```
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Logical representation of an AND gate.
/// </summary>
public class AndGate : Circuit
{
    public AndGate() : this(Vector2.zero) { }

    public AndGate(Vector2 startingPos) : base("AND", 2, 1, startingPos) { }

    /// <summary>
    /// Returns an output to update if the output has changed due to alterations in input power statuses.
    /// </summary>
    /// <returns>The list of outputs that should have their connections called.</returns>
    protected override List<Output> UpdateOutputs()
    {
        bool outputStatus = Outputs[0].Powered;
        List<Output> outputs = new List<Output>();

        // AND gate representation
        Outputs[0].Powered = Inputs[0].Powered && Inputs[1].Powered;

        if (outputStatus != Outputs[0].Powered || MaterialNotMatching())
            outputs.Add(Outputs[0]);

        return outputs;
    }

    /// <summary>
    /// Checks all outputs to determine if the output node material is not matching its power status.<br/><br/>
    /// This is utilized within custom circuits to force update calls that would normally not occur due to the nature of UpdateOutputs().
    /// </summary>
    /// <returns>Whether any output material does not match its power status.</returns>
    private bool MaterialNotMatching()
    {
        if (Outputs[0].StatusRenderer == null) return false;

        return (Outputs[0].Powered && Outputs
            [0].StatusRenderer.sharedMaterial !=
            CircuitVisualizer.Instance.PowerOnMaterial) ||
                (!Outputs[0].Powered && Outputs
```

```
                    [0].StatusRenderer.sharedMaterial !=
                    CircuitVisualizer.Instance.PowerOffMaterial);
41      }
42  }
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical representation of a BUFFER gate.
6  /// </summary>
7  public class Buffer : Circuit
8  {
9      public Buffer() : this(Vector2.zero) { }
10
11     public Buffer(Vector2 startingPos) : base("BUFFER", 1, 1, startingPos)
         { }
12
13     /// <summary>
14     /// Returns an output to update if the output has changed due to
         alterations in input power statuses.
15     /// </summary>
16     /// <returns>The list of outputs that should have their connections
         called.</returns>
17     protected override List<Output> UpdateOutputs()
18     {
19         bool outputStatus = Outputs[0].Powered;
20         List<Output> outputs = new List<Output>();
21
22         // BUFFER gate representation
23         Outputs[0].Powered = Inputs[0].Powered;
24
25         if (outputStatus != Outputs[0].Powered || MaterialNotMatching())
             outputs.Add(Outputs[0]);
26
27         return outputs;
28     }
29
30     /// <summary>
31     /// Checks all outputs to determine if the output node material is not
         matching its power status.<br/><br/>
32     /// This is utilized within custom circuits to force update calls that
         would normally not occur due to the nature of UpdateOutputs().
33     /// </summary>
34     /// <returns>Whether any output material does not match its power
         status.</returns>
35     private bool MaterialNotMatching()
36     {
37         if (Outputs[0].StatusRenderer == null) return false;
38
39         return (Outputs[0].Powered && Outputs
             [0].StatusRenderer.sharedMaterial !=
             CircuitVisualizer.Instance.PowerOnMaterial) ||
40               (!Outputs[0].Powered && Outputs
```

```
                    [0].StatusRenderer.sharedMaterial !=
                    CircuitVisualizer.Instance.PowerOffMaterial);
41      }
42  }
```

```
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Logical representation of a DISPLAY.
/// </summary>
public class Display : Circuit
{
    /// <summary>
    /// Similar to <seealso cref="pins"/>, each value refers to the material that is rendered for a corresponding pin when the user's cursor hovers over its corresponding node.
    /// </summary>
    private GameObject[] previewPins = new GameObject[8];

    /// <summary>
    /// Similar to <seealso cref="previewPins"/>, each value refers to the material that is rendered for a corresponding pin when its corresponding node is powered.
    /// </summary>
    private MeshRenderer[] pins = new MeshRenderer[8];

    public Display() : this(Vector2.zero) { }

    public Display(Vector2 startingPos) : base("DISPLAY", 8, 0, startingPos) { }

    /// <summary>
    /// Updates each pin based on the power status of its corresponding node.
    /// </summary>
    /// <returns>An empty list of outputs.</returns>
    protected override List<Output> UpdateOutputs()
    {
        for (int i = 0; i < 8; i++) pins[i].material = Inputs[i].Powered ? CircuitVisualizer.Instance.PowerOnMaterial : CircuitVisualizer.Instance.PowerOffMaterial;

        // Always returns an empty list as a DISPLAY has no output nodes.
        return new List<Output>();
    }

    // Getter and setter method
    public GameObject[] PreviewPins { get { return previewPins; } set { previewPins = value; } }

    // Setter method
    public MeshRenderer[] Pins { set { pins = value; } }
}
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical representation of an INPUT gate.
6  /// </summary>
7  public class InputGate : Circuit
8  {
9      /// <summary>
10     /// Powered status unique to an INPUT gate.
11     /// </summary>
12     private bool powered;
13
14     public InputGate() : this(Vector2.zero) { }
15
16     public InputGate(Vector2 startingPos) : base("INPUT", 0, 1,
         startingPos) { }
17
18     /// <summary>
19     /// Returns an output to update if the output has changed due to
         alterations in input power statuses.
20     /// </summary>
21     /// <returns>The list of outputs that should have their connections
         called.</returns>
22     protected override List<Output> UpdateOutputs()
23     {
24         bool outputStatus = Outputs[0].Powered;
25         List<Output> outputs = new List<Output>();
26
27         // INPUT gate representation
28         Outputs[0].Powered = powered;
29
30         if (outputStatus != Outputs[0].Powered) outputs.Add(Outputs[0]);
31
32         return outputs;
33     }
34
35     /// <summary>
36     /// Getter and setter method; setting the powered value of the input
         node will also create an update call.
37     /// </summary>
38     public bool Powered { get { return powered; } set { powered = value;
         Update(); UpdateChildren(); } }
39 }
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical representation of an NAND (NOT AND) gate.<br/><br/>
6  /// The NAND gate is also a universal gate.
7  /// </summary>
8  public class NAndGate : Circuit
9  {
10     public NAndGate() : this(Vector2.zero) { }
11
12     public NAndGate(Vector2 startingPos) : base("NAND", 2, 1, startingPos)
         { }
13
14     /// <summary>
15     /// Returns an output to update if the output has changed due to
         alterations in input power statuses.
16     /// </summary>
17     /// <returns>The list of outputs that should have their connections
         called.</returns>
18     protected override List<Output> UpdateOutputs()
19     {
20         bool outputStatus = Outputs[0].Powered;
21         List<Output> outputs = new List<Output>();
22
23         // NAND gate representation
24         Outputs[0].Powered = !(Inputs[0].Powered && Inputs[1].Powered);
25
26         if (outputStatus != Outputs[0].Powered || MaterialNotMatching())
             outputs.Add(Outputs[0]);
27
28         return outputs;
29     }
30
31     /// <summary>
32     /// Checks all outputs to determine if the output node material is not
         matching its power status.<br/><br/>
33     /// This is utilized within custom circuits to force update calls that
         would normally not occur due to the nature of UpdateOutputs().
34     /// </summary>
35     /// <returns>Whether any output material does not match its power
         status.</returns>
36     private bool MaterialNotMatching()
37     {
38         if (Outputs[0].StatusRenderer == null) return false;
39
40         return (Outputs[0].Powered && Outputs
             [0].StatusRenderer.sharedMaterial !=
             CircuitVisualizer.Instance.PowerOnMaterial) ||
```

```
41                  (!Outputs[0].Powered && Outputs
                      [0].StatusRenderer.sharedMaterial !=
                      CircuitVisualizer.Instance.PowerOffMaterial);
42      }
43  }
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical representation of an NOR (NOT OR) gate.<br/><br/>
6  /// The NOR gate is also a universal gate.
7  /// </summary>
8  public class NOrGate : Circuit
9  {
10     public NOrGate() : this(Vector2.zero) { }
11
12     public NOrGate(Vector2 startingPos) : base("NOR", 2, 1, startingPos)
         { }
13
14     /// <summary>
15     /// Returns an output to update if the output has changed due to
         alterations in input power statuses.
16     /// </summary>
17     /// <returns>The list of outputs that should have their connections
         called.</returns>
18     protected override List<Output> UpdateOutputs()
19     {
20         bool outputStatus = Outputs[0].Powered;
21         List<Output> outputs = new List<Output>();
22
23         // NOR gate representation
24         Outputs[0].Powered = !(Inputs[0].Powered || Inputs[1].Powered);
25
26         if (outputStatus != Outputs[0].Powered || MaterialNotMatching())
             outputs.Add(Outputs[0]);
27
28         return outputs;
29     }
30
31     /// <summary>
32     /// Checks all outputs to determine if the output node material is not
         matching its power status.<br/><br/>
33     /// This is utilized within custom circuits to force update calls that
         would normally not occur due to the nature of UpdateOutputs().
34     /// </summary>
35     /// <returns>Whether any output material does not match its power
         status.</returns>
36     private bool MaterialNotMatching()
37     {
38         if (Outputs[0].StatusRenderer == null) return false;
39
40         return (Outputs[0].Powered && Outputs
             [0].StatusRenderer.sharedMaterial !=
             CircuitVisualizer.Instance.PowerOnMaterial) ||
```

```
41                  (!Outputs[0].Powered && Outputs
                      [0].StatusRenderer.sharedMaterial !=
                      CircuitVisualizer.Instance.PowerOffMaterial);
42      }
43  }
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical representation of a NOT gate.
6  /// </summary>
7  public class NotGate : Circuit
8  {
9      public NotGate() : this(Vector2.zero) { }
10
11     public NotGate(Vector2 startingPos) : base("NOT", 1, 1, startingPos)
         { }
12
13     /// <summary>
14     /// Returns an output to update if the output has changed due to
         alterations in input power statuses.
15     /// </summary>
16     /// <returns>The list of outputs that should have their connections
         called.</returns>
17     protected override List<Output> UpdateOutputs()
18     {
19         bool outputStatus = Outputs[0].Powered;
20         List<Output> outputs = new List<Output>();
21
22         // NOT gate representation
23         Outputs[0].Powered = !Inputs[0].Powered;
24
25         if (outputStatus != Outputs[0].Powered || MaterialNotMatching())
             outputs.Add(Outputs[0]);
26
27         return outputs;
28     }
29
30     /// <summary>
31     /// Checks all outputs to determine if the output node material is not
         matching its power status.<br/><br/>
32     /// This is utilized within custom circuits to force update calls that
         would normally not occur due to the nature of UpdateOutputs().
33     /// </summary>
34     /// <returns>Whether any output material does not match its power
         status.</returns>
35     private bool MaterialNotMatching()
36     {
37         if (Outputs[0].StatusRenderer == null) return false;
38
39         return (Outputs[0].Powered && Outputs
             [0].StatusRenderer.sharedMaterial !=
             CircuitVisualizer.Instance.PowerOnMaterial) ||
40                 (!Outputs[0].Powered && Outputs
```

```
                    [0].StatusRenderer.sharedMaterial !=
                    CircuitVisualizer.Instance.PowerOffMaterial);
41      }
42  }
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical representation of an OR gate.
6  /// </summary>
7  public class OrGate : Circuit
8  {
9      public OrGate() : this(Vector2.zero) { }
10
11     public OrGate(Vector2 startingPos) : base("OR", 2, 1, startingPos) { }
12
13     /// <summary>
14     /// Returns an output to update if the output has changed due to
         alterations in input power statuses.
15     /// </summary>
16     /// <returns>The list of outputs that should have their connections
         called.</returns>
17     protected override List<Output> UpdateOutputs()
18     {
19         bool outputStatus = Outputs[0].Powered;
20         List<Output> outputs = new List<Output>();
21
22         // OR gate representation
23         Outputs[0].Powered = Inputs[0].Powered || Inputs[1].Powered;
24
25         if (outputStatus != Outputs[0].Powered || MaterialNotMatching())
             outputs.Add(Outputs[0]);
26
27         return outputs;
28     }
29
30     /// <summary>
31     /// Checks all outputs to determine if the output node material is not
         matching its power status.<br/><br/>
32     /// This is utilized within custom circuits to force update calls that
         would normally not occur due to the nature of UpdateOutputs().
33     /// </summary>
34     /// <returns>Whether any output material does not match its power
         status.</returns>
35     private bool MaterialNotMatching()
36     {
37         if (Outputs[0].StatusRenderer == null) return false;
38
39         return (Outputs[0].Powered && Outputs
             [0].StatusRenderer.sharedMaterial !=
             CircuitVisualizer.Instance.PowerOnMaterial) ||
40               (!Outputs[0].Powered && Outputs
                   [0].StatusRenderer.sharedMaterial !=
```

```
                    CircuitVisualizer.Instance.PowerOffMaterial);
41      }
42  }
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical representation of an XOR (EXCLUSIVE OR) gate.
6  /// </summary>
7  public class XOrGate : Circuit
8  {
9      public XOrGate() : this(Vector2.zero) { }
10
11     public XOrGate(Vector2 startingPos) : base("XOR", 2, 1, startingPos)
         { }
12
13     /// <summary>
14     /// Returns an output to update if the output has changed due to
         alterations in input power statuses.
15     /// </summary>
16     /// <returns>The list of outputs that should have their connections
         called.</returns>
17     protected override List<Output> UpdateOutputs()
18     {
19         bool outputStatus = Outputs[0].Powered;
20         List<Output> outputs = new List<Output>();
21
22         // XOR gate representation
23         Outputs[0].Powered = Inputs[0].Powered && !Inputs[1].Powered || !
             Inputs[0].Powered && Inputs[1].Powered;
24
25         if (outputStatus != Outputs[0].Powered || MaterialNotMatching())
             outputs.Add(Outputs[0]);
26
27         return outputs;
28     }
29
30     /// <summary>
31     /// Checks all outputs to determine if the output node material is not
         matching its power status.<br/><br/>
32     /// This is utilized within custom circuits to force update calls that
         would normally not occur due to the nature of UpdateOutputs().
33     /// </summary>
34     /// <returns>Whether any output material does not match its power
         status.</returns>
35     private bool MaterialNotMatching()
36     {
37         if (Outputs[0].StatusRenderer == null) return false;
38
39         return (Outputs[0].Powered && Outputs
             [0].StatusRenderer.sharedMaterial !=
             CircuitVisualizer.Instance.PowerOnMaterial) ||
```

```
40                  (!Outputs[0].Powered && Outputs
                      [0].StatusRenderer.sharedMaterial !=
                      CircuitVisualizer.Instance.PowerOffMaterial);
41      }
42  }
```

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using UnityEngine;
4
5  /// <summary>
6  /// Circuit is the parent of every other concretized circuit, containing
     several predefined constructors, methods, and values.<br/><br/>
7  /// Each circuit must implement the abstract method <seealso
     cref="UpdateOutputs"/> with the logic to update its outputs.
8  /// </summary>
9  public abstract class Circuit
10 {
11     /// <summary>
12     /// The time it takes for an update call to occur. This value is
         measured in seconds.
13     /// </summary>
14     public readonly static float clockSpeed = 0.075f;
15
16     /// <summary>
17     /// The custom circuit associated with this input.<br/><br/>
18     /// This value will not be null if and only if the circuit is
         internally within a custom circuit.
19     /// </summary>
20     public CustomCircuit customCircuit;
21
22     /// <summary>
23     /// Whether the circuit should have a representative in-scene mesh
         generated by <see cref="CircuitVisualizer"/>.
24     /// </summary>
25     private bool visible;
26
27     /// <summary>
28     /// The physical mesh generated by <see cref="CircuitVisualizer"/> for
          this circuit.<br/><br/>
29     /// This value will be null if and only if the circuit is internally
         within a custom circuit, i.e. no mesh will be generated.
30     /// </summary>
31     private GameObject physicalObject;
32
33     /// <summary>
34     /// The list of input nodes belonging to the circuit.
35     /// </summary>
36     private Input[] inputs;
37
38     /// <summary>
39     /// The list of output nodes belonging to this circuit.
40     /// </summary>
41     private Output[] outputs;
42
```

```
43      /// <summary>
44      /// The list of outputs belonging to this circuit whose power stauses
          have changed after calling <seealso cref="UpdateOutputs"/>.<br/><br/
          >
45      /// Functionally, if an output does not have its power status change
          before and after an UpdateOutputs() call, it will short circuit and
          not call any circuits it is connected to.
46      /// </summary>
47      private List<Output> outputsToUpdate;
48
49      /// <summary>
50      /// The name of the circuit.
51      /// </summary>
52      private string circuitName;
53
54      /// <summary>
55      /// Input represents all required members of an input node that belong
            to a circuit.<br/><br/>
56      /// An input can only have one connection.
57      /// </summary>
58      public class Input
59      {
60          public Input(Circuit parentCircuit) { this.parentCircuit =
              parentCircuit; }
61
62          /// <summary>
63          /// Whether the input is powered.
64          /// </summary>
65          private bool powered;
66
67          /// <summary>
68          /// The circuit the input composes.
69          /// </summary>
70          private Circuit parentCircuit;
71
72          /// <summary>
73          /// The connection related to the input, if any.
74          /// </summary>
75          private CircuitConnector.Connection connection;
76
77          /// <summary>
78          /// Contains the material that visually displays whether the input
                is powered or not.
79          /// </summary>
80          private MeshRenderer statusRenderer;
81
82          /// <summary>
83          /// The output connecting to the input, if any.
84          /// </summary>
```

