```
using System;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.XR;
using static UnityEditor.PlayerSettings;

/// <summary>
/// BehaviorManager primarily handles the bulk of all dynamic and scripted non-UI events triggerable by the user.
/// </summary>
public class BehaviorManager : MonoBehaviour
{
    // Singleton state reference
    private static BehaviorManager instance;

    /// <summary>
    /// The current state that the editor scene is in.
    /// </summary>
    public enum GameState { GRID_HOVER, CIRCUIT_HOVER, CIRCUIT_MOVEMENT, CIRCUIT_PLACEMENT, IO_HOVER, IO_PRESS, USER_INTERFACE, WIRE_HOVER, WIRE_PRESS }

    /// <summary>
    /// Utilized alongside a <seealso cref="GameState"/> to determine the consequences of each game state.<br/><br/>
    /// <seealso cref="UNRESTRICTED"/>: nothing occurs.<br/>
    /// <seealso cref="LOCKED"/>: most/all other non-UI elements are locked; UI is still enabled.<br/>
    /// <seealso cref="PAUSED"/>: all non-UI elements are locked.
    /// </summary>
    public enum StateType { UNRESTRICTED, LOCKED, PAUSED }

    /// <summary>
    /// Cancels most non-UI and UI related events when pressed.<br/><br/>
    /// More often than not, an alternate option to cancel such events also occur with the right mouse button.
    /// </summary>
    [SerializeField]
    KeyCode cancelKey;

    /// <summary>
    /// Displays the name of any hovered inputs/outputs belonging to a custom circuit, if any.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI ioText;

```

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

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

```
127                 currentPreviewPin = null;
128             }
129
130             // Disables the currently hovered display pin, if any
131             // This specifically occurs once as ioText is externally used
                  and can have non-empty text.
132             if (doOnce && ioText.text != "") ioText.text = "";
133
134             doOnce = false;
135             return;
136         }
137
138         // Otherwise, checks for all relevant events by beginning to
              raycast.
139         doOnce = true;
140         ray = CameraMovement.Instance.PlayerCamera.ScreenPointToRay
              (Input.mousePosition);
141
142         // If nothing is raycasted, also return and disable set values.
143         if (!Physics.Raycast(ray, out RaycastHit hitInfo))
144         {
145             if (currentPreviewPin != null)
146             {
147                 currentPreviewPin.SetActive(false);
148                 currentPreviewPin = null;
149             }
150
151             if (ioText.text != "") ioText.text = "";
152
153             return;
154         }
155
156         GameObject hitObj = hitInfo.transform.gameObject;
157
158         // If hovered on an input or output belonging to a custom circuit,
               obtain its label.
159         if ((hitObj.layer == 9 || hitObj.layer == 10) &&
              hitObj.GetComponentInParent<CircuitReference>().Circuit.GetType
              () == typeof(CustomCircuit))
160         {
161             CustomCircuit customCircuit = (CustomCircuit)
                  hitObj.GetComponentInParent<CircuitReference>().Circuit;
162
163             int index;
164             string label;
165
166             // Is an input, therefore looks in relevant input variables.
167             if (hitObj.layer == 9)
168             {
```

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

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

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

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

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

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

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

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

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

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

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

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

```
628                     LateUpdate();
629                     return;
630                 }
631 
632                 // Placement is cancelled; delete the circuit.
633                 if (Input.GetKeyDown(cancelKey) ||
                      Input.GetMouseButtonDown(1))
634                 {
635                     Cursor.visible = true;
636                     CircuitCaller.Destroy(currentCircuit);
637                     currentCircuit = null;
638                     stateType = StateType.UNRESTRICTED;
639                 }
640 
641                 break;
642         }
643     }
644 
645     // Getter and setter methods
646     public GameState UnpausedGameState { get { return unpausedGameState; }
        set { unpausedGameState = value; } }
647 
648     public StateType UnpausedStateType { get { return unpausedStateType; }
        set { unpausedStateType = value; } }
649 
650     // Getter methods
651     public static BehaviorManager Instance { get { return instance; } }
652 
653     public bool LockUI { get { return lockUI; } set { lockUI = value; } }
654 
655     public GameState CurrentGameState { get { return gameState; } }
656 
657     public int IOLayerCheck { get { return ioLayerCheck; } }
658 
659     public StateType CurrentStateType { get { return stateType; } }
660 }
```