
                      ().Output);
175                 label = customCircuit.PreviewStructure.OutputLabels
                      [index];
176             }
177
178             ioText.text = label;
179         }
180
181         // Otherwise, therre is no label to display
182         else if (ioText.text != "") ioText.text = "";
183
184         // If hovered on an input belonging to a display, enable its
              corresponding preview pin
185         if (hitObj.layer == 9 &&
              hitObj.GetComponentInParent<CircuitReference>().Circuit.GetType
              () == typeof(Display))
186         {
187             // Occurs if still hovered on the same input node
188             if (currentPreviewPin == hitObj.transform) return;
189
190             Display display = (Display)
                  hitObj.GetComponentInParent<CircuitReference>().Circuit;
191             int index = -1;
192
193             // Determines which preview pin should be enabled
194             for (int i = 0; i < 8; i++)
195             {
196                 if (display.Inputs[i].Transform.gameObject == hitObj)
197                 {
198                     index = i;
199                     break;
200                 }
201             }
202
203             // If the last frame focused on a separate preview pin,
                  disable that first
204             if (currentPreviewPin != null) currentPreviewPin.SetActive
                  (false);
205
206             // Enable the current preview pin
```

```
207             currentPreviewPin = display.PreviewPins[index];
208             currentPreviewPin.SetActive(true);
209         }
210
211         // Otherwise, disable the current preview pin, if any
212         else if (currentPreviewPin != null)
213         {
214             currentPreviewPin.SetActive(false);
215             currentPreviewPin = null;
216         }
217     }
218
219     // Obtains a new game state/state type, and if applicable, listens to
        input/events corresponding to the game state.
220     private void LateUpdate()
221     {
222         gameState = UpdateGameState();
223         GameStateListener();
224     }
225
226     /// <summary>
227     /// Obtains a new GameState by performing a raycast in combination
        with the current game state.
228     /// </summary>
229     /// <returns>The new game state to switch to</returns>
230     private GameState UpdateGameState()
231     {
232         // Current state is UI
233         if (EventSystem.current.IsPointerOverGameObject() || lockUI)
234         {
235             if (gameState == GameState.USER_INTERFACE) return
                gameState; // Last state was UI, return.
236
237             // The UI state pauses the previous game state/state type,
                storing it in separate paused values.
238             unpausedGameState = gameState;
239             unpausedStateType = stateType;
240             stateType = StateType.PAUSED;
241             Cursor.visible = true;
242             CursorManager.SetMouseTexture(true);
243             return GameState.USER_INTERFACE;
244         }
245
246         // Current game state is not UI but the previous game state was.
247         // Therefore, restore the game state/state type present before the
            user hovered onto UI.
248         if (gameState == GameState.USER_INTERFACE)
249         {
250             gameState = unpausedGameState;
```

```
251             stateType = unpausedStateType;
252
253             // Conditions for a visible cursor
254             Cursor.visible = unpausedGameState !=
                  GameState.CIRCUIT_MOVEMENT && unpausedGameState !=
                  GameState.CIRCUIT_PLACEMENT;
255         }
256
257         // Locked states must change manually, not automatically.
258         if (stateType == StateType.LOCKED) return gameState;
259
260         // The raycast reached nothing -- defaults to the grid hover
              state.
261         if (!Physics.Raycast(ray, out RaycastHit hitInfo))
262         {
263             stateType = StateType.UNRESTRICTED;
264             CursorManager.SetMouseTexture(true);
265             return GameState.GRID_HOVER;
266         }
267
268         GameObject hitObject = hitInfo.transform.gameObject;
269
270         // Mouse is on top of a circuit & LMB and/or RMB have been pressed
271         if (gameState == GameState.CIRCUIT_HOVER &&
              (Input.GetMouseButtonDown(0) || Input.GetMouseButtonDown(1)))
272         {
273             currentCircuit =
                  hitObject.GetComponentInParent<CircuitReference>().Circuit;
274
275             // Left click (or both): circuit movement begins.
276             if (Input.GetMouseButtonDown(0))
277             {
278                 CircuitPress();
279                 stateType = StateType.LOCKED;
280                 CursorManager.SetMouseTexture(false);
281                 Cursor.visible = false;
282             }
283
284             // Right click: destroy current circuit.
285             else
286             {
287                 CircuitCaller.Destroy(currentCircuit);
288                 stateType = StateType.UNRESTRICTED;
289             }
290
291             return GameState.CIRCUIT_MOVEMENT;
292         }
293
294         // Mouse is on top of a circuit
```