```
85          private Output parentOutput;
86
87          /// <summary>
88          /// Transform of the GameObject representing the input, if any.
89          /// </summary>
90          private Transform transform;
91
92          // Getter and setter methods
93          public bool Powered { get { return powered; } set { powered =
              value; } }
94
95          public Circuit ParentCircuit { get { return parentCircuit; } set
              { parentCircuit = value; } }
96
97          public CircuitConnector.Connection Connection { get { return
              connection; } set { connection = value;  } }
98
99          public MeshRenderer StatusRenderer { get { return
              statusRenderer; } set { statusRenderer = value; } }
100
101         public Output ParentOutput { get { return parentOutput; } set
              { parentOutput = value; } }
102
103         public Transform Transform { get { return transform; } set
              { transform = value; } }
104     }
105
106     /// <summary>
107     /// Output represents all required members of an output node that
          belong to a circuit.<br/><br/>
108     /// An output can more than one connection.
109     /// </summary>
110     public class Output
111     {
112         public Output(Circuit parentCircuit) { this.parentCircuit =
              parentCircuit; }
113
114         /// <summary>
115         /// Whether the output is powered.
116         /// </summary>
117         private bool powered;
118
119         /// <summary>
120         /// The circuit the output composes.
121         /// </summary>
122         private Circuit parentCircuit;
123
124         /// <summary>
125         /// The connections related to the output, if any.
```

```
126         /// </summary>
127         private List<CircuitConnector.Connection> connections = new
              List<CircuitConnector.Connection>();
128
129         /// <summary>
130         /// The inputs connecting to the output, if any.
131         /// </summary>
132         private List<Input> childInputs = new List<Input>();
133
134         /// <summary>
135         /// Contains the material that visually displays whether the
              output is powered or not.
136         /// </summary>
137         private MeshRenderer statusRenderer;
138
139         /// <summary>
140         /// Transform of the GameObject representing the output, if any.
141         /// </summary>
142         private Transform transform;
143
144         // Getter and setter methods
145         public bool Powered { get { return powered; } set { powered =
              value; } }
146
147         public Circuit ParentCircuit { get { return parentCircuit; } set
              { parentCircuit = value; } }
148
149         public List<CircuitConnector.Connection> Connections { get
              { return connections; } set { connections = value; } }
150
151         public List<Input> ChildInputs { get { return childInputs; } set
              { childInputs = value; } }
152
153         public MeshRenderer StatusRenderer { get { return
              statusRenderer; } set { statusRenderer = value; } }
154
155         public Transform Transform { get { return transform; } set
              { transform = value; } }
156     }
157
158     /// <summary>
159     /// UpdateCall represents an attempt to alter an input node from a
          given output node.<br/><br/>
160     /// An update call does not occur instantly, rather after <seealso
          cref="clockSpeed"/> seconds have passed.<br/>
161     /// This prevents any potential stack overflows caused by loops within
           circuits.
162     /// </summary>
163     public class UpdateCall
```

```
164     {
165         /// <summary>
166         /// Whether the input should be powered.
167         /// </summary>
168         private bool powered;
169 
170         /// <summary>
171         /// The input pertaining to this update call.
172         /// </summary>
173         private Input input;
174 
175         /// <summary>
176         ///  The output pertaining to this update call.
177         /// </summary>
178         private Output output;
179 
180         public UpdateCall(bool powered, Input input, Output output)
181         {
182             this.powered = powered;
183             this.input = input;
184             this.output = output;
185         }
186 
187         // Getter methods
188         public bool Powered { get { return powered; } }
189 
190         public Input Input { get { return input; } }
191 
192         public Output Output { get { return output; } }
193     }
194 
195     /// <summary>
196     /// Utilized by custom circuits to initialize a circuit with a variable number of input and output nodes.<br/><br/>
197     /// With this constructor, it is expected that <seealso cref="Inputs"/> and <seealso cref="Outputs"/> will be overriden within <see cref="CustomCircuit"/>.
198     /// </summary>
199     /// <param name="circuitName">Name of the circuit.</param>
200     /// <param name="startingPosition">Starting position of the circuit.</param>
201     public Circuit(string circuitName, Vector2 startingPosition) : this(circuitName, 0, 0, startingPosition, false) { }
202 
203     /// <summary>
204     /// Utilized by inherited circuits to determine the specific number of input and output nodes.
205     /// </summary>
206     /// <param name="circuitName">Name of the circuit.</param>
```

```
207     /// <param name="numInputs">Number of inputs associated with the circuit.</param>
208     /// <param name="numOutputs">Number of outputs associated with the circuit.</param>
209     /// <param name="startingPosition">Starting position of the circuit.</param>
210     public Circuit(string circuitName, int numInputs, int numOutputs, Vector2 startingPosition) : this(circuitName, numInputs, numOutputs, startingPosition, true) { }
211
212     /// <summary>
213     /// Primary constructor that all other constructors reference.
214     /// </summary>
215     /// <param name="circuitName">Name of the circuit.</param>
216     /// <param name="numInputs">Number of inputs associated with the circuit.</param>
217     /// <param name="numOutputs">Number of outputs associated with the circuit.</param>
218     /// <param name="startingPosition">Starting position of the circuit.</param>
219     /// <param name="createIO">Whether each input and output should be initialized.</param>
220     public Circuit(string circuitName, int numInputs, int numOutputs, Vector2 startingPosition, bool createIO)
221     {
222         this.circuitName = circuitName;
223
224         // Initializes inputs and outputs if specified
225         if (createIO)
226         {
227             inputs = new Input[numInputs];
228             outputs = new Output[numOutputs];
229
230             for (int i = 0; i < numInputs; i++) { inputs[i] = new Input(this); }
231
232             for (int i = 0; i < numOutputs; i++) { outputs[i] = new Output(this); }
233         }
234
235         /* Determines whether this circuit is meant to be visible.
236          * Within this project, Vector2.PositiveInfinity implicitly defines an invisible circuit.
237          * The only circuits that are invisible are ones that are part of custom circuits.
238          */
239         visible = startingPosition.x != float.PositiveInfinity &&
240           startingPosition.y != float.PositiveInfinity;
```

```
241         // Creates a corresponding mesh if the circuit is visible.
242         if (visible) CircuitVisualizer.Instance.VisualizeCircuit(this,
              startingPosition);
243     }
244
245     /// <summary>
246     /// Alternate signature of UpdateCircuit() that assumes the specified
          output is not null.
247     /// </summary>
248     /// <param name="input">The input to update.</param>
249     /// <param name="output">The output that caused the update.</param>
250     public static void UpdateCircuit(Input input, Output output)
          { UpdateCircuit(output.Powered, input, output); }
251
252     /// <summary>
253     /// Updates the circuit belonging to the specified input based on the
          given power status.<br/><br/>
254     /// Afterward, the circuit belonging to the specified input will
          update all circuits connected to its output(s).
255     /// </summary>
256     /// <param name="powered">Whether the specified input should be
          powered.</param>
257     /// <param name="input">The input to update.</param>
258     /// <param name="output">The output that caused the update.</param>
259     public static void UpdateCircuit(bool powered, Input input, Output
          output)
260     {
261         input.Powered = powered;
262
263         // Updates input power status material, if applicable
264         if (input.StatusRenderer != null) input.StatusRenderer.material =
              powered ? CircuitVisualizer.Instance.PowerOnMaterial :
              CircuitVisualizer.Instance.PowerOffMaterial;
265
266         // Updates the connection wire material associated with the input,
              if applicable
267         if (input.Connection != null)
              CircuitConnector.UpdateConnectionMaterial(input.Connection,
              powered);
268
269         input.ParentOutput = output;
270         input.ParentCircuit.Update();
271         input.ParentCircuit.UpdateChildren();
272     }
273
274     /// <summary>
275     /// Obtains the outputs that should be accessed by <seealso
          cref="UpdateChildren"/> as well as updating their <seealso
          cref="Output.statusRenderer"/> materials.
```

```
276     /// </summary>
277     public void Update()
278     {
279         // If all outputs should be checked, disregard any potential short circuiting optimization.
280         bool shouldCheckAllOutputs = customCircuit != null && customCircuit.finalOutputs.Count > 0;
281 
282         outputsToUpdate = UpdateOutputs();
283 
284         if (shouldCheckAllOutputs) { outputsToUpdate = Outputs.ToList(); }
285 
286         UpdateStatuses();
287     }
288 
289     /// <summary>
290     /// Calls and updates all connections associated to each valid output.<br/><br/>
291     /// This method can be called recursively, i.e. trigger a chain reaction.
292     /// </summary>
293     public void UpdateChildren()
294     {
295         List<UpdateCall> updateList = new List<UpdateCall>();
296 
297         foreach (Output output in outputsToUpdate)
298         {
299             if (customCircuit != null && customCircuit.finalOutputs.Contains(output)) customCircuit.finalOutputs.Remove(output);
300 
301             foreach (Input input in output.ChildInputs) updateList.Add(new UpdateCall(output.Powered, input, output));
302         }
303 
304         CircuitCaller.InitiateUpdateCalls(updateList);
305     }
306 
307     /// <summary>
308     /// Updates the materials of each valid output.
309     /// </summary>
310     private void UpdateStatuses()
311     {
312         foreach (Output output in outputsToUpdate)
313         {
314             if (output.StatusRenderer == null) continue;
315 
316             output.StatusRenderer.material = output.Powered ? CircuitVisualizer.Instance.PowerOnMaterial :
```

```
                CircuitVisualizer.Instance.PowerOffMaterial;
317         }
318     }
319
320     /// <summary>
321     /// Abstract implementation representing the input to output logic of a circuit.<br/>
322     /// Utilizes all inputs to recalculate the power status of each output.
323     /// </summary>
324     /// <returns>The list of outputs that have changed before and during this method.</returns>
325     protected abstract List<Output> UpdateOutputs();
326
327     // Getter and setter methods
328     public bool Visible { get { return visible; } set { visible = value; } }
329
330     public GameObject PhysicalObject { get { return physicalObject; } set { physicalObject = value; } }
331
332     public Input[] Inputs { get { return inputs; } set { inputs = value; } }
333
334     public Output[] Outputs { get { return outputs; } set { outputs = value; } }
335
336     public string CircuitName { get { return circuitName; } set { circuitName = value; } }
337 }
```

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  /// <summary>
5  /// Logical reprentation of a custom circuit consisting of a variable number/type of other circuits.
6  /// </summary>
7  public class CustomCircuit : Circuit
8  {
9      /// <summary>
10     /// The current custom circuit that is being rendered.<br/><br/>
11     ///
12     /// This value is utilized to differentiate between external and internal (part of a custom circuit) custom circuits.
13     /// </summary>
14     private static CustomCircuit currentCustomCircuit;
15
16     /// <summary>
17     /// Whether or not the custom circuit has been removed and therefore deferenced by its child circuits.
18     /// </summary>
19     private bool shouldDereference;
20
21     /// <summary>
22     /// The list of all internal circuits within the custom circuit.
23     /// </summary>
24     private List<Circuit> circuitList = new List<Circuit>();
25
26     /// <summary>
27     /// The parent GameObject under which all internal connections are attached.
28     /// </summary>
29     private GameObject connections;
30
31     /// <summary>
32     /// The list of all inputs within the custom circuit that have no connections.<br/><br/>
33     /// All empty inputs are rendered by <see cref="CircuitVisualizer"/>, meaning they can be externally connected to other circuits within a scene.
34     /// </summary>
35     private List<Input> emptyInputs = new List<Input>();
36
37     /// <summary>
38     /// The list of all inputs within the custom circuit.
39     /// </summary>
40     private List<Input> inputs = new List<Input>();
41
42     /// <summary>
```

```
43      /// The list of all outputs within the custom circuit that have no
          connections.<br/><br/>
44      /// All empty outputs are rendered by <see cref="CircuitVisualizer"/>,
           meaning they can be externally connected to other circuits within a
           scene.
45      /// </summary>
46      private List<Output> emptyOutputs = new List<Output>();
47
48      /// <summary>
49      /// The list of all empty outputs yet to have received an update
          call.<br/><br/>
50      /// This list is utilized to ensure that any placed custom circuit is
          properly updated by allowing for update call overrides that would
          otherwise not occur.
51      /// </summary>
52      public List<Output> finalOutputs;
53
54      /// <summary>
55      /// The list of all outputs within the custom circuit.
56      /// </summary>
57      private List<Output> outputs = new List<Output>();
58
59      /// <summary>
60      /// The preview structure the custom circuit is referring to.
61      /// </summary>
62      private PreviewStructure previewStructure;
63
64      /// <summary>
65      /// Alternate signature intended for creating custom circuits that is
          not inside a custom circuit, i.e. external.
66      /// </summary>
67      /// <param name="previewStructure"></param>
68      public CustomCircuit(PreviewStructure previewStructure) : this
          (previewStructure, Vector2.zero, true) {}
69
70      /// <summary>
71      /// Primary constructor for instantiating any custom circuit.
72      /// </summary>
73      /// <param name="previewStructure">The preview structure the custom
          circuit is referring to.</param>
74      /// <param name="startingPos">The in-scene position of the circuit
          (not applicable if the custom circuit is not visible).</param>
75      /// <param name="isFirst">Whether the custom circuit is external, in
          which case it will be visibly rendered.</param>
76      public CustomCircuit(PreviewStructure previewStructure, Vector2
          startingPos, bool isFirst) : base(previewStructure.Name,
          Vector2.positiveInfinity)
77      {
78          // If this custom circuit is external, it should be marked as the
```

```
            current custom circuit to be built as well as visible.
79          if (isFirst) { shouldDereference = false; currentCustomCircuit =
              this; Visible = true; }
80
81          CircuitName = previewStructure.Name;
82          this.previewStructure = previewStructure;
83          CreateCircuit(startingPos);
84      }
85
86      private void CreateCircuit(Vector2 startingPos)
87      {
88          connections = new GameObject("Connections [CUSTOM CIRCUIT]");
89
90          // Intantiates each internal circuit within the custom circuit
91          foreach (CircuitIdentifier circuitIdentifier in
              previewStructure.Circuits)
92          {
93              Circuit circuit = CircuitIdentifier.RestoreCircuit
                  (circuitIdentifier, false);
94
95              // All non-custom circuits are designated as the child of the
                  current custom circuit
96              if (circuit.GetType() != typeof(CustomCircuit))
                  circuit.customCircuit = this;
97
98              circuitList.Add(circuit);
99
100             foreach (Input input in circuit.Inputs) inputs.Add(input);
101
102             foreach (Output output in circuit.Outputs) outputs.Add
                  (output);
103         }
104
105         int inputAmount = previewStructure.InputLabels.Count;
106
107         // Restores all empty inputs as designated by the assigned preview
               structure.
108         for (int i = 0; i < inputAmount; i++) emptyInputs.Add(inputs
              [previewStructure.InputOrders.IndexOf(i)]);
109
110         int outputAmount = previewStructure.OutputLabels.Count;
111
112         // Restores all empty outputs as designated by the assigned
              preview structure.
113         for (int i = 0; i < outputAmount; i++) emptyOutputs.Add(outputs
              [previewStructure.OutputOrders.IndexOf(i)]);
114
115         // Sets the inputs and outputs as ONLY the empty inputs and
              outputs.
```

```
116         Inputs = emptyInputs.ToArray(); Outputs = emptyOutputs.ToArray();
117
118         int index = 0;
119
120         finalOutputs = new List<Output>(emptyOutputs);
121
122         // If the custom circuit is external/visible (synonymous with one another), render it into the scene.
123         if (Visible) CircuitVisualizer.Instance.VisualizeCustomCircuit (this, startingPos);
124
125         List<UpdateCall> updateCalls = new List<UpdateCall>();
126
127         // Within the custom circuit, reinstate every connection.
128         foreach (InternalConnection internalConnection in previewStructure.Connections)
129         {
130             CircuitConnector.Connection connection = connections.AddComponent<CircuitConnector.Connection>();
131             Input input = inputs[internalConnection.InputIndex];
132             Output output = outputs[internalConnection.OutputIndex];
133
134             // Sets all values of the current connection
135             connection.Input = input;
136             connection.Output = output;
137             input.Connection = connection;
138             input.ParentOutput = output;
139             output.Connections.Add(connection);
140             output.ChildInputs.Add(input);
141             updateCalls.Add(new UpdateCall(output.Powered, input, output));
142             index++;
143         }
144
145         // Begins to call each connection.
146         CircuitCaller.InitiateUpdateCalls(updateCalls);
147
148         // Begins the chain reaction to inevitably update the outputs.
149         UpdateOutputs();
150
151         /* Implies that the current custom circuit is a part of another custom circuit.
152          * As such, it points its custom circuit to the external custom circuit (parent).
153          * Furthermore, the GameObject holding its connection information becomes the child of the parent's connection GameObject.
154          */
155         if (!Visible)
156         {
```

```
157             customCircuit = currentCustomCircuit;
158             connections.transform.SetParent
                  (customCircuit.Connections.transform);
159         }
160
161         // Implies the current custom circuit IS the external custom
              curcuit (i.e. currentCustomCircuit == null --> parent custom
              circuit).
162         else currentCustomCircuit = null;
163     }
164
165     /// <summary>
166     /// Utilized after the instantiation of a custom circuit to update its
          logic to default status.<br/><br/>
167     /// Since a custom circuit does not store the exact predicate that
          controls the output, this method aims to bring about a chain
          reaction from the known inputs to eventually update the outputs in
          variable time.<br/><br/>
168     /// Furthermore, a custom circuit never has its UpdateOutputs() method
           accessed; as such, the return value is not necessary and thus
          yields null.
169     /// </summary>
170     protected override List<Output> UpdateOutputs()
171     {
172         foreach (Input input in emptyInputs) UpdateCircuit(false, input,
              null);
173
174         return null;
175     }
176
177     // Getter and setter method
178     public bool ShouldDereference { get { return shouldDereference; } set
          { shouldDereference = value; } }
179
180     // Getter methods
181     public GameObject Connections { get { return connections; } }
182
183     public PreviewStructure PreviewStructure { get { return
          previewStructure; } }
184 }
```

```
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     }
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// CameraMovement handles all player movement within the editor scene.<br/><br/>
6  /// Some behaviors (e.g. scrolling) will be enabled or disabled based on the state of several other scripts.
7  /// </summary>
8  public class CameraMovement : MonoBehaviour
9  {
10     // Singleton state reference
11     private static CameraMovement instance;
12
13     /// <summary>
14     /// The primary camera utilized by the player.
15     /// </summary>
16     [SerializeField]
17     Camera playerCamera;
18
19     /// <summary>
20     /// The speed under which the player can move around scenes.
21     /// </summary>
22     [SerializeField]
23     float movementSpeed;
24
25     /// <summary>
26     /// The speed under which the player can scroll around scenes.
27     /// </summary>
28     [SerializeField]
29     float scrollSpeed;
30
31     /// <summary>
32     /// How low and high the player can vertically go.
33     /// </summary>
34     [SerializeField]
35     float minHeight, maxHeight;
36
37     /// <summary>
38     /// Moves the player up and down respectively.
39     /// </summary>
40     [SerializeField]
41     KeyCode upKey, downKey;
42
43     /// <summary>
44     /// Keeps track of the mouse position in the current frame.
45     /// </summary>
46     private Vector3 mousePosCurrent;
47
```

```
48      // Enforces a singleton state pattern and initializes camera values.
49      private void Awake()
50      {
51          if (instance != null)
52          {
53              Destroy(this);
54              throw new Exception("CameraMovement instance already
                  established; terminating.");
55          }
56
57          instance = this;
58          ClampPos();
59      }
60
61      private void Start() { mousePosCurrent =
          Coordinates.Instance.MousePos; }
62
63      // Listens to key inputs and updates movements each frame.
64      private void Update()
65      {
66          float x, y, z;
67
68          Vector3 mousePosPrev = mousePosCurrent;
69
70          mousePosCurrent = Coordinates.Instance.MousePos;
71
72          // If the scene is paused or there is an override, no movement can
              occur.
73          if (BehaviorManager.Instance.CurrentStateType ==
              BehaviorManager.StateType.PAUSED && !
              IOAssigner.Instance.MovementOverride) return;
74
75          // Otherwise, some/all movement features are allowed depending on
              whether the game is unrestricted or locked.
76          // X-Z movement via mouse drag
77          if (Input.GetMouseButton(0) &&
              BehaviorManager.Instance.CurrentStateType ==
              BehaviorManager.StateType.UNRESTRICTED &&
              BehaviorManager.Instance.CurrentGameState !=
              BehaviorManager.GameState.CIRCUIT_HOVER)
78          {
79              Vector3 mousePosDelta = mousePosPrev - mousePosCurrent;
80
81              x = mousePosDelta.x;
82              z = mousePosDelta.z;
83          }
84
85          // X-Z movement via WASD
86          else
```

```
87          {
88              // Obtains x and z axis values based on input
89              x = Input.GetAxisRaw("Horizontal") * movementSpeed *
                  Time.deltaTime;
90              z = Input.GetAxisRaw("Vertical") * movementSpeed *
                  Time.deltaTime;
91          }
92
93          // Y movement via scroll wheel
94          if (Mathf.Abs(Input.mouseScrollDelta.y) > 0)
95          {
96              y = -Input.mouseScrollDelta.y * scrollSpeed * Time.deltaTime;
97          }
98
99          // Y movement via upKey and/or downKey
100         else
101         {
102             y = 0;
103
104             // Determines y axis values (holding both "upKey" and
                  "downKey" will negate one another)
105             if (Input.GetKey(upKey))
106             {
107                 y += movementSpeed * Time.deltaTime;
108             }
109
110             if (Input.GetKey(downKey))
111             {
112                 y -= movementSpeed * Time.deltaTime;
113             }
114         }
115
116         // Adds obtained values and updates position
117         transform.position += x * Vector3.right + y * -CameraRay.direction
              + z * Vector3.forward;
118         ClampPos();
119         mousePosCurrent = Coordinates.Instance.MousePos;
120     }
121
122     /// <summary>
123     /// Clamps values to ensure the user cannot traverse out of bounds.
124     /// </summary>
125     private void ClampPos()
126     {
127         Vector3 pos = transform.position;
128
129         pos.y = Mathf.Clamp(pos.y, minHeight, maxHeight);
130         transform.position = pos;
131     }
```

```
132
133     // Getter methods
134     public static CameraMovement Instance { get { return instance; } }
135
136     public Camera PlayerCamera { get { return playerCamera; } }
137
138     private Ray CameraRay { get { return playerCamera.ScreenPointToRay
          (Input.mousePosition); } }
139 }
```

```
1  using System;
2  using TMPro;
3  using UnityEngine;
4  using UnityEngine.UI;
5
6  /// <summary>
7  /// Coordinates keeps track of the world position as well as the grid
     snapping mode.
8  /// </summary>
9  public class Coordinates : MonoBehaviour
10 {
11     // Singleton state reference
12     private static Coordinates instance;
13
14     /// <summary>
15     /// Dictates how the current world position within the editor scene
         should be interpreted.<br/>
16     /// This modified position is utilized for several actions within the
         scene, such as placing wires and moving circuits.<br/><br/>
17     /// <seealso cref="GRID"/>: snap current mouse position to the visual
         grid.<br/>
18     /// <seealso cref="NONE"/>: keep the current mouse position as is.
19     /// </summary>
20     public enum SnappingMode { GRID, NONE }
21
22     /// <summary>
23     /// The transparency value of <seealso cref="gridStatus"/> when
         <seealso cref="SnappingMode.NONE"/> is enabled.
24     /// </summary>
25     [SerializeField] float gridTransparencyConstant;
26
27     /// <summary>
28     /// In-scene icon that visualizes the status of <seealso
         cref="snappingMode"/>.
29     /// </summary>
30     [SerializeField] Image gridStatus;
31
32     /// <summary>
33     /// Toggles the <seealso cref="SnappingMode"/> currently not in use.
34     /// </summary>
35     [SerializeField] KeyCode snapToggleKey;
36
37     /// <summary>
38     /// Displays the current world coordinates to the user.
39     /// </summary>
40     [SerializeField] TextMeshProUGUI coordinateText;
41
42     /// <summary>
43     /// Stores the inspector-assigned color of <seealso cref="gridStatus"/
```

```
        >.
    /// </summary>
    private Color gridStatusColor;

    /// <summary>
    /// Utilized to perform a raycast to calculate <seealso cref="mousePos"/>.
    /// </summary>
    private Plane raycastPlane;

    /// <summary>
    /// The current <seealso cref="SnappingMode"/>.
    /// </summary>
    private SnappingMode snappingMode;

    /// <summary>
    /// Stores the calculated mouse to world position.
    /// </summary>
    private Vector3 mousePos;

    private void Update()
    {
        // If the snap toggle key is pressed at a valid time, switch states.
        if (Input.GetKeyDown(snapToggleKey) && BehaviorManager.Instance.CurrentStateType != BehaviorManager.StateType.PAUSED)
        {
            snappingMode = snappingMode == SnappingMode.GRID ? SnappingMode.NONE : SnappingMode.GRID;
            CurrentSnappingMode = snappingMode; // Ensures the UI is also updated.
        }
    }

    private void Awake()
    {
        // Enforces a singleton state pattern
        if (instance != null)
        {
            Destroy(this);
            throw new Exception("Coordinates instance already established; terminating.");
        }

        instance = this;

        // Initializes private values
        raycastPlane = new Plane(Vector3.down,
```

```
            GridMaintenance.Instance.GridHeight);
85          gridStatusColor = gridStatus.color;
86      }
87
88      /// <summary>
89      /// Snaps the specified position to the grid.
90      /// </summary>
91      /// <param name="normalPos">The position that should be snapped to the
           grid.</param>
92      /// <returns>The grid position.</returns>
93      public static Vector3 NormalToGridPos(Vector3 normalPos) { return new
          Vector3((int)(normalPos.x + 0.5f * Mathf.Sign(normalPos.x)),
          GridMaintenance.Instance.GridHeight, (int)(normalPos.z + 0.5f *
          Mathf.Sign(normalPos.z))); }
94
95      // Getter methods
96      public static Coordinates Instance { get { return instance; } }
97
98      /// <summary>
99      /// Returns a new ray from the camera to the current mouse position.
100     /// </summary>
101     private Ray CameraRay { get { return
          CameraMovement.Instance.PlayerCamera.ScreenPointToRay
          (Input.mousePosition); } }
102
103     /// <summary>
104     /// Calculates and returns the current grid position.
105     /// </summary>
106     public Vector3 GridPos { get { return NormalToGridPos(mousePos); } }
107
108     /// <summary>
109     /// Calculates and returns the current mouse position.
110     /// </summary>
111     public Vector3 MousePos
112     {
113         get
114         {
115             Ray ray = CameraRay;
116
117             if (raycastPlane.Raycast(ray, out float distance))
118             {
119                 mousePos = ray.GetPoint(distance);
120
121                 // Updates the coordinates UI if the game is not currently
                       paused
122                 if (BehaviorManager.Instance.CurrentStateType !=
                      BehaviorManager.StateType.PAUSED) coordinateText.text =
                      "(" + mousePos.x.ToString("0.0") + ", " +
                      mousePos.z.ToString("0.0") + ")";
```

```
123
124                 return new Vector3(mousePos.x,
                      GridMaintenance.Instance.GridHeight, mousePos.z);
125             }
126
127             throw new Exception("Unable to obtain new mouse position --
                  raycast failed.");
128         }
129     }
130
131     /// <summary>
132     /// Returns a modified version of <seealso cref="mousePos"/> based on
          <seealso cref="snappingMode"/>.
133     /// </summary>
134     public Vector3 ModePos { get { return snappingMode ==
          SnappingMode.GRID ? GridPos : MousePos; } }
135
136     /// <summary>
137     /// Serves as a getter method as well as a setter method for both
          <seealso cref="snappingMode"/> and <seealso cref="gridStatus"/>.
138     /// </summary>
139     public SnappingMode CurrentSnappingMode { get { return snappingMode; }
140         set
141         {
142             snappingMode = value;
143
144             if (value == SnappingMode.GRID)
145             {
146                 gridStatus.color = gridStatusColor;
147             }
148
149             else
150             {
151                 Color temp = gridStatusColor;
152
153                 temp.a = gridTransparencyConstant;
154                 gridStatus.color = temp;
155             }
156         }
157     }
158 }
```

```cs
using System;
using UnityEngine;

/// <summary>
/// DisplayReference is assigned to and stores values unique to the Display prefab.
/// </summary>
public class DisplayReference : MonoBehaviour
{
    /// <summary>
    /// The inputs of the display.
    /// </summary>
    [SerializeField]
    GameObject[] inputs = new GameObject[8];

    /// <summary>
    /// The preview pins (on hover) corresponding to each value within <seealso cref="inputs"/>.
    /// </summary>
    [SerializeField]
    GameObject[] previewPins = new GameObject[8];

    /// <summary>
    /// The input statuses of the display.
    /// </summary>
    [SerializeField]
    MeshRenderer[] inputStatuses = new MeshRenderer[8];

    /// <summary>
    /// The pins (on power) corresponding to each value within <seealso cref="inputs"/>.
    /// </summary>
    [SerializeField]
    MeshRenderer[] pins = new MeshRenderer[8];

    // Ensures the sizes of each array cannot be modified within the inspector
    private void OnValidate()
    {
        if (inputs.Length != 8) Array.Resize(ref inputs, 8);

        if (inputStatuses.Length != 8) Array.Resize(ref inputStatuses, 8);

        if (pins.Length != 8) Array.Resize(ref pins, 8);

        if (previewPins.Length != 8) Array.Resize(ref previewPins, 8);
    }

    // Getter methods
```

```
46      public GameObject[] Inputs { get { return inputs; } }
47
48      public GameObject[] PreviewPins { get { return previewPins; } }
49
50      public MeshRenderer[] InputStatuses { get { return inputStatuses; } }
51
52      public MeshRenderer[] Pins { get { return pins; } }
53  }
```

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// EditorStructureManager accesses the current <see cref="EditorStructure"/> in use and directly manages its serialization/deserialization.
/// </summary>
public class EditorStructureManager : MonoBehaviour
{
    // Singleton state reference
    private static EditorStructureManager instance;

    /// <summary>
    /// List of instantiated circuits in the current editor scene.
    /// </summary>
    [HideInInspector]
    List<Circuit> circuits = new List<Circuit>();

    /// <summary>
    /// Index representation of all currently bookmarked circuits.
    /// </summary>
    [HideInInspector]
    List<int> bookmarks = new List<int>();

    /// <summary>
    /// List of instantiated connections in the current editor scene.
    /// </summary>
    [HideInInspector]
    List<CircuitConnector.Connection> connections = new List<CircuitConnector.Connection>();

    /// <summary>
    /// Whether a prompt to save the scene should appear or not when attempting to save.<br/><br/>
    /// False after a successful save and true if any important actions occur within a scene.
    /// </summary>
    private bool displaySavePrompt = false;

    // Enforces a singleton state pattern
    private void Awake()
    {
        if (instance != null)
        {
            Destroy(this);
            throw new Exception("EditorStructureManager instance already
```

```
                established; terminating.");
45          }
46
47          instance = this;
48      }
49
50      // Begins the deserialization process as the scene is loaded.
51      private void Start() { Deserialize(); }
52
53      /// <summary>
54      /// Begins the serialization process by calling its corresponding
          coroutine.
55      /// </summary>
56      public void Serialize()
57      {
58          displaySavePrompt = false;
59          StartCoroutine(SerializeCoroutine());
60      }
61
62      /// <summary>
63      /// Gathers all relevant information from the current scene and loads
          it into the current <see cref="EditorStructure"/> before saving it
          to the directory.
64      /// </summary>
65      public IEnumerator SerializeCoroutine()
66      {
67          // By skipping a frame, the UI loading screen is able to appear
              for more complex scenes.
68          yield return null;
69
70          // Obtains the current editor structure to override
71          int sceneIndex = MenuLogicManager.Instance.CurrentSceneIndex;
72          EditorStructure editorStructure =
              MenuSetupManager.Instance.EditorStructures[sceneIndex];
73
74          List<bool> isPoweredInput = new List<bool>();
75          List<CircuitIdentifier> circuitIdentifiers = new
              List<CircuitIdentifier>();
76
77          // Iterates through all circuits within the scene to assign all
              circuit identifiers and powered inputs.
78          foreach (Circuit circuit in circuits)
79          {
80              circuitIdentifiers.Add(new CircuitIdentifier(circuit));
81              isPoweredInput.Add(circuit.GetType() == typeof(InputGate) &&
                  ((InputGate)circuit).Powered);
82          }
83
84          // After obtaining all relevant values, override the editor
```