```
295         if (hitObject.layer == 8) // 8 --> circuit base layer
296         {
297             stateType = StateType.UNRESTRICTED;
298             CursorManager.SetMouseTexture(false);
299             return GameState.CIRCUIT_HOVER;
300         }
301
302         // Mouse is on top of an input/output & LMB and/or RMB and/or MMB
              have been pressed
303         if (gameState == GameState.IO_HOVER && (Input.GetMouseButtonDown
              (0) || Input.GetMouseButtonDown(1) || Input.GetMouseButtonDown
              (2)))
304         {
305             ioLMB = Input.GetMouseButtonDown(0);
306             stateType = StateType.LOCKED;
307
308             // Left click (perahsp with other inputs, but LMB has the
                  highest preference): begins the connection process
309             if (ioLMB)
310             {
311                 IOLMBPress(hitObject);
312                 CursorManager.SetMouseTexture(true);
313             }
314
315             // Right click or middle mouse button: alternate press
316             // RMB: deletes all connections attached to the input/output
                  in question
317             // MMB (only applicable if hovered onto an input gate's
                  output): switch power states
318             else IOAlternatePress(hitObject);
319
320             return GameState.IO_PRESS;
321         }
322
323         // Mouse is on top of any input or output
324         if (hitObject.layer == 9 || hitObject.layer == 10)
325         {
326             stateType = StateType.UNRESTRICTED;
327             CursorManager.SetMouseTexture(false);
328             return GameState.IO_HOVER;
329         }
330
331         // Mouse is on top of a wire & RMB has been pressed
332         if (gameState == GameState.WIRE_HOVER && Input.GetMouseButtonDown
              (1))
333         {
334             stateType = StateType.LOCKED;
335             CursorManager.SetMouseTexture(true);
336             WirePress(hitObject); // Deletes the wire and its
```

```
                corresponding connection
337             return GameState.WIRE_PRESS;
338         }
339
340         // Mouse is on top of a wire
341         if (hitObject.layer == 11)
342         {
343             stateType = StateType.UNRESTRICTED;
344             CursorManager.SetMouseTexture(false);
345             return GameState.WIRE_HOVER;
346         }
347
348         // If none of the other conditions were met, default to the grid
              hover state instead.
349         stateType = StateType.UNRESTRICTED;
350         CursorManager.SetMouseTexture(true);
351         return GameState.GRID_HOVER;
352     }
353
354     /// <summary>
355     /// Begins the connection process.
356     /// </summary>
357     /// <param name="hitObject">The GameObject that was raycasted.</param>
358     private void IOLMBPress(GameObject hitObject)
359     {
360         Vector3 startingPos;
361
362         // Input layer was pressed; next press should be on an output
              layer
363         if (hitObject.layer == 9)
364         {
365             currentInput =
                  hitObject.GetComponent<CircuitVisualizer.InputReference>
                  ().Input;
366             ioLayerCheck = 10;
367             startingPos = currentInput.Transform.position;
368         }
369
370         // Output layer was pressed; next press should be on an input
              layer
371         else
372         {
373             currentOutput =
                  hitObject.GetComponent<CircuitVisualizer.OutputReference>
                  ().Output;
374             ioLayerCheck = 9;
375             startingPos = currentOutput.Transform.position;
376         }
377
```

```
378             CircuitConnector.Instance.BeginConnectionProcess(startingPos);
379         }
380
381         /// <summary>
382         /// Based on context, deletes all connections belonging to an input or
                output node or alternates the power status of an input gate.
383         /// </summary>
384         /// <param name="hitObject"></param>
385         private void IOAlternatePress(GameObject hitObject)
386         {
387             // If the raycasted object was an input and the MMB is pressed,
                  alternate its input
388             if (Input.GetMouseButtonDown(2))
389             {
390                 if (hitObject.layer == 10 &&
                      hitObject.GetComponentInParent<CircuitReference>
                      ().Circuit.GetType() == typeof(InputGate))
391                 {
392                     InputGate gate = (InputGate)
                          hitObject.GetComponentInParent<CircuitReference>
                          ().Circuit;
393
394                     gate.Powered = !gate.Powered;
395                     EditorStructureManager.Instance.DisplaySavePrompt =
                          true; // Important enough to trigger the save prompt
396                 }
397
398             }
399
400             // RMB on an input -- begin disconnection process
401             else if (hitObject.layer == 9)
402             {
403                 Circuit.Input input =
                      hitObject.GetComponent<CircuitVisualizer.InputReference>
                      ().Input;
404
405                 // If there is a connection, disconnect it.
406                 if (input.Connection != null) CircuitConnector.Disconnect
                      (input.Connection);
407             }
408
409             // RMB on an output -- begin disconnection process
410             else
411             {
412                 Circuit.Output output =
                      hitObject.GetComponent<CircuitVisualizer.OutputReference>
                      ().Output;
413                 List<CircuitConnector.Connection> connections = new
                      List<CircuitConnector.Connection>(output.Connections);
```

```
414
415             // Disconnects each connection associated with this output, if any.
416             foreach (CircuitConnector.Connection connection in connections) CircuitConnector.Disconnect(connection);
417         }
418
419         stateType = StateType.UNRESTRICTED;
420     }
421
422     /// <summary>
423     /// Called after a new circuit has been instantiated; sets initial values.
424     /// </summary>
425     /// <param name="currentCircuit">The circuit that has just been created.</param>
426     public void CircuitPlacement(Circuit currentCircuit)
427     {
428         // If there is a already a circuit in the process of being placed, destroy it.
429         if (this.currentCircuit != null) { CircuitCaller.Destroy(this.currentCircuit); }
430
431         this.currentCircuit = currentCircuit;
432         currentCircuit.PhysicalObject.transform.position = Coordinates.Instance.MousePos;
433     }
434
435     /// <summary>
436     /// Deletes a wire GameObject and its associated connection
437     /// </summary>
438     /// <param name="hitObject">The wire to delete.</param>
439     private void WirePress(GameObject hitObject)
440     {
441         CircuitConnector.Connection connection;
442
443         // Determines if the raycasted object is the parent mesh (aka not the starting/ending wire mesh).
444         if (hitObject.transform.parent == null)
445         {
446             connection = hitObject.GetComponent<CircuitConnector.Connection>();
447             Destroy(hitObject.transform.gameObject);
448         }
449
450         // Otherwise, is a starting or ending wire.
451         else
452         {
453             connection =
```

```
                hitObject.GetComponentInParent<CircuitConnector.Connection>
                ();
454             Destroy(hitObject.transform.parent.parent.gameObject);
455         }
456
457         CircuitConnector.Disconnect(connection); // Disconnects the logic
              associated with the connection
458         stateType = StateType.UNRESTRICTED;
459     }
460
461     /// <summary>
462     /// Called after a circuit has been pressed; sets initial values.
463     /// </summary>
464     private void CircuitPress()
465     {
466         Vector3 mousePos = Coordinates.Instance.MousePos;
467
468         startingPos = currentCircuit.PhysicalObject.transform.position;
469         endingOffset = startingOffset = mousePos;
470     }
471
472     /// <summary>
473     /// Cancels the connection process.
474     /// </summary>
475     public void CancelWirePlacement()
476     {
477         CircuitConnector.Instance.CancelConnectionProcess();
478         currentInput = null; currentOutput = null;
479     }
480
481     /// <summary>
482     /// Cancels the circuit movement process.
483     /// </summary>
484     public void CancelCircuitMovement()
485     {
486         Cursor.visible = true;
487         currentCircuit = null;
488     }
489
490     /// <summary>
491     /// Called after obtaining a new game state; listens to input/events
        corresponding to the game state.
492     /// </summary>
493     private void GameStateListener()
494     {
495         switch (gameState)
496         {
497             case GameState.GRID_HOVER:
498                 // Opens the bookmarked circuits menu.
```