```
            structure
85          editorStructure.InGridMode =
              Coordinates.Instance.CurrentSnappingMode ==
              Coordinates.SnappingMode.GRID;
86          editorStructure.IsPoweredInput = isPoweredInput;
87          editorStructure.Circuits = circuitIdentifiers;
88          editorStructure.Bookmarks = bookmarks;
89          editorStructure.BookmarkIDs = TaskbarManager.Instance.BookmarkIDs;
90          editorStructure.CameraLocation =
              CameraMovement.Instance.PlayerCamera.transform.position;
91
92          // Saves the new editor structure to the directory and begins
              serializing all connections within the scene.
93          MenuSetupManager.Instance.UpdateEditorStructure(sceneIndex,
              editorStructure);
94          MenuSetupManager.Instance.GenerateConnections(true, sceneIndex,
              connections);
95
96          TaskbarManager.Instance.CloseMenu();
97      }
98
99      /// <summary>
100     /// Opens an editor scene based on a specified <see
          cref="EditorStructure"/>, restoring components based on its save
          files.
101     /// </summary>
102     public void Deserialize()
103     {
104         // Obtains the current editor structure to reference
105         int sceneIndex = MenuLogicManager.Instance.CurrentSceneIndex;
106         EditorStructure editorStructure =
              MenuSetupManager.Instance.EditorStructures[sceneIndex];
107
108         // If this is the first time the scene has been opened, set
              default values, save to directory, and return.
109         if (MenuLogicManager.Instance.FirstOpen)
110         {
111             bool inGridMode = Coordinates.Instance.CurrentSnappingMode ==
                  Coordinates.SnappingMode.GRID;
112             Vector3 cameraLocation =
                  CameraMovement.Instance.PlayerCamera.transform.position;
113
114             editorStructure.InGridMode = inGridMode;
115             editorStructure.CameraLocation = cameraLocation;
116             TaskbarManager.Instance.RestoreCustomCircuits(); // Ensures
                  all custom circuits still appear in the add menu
117             MenuSetupManager.Instance.UpdateEditorStructure(sceneIndex,
                  editorStructure); // Saves to directory
118             return;
```

```
119         }
120
121         Coordinates.Instance.CurrentSnappingMode =
              editorStructure.InGridMode ? Coordinates.SnappingMode.GRID :
              Coordinates.SnappingMode.NONE;
122         CameraMovement.Instance.PlayerCamera.transform.position =
              editorStructure.CameraLocation;
123
124         // Instantiates all circuits back to the scene
125         foreach (CircuitIdentifier circuitIdentifier in
              editorStructure.Circuits)
126         {
127             circuits.Add(CircuitIdentifier.RestoreCircuit
                (circuitIdentifier));
128         }
129
130         // Restores all connections back to the scene
131         MenuSetupManager.Instance.RestoreConnections(sceneIndex);
132
133         int index = 0;
134
135         // After all circuits have been established and connected, turn on
              any previously powered input gate.
136         foreach (bool isPoweredInput in editorStructure.IsPoweredInput)
137         {
138             if (isPoweredInput) ((InputGate)circuits[index]).Powered =
                  true;
139             index++;
140         }
141
142         // Lastly, all bookmarks and custom circuits are restored to their
              respective menus.
143         TaskbarManager.Instance.RestoreBookmarks
              (editorStructure.Bookmarks, editorStructure.BookmarkIDs);
144         TaskbarManager.Instance.RestoreCustomCircuits();
145     }
146
147     // Getter methods
148     public static EditorStructureManager Instance { get { return
          instance; } }
149
150     public bool DisplaySavePrompt { get { return displaySavePrompt; } set
          { displaySavePrompt = value; } }
151
152     public List<Circuit> Circuits { get { return circuits; } }
153
154     public List<CircuitConnector.Connection> Connections { get { return
          connections; } }
155
```

```
156     public List<int> Bookmarks { get { return bookmarks; } }
157 }
```

```
using System;
using System.Collections.Generic;
using TMPro;
using UnityEngine;

/// <summary>
/// IOAssigner is enabled in the editor scene after a custom circuit is validated to assign input/output orders and labels.
/// </summary>
public class IOAssigner : MonoBehaviour
{
    // Singleton state reference
    private static IOAssigner instance;

    /// <summary>
    /// The type of text <seealso cref="hoverText"/> should display.<br/><br/>
    /// <seealso cref="NONE"/>: prompts the user to hover on valid inputs/outputs.<br/>
    /// <seealso cref="INPUT"/>: currently hovered onto an input; display its potential order.<br/>
    /// <seealso cref="OUTPUT"/>: currently hovered onto an output; display its potential order.
    /// </summary>
    private enum TextMode { NONE, INPUT, OUTPUT }

    /// <summary>
    /// Exits the IOAssigner phase; the circuit must be validated again to reach this point.
    /// </summary>
    [SerializeField]
    KeyCode exitKey;

    /// <summary>
    /// The material applied to all empty inputs that are active and not being hovered on.
    /// </summary>
    [SerializeField]
    Material emptyInputMaterial;

    /// <summary>
    /// The material applied to all empty outputs that are active and not being hovered on.
    /// </summary>
    [SerializeField]
    Material emptyOutputMaterial;

    /// <summary>
    /// The material applied to an empty input/output that is currently
```

```
        hovered on.
42      /// </summary>
43      [SerializeField]
44      Material hoveredMaterial;
45
46      /// <summary>
47      /// Displays text determined by the current <seealso cref="TextMode"/
          >.
48      /// </summary>
49      [SerializeField]
50      TextMeshProUGUI hoverText;
51
52      /// <summary>
53      /// Whether the user has clicked on an empty input/output to bring
          about the label composition UI.
54      /// </summary>
55      private bool labelSelectionMode;
56
57      /// <summary>
58      /// Bypasses the movement system; prevents movement when composing a
          label.
59      /// </summary>
60      private bool movementOverride;
61
62      /// <summary>
63      /// The current selected input.
64      /// </summary>
65      private Circuit.Input currentInput;
66
67      /// <summary>
68      /// The list of empty inputs given to IOAssigner by <see
          cref="PreviewStructureManager"/>.<br/><br/>
69      /// These inputs are guaranteed to be valid, and the role of
          IOAssigner is to have the user label and order them to their liking.
70      /// </summary>
71      private List<Circuit.Input> emptyInputs;
72
73      /// <summary>
74      /// Contains all elements from <seealso cref="emptyInputs"/>, now
          reordered by the user.<br/><br/>
75      /// </summary>
76      private List<Circuit.Input> orderedInputs;
77
78      /// <summary>
79      /// The current selected output.
80      /// </summary>
81      private Circuit.Output currentOutput;
82
83      /// <summary>
```

```
84      /// The list of empty outputs given to IOAssigner by <see
          cref="PreviewStructureManager"/>.<br/><br/>
85      /// These outputs are guaranteed to be valid, and the role of
          IOAssigner is to have the user label and order them to their liking.
86      /// </summary>
87      private List<Circuit.Output> emptyOutputs;
88
89      /// <summary>
90      /// Contains all elements from <seealso cref="emptyOutputs"/>, now
          reordered by the user.<br/><br/>
91      /// </summary>
92      private List<Circuit.Output> orderedOutputs;
93
94      /// <summary>
95      /// The current valid GameObject hovered on by the user.<br/><br/>
96      /// This GameObject is guaranteed to either contain an <see
          cref="Circuit.Input"/> in <seealso cref="emptyInputs"/> or an <see
          cref="Circuit.Output"/> in <seealso cref="emptyOutputs"/>.
97      /// </summary>
98      private GameObject currentHover;
99
100     /// <summary>
101     /// Number of inputs and outputs have been successfully assigned and
          labeled by the user.
102     /// </summary>
103     private int inputCount, outputCount;
104
105     /// <summary>
106     /// The number of inputs and outputs that should be assigned for
          IOAssigner to complete its task.<br/><br/>
107     /// Both values initially are equal to their respective lengths of
          <seealso cref="emptyInputs"/> and <seealso cref="emptyOutputs"/>,
          but both lists have user-assigned elements removed.
108     /// </summary>
109     private int targetInputCount, targetOutputCount;
110
111     /// <summary>
112     /// Contain the respective labels for each user-assigned input and
          output.
113     /// </summary>
114     private List<string> inputLabels, outputLabels;
115
116     // Enforces a singleton state pattern and disables update calls.
117     private void Awake()
118     {
119         if (instance != null)
120         {
121             Destroy(this);
122             throw new Exception("IOAssigner instance already established;
```

```
                terminating.");
123         }
124
125         instance = this;
126         enabled = false; // After completing instance assignment, disable
              update calls until script is needed.
127     }
128
129     private void Update()
130     {
131         // Currently composing a label for an empty input or output.
132         if (labelSelectionMode)
133         {
134             // Exit conditions for quitting the labeling interface.
135             if (Input.GetKeyDown(exitKey) || Input.GetMouseButtonDown(1))
                  { CancelIOPress(); }
136
137             return;
138         }
139
140         // Exit conditions for quitting IOAssigner.
141         if (Input.GetKeyDown(exitKey) || Input.GetMouseButtonDown(1))
              { Exit(); return; }
142
143         Ray ray = CameraMovement.Instance.PlayerCamera.ScreenPointToRay
              (Input.mousePosition);
144
145         // Checks to see if the raycast hit ANY input or output
146         if (Physics.Raycast(ray, out RaycastHit hitInfo) &&
              (hitInfo.transform.gameObject.layer == 9 ||
              hitInfo.transform.gameObject.layer == 10))
147         {
148             Circuit.Input input = null; Circuit.Output output = null;
149             GameObject hitObject = hitInfo.transform.gameObject;
150
151             // Is an input
152             if (hitObject.layer == 9)
153             {
154                 input =
                      hitObject.GetComponent<CircuitVisualizer.InputReference>
                      ().Input;
155
156                 // If not within emptyInputs, not valid.
157                 if (!emptyInputs.Contains(input)) return;
158             }
159
160             // Is an output
161             else
162             {
```

```
163                 output =
                      hitObject.GetComponent<CircuitVisualizer.OutputReference
                      >().Output;
164
165                 // If not within emptyOutputs, not valid.
166                 if (!emptyOutputs.Contains(output)) return;
167             }
168
169             // Updates the interface and input/output material if the
                  current hit object is not the hovered object from the last
                  frame.
170             if (currentHover != hitObject)
171             {
172                 // If the last hit object was something, restore its
                      default material.
173                 if (currentHover != null)
                      currentHover.GetComponent<MeshRenderer>().material =
                      currentHover.layer == 9 ? emptyInputMaterial :
                      emptyOutputMaterial;
174
175                 // Update material and set text based on GameObject layer
                      (i.e. input or output)
176                 hitObject.GetComponent<MeshRenderer>().material =
                      hoveredMaterial;
177                 SetHoverText(hitObject.layer == 9 ? TextMode.INPUT :
                      TextMode.OUTPUT);
178
179                 // Store as current hover object
180                 currentHover = hitObject;
181             }
182
183             // Also begins the labeling process if LMB is pressed
184             if (Input.GetMouseButtonDown(0)) OnIOPress(input, output);
185         }
186
187         // Restores to default values if the raycast was unsuccessful.
188         else if (currentHover != null)
189         {
190             currentHover.GetComponent<MeshRenderer>().material =
                  currentHover.layer == 9 ? emptyInputMaterial :
                  emptyOutputMaterial;
191             currentHover = null;
192             SetHoverText(TextMode.NONE);
193         }
194     }
195
196     /// <summary>
197     /// Enables IOAssigner and starts the labeling/ordering process.
198     /// </summary>
```

```
199     /// <param name="emptyInputs">The empty inputs of the prospective
          custom circuit.</param>
200     /// <param name="emptyOutputs">The empty outputs of the prospective
          custom circuit.</param>
201     public void Initialize(List<Circuit.Input> emptyInputs,
          List<Circuit.Output> emptyOutputs)
202     {
203         labelSelectionMode = false; movementOverride = true;
204         inputCount = outputCount = 0;
205         targetInputCount = emptyInputs.Count; targetOutputCount =
              emptyOutputs.Count;
206         this.emptyInputs = emptyInputs; this.emptyOutputs = emptyOutputs;
207         orderedInputs = new List<Circuit.Input>(); orderedOutputs = new
              List<Circuit.Output>();
208         inputLabels = new List<string>(); outputLabels = new List<string>
              ();
209         hoverText.gameObject.SetActive(true);
210
211         // Switches the materials of all empty inputs and outputs to
              highlight them.
212         foreach (Circuit.Input input in emptyInputs)
              input.Transform.GetComponent<MeshRenderer>().material =
              emptyInputMaterial;
213
214         foreach (Circuit.Output output in emptyOutputs)
              output.Transform.GetComponent<MeshRenderer>().material =
              emptyOutputMaterial;
215
216         SetHoverText(TextMode.NONE);
217         Update();
218         enabled = true; // Enables frame-by-frame update calls from Unity.
219     }
220
221     /// <summary>
222     /// Disables IOAssigner and exists the labeling/ordering process.
223     /// </summary>
224     private void Exit()
225     {
226         movementOverride = false;
227
228         // If there are any empty inputs and/or outputs left, restore
              their default material.
229         foreach (Circuit.Input input in emptyInputs)
              input.Transform.GetComponent<MeshRenderer>().material =
              CircuitVisualizer.Instance.InputMaterial;
230
231         foreach (Circuit.Output output in emptyOutputs)
              output.Transform.GetComponent<MeshRenderer>().material =
              CircuitVisualizer.Instance.OutputMaterial;
```

```
232
233         hoverText.gameObject.SetActive(false);
234         enabled = false; // Disables frame-by-frame update calls from Unity.
235         TaskbarManager.Instance.CloseMenu();
236     }
237
238     /// <summary>
239     /// Cancels the label selection process for the current input or output.
240     /// </summary>
241     public void CancelIOPress()
242     {
243         movementOverride = true; labelSelectionMode = false;
244         TaskbarManager.Instance.CloseMenu();
245         TaskbarManager.Instance.NullState();
246     }
247
248     /// <summary>
249     /// Begins the label selection process for the current input or output.<br/><br/>
250     /// While there is both an input and output parameter, one of them will always be null.
251     /// </summary>
252     /// <param name="input">The input to label.</param>
253     /// <param name="output">The output to label.</param>
254     private void OnIOPress(Circuit.Input input, Circuit.Output output)
255     {
256         movementOverride = false; labelSelectionMode = true;
257         currentInput = input; currentOutput = output;
258         TaskbarManager.Instance.CloseMenu();
259         TaskbarManager.Instance.OpenLabelMenu(input != null);
260     }
261
262     /// <summary>
263     /// Successfully completes the label selection process for the current input or output.
264     /// </summary>
265     /// <param name="inputField">The text box to extract the label name from.</param>
266     public void ConfirmIOPress(TMP_InputField inputField)
267     {
268         string labelName = inputField.text;
269
270         // Implies the current labeling was done for an input
271         if (currentInput != null)
272         {
273             // Moves the current input to the ordered list and out of the empty list.
```

```
274             inputCount++;
275             emptyInputs.Remove(currentInput);
276             orderedInputs.Add(currentInput);
277             inputLabels.Add(labelName);
278         }
279 
280         // Implies the current labeling was done for an output
281         else
282         {
283             // Moves the current output to the ordered list and out of the empty list.
284             outputCount++;
285             emptyOutputs.Remove(currentOutput);
286             orderedOutputs.Add(currentOutput);
287             outputLabels.Add(labelName);
288         }
289 
290         inputField.text = "";
291         currentHover.GetComponent<MeshRenderer>().material = currentHover.layer == 9 ? CircuitVisualizer.Instance.InputMaterial : CircuitVisualizer.Instance.OutputMaterial;
292         currentHover = null;
293         SetHoverText(TextMode.NONE);
294         labelSelectionMode = false;
295         TaskbarManager.Instance.CloseMenu();
296         TaskbarManager.Instance.NullState();
297 
298         // All inputs/outputs have been labeled and/or ordered
299         if (inputCount == targetInputCount && outputCount == targetOutputCount)
300         {
301             hoverText.gameObject.SetActive(false);
302             enabled = false;
303             PreviewStructureManager.Instance.CreateCustomCircuit(orderedInputs, orderedOutputs, inputLabels, outputLabels); // Finally creates the custom circuit.
304         }
305 
306         else movementOverride = true;
307     }
308 
309     /// <summary>
310     /// Modifies the value of <seealso cref="hoverText"/> based on the assigned text mode.
311     /// </summary>
312     /// <param name="textMode">The current text mode.</param>
313     private void SetHoverText(TextMode textMode)
314     {
```