```
499                 if (Input.GetMouseButtonDown(1) &&
                      TaskbarManager.Instance.CurrentMenu == null &&
                      TaskbarManager.Instance.ReopenBookmarks)
                      TaskbarManager.Instance.OpenBookmarks();
500
501                 return;
502             case GameState.IO_PRESS:
503                 if (!ioLMB) return; // The left mouse button was not
                      pressed, therefore the corresponding connection code
                      should be skipped.
504
505                 // Checks to see if the user is hovered on a valid
                      GameObject to complete the connection process.
506                 if (Physics.Raycast
                      (CameraMovement.Instance.PlayerCamera.ScreenPointToRay
                      (Input.mousePosition), out RaycastHit hitInfo) &&
                      hitInfo.transform.gameObject.layer == ioLayerCheck)
507                 {
508                     // Output layer was initially pressed, therefore this
                     is an input node
509                     if (ioLayerCheck == 9) currentInput =
                     hitInfo.transform.GetComponent<CircuitVisualizer.InputRe
                     ference>().Input;
510
511                     // Input layer was initially pressed, therefore this
                     is an output node
512                     else currentOutput =
                     hitInfo.transform.GetComponent<CircuitVisualizer.OutputR
                     eference>().Output;
513
514                     CursorManager.SetMouseTexture(false);
515
516                     // The user completes the connection process by
                     hovering on a valid input AND pressing the left mouse
                     button.
517                     if (Input.GetMouseButtonDown(0))
518                     {
519                         EditorStructureManager.Instance.DisplaySavePrompt
                     = true; // Important enough to trigger the save prompt
520
521                         // Disconnects the current connection to the
                     input, if there is one
522                         if (currentInput.ParentOutput != null)
                     CircuitConnector.Disconnect(currentInput.Connection);
523
524                         CircuitConnector.Connection connection =
                     CircuitConnector.Instance.CurrentConnection;
525
526                         CircuitConnector.Connect(currentInput,
```

```
                    currentOutput); // Ensures the connection is logically accounted for
527
528                     // If the order of selection was not output -> input, the starting and ending wires are swapped with one another.
529                     // This occurs because the starting wire is always associated with the input node, hence the GameObjects are swapped to maintain this rule.
530                     if (ioLayerCheck == 10)
531                     {
532                         GameObject temp = connection.StartingWire;
533
534                         // Swaps the starting and ending wires within the connection
535                         connection.StartingWire = connection.EndingWire;
536                         connection.EndingWire = temp;
537
538                         // Ensures the serialization process works as intended by keeping the hierarchy order of the wires the same, regardless of connection order.
539                         if (connection.StartingWire != connection.EndingWire)
540                         {
541                             connection.StartingWire.name = "Starting Wire";
542                             connection.EndingWire.name = "Ending Wire";
543 connection.StartingWire.transform.SetAsFirstSibling();
544                         }
545                     }
546
547                     stateType = StateType.UNRESTRICTED;
548                     currentInput = null; currentOutput = null;
549                     return;
550                 }
551             }
552
553             else CursorManager.SetMouseTexture(true);
554
555             // Cancels the connection process.
556             if (Input.GetKeyDown(cancelKey) || Input.GetMouseButtonDown(1))
557             {
558                 CancelWirePlacement();
559                 stateType = StateType.UNRESTRICTED;
560             }
```