```
315         // Default text
316         string text = "hover over and select inputs/outputs to determine
              their order & label";
317
318         // Modifies the text if an input or output is implied.
319         switch (textMode)
320         {
321             case TextMode.INPUT:
322                 text = "input #" + (inputCount + 1);
323                 break;
324             case TextMode.OUTPUT:
325                 text = "output #" + (outputCount + 1);
326                 break;
327         }
328
329         hoverText.text = text;
330     }
331
332     // Getter methods
333     public static IOAssigner Instance { get { return instance; } }
334
335     public bool MovementOverride { get { return movementOverride; } }
336 }
```

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// PreviewStructureManager controls the primary logic involved with creating custom circuits within editor scenes.
/// </summary>
public class PreviewStructureManager : MonoBehaviour
{
    // Singleton state reference
    private static PreviewStructureManager instance;

    /// <summary>
    /// Denotes whether each internal circuit within the custom circuit has been reached.<br/><br/>
    /// Functionally, this list is used to run the depth-first search (DFS) algorithm to determine whether all circuits in an editor scene are connected.
    /// </summary>
    private bool[] reachedCircuits;

    /// <summary>
    /// List of inputs with no connections and all inputs respectively.
    /// </summary>
    private List<Circuit.Input> emptyInputs,
        inputs;

    /// <summary>
    /// List of outputs with no connections and all outputs respectively.
    /// </summary>
    private List<Circuit.Output> emptyOutputs,
        outputs;

    /// <summary>
    /// The number of circuits that have been reached.<br/><br/>
    /// Functionally, circuitCount is utilized alongside <seealso cref="reachedCircuits"/> to determine whether all circuits in an editor scene are connected.
    /// </summary>
    private int circuitCount;

    /// <summary>
    /// The prospective name for the current custom circuit.<br/><br/>
    /// If all validation tests succeed, it will be utilized as the name of the custom circuit.
    /// </summary>
    private string currentName;
```

```
43
44      // Enforces a singleton state pattern
45      private void Awake()
46      {
47          if (instance != null)
48          {
49              Destroy(this);
50              throw new Exception("PreviewStructureManager instance already
                  established; terminating.");
51          }
52
53          instance = this;
54      }
55
56      /// <summary>
57      /// Calls the coroutine that begins the circuit creation process,
            namely its validation tests.
58      /// </summary>
59      /// <param name="name">The prospective name of the custom circuit to
            use.</param>
60      public void VerifyPreviewStructure(string name) { StartCoroutine
            (VerifyPreviewStructureCoroutine(name)); }
61
62      /// <summary>
63      /// Performs a series of tests to verify the validity of a prospective
             custom circuit based on the current editor scene.
64      /// </summary>
65      /// <param name="name">The prospective name of the custom circuit to
            use.</param>
66      private IEnumerator VerifyPreviewStructureCoroutine(string name)
67      {
68          // Skipping a frame ensures the UI dialog for verifying a custom
              circuit will show.
69          yield return null;
70
71          // Validation test #1: non-empty name
72          if (name == "")
73          {
74              TaskbarManager.Instance.CircuitSaveError("The custom circuit
                  must not have an empty name.");
75              yield break;
76          }
77
78          // Validation test #2: unique name
79          foreach (PreviewStructure previewStructure in
              MenuSetupManager.Instance.PreviewStructures)
80          {
81              if (previewStructure.Name == name)
82              {
```

```
83                  TaskbarManager.Instance.CircuitSaveError("The custom
                      circuit must have a unique name.");
84                  yield break;
85              }
86          }
87
88          // Validation test #3: >= 1 circuits
89          if (EditorStructureManager.Instance.Circuits.Count == 0)
90          {
91              TaskbarManager.Instance.CircuitSaveError("The custom circuit
                  must consist of (1) or more circuits.");
92              yield break;
93          }
94
95          // Validation test #4: no input/display gates
96          foreach (Circuit circuit in
              EditorStructureManager.Instance.Circuits)
97          {
98              Type type = circuit.GetType();
99
100             if (type == typeof(InputGate) || type == typeof(Display))
101             {
102                 TaskbarManager.Instance.CircuitSaveError("The custom
                      circuit must not consist of any input gates or
                      displays.");
103                 yield break;
104             }
105         }
106
107         // Validation test #5: all circuits are connected
108         reachedCircuits = new bool
              [EditorStructureManager.Instance.Circuits.Count];
109         emptyInputs = new List<Circuit.Input>(); inputs = new
              List<Circuit.Input>();
110         emptyOutputs = new List<Circuit.Output>(); outputs = new
              List<Circuit.Output>();
111         circuitCount = 0;
112         CircuitConnectionTest(EditorStructureManager.Instance.Circuits
              [0]); // Begins the DFS algorithm
113
114         if (circuitCount != reachedCircuits.Length)
115         {
116             TaskbarManager.Instance.CircuitSaveError("The custom circuit
                  must be entirely connected.");
117             yield break;
118         }
119
120         // Validation test #6: >= 1 empty outputs
121         if (emptyOutputs.Count == 0)
```

```
122         {
123             TaskbarManager.Instance.CircuitSaveError("The custom circuit must have (1) or more empty outputs.");
124             yield break;
125         }
126 
127         /// All validation tests completed ///
128 
129         currentName = name;
130         TaskbarManager.Instance.CloseMenu();
131         TaskbarManager.Instance.NullState();
132 
133         // Begins the process in which the user assigns the order and labels of all empty inputs and outputs.
134         IOAssigner.Instance.Initialize(emptyInputs, emptyOutputs);
135     }
136 
137     /// <summary>
138     /// Starts the coroutine involved in finally creating a custom circuit.<br/><br/>
139     /// This method is specifically called by <see cref="IOAssigner"/> after all empty inputs and outputs have been ordered by the user (as well as any respective labling).
140     /// </summary>
141     /// <param name="orderedInputs"></param>
142     /// <param name="orderedOutputs"></param>
143     /// <param name="inputLabels"></param>
144     /// <param name="outputLabels"></param>
145     public void CreateCustomCircuit(List<Circuit.Input> orderedInputs, List<Circuit.Output> orderedOutputs, List<string> inputLabels, List<string> outputLabels)
146     {
147         StartCoroutine(CreatePreviewStructure(orderedInputs, orderedOutputs, inputLabels, outputLabels));
148     }
149 
150     /// <summary>
151     /// Serializes a custom circuit as well as its corresponding preview structure.
152     /// </summary>
153     /// <param name="orderedInputs">The list of empty inputs, ordered.</param>
154     /// <param name="orderedOutputs">The list of empty outputs, ordered.</param>
155     /// <param name="inputLabels">Labels associated with each ordered input.</param>
156     /// <param name="outputLabels">Labels associated with each ordered output.</param>
157     private IEnumerator CreatePreviewStructure(List<Circuit.Input>
```

```
          orderedInputs, List<Circuit.Output> orderedOutputs, List<string>
          inputLabels, List<string> outputLabels)
158     {
159         TaskbarManager.Instance.OnSuccessfulPreviewVerification();
160
161         // Skipping a frame ensures the UI dialog for creating a custom
              circuit will show.
162         yield return null;
163
164         List<CircuitIdentifier> circuitIdentifiers = new
              List<CircuitIdentifier>();
165         List<int> inputOrders = new List<int>(), outputOrders = new
              List<int>();
166         PreviewStructure previewStructure = new PreviewStructure
              (currentName);
167
168         // Serializes each circuit by instanting CircuitIdentifier
              references.
169         foreach (Circuit circuit in
              EditorStructureManager.Instance.Circuits)
170         {
171             circuitIdentifiers.Add(new CircuitIdentifier(circuit));
172
173             foreach (Circuit.Input input in circuit.Inputs) { inputs.Add
                (input); inputOrders.Add(orderedInputs.IndexOf(input)); }
174
175             foreach (Circuit.Output output in circuit.Outputs)
                { outputs.Add(output); outputOrders.Add
                (orderedOutputs.IndexOf(output)); }
176         }
177
178         previewStructure.Circuits = circuitIdentifiers;
179         previewStructure.ID = UniqueID; // Assigns a unique ID to the
              preview structure.
180         previewStructure.InputOrders = inputOrders;
181         previewStructure.OutputOrders = outputOrders;
182         previewStructure.InputLabels = inputLabels;
183         previewStructure.OutputLabels = outputLabels;
184         previewStructure.CameraLocation =
              CameraMovement.Instance.PlayerCamera.transform.position;
185
186         List<InternalConnection> internalConnections = new
              List<InternalConnection>();
187
188         // Serializes each connection by assigning index values to each
              input/output pair within an InternalConnection instance.
189         foreach (CircuitConnector.Connection connection in
              EditorStructureManager.Instance.Connections)
190         {
```

```
191             internalConnections.Add(new InternalConnection(
192                 inputs.IndexOf(connection.Input),
193                 outputs.IndexOf(connection.Output)
194                 ));
195         }
196 
197         previewStructure.Connections = internalConnections;
198 
199         // Adds preview structure and its connections to the save directory and add menu.
200         MenuSetupManager.Instance.PreviewStructures.Add(previewStructure);
201         MenuSetupManager.Instance.GenerateConnections(false, previewStructure.ID, EditorStructureManager.Instance.Connections);
202         MenuSetupManager.Instance.UpdatePreviewStructure (previewStructure);
203         TaskbarManager.Instance.AddCustomCircuitPanel(previewStructure.ID, false);
204         TaskbarManager.Instance.OnSuccessfulPreviewStructure();
205     }
206 
207     /// <summary>
208     /// Performs a depth-first search starting at the first placed circuit to determine whether the scene represents a complete graph. <br/>
209     /// At the same time, any circuit input or output without a connection is stored for the next test.
210     /// </summary>
211     private void CircuitConnectionTest(Circuit currentCircuit)
212     {
213         while (currentCircuit.customCircuit != null)
214         {
215             currentCircuit = currentCircuit.customCircuit;
216         }
217 
218         int index = EditorStructureManager.Instance.Circuits.IndexOf (currentCircuit);
219 
220         if (reachedCircuits[index]) return;
221 
222         reachedCircuits[index] = true;
223         circuitCount++;
224 
225         foreach (Circuit.Input input in currentCircuit.Inputs)
226         {
227             if (input.ParentOutput == null) { emptyInputs.Add(input); continue; }
228 
229             CircuitConnectionTest(input.ParentOutput.ParentCircuit);
230         }
```

```
231
232         foreach (Circuit.Output output in currentCircuit.Outputs)
233         {
234             if (output.ChildInputs.Count == 0) { emptyOutputs.Add(output);
                continue; }
235
236             foreach (Circuit.Input input in output.ChildInputs)
237             {
238                 CircuitConnectionTest(input.ParentCircuit);
239             }
240         }
241     }
242
243     /// <summary>
244     /// Returns a new unique ID for a new preview structure.<br/><br/>
245     /// A unique ID starts from 0 and increments onward.
246     /// </summary>
247     private int UniqueID
248     {
249         get
250         {
251             int currentID = 0;
252
253             // Keeps incrementing the current ID until it is unique
254             // This system ensures that if an ID that is not the largest
                is removed, it will be recycled in future custom circuit
                creations.
255             while (true)
256             {
257                 if (!
                    MenuSetupManager.Instance.PreviewStructureIDs.Contains
                    (currentID))
258                 {
259                     MenuSetupManager.Instance.PreviewStructureIDs.Add
                    (currentID);
260                     return currentID;
261                 }
262
263                 currentID++;
264             }
265         }
266     }
267
268     // Getter method
269     public static PreviewStructureManager Instance { get { return
        instance; } }
270 }
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// BackgroundParallax captures any mouse movement and proportionally moves
       an assigned background to emulate a parallax effect.
6  /// </summary>
7  public class BackgroundParallax : MonoBehaviour
8  {
9      // Singleton state reference
10     private static BackgroundParallax instance;
11
12     /// <summary>
13     /// Controls how much the captured mouse movement alters the
         background.
14     /// </summary>
15     [SerializeField]
16     float parallaxStrength;
17
18     /// <summary>
19     /// The background that the parallax is applied on.
20     /// </summary>
21     [SerializeField]
22     RectTransform backgroundTransform;
23
24     /// <summary>
25     /// Stores prior mouse positions to calculate the delta movement
         between frames.
26     /// </summary>
27     private Vector2 mousePos;
28
29     // Enforces a singleton state pattern
30     private void Awake()
31     {
32         if (instance != null)
33         {
34             Destroy(this);
35             throw new Exception("BackgroundParallax instance already
                 established; terminating.");
36         }
37
38         instance = this;
39     }
40
41     private void Start() { mousePos = Input.mousePosition; }
42
43     private void Update()
44     {
45         // Captures the difference in mouse position between frames,
```

```
            proportionally moving the background.
        Vector2 prevMousePos = mousePos, mouseDelta;

        mousePos = Input.mousePosition;
        mouseDelta = (prevMousePos - mousePos) * parallaxStrength;
        backgroundTransform.offsetMin += mouseDelta;
    }

    // Getter method
    public static BackgroundParallax Instance { get { return instance; } }
}
```

```
1  using UnityEngine;
2  using UnityEngine.EventSystems;
3
4  /// <summary>
5  /// ButtonRightClick is assigned to the save slot buttons within the menu,
     enabling for the deletion of a scene.
6  /// </summary>
7  public class ButtonRightClick : MonoBehaviour, IPointerClickHandler
8  {
9      /// <summary>
10     /// Stores which editor scene this instance is assigned to.
11     /// </summary>
12     [SerializeField] int saveIndex;
13
14     /// <summary>
15     /// Begins the scene deletion process if the captured input was a right
          click.
16     /// </summary>
17     public void OnPointerClick(PointerEventData eventData)
18     {
19         if (eventData.button == PointerEventData.InputButton.Right)
             MenuInterfaceManager.Instance.BeginSceneDeletion(saveIndex);
20     }
21 }
```

```
1  using UnityEngine;
2  using UnityEngine.UI;
3
4  /// <summary>
5  /// CustomCircuitButtons is utilized to add delegate listeners to its
     specified custom circuit within the menu for viewing and deleting.
6  /// </summary>
7  public class CustomCircuitButtons : MonoBehaviour
8  {
9      /// <summary>
10     /// The in-scene buttons pertaining to deletion and viewing a custom
         circuit.
11     /// </summary>
12     [SerializeField]
13     Button deleteButton, viewButton;
14
15     // Getter methods
16     public Button DeleteButton { get { return deleteButton; } }
17
18     public Button ViewButton { get { return viewButton; } }
19 }
```

```
using System;
using System.Collections.Generic;
using TMPro;
using UnityEngine;

/// <summary>
/// MenuInterfaceManager handles all UI interactions and transitions within the menu scene.
/// </summary>
public class MenuInterfaceManager : MonoBehaviour
{
    // Singleton state reference
    private static MenuInterfaceManager instance;

    /// <summary>
    /// The colors of save slots when uncreated and created. respectively.
    /// </summary>
    [SerializeField]
    Color defaultColor,
        saveColor;

    /// <summary>
    /// How much the height of <seealso cref="deleteErrorTransform"/> should increase for each additional error message.
    /// </summary>
    [SerializeField] float deleteErrorMessageSize;

    /// <summary>
    /// Attempts to exit the <seealso cref="currentInterface"/> in use.<br/><br/>
    /// For some interfaces, this will not instantly occur (such as going back to the options menu from the guide).
    /// </summary>
    [SerializeField] KeyCode cancelKey;

    /// <summary>
    /// Utilized to add and reference custom circuits from the directory.
    /// </summary>
    [SerializeField]
    GameObject customCircuitPanel,
        customCircuitPrefab;

    /// <summary>
    /// Set to visible when an interface is opened.<br/><br/>
    /// Within the scene, this should be a semi-transparent background overlayed onto everything except the currently opened interface.
    /// </summary>
    [SerializeField]
    GameObject transparentBackground,
```

```
45          transparentBackgroundUpper; // Utilized for multi-level interfaces
               such as the options interface.
46
47      [Space(10)]
48      /// <summary>
49      /// List of all interfaces within the scene.
50      /// </summary>
51      [SerializeField]
52      GameObject customCircuitsInterface, // Displays all currently open
          custom circuits and allows the user to view and/or delete them;
          opened from the options menu.
53          deleteErrorInterface, // The error log(s) displayed when a custom
              circuit cannot be deleted for any reason(s).
54          guideInterface, // Displays the guide prefab; opened from the
              options menu.
55          optionSelectionInterface, // Allows the user to activate
              customCircuitsInterface as well as guideInterface.
56          optionsInterface, // The parent of optionSelectionInterface
              indicating that the options menu is open.
57          sceneDeletionInterface, // Prompts the user to confirm deleting an
               editor structure (save slot).
58          sceneNameInterface; // Prompts the user to compose a name for an
              editor structure (save slot).
59
60      [Space(10)]
61      /// <summary>
62      /// Utilized to expand the vertical height of the error interface by
          factors of <seealso cref="deleteErrorMessageSize"/>.
63      /// </summary>
64      [SerializeField]
65      RectTransform deleteErrorTransform;
66
67      /// <summary>
68      /// The text components for the delete and scene creation interfaces
          utilized to display the cause(s) of error.
69      /// </summary>
70      [SerializeField]
71      TextMeshProUGUI deleteErrorText,
72      sceneNameError;
73
74      /// <summary>
75      /// The input field within <seealso cref="sceneNameInterface"/> that
          the user uses to compose a scene name.
76      /// </summary>
77      [SerializeField]
78      TMP_InputField sceneNameInputField;
79
80      /// <summary>
81      /// Text components of all 3 save slots within the menu used for
```

```
          setting and deleting their names.
 82       /// </summary>
 83       [SerializeField]
 84       TextMeshProUGUI save1,
 85           save2,
 86           save3;
 87
 88       /// <summary>
 89       /// The current UI interface in use.
 90       /// </summary>
 91       private GameObject currentInterface;
 92
 93       /// <summary>
 94       /// Index of the current <see cref="EditorStructure"/> loaded into the
             scene.
 95       /// </summary>
 96       private int currentSceneIndex = -1;
 97
 98       // Enforces a singleton state pattern
 99       private void Awake()
100       {
101           if (instance != null)
102           {
103               Destroy(this);
104               throw new Exception("MenuInterfaceManager instance already
                      established; terminating.");
105           }
106
107           instance = this;
108       }
109
110       // Loads in all serialized editor scenes and preview structures.
111       private void Start()
112       {
113           UpdateInterface();
114           AddCustomBookmarks();
115           CursorManager.SetMouseTexture(true);
116           enabled = false;
117       }
118
119       private void Update()
120       {
121           // Default exit controls for all interfaces except the options
                  interface.
122           if (currentInterface != optionsInterface)
123           {
124               if (Input.GetKeyDown(cancelKey) || Input.GetMouseButtonDown
                      (1)) CancelCurrentSubmission();
125           }
```

```
126
127         // Otherwise, the options interface is opened and should utilize a different transition scheme.
128         else if (Input.GetKeyDown(cancelKey) || Input.GetMouseButtonDown(1))
129         {
130             // If at the root, exit the interface.
131             if (optionSelectionInterface.activeSelf)
                CancelCurrentSubmission();
132
133             // If within the custom circuits interface whilst the delete error interface is not open, return to the root.
134             else if (customCircuitsInterface.activeSelf && !deleteErrorInterface.activeSelf)
135             {
136                 customCircuitsInterface.SetActive(false);
137                 optionSelectionInterface.SetActive(true);
138             }
139
140             // If within the guide interface, return to the root.
141             else if (guideInterface.activeSelf)
142             {
143                 guideInterface.SetActive(false);
144                 optionSelectionInterface.SetActive(true);
145             }
146
147             // If within the guide delete error interface, re-adjust background layers to make the custom circuit interface accessible.
148             else if (deleteErrorInterface.activeSelf)
149             {
150                 transparentBackgroundUpper.SetActive(false);
151                 transparentBackground.SetActive(true);
152                 deleteErrorInterface.SetActive(false);
153             }
154         }
155     }
156
157     /// <summary>
158     /// Opens an editor scene; called by pressing one of the valid save buttons.
159     /// </summary>
160     /// <param name="sceneIndex"></param>
161     public void OpenScene(int sceneIndex) { MenuLogicManager.Instance.OpenScene(sceneIndex); }
162
163     /// <summary>
164     /// Instantiates a <seealso cref="customCircuitPrefab"/> button for each preview structure.
```

```
165     /// </summary>
166     private void AddCustomBookmarks()
167     {
168         List<PreviewStructure> previewStructures =
              MenuSetupManager.Instance.PreviewStructures;
169
170         foreach (PreviewStructure previewStructure in previewStructures)
171         {
172             GameObject current = Instantiate(customCircuitPrefab,
                  customCircuitPanel.transform);
173
174             current.GetComponentInChildren<TextMeshProUGUI>().text =
                  previewStructure.Name;
175
176             // Adds the relevant listeners to ensure the custom circuit
                  can be deleted and viewed by their respective buttons.
177             current.GetComponent<CustomCircuitButtons>
                  ().DeleteButton.onClick.AddListener(delegate { DeletePreview
                  (current, previewStructure); });
178             current.GetComponent<CustomCircuitButtons>
                  ().ViewButton.onClick.AddListener(delegate { PreviewScene
                  (previewStructure); });
179         }
180     }
181
182     /// <summary>
183     /// Deletes a preview structure from the game and directory given it
          is not in use.
184     /// </summary>
185     /// <param name="button">The button assigned this method by a
          delegate.</param>
186     /// <param name="previewStructure">The preview structure to attempt
          deletion on.</param>
187     public void DeletePreview(GameObject button, PreviewStructure
          previewStructure)
188     {
189         // Obtains a list of all error messages in an attempt to delete
              the preview structure
190         List<string> errorMessages =
              MenuLogicManager.CanDeleteCustomCircuit(previewStructure);
191
192         // If there are no errors, proceed with deletion.
193         if (errorMessages.Count == 0)
194         {
195             MenuSetupManager.Instance.DeletePreviewStructure
                  (previewStructure);
196             Destroy(button);
197         }
198
```

```
199         // Otherwise, open the delete error interface and display errors.
200         else
201         {
202             transparentBackgroundUpper.SetActive(true);
203             transparentBackground.SetActive(false);
204             deleteErrorInterface.SetActive(true);
205             deleteErrorTransform.sizeDelta = new Vector2
                  (deleteErrorTransform.sizeDelta.x, deleteErrorMessageSize *
                  errorMessages.Count);
206             deleteErrorText.text = "";
207 
208             int index = 0;
209 
210             // Adds each error
211             foreach (string errorMessage in errorMessages)
212             {
213                 index++;
214                 deleteErrorText.text += "- " + errorMessage + (index !=
                      errorMessages.Count ? "\n\n" : "");
215             }
216         }
217     }
218 
219     /// <summary>
220     /// Referenced when the user acknowledges and closes the delete error
          interface; called by pressing an in-scene button.
221     /// </summary>
222     public void OnCircuitDeleteErrorConfirm()
223     {
224         transparentBackgroundUpper.SetActive(false);
225         transparentBackground.SetActive(true);
226         deleteErrorInterface.SetActive(false);
227     }
228 
229     /// <summary>
230     /// Opens a preview structure whose internal components the user
          wishes to inspect.<br/><br/>
231     /// This method is assigned as a listener to the delete button within
          each instantiated <seealso cref="customCircuitPrefab"/>.
232     /// </summary>
233     /// <param name="previewStructure">The preview structure to open.</
          param>
234     public void PreviewScene(PreviewStructure previewStructure)
          { MenuLogicManager.Instance.OpenPreview(previewStructure); }
235 
236     /// <summary>
237     /// Sets the names of all save slots corresponding to serialized
          editor structures.
238     /// </summary>
```

```
239     public void UpdateInterface()
240     {
241         EditorStructure[] editorStructures =
              MenuSetupManager.Instance.EditorStructures;
242
243         if (editorStructures[0] != null) { save1.text = editorStructures
              [0].Name; save1.color = saveColor; }
244
245         if (editorStructures[1] != null) { save2.text = editorStructures
              [1].Name; save2.color = saveColor; }
246
247         if (editorStructures[2] != null) { save3.text = editorStructures
              [2].Name; save3.color = saveColor; }
248     }
249
250     /// <summary>
251     /// Opens <seealso cref="sceneDeletionInterface"/> prompting a user to
          acknowledge their action to delete a editor structure (save slot).
252     /// </summary>
253     /// <param name="sceneIndex">The index of the prospective editor
          structure to delete</param>
254     public void BeginSceneDeletion(int sceneIndex)
255     {
256         if (MenuSetupManager.Instance.EditorStructures[sceneIndex] ==
              null) return;
257
258         currentSceneIndex = sceneIndex;
259         BeginInterface(sceneDeletionInterface);
260     }
261
262     /// <summary>
263     /// Opens <seealso cref="sceneNameInterface"/> prompting a user to
          compose a name to create an editor structure.
264     /// </summary>
265     /// <param name="sceneIndex">The index of the prospective editor
          structure to create.</param>
266     public void BeginSceneNameSubmission(int sceneIndex)
267     {
268         currentSceneIndex = sceneIndex;
269         BeginInterface(sceneNameInterface);
270     }
271
272     /// <summary>
273     /// Opens the options interface; called by pressing an in-scene
          button.
274     /// </summary>
275     public void OpenOptionsInterface()
276     {
277         BeginInterface(optionsInterface);
```

```
278         optionSelectionInterface.SetActive(true);
              customCircuitsInterface.SetActive(false);
              guideInterface.SetActive(false);
279     }
280
281     /// <summary>
282     /// Opens the guide interface; called by pressing an in-scene button.
283     /// </summary>
284     public void OpenGuide()
285     {
286         optionSelectionInterface.SetActive(false);
287         guideInterface.SetActive(true);
288     }
289
290     /// <summary>
291     /// Opens the custom circuit interface; called by pressing an in-scene
          button.
292     /// </summary>
293     public void OpenCustomCircuits()
294     {
295         optionSelectionInterface.SetActive(false);
296         customCircuitsInterface.SetActive(true);
297     }
298
299     /// <summary>
300     /// Restores the text of a save slot to its default values after a
        success editor scene deletion.
301     /// </summary>
302     public void SceneDeleteSubmission()
303     {
304         MenuSetupManager.Instance.DeleteEditorStructure
              (currentSceneIndex);
305         TextMeshProUGUI currentText;
306
307         if (currentSceneIndex == 0)
308         {
309             currentText = save1;
310         }
311
312         else if (currentSceneIndex == 1)
313         {
314             currentText = save2;
315         }
316
317         else
318         {
319             currentText = save3;
320         }
321
```

```
322         CancelCurrentSubmission();
323         currentText.text = "new save";
324         currentText.color = defaultColor;
325 
326     }
327 
328     /// <summary>
329     /// Checks the name in <seealso cref="sceneNameInputField"/> and creates an editor scene if it is valid.
330     /// </summary>
331     public void SceneNameSubmission()
332     {
333         string submission = sceneNameInputField.text.ToLower().Trim();
334 
335         // Cannot be empty
336         if (submission == string.Empty) sceneNameError.text = "scene name must be non-empty";
337 
338         // Must be unique
339         else if (CurrentSceneNames.Contains(submission)) sceneNameError.text = "scene name must be unique";
340 
341         // Creates and opens the editor structure.
342         else MenuLogicManager.Instance.CreateScene(currentSceneIndex, submission);
343     }
344 
345     /// <summary>
346     /// Closes the current interface in use.
347     /// </summary>
348     public void CancelCurrentSubmission()
349     {
350         if (currentInterface == sceneNameInterface) sceneNameInputField.text = sceneNameError.text = "";
351 
352         BackgroundParallax.Instance.enabled = true;
353         ToggleCurrentInterface(false);
354         currentSceneIndex = -1;
355         currentInterface = null;
356         enabled = false;
357     }
358 
359     // Exits the game; called by pressing an in-scene button.
360     public void Quit() { Application.Quit(); }
361 
362     /// <summary>
363     /// Opens an interface.
364     /// </summary>
365     /// <param name="newInterface">The interface to open.</param>
```

```
366     private void BeginInterface(GameObject newInterface)
367     {
368         if (currentInterface != null) return;
369 
370         BackgroundParallax.Instance.enabled = false;
371         currentInterface = newInterface;
372         ToggleCurrentInterface(true);
373         enabled = true;
374     }
375 
376     /// <summary>
377     /// Sets the visibility of the current interface.
378     /// </summary>
379     /// <param name="visible">Whether the current interface should be
        visible.</param>
380     private void ToggleCurrentInterface(bool visible)
381     {
382         currentInterface.SetActive(visible);
          transparentBackground.SetActive(visible);
383     }
384 
385     // Getter methods
386     public static MenuInterfaceManager Instance { get { return
        instance; } }
387 
388     /// <summary>
389     /// Returns the named list of all editor structures.
390     /// </summary>
391     private List<string> CurrentSceneNames
392     {
393         get
394         {
395             EditorStructure[] editorStructures =
                MenuSetupManager.Instance.EditorStructures;
396             List<string> currentSceneNames = new List<string>();
397 
398             foreach (EditorStructure editorStructure in editorStructures)
                if (editorStructure != null) currentSceneNames.Add
                (editorStructure.Name);
399 
400             return currentSceneNames;
401         }
402     }
403 }
```

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

/// <summary>
/// MenuLogicManager handles all scene transitions as well as validation tests for deleting custom circuits.
/// </summary>
public class MenuLogicManager : MonoBehaviour
{
    // Singleton state reference
    private static MenuLogicManager instance;

    /// <summary>
    /// Whether the current editor scene has just been created or not.<br/><br/>
    /// If it is the first time, this value indicates that default values must be initialized in the editor scene.
    /// </summary>
    private bool firstOpen;

    /// <summary>
    /// The index of the current editor scene to open.
    /// </summary>
    private int currentSceneIndex;

    /// <summary>
    /// The current preview structure in the preview scene.
    /// </summary>
    private PreviewStructure currentPreviewStructure;

    /// <summary>
    /// Utilized within <seealso cref="TraversalTest(PreviewStructure, PreviewStructure)"/> to sort all preview structures into these respective lists.<br/>
    /// If a circuit is invalid, it means that it is being utilized somewhere and cannot be deleted. A valid circuit can be deleted.
    /// </summary>
    private List<PreviewStructure> invalidCircuits, validCircuits;

    // Enforces a singleton state pattern
    private void Awake()
    {
        if (instance != null)
        {
            Destroy(this);
            throw new Exception("MenuLogicManager instance already established; terminating.");
```

```
43          }
44
45          instance = this;
46      }
47
48      /// <summary>
49      /// Attempts to open an editor scene by index.
50      /// </summary>
51      /// <param name="sceneIndex">Index of the editor structure to open.</
          param>
52      public void OpenScene(int sceneIndex)
53      {
54          bool isCreated = MenuSetupManager.Instance.EditorStructures
            [sceneIndex] != null;
55
56          firstOpen = !isCreated;
57
58          // Opens the scene only if it has not been just created; otherwise
             completes the setup process.
59          if (isCreated) ImportScene(sceneIndex); else
            MenuInterfaceManager.Instance.BeginSceneNameSubmission
            (sceneIndex);
60      }
61
62      /// <summary>
63      /// Creates an editor scene.
64      /// </summary>
65      /// <param name="sceneIndex">Index of the editor structure to
          create.</param>
66      /// <param name="name">Name of the editor structure to create.</param>
67      public void CreateScene(int sceneIndex, string name)
68      {
69          MenuSetupManager.Instance.EditorStructures[sceneIndex] = new
            EditorStructure(name);
70          ImportScene(sceneIndex);
71      }
72
73      /// <summary>
74      /// Opens a preview structure.
75      /// </summary>
76      /// <param name="previewStructure">The preview structure to open.</
          param>
77      public void OpenPreview(PreviewStructure previewStructure)
78      {
79          currentPreviewStructure = previewStructure;
80          SceneManager.LoadScene(2);
81      }
82
83      /// <summary>
```

```
84      /// Opens a scene that has already been created beforehand.
85      /// </summary>
86      /// <param name="sceneIndex"></param>
87      private void ImportScene(int sceneIndex)
88      {
89          currentSceneIndex = sceneIndex;
90          SceneManager.LoadScene(1);
91      }
92
93      /// <summary>
94      /// Runs several validation tests that determine whether a custom
          circuit can be deleted.<br/><br/>.
95      /// If there are no error messages returned, then the custom circuit
          can be deleted.
96      /// </summary>
97      /// <param name="_previewStructure">The prospect custom circuit/
          preview structure to delete.</param>
98      /// <returns></returns>
99      public static List<string> CanDeleteCustomCircuit(PreviewStructure
          _previewStructure)
100     {
101         List<string> errorMessages = new List<string>();
102         instance.invalidCircuits = new List<PreviewStructure>();
            instance.validCircuits = new List<PreviewStructure>();
103
104         // Marks each preview structure as either valid or invalid
105         foreach (PreviewStructure previewStructure in
            MenuSetupManager.Instance.PreviewStructures)
            instance.TraversalTest(previewStructure, _previewStructure);
106
107         // Tests #1/#2: not placed or part of a bookmark directly (top-
            most custom circuit) or indirectly (inside of another custom
            circuit) within an editor scene.
108         foreach (EditorStructure editorStructure in
            MenuSetupManager.Instance.EditorStructures)
109         {
110             if (editorStructure == null) continue;
111
112             // Placed circuit test
113             foreach (CircuitIdentifier circuitIdentifier in
                editorStructure.Circuits)
114             {
115                 if (circuitIdentifier.previewStructureID == -1) continue;
116
117                 PreviewStructure previewStructure =
                    MenuSetupManager.Instance.PreviewStructures
                    [MenuSetupManager.Instance.PreviewStructureIDs.IndexOf
                    (circuitIdentifier.previewStructureID)];
118
```

```
119                 if (instance.invalidCircuits.Contains(previewStructure))
                      { errorMessages.Add("Circuit is directly and/or
                      indirectly referenced by a placed circuit in an editor
                      scene"); break; }
120             }
121
122             // Bookmark test
123             foreach (int customCircuitID in editorStructure.BookmarkIDs)
124             {
125                 if (customCircuitID == -1) continue;
126
127                 PreviewStructure previewStructure =
                      MenuSetupManager.Instance.PreviewStructures
                      [MenuSetupManager.Instance.PreviewStructureIDs.IndexOf
                      (customCircuitID)];
128
129                 if (instance.invalidCircuits.Contains(previewStructure))
                      { errorMessages.Add("Circuit is directly and/or
                      indirectly referenced by a bookmark in an editor
                      scene"); break; }
130             }
131         }
132
133         // Test #3: not directly (top-most custom circuit) or indirectly
              (inside of another custom circuit) within a custom circuit.
134         foreach (PreviewStructure previewStructure in
              MenuSetupManager.Instance.PreviewStructures)
135         {
136             if (previewStructure == _previewStructure) continue;
137
138             if (instance.invalidCircuits.Contains(previewStructure))
                  { errorMessages.Add("Circuit is directly and/or indirectly
                  referenced by another custom circuit"); break; }
139         }
140
141         // Returns the list of all obtained error messages, if any.
142         return errorMessages;
143     }
144
145     /// <summary>
146     /// Sorts the current preview structure as invalid (contains the
          target circuit) or valid by running a modified depth-first
          search.<br/><br/>
147     /// Essentially, the user inputs a starting preview structure. Until a
           starting circuit (e.g. AND) is reached or there is nothing else to
          explore, this method:<br/>
148     /// - Recursively calls this method on all of its circuit components
          that are custom circuits granted they have not already been sorted
          into valid or invalid circuits.
```

```
149     /// - Adds the current circuit as an invalid preview structure if any of its children contained the target custom circuit, and returns.<br/>
150     /// - Adds the current circuit as a valid preview structure if none of its child custom circuits contained the target custom circuit.
151     /// </summary>
152     /// <param name="current"></param>
153     /// <param name="target"></param>
154     private void TraversalTest(PreviewStructure current, PreviewStructure target)
155     {
156         // If the target preview structure was recursively called by TraversalTest, then add it.
157         if (current == target) { if (!invalidCircuits.Contains(current)) { invalidCircuits.Add(current); } return; }
158 
159         // If already designated as an invalid/valid circuit, return.
160         else if (invalidCircuits.Contains(current) || validCircuits.Contains(current)) return;
161 
162         // Traverse through each custom circuit and recursively call this method.
163         // Once done, check to see if the target preview structure was detected. If so, then also add this preview structure.
164         foreach (CircuitIdentifier circuitIdentifier in current.Circuits)
165         {
166             if (circuitIdentifier.previewStructureID != -1)
167             {
168                 PreviewStructure previewStructure = MenuSetupManager.Instance.PreviewStructures[MenuSetupManager.Instance.PreviewStructureIDs.IndexOf(circuitIdentifier.previewStructureID)];
169 
170                 TraversalTest(previewStructure, target);
171 
172                 if (invalidCircuits.Contains(previewStructure)) { invalidCircuits.Add(current); return; }
173             }
174         }
175 
176         // If not invalid, then must be valid.
177         validCircuits.Add(current);
178     }
179 
180     // Getter methods
181     public static MenuLogicManager Instance { get { return instance; } }
182 
183     public bool FirstOpen { get { return firstOpen; } }
184 
```

```
185     public int CurrentSceneIndex { get { return currentSceneIndex; } }
186
187     public PreviewStructure CurrentPreviewStructure { get { return
          currentPreviewStructure; } }
188 }
```

```
1  using System;
2  using System.Collections.Generic;
3  using System.IO;
4  using System.Linq;
5  using UnityEngine;
6
7  /// <summary>
8  /// MenuSetupManager serves as the primary script for persistence and
     communication between the menu and editor/preview scenes.
9  /// </summary>
10 public class MenuSetupManager : MonoBehaviour
11 {
12     // Singleton state reference
13     private static MenuSetupManager instance;
14
15     /// <summary>
16     /// The list of persistent scripts that should be loaded after
         MenuSetupManager.
17     /// </summary>
18     private Type[] componentsToAdd = new Type[]
19     {
20         typeof(MenuLogicManager)
21     };
22
23     /// <summary>
24     /// The list of editor scenes that exist within the project.
25     /// </summary>
26     private EditorStructure[] editorStructures = new EditorStructure[3];
27
28     private FileAttributes fileAttributes = FileAttributes.Normal;
29
30     /// <summary>
31     /// List of custom circuit IDs corresponding to each element within
         <seealso cref="previewStructures"/>.<br/><br/>
32     /// This list is primarily utilized to find a <see
         cref="PreviewStructure"/> within <seealso cref="previewStructures"/>
          through the IndexOf() method.
33     /// </summary>
34     private List<int> previewStructureIDs = new List<int>();
35
36     /// <summary>
37     /// The list of custom circuits created by the user.<br/><br/>
38     /// Note: a <see cref="PreviewStructure"/> is synonymous with a custom
          circuit; however a preview structure tends to refer to the actual
         scene where the custom circuit can be internally viewed.
39     /// </summary>
40     private List<PreviewStructure> previewStructures = new
         List<PreviewStructure>();
41
```

```
42      /// <summary>
43      /// Static constants representing folder names and file names used for
            serialization.
44      /// </summary>
45      private readonly string editorFolder = "EditorSaves",
46          editorPrefab1Name = "PREFABS_0",
47          editorPrefab2Name = "PREFABS_1",
48          editorPrefab3Name = "PREFABS_2",
49          previewFolder = "PreviewSaves",
50          previewSubdirectory = "CUSTOM_",
51          save1Name = "SAVE_0.json",
52          save2Name = "SAVE_1.json",
53          save3Name = "SAVE_2.json";
54
55      // Enforces a singleton state pattern and imports all serialized
            information.
56      private void Awake()
57      {
58          if (instance != null)
59          {
60              Destroy(this);
61              return;
62          }
63
64          instance = this;
65          DontDestroyOnLoad(this);
66          ImportJSONInformation();
67
68          foreach (Type type in componentsToAdd) gameObject.AddComponent
                (type);
69      }
70
71      /// <summary>
72      /// Deletes a requested editor scene within the running game as well
            as in the save directory.
73      /// </summary>
74      /// <param name="sceneIndex">The editor scene to delete (0-2).</param>
75      public void DeleteEditorStructure(int sceneIndex)
76      {
77          editorStructures[sceneIndex] = null;
78
79          // Obtains the save path pertaining to the requested editor scene.
80          string editorPath = Application.persistentDataPath + "/" +
                editorFolder + "/";
81          string prefabPath = Application.persistentDataPath + "/" +
                editorFolder + "/";
82
83          if (sceneIndex == 0) prefabPath += editorPrefab1Name; else if
                (sceneIndex == 1) prefabPath += editorPrefab2Name; else
```

```
            prefabPath += editorPrefab3Name;
84
85          if (sceneIndex == 0) editorPath += save1Name; else if (sceneIndex
              == 1) editorPath += save2Name; else editorPath += save3Name;
86
87          File.WriteAllText(editorPath, ""); // Deletes the editor structure
               JSON file
88          prefabPath += "/";
89
90          // Deletes all connection JSON files, if any
91          string[] filePaths = Directory.GetFiles(prefabPath);
92
93          foreach (string file in filePaths) File.Delete(file);
94      }
95
96      /// <summary>
97      /// Overrides a requested editor scene as a consequence of a new save.
98      /// </summary>
99      /// <param name="sceneIndex">The editor scene to update (0-2).</param>
100     /// <param name="editorStructure">The editor scene object to update.</
          param>
101     public void UpdateEditorStructure(int sceneIndex, EditorStructure
          editorStructure)
102     {
103         string editorPath = Application.persistentDataPath + "/" +
              editorFolder + "/";
104
105         if (sceneIndex == 0) editorPath += save1Name; else if (sceneIndex
              == 1) editorPath += save2Name; else editorPath += save3Name;
106
107         File.WriteAllText(editorPath, JsonUtility.ToJson
              (editorStructure));
108     }
109
110     /// <summary>
111     /// Overrides a requested preview structure as a consequence of a new
          save.
112     /// </summary>
113     /// <param name="previewStructure">The preview structure object to
          update.</param>
114     public void UpdatePreviewStructure(PreviewStructure previewStructure)
115     {
116         string previewPath = Application.persistentDataPath + "/" +
              previewFolder + "/" + previewSubdirectory + previewStructure.ID
              + "/SAVE.json";
117
118         File.WriteAllText(previewPath, JsonUtility.ToJson
              (previewStructure));
119     }
```

```
120
121     /// <summary>
122     /// Serializes all connections pertaining to either a preview or editor structure.
123     /// </summary>
124     /// <param name="isEditor">Whether the referenced connections belong to an editor scene.</param>
125     /// <param name="generateIndex">The editor scene to update (0-2) if the referenced connections belong to an editor scene.</param>
126     /// <param name="connections">The connections to serialize.</param>
127     public void GenerateConnections(bool isEditor, int generateIndex, List<CircuitConnector.Connection> connections)
128     {
129         // Obtains the path to save all connections to.
130         string path = Application.persistentDataPath + "/" + (isEditor ? editorFolder : previewFolder) + "/";
131
132         if (isEditor)
133         {
134             if (generateIndex == 0) path += editorPrefab1Name; else if (generateIndex == 1) path += editorPrefab2Name; else path += editorPrefab3Name;
135         }
136
137         else
138         {
139             path += previewSubdirectory + generateIndex;
140
141             if (!Directory.Exists(path))
142             {
143                 Directory.CreateDirectory(Application.persistentDataPath + "/" + previewFolder + "/" + previewSubdirectory + generateIndex);
144             }
145         }
146
147         // In case the folder is already populated, all files are cleared from the obtained directory.
148         string[] filePaths = Directory.GetFiles(path);
149
150         foreach (string file in filePaths) File.Delete(file);
151
152         // No point in continuing if there are no connections.
153         if (connections.Count == 0) return;
154
155         int index = 0;
156
157         // Traverses through each connection and generates a corresponding JSON file.
```

```
158         foreach (CircuitConnector.Connection connection in connections)
159         {
160             int inputCircuitIndex;
161             int outputCircuitIndex;
162             int inputIndex;
163             int outputIndex;
164 
165             // Runs if the input belongs to a custom circuit
166             // customCircuit == null --> a non-custom circuit
167             if (connection.Input.ParentCircuit.customCircuit != null)
168             {
169                 /* A custom circuit can be put inside of another custom
                      circuit recursively.
170                  * Therefore, to obtain the top-most (actual) custom
                       circuit located in the scene, some calculations must
                       occur.
171                  * The primary condition for this is to keep accessing the
                        custom circuit of the parent until it is null.
172                  * If it is null, that means the current custom circuit is
                        at the top-most level.
173                  * This essentially emulates a linked-list property, where
                        the head is the node with no parent.
174                  */
175                 Circuit actualCircuit =
                      connection.Input.ParentCircuit.customCircuit;
176 
177                 // Obtains the top-most custom circuit
178                 while (actualCircuit.customCircuit != null) actualCircuit
                      = actualCircuit.customCircuit;
179 
180                 inputCircuitIndex =
                      EditorStructureManager.Instance.Circuits.IndexOf
                      (actualCircuit);
181                 inputIndex = Array.IndexOf(actualCircuit.Inputs,
                      connection.Input);
182             }
183 
184             // Runs if the input belongs to a non-custom circuit
185             else
186             {
187                 inputCircuitIndex =
                      EditorStructureManager.Instance.Circuits.IndexOf
                      (connection.Input.ParentCircuit);
188                 inputIndex = Array.IndexOf
                      (connection.Input.ParentCircuit.Inputs,
                      connection.Input);
189             }
190 
191             // Runs if the output belongs to a custom circuit
```

```
192             if (connection.Output.ParentCircuit.customCircuit != null)
193             {
194                 Circuit actualCircuit =
                      connection.Output.ParentCircuit.customCircuit;
195
196                 while (actualCircuit.customCircuit != null) actualCircuit
                      = actualCircuit.customCircuit;
197
198                 outputCircuitIndex =
                      EditorStructureManager.Instance.Circuits.IndexOf
                      (actualCircuit);
199                 outputIndex = Array.IndexOf(actualCircuit.Outputs,
                      connection.Output);
200             }
201
202             // Runs if the output belongs to a non-custom circuit
203             else
204             {
205                 outputCircuitIndex =
                      EditorStructureManager.Instance.Circuits.IndexOf
                      (connection.Output.ParentCircuit);
206                 outputIndex = Array.IndexOf
                      (connection.Output.ParentCircuit.Outputs,
                      connection.Output);
207             }
208
209             // Creates a corresponding connection identifier from the
                  obtained indeces and saves to the obtained directory.
210             CircuitConnectorIdentifier circuitConnectionIdentifier = new
                  CircuitConnectorIdentifier(inputCircuitIndex,
                  outputCircuitIndex, inputIndex, outputIndex);
211             ConnectionSerializer.SerializeConnection(connection,
                  circuitConnectionIdentifier, path + "/CONNECTION_" + index +
                   ".json");
212             index++;
213         }
214     }
215
216     /// <summary>
217     /// Editor structure variation of the RestoreConnections() method
          utilized to restore serialized connections.
218     /// </summary>
219     /// <param name="sceneIndex">The editor scene to delete (0-2).</param>
220     public void RestoreConnections(int sceneIndex)
221     {
222         // Obtains the path to access the connection JSON files from.
223         string prefabPath = Application.persistentDataPath + "/" +
              editorFolder + "/";
224
```

```
225         if (sceneIndex == 0) prefabPath += editorPrefab1Name; else if
              (sceneIndex == 1) prefabPath += editorPrefab2Name; else
              prefabPath += editorPrefab3Name;
226
227         prefabPath += "/";
228
229         // Calls the primary method with the obtained directory.
230         RestoreConnections(prefabPath, true);
231     }
232
233     /// <summary>
234     /// Preview structure variation of the RestoreConnections() method
          utilized to restore serialized connections.
235     /// </summary>
236     /// <param name="previewStructure">The preview structure object to
          access.</param>
237     public void RestoreConnections(PreviewStructure previewStructure)
          { RestoreConnections(Application.persistentDataPath + "/" +
          previewFolder + "/" + previewSubdirectory + previewStructure.ID +
          "/", false); }
238
239     /// <summary>
240     /// Deserializes saved connections and restores them to the relevant
          scene.
241     /// </summary>
242     /// <param name="prefabPath">The path to access the serialized
          connection files from.</param>
243     /// <param name="isEditor">Whether the path points to an editor
          structure.</param>
244     private void RestoreConnections(string prefabPath, bool isEditor)
245     {
246         string[] filePaths = Directory.GetFiles(prefabPath);
247
248         // Ensures only JSON files are being accessed.
249         List<string> prefabFilePaths = filePaths.Where(filePath =>
              filePath.EndsWith(".json")).ToList();
250
251         // Iterates through each connection file and restores it to the
              scene.
252         foreach (string prefabFilePath in prefabFilePaths)
253         {
254             // Implies the current JSON is a save file rather than a
                  connection file, and should therefore be skipped.
255             if (prefabFilePath.EndsWith("SAVE.json")) continue;
256
257             // Simultaneously creates the connection mesh as well as all
                  relevant values in the form of a
                  ConnectionSerializerRestorer object
258             ConnectionSerializerRestorer connectionParent =
```

```
                CircuitVisualizer.Instance.VisualizeConnection
                (JsonUtility.FromJson<ConnectionSerializer>(File.ReadAllText
                (prefabFilePath)));
259
260             // Depending on whether the scene is in the editor or not, the
                   method of obtaining the connection's input and output will
                differ.
261             Circuit.Input input = isEditor ?
262                 EditorStructureManager.Instance.Circuits
                    [connectionParent.circuitConnectorIdentifier.InputCircui
                    tIndex].Inputs
                    [connectionParent.circuitConnectorIdentifier.InputIndex]
                     :
263                 PreviewManager.Instance.Circuits
                    [connectionParent.circuitConnectorIdentifier.InputCircui
                    tIndex].Inputs
                    [connectionParent.circuitConnectorIdentifier.InputIndex]
                    ;
264             Circuit.Output output = isEditor ?
265                 EditorStructureManager.Instance.Circuits
                    [connectionParent.circuitConnectorIdentifier.OutputCircu
                    itIndex].Outputs
                    [connectionParent.circuitConnectorIdentifier.OutputIndex
                    ] :
266                 PreviewManager.Instance.Circuits
                    [connectionParent.circuitConnectorIdentifier.OutputCircu
                    itIndex].Outputs
                    [connectionParent.circuitConnectorIdentifier.OutputIndex
                    ];
267
268             // Names and restores the connection within the scene by
                  setting the parent circuits of the input and output.
269             connectionParent.parentObject.name = "Connection";
270             CircuitConnector.ConnectRestoration
                  (connectionParent.parentObject, input, output,
                  connectionParent.endingWire, connectionParent.startingWire,
                  isEditor);
271         }
272     }
273
274     /// <summary>
275     /// Deletes a requested preview structure within the running game as
          well as in the save directory.
276     /// </summary>
277     /// <param name="previewStructure">The preview structure object to
          delete.</param>
278     public void DeletePreviewStructure(PreviewStructure previewStructure)
279     {
280         int index = previewStructures.IndexOf(previewStructure);
```

```
281         string folderPath = Application.persistentDataPath + "/" +
              previewFolder + "/" + previewSubdirectory + previewStructure.ID;
282
283         previewStructures.Remove(previewStructure);
284         previewStructureIDs.Remove(previewStructureIDs[index]);
285
286         string[] filePaths = Directory.GetFiles(folderPath + "/");
287
288         foreach (string file in filePaths) File.Delete(file);
289
290         Directory.Delete(folderPath);
291     }
292
293     /// <summary>
294     /// Extracts existing JSON data from the game directory to populate
          all editor and preview structures.<br/><br/>
295     /// This method is called on startup before anything else.
296     /// </summary>
297     private void ImportJSONInformation()
298     {
299         // Ensures the base editor and preview save folders are created if
              they were removed
300         if (!Directory.Exists(Application.persistentDataPath + "/" +
              editorFolder))
301         {
302             Directory.CreateDirectory(Application.persistentDataPath + "/"
                  + editorFolder);
303         }
304
305         if (!Directory.Exists(Application.persistentDataPath + "/" +
              previewFolder))
306         {
307             Directory.CreateDirectory(Application.persistentDataPath + "/"
                  + previewFolder);
308         }
309
310         //  Ensures the editor subdirectory save folders are created if
              they were removed
311         if (!Directory.Exists(Application.persistentDataPath + "/" +
              editorFolder + "/" + editorPrefab1Name))
312         {
313             Directory.CreateDirectory(Application.persistentDataPath + "/"
                  + editorFolder + "/" + editorPrefab1Name);
314         }
315
316         if (!Directory.Exists(Application.persistentDataPath + "/" +
              editorFolder + "/" + editorPrefab2Name))
317         {
318             Directory.CreateDirectory(Application.persistentDataPath + "/"
```

```
                + editorFolder + "/" + editorPrefab2Name);
        }

        if (!Directory.Exists(Application.persistentDataPath + "/" +
          editorFolder + "/" + editorPrefab3Name))
        {
            Directory.CreateDirectory(Application.persistentDataPath + "/"
               + editorFolder + "/" + editorPrefab3Name);
        }

        string editorPath = Application.persistentDataPath + "/" +
          editorFolder + "/";

        // Ensures the relevant editor JSON save files are created if they
           were removed, otherwise they are loaded into the game.
        if (!File.Exists(editorPath + save1Name))
        {
            File.Create(editorPath + save1Name);
            File.SetAttributes(editorPath + save1Name, fileAttributes);
        }

        else
        {
            editorStructures[0] = JsonUtility.FromJson<EditorStructure>
              (File.ReadAllText(editorPath + save1Name));
        }

        if (!File.Exists(editorPath + save2Name))
        {
            File.Create(editorPath + save2Name);
            File.SetAttributes(editorPath + save2Name, fileAttributes);
        }

        else
        {
            editorStructures[1] = JsonUtility.FromJson<EditorStructure>
              (File.ReadAllText(editorPath + save2Name));
        }

        if (!File.Exists(editorPath + save3Name))
        {
            File.Create(editorPath + save3Name);
            File.SetAttributes(editorPath + save3Name, fileAttributes);
        }

        else
        {
            editorStructures[2] = JsonUtility.FromJson<EditorStructure>
              (File.ReadAllText(editorPath + save3Name));
```

```
360         }
361
362         string[] previewFilePaths = Directory.GetDirectories
              (Application.persistentDataPath + "/" + previewFolder);
363
364         // Traverses through all valid preview save files and loads them
              into the game.
365         foreach (string filePath in previewFilePaths)
366         {
367             string[] previewFiles = Directory.GetFiles(filePath);
368             string jsonFile = previewFiles.FirstOrDefault(s => s.EndsWith
                ("SAVE.json"));
369
370             if (jsonFile == null) throw new Exception("Preview structure
                JSON modified outside the script; terminating.");
371
372             try
373             {
374                 PreviewStructure previewStructure =
                    JsonUtility.FromJson<PreviewStructure>(File.ReadAllText
                    (jsonFile));
375
376                 previewStructures.Add(previewStructure);
377                 PreviewStructureIDs.Add(previewStructure.ID);
378             }
379
380             catch
381             {
382                 throw new Exception("Preview structure JSON modified
                    outside the script; terminating.");
383             }
384         }
385     }
386
387     // Getter methods
388     public static MenuSetupManager Instance { get { return instance; } }
389
390     public EditorStructure[] EditorStructures { get { return
          editorStructures; } }
391
392     public List<int> PreviewStructureIDs { get { return
          previewStructureIDs; } }
393
394     public List<PreviewStructure> PreviewStructures { get { return
          previewStructures; } }
395 }
```

```cs
using System;
using TMPro;
using UnityEngine;
using UnityEngine.EventSystems;

/// <summary>
/// BehaviorManagerPreview handles game state actions transitions within a preview scene.
/// </summary>
public class BehaviorManagerPreview : MonoBehaviour
{
    // Singleton state reference
    private static BehaviorManagerPreview instance;

    /// <summary>
    /// The material utilized for empty inputs or outputs the user is currently hovered on.
    /// </summary>
    [SerializeField]
    Material selectedMaterial;

    /// <summary>
    /// Displays the current world position.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI coordinatesText;

    /// <summary>
    /// Displays the current label of the empty input or output hovered on, if applicable.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI labelText;

    /// <summary>
    /// Whether the user is currently hovered onto an empty input or output.
    /// </summary>
    private bool isInput;

    /// <summary>
    /// Whether the user is currently hovered onto a UI element.
    /// </summary>
    private bool isUILocked;

    /// <summary>
    /// The current GameObject raycasted to; guaranteed to be an input or output.
    /// </summary>
```

```
45      private GameObject currentHitObject;
46
47      /// <summary>
48      /// The current index of the empty input or output that the user is
          hovered on.
49      /// </summary>
50      private int labelIndex;
51
52      /// <summary>
53      /// The global grid height that all raycasts are cast on.
54      /// </summary>
55      private Plane coordinatesPlane;
56
57      /// <summary>
58      /// Default text utilized when the user is not hovered onto an empty
          input or output.
59      /// </summary>
60      private readonly string defaultHoverText = "hover over and select
          inputs/outputs to view their order & label";
61
62      // Enforces a singleton state pattern and establishes the grid plane.
63      private void Awake()
64      {
65          if (instance != null)
66          {
67              Destroy(this);
68              throw new Exception("BehaviorManagerPreview instance already
                  established; terminating.");
69          }
70
71          instance = this;
72          coordinatesPlane = new Plane(Vector3.down,
              GridMaintenance.Instance.GridHeight);
73      }
74
75      private void Start() { labelText.text = defaultHoverText; }
76
77      private void Update()
78      {
79          // If hovered onto UI, reset to default values
80          if (EventSystem.current.IsPointerOverGameObject()) { isUILocked =
              true; State(null); return; }
81
82          isUILocked = false; // Otherwise, game is not paused.
83
84          bool raycastHit = Physics.Raycast
              (CameraMovementPreview.Instance.PlayerCamera.ScreenPointToRay
              (Input.mousePosition), out RaycastHit hitInfo);
85
```

```
86          // If raycast invalid, reset to default values
87          if (!raycastHit) { State(null); return; }
88
89          // If raycast GameObject is not an input or output, reset to default values
90          if (hitInfo.transform.gameObject.layer != 9 && hitInfo.transform.gameObject.layer != 10) { State(null); return; }
91
92          State(hitInfo.transform.gameObject);
93      }
94
95      /// <summary>
96      /// Exhibits different text states based on hit object properties.<br/><br/>
97      /// This text is then written to <seealso cref="labelText"/>.
98      /// </summary>
99      /// <param name="hitObject"></param>
100     private void State(GameObject hitObject)
101     {
102         // Already completed calculations for the same hit object.
103         if (currentHitObject == hitObject) return;
104
105         // Otherwise, restore previoous hit object to default values.
106         if (currentHitObject != null) currentHitObject.GetComponent<MeshRenderer>().material = currentHitObject.layer == 9 ? PreviewManager.Instance.InputMaterial : PreviewManager.Instance.OutputMaterial;
107
108         // UpdateLabelIndex(hitObject) != -1 implies it is an empty input or outrput.
109         if (hitObject != null && UpdateLabelIndex(hitObject) != -1)
110         {
111             // Obtains the label text
112             string newLabelText = isInput ? MenuLogicManager.Instance.CurrentPreviewStructure.InputLabels[labelIndex] : MenuLogicManager.Instance.CurrentPreviewStructure.OutputLabels[labelIndex];
113
114             // Sets text values
115             hitObject.GetComponent<MeshRenderer>().material = selectedMaterial;
116             labelText.text = (isInput ? "input" : "output") + " #" + (labelIndex + 1) + (newLabelText != "" ? " - " + newLabelText : "");
117             currentHitObject = hitObject;
118         }
```

```
119
120         // Is null and/or invalid input/output; restore default values.
121         else
122         {
123             labelText.text = defaultHoverText;
124
125             if (hitObject == null) currentHitObject = null;
126         }
127     }
128
129     /// <summary>
130     /// Obtains the current mouse to world position.
131     /// </summary>
132     /// <returns>The current world position.</returns>
133     public Vector3 UpdateCoordinates()
134     {
135         Ray raycastRay =
              CameraMovementPreview.Instance.PlayerCamera.ScreenPointToRay
              (Input.mousePosition);
136
137         if (coordinatesPlane.Raycast(raycastRay, out float distance))
138         {
139             Vector3 point = raycastRay.GetPoint(distance);
140
141             // If the UI is not locked, also update the coordinates UI
                  text.
142             if (!isUILocked) coordinatesText.text = "(" + point.x.ToString
                  ("0.0") + ", " + point.z.ToString("0.0") + ")";
143
144             return new Vector3(point.x,
                  GridMaintenance.Instance.GridHeight, point.z);
145         }
146
147         throw new Exception("Unable to obtain new mouse position --
              raycast failed.");
148     }
149
150     /// <summary>
151     /// Obtains the index of the empty input or output belonging to the
          given hit object.
152     /// </summary>
153     /// <param name="hitObject">The raycasted game object.</param>
154     /// <returns>The index of the empty input or output.</returns>
155     private int UpdateLabelIndex(GameObject hitObject)
156     {
157         if (hitObject.layer == 9)
158         {
159             isInput = true;
160             labelIndex =
```

```
                MenuLogicManager.Instance.CurrentPreviewStructure.InputOrder
                s[PreviewManager.Instance.Inputs.IndexOf
                (hitObject.GetComponent<CircuitVisualizer.InputReference>
                ().Input)];
161         }
162
163         else
164         {
165             isInput = false;
166             labelIndex =
                MenuLogicManager.Instance.CurrentPreviewStructure.OutputOrde
                rs[PreviewManager.Instance.Outputs.IndexOf
                (hitObject.GetComponent<CircuitVisualizer.OutputReference>
                ().Output)];
167         }
168
169         return labelIndex;
170     }
171
172     // Getter methods
173     public static BehaviorManagerPreview Instance { get { return
          instance; } }
174
175     public bool IsUILocked { get { return isUILocked; } }
176 }
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// CameraMovementPreview handles all player movement within the preview scene.<br/><br/>
6  /// Some behaviors (e.g. scrolling) will be enabled or disabled based on the state of several other scripts.
7  /// </summary>
8  public class CameraMovementPreview : MonoBehaviour
9  {
10     // Singleton state reference
11     private static CameraMovementPreview instance;
12
13     /// <summary>
14     /// The primary camera utilized by the player.
15     /// </summary>
16     [SerializeField]
17     Camera playerCamera;
18
19     /// <summary>
20     /// The speed under which the player can move around scenes.
21     /// </summary>
22     [SerializeField]
23     float movementSpeed;
24
25     /// <summary>
26     /// The speed under which the player can scroll around scenes.
27     /// </summary>
28     [SerializeField]
29     float scrollSpeed;
30
31     /// <summary>
32     /// How low and high the player can vertically go.
33     /// </summary>
34     [SerializeField]
35     float minHeight, maxHeight;
36
37     /// <summary>
38     /// Moves the player up and down respectively.
39     /// </summary>
40     [SerializeField]
41     KeyCode upKey, downKey;
42
43     /// <summary>
44     /// Keeps track of the mouse position in the current frame.
45     /// </summary>
46     private Vector3 mousePosCurrent;
47
```

```
48      // Enforces a singleton state pattern and initializes camera values.
49      private void Awake()
50      {
51          if (instance != null)
52          {
53              Destroy(this);
54              throw new Exception("CameraMovementPreview instance already
                  established; terminating.");
55          }
56
57          instance = this;
58          ClampPos();
59      }
60
61      private void Start() { mousePosCurrent =
            BehaviorManagerPreview.Instance.UpdateCoordinates(); }
62
63      // Listens to key inputs and updates movements each frame.
64      private void Update()
65      {
66          float x, y, z;
67
68          Vector3 mousePosPrev = mousePosCurrent;
69
70          mousePosCurrent =
              BehaviorManagerPreview.Instance.UpdateCoordinates();
71
72          // X-Z movement via mouse drag
73          if (Input.GetMouseButton(0) && !
              BehaviorManagerPreview.Instance.IsUILocked)
74          {
75              Vector3 mousePosDelta = mousePosPrev - mousePosCurrent;
76
77              x = mousePosDelta.x;
78              z = mousePosDelta.z;
79          }
80
81          // X-Z movement via WASD
82          else
83          {
84              // Obtains x and z axis values based on input
85              x = Input.GetAxisRaw("Horizontal") * movementSpeed *
                    Time.deltaTime;
86              z = Input.GetAxisRaw("Vertical") * movementSpeed *
                    Time.deltaTime;
87          }
88
89          // Y movement via scroll wheel
90          if (Mathf.Abs(Input.mouseScrollDelta.y) > 0)
```

```
91          {
92              y = -Input.mouseScrollDelta.y * scrollSpeed * Time.deltaTime;
93          }
94
95          // Y movement via upKey and/or downKey
96          else
97          {
98              y = 0;
99
100             // Determines y axis values (holding both "upKey" and
                  "downKey" will negate one another)
101             if (Input.GetKey(upKey))
102             {
103                 y += movementSpeed * Time.deltaTime;
104             }
105
106             if (Input.GetKey(downKey))
107             {
108                 y -= movementSpeed * Time.deltaTime;
109             }
110         }
111
112         // Adds obtained values and updates position
113         transform.position += x * Vector3.right + y * -CameraRay.direction
                + z * Vector3.forward;
114         ClampPos();
115         mousePosCurrent =
              BehaviorManagerPreview.Instance.UpdateCoordinates();
116     }
117
118     /// <summary>
119     /// Clamps values to ensure the user cannot traverse out of bounds.
120     /// </summary>
121     private void ClampPos()
122     {
123         Vector3 pos = transform.position;
124
125         pos.y = Mathf.Clamp(pos.y, minHeight, maxHeight);
126         transform.position = pos;
127     }
128
129     // Getter methods
130     public static CameraMovementPreview Instance { get { return
          instance; } }
131
132     public Camera PlayerCamera { get { return playerCamera; } }
133
134     private Ray CameraRay { get { return playerCamera.ScreenPointToRay
          (Input.mousePosition); } }
```

```
135 }
```

```
using System;
using System.Collections.Generic;
using TMPro;
using UnityEngine;

/// <summary>
/// PreviewManager deserializes the current preview structure within the preview scene.
/// </summary>
public class PreviewManager : MonoBehaviour
{
    // Singleton state reference
    private static PreviewManager instance;

    /// <summary>
    /// Material used for empty inputs and outputs respectively.
    /// </summary>
    [SerializeField]
    Material inputMaterial,
        outputMaterial;

    /// <summary>
    /// Displays the current input or output label being hovered on, if any.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI nameText;

    /// <summary>
    /// List of instantiated circuits in the scene.
    /// </summary>
    private List<Circuit> circuits = new List<Circuit>();

    /// <summary>
    /// List of all inputs from circuits in the scene.
    /// </summary>
    private List<Circuit.Input> inputs = new List<Circuit.Input>();

    /// <summary>
    /// List of all outputs from circuits in the scene.
    /// </summary>
    private List<Circuit.Output> outputs = new List<Circuit.Output>();

    /// <summary>
    /// The preview structure to deserialize and load into the preview scene.
    /// </summary>
    private PreviewStructure previewStructure;

```

```
47      // Enforces a singleton state pattern
48      private void Awake()
49      {
50          if (instance != null)
51          {
52              Destroy(this);
53              throw new Exception("PreviewManager instance already
                  established; terminating.");
54          }
55
56          instance = this;
57      }
58
59      // Begins the deserialization process
60      private void Start()
61      {
62          CursorManager.SetMouseTexture(true);
63          previewStructure =
                MenuLogicManager.Instance.CurrentPreviewStructure;
64          nameText.text = previewStructure.Name;
65          Deserialize();
66          UpdateIOMaterials();
67      }
68
69      /// <summary>
70      /// Deserializes the current preview structure.<br/><br/>
71      /// The restored values include the circuits, connections, and camera
            position.
72      /// </summary>
73      private void Deserialize()
74      {
75          foreach (CircuitIdentifier circuitIdentifier in
                previewStructure.Circuits) circuits.Add
                (CircuitIdentifier.RestoreCircuit(circuitIdentifier));
76
77          MenuSetupManager.Instance.RestoreConnections(previewStructure);
78          CameraMovementPreview.Instance.PlayerCamera.transform.position =
                previewStructure.CameraLocation;
79      }
80
81      /// <summary>
82      /// Iterates through each circuit in the scene to change the material
            of empty inputs and outputs.
83      /// </summary>
84      private void UpdateIOMaterials()
85      {
86          int inputIndex