```
561
562                 break;
563             case GameState.CIRCUIT_MOVEMENT:
564                 // Cancels the circuit movement process if the left mouse
                      button is not held.
565                 if (!Input.GetMouseButton(0))
566                 {
567                     CancelCircuitMovement();
568                     stateType = StateType.UNRESTRICTED;
569                     return;
570                 }
571
572                 // Calculates the delta mouse movement from the last frame
573                 endingOffset = Coordinates.Instance.MousePos;
574                 prevDeltaPos = deltaPos;
575                 deltaPos = endingOffset - startingOffset + startingPos;
576
577                 // Snaps the obtained position to the grid if grid
                      snapping is enabled.
578                 if (Coordinates.Instance.CurrentSnappingMode ==
                      Coordinates.SnappingMode.GRID) deltaPos =
                      Coordinates.NormalToGridPos(deltaPos);
579
580                 currentCircuit.PhysicalObject.transform.position =
                      deltaPos;
581
582                 if (prevDeltaPos != deltaPos) // Ensures the circuit has
                      moved from its previous position before updating the
                      transforms of both wire GameObjects.
583                 {
584                     EditorStructureManager.Instance.DisplaySavePrompt =
                        true; // Important enough to trigger the save prompt
585
586                     // Updates the position/scale each valid connection
                        associated with the inputs of the moved circuit.
587                     // This occurs so that each physical wire continues to
                        stretch/shrink and follow each circuit within the
                        scene.
588                     foreach (Circuit.Input input in currentCircuit.Inputs)
589                     {
590                         if (input.Connection != null)
591                         {
592                             bool isCentered = input.Connection.EndingWire
                        == input.Connection.StartingWire;
593                             Vector3 fromPos = isCentered ?
                        input.Connection.Output.Transform.position :
                        input.Connection.EndingWire.transform.position;
594
595                             CircuitConnector.UpdatePosition
```

```
                    (input.Connection.EndingWire, fromPos,
                    input.Transform.position, isCentered);
596                      }
597                  }
598
599                  // Updates the position/scale each valid connection
                    associated with the outputs of the moved circuit.
600                  // This occurs so that each physical wire continues to
                     stretch/shrink and follow each circuit within the
                    scene.
601                  foreach (Circuit.Output output in
                    currentCircuit.Outputs)
602                  {
603                      foreach (CircuitConnector.Connection connection in
                     output.Connections)
604                      {
605                          bool isCentered = connection.EndingWire ==
                    connection.StartingWire;
606                          Vector3 fromPos = isCentered ?
                    connection.Input.Transform.position :
                    connection.StartingWire.transform.position;
607
608                          CircuitConnector.UpdatePosition
                    (connection.StartingWire, fromPos,
                    output.Transform.position, isCentered);
609                      }
610                  }
611              }
612
613              break;
614          case GameState.CIRCUIT_PLACEMENT:
615              // Until its placement is confirmed, the circuit follows
                   the mouse cursor.
616              currentCircuit.PhysicalObject.transform.position =
                   Coordinates.Instance.ModePos;
617
618              // Placement is confirmed
619              if (Input.GetMouseButtonDown(0))
620              {
621                  Cursor.visible = true;
622                  EditorStructureManager.Instance.Circuits.Add
                    (currentCircuit); // Adds circuit for potential
                    serialization
623                  EditorStructureManager.Instance.DisplaySavePrompt =
                    true;
624                  currentCircuit = null;
625                  stateType = StateType.UNRESTRICTED;
626                  LateUpdate();
627                  return;
```

```
628                 }
629
630                 // Placement is cancelled; delete the circuit.
631                 if (Input.GetKeyDown(cancelKey) ||
                      Input.GetMouseButtonDown(1))
632                 {
633                     Cursor.visible = true;
634                     CircuitCaller.Destroy(currentCircuit);
635                     currentCircuit = null;
636                     stateType = StateType.UNRESTRICTED;
637                 }
638
639                 break;
640             }
641         }
642
643         // Getter and setter methods
644         public GameState UnpausedGameState { get { return unpausedGameState; }
              set { unpausedGameState = value; } }
645
646         public StateType UnpausedStateType { get { return unpausedStateType; }
              set { unpausedStateType = value; } }
647
648         // Getter methods
649         public static BehaviorManager Instance { get { return instance; } }
650
651         public bool LockUI { get { return lockUI; } set { lockUI = value; } }
652
653         public GameState CurrentGameState { get { return gameState; } }
654
655         public int IOLayerCheck { get { return ioLayerCheck; } }
656
657         public StateType CurrentStateType { get { return stateType; } }
658     }
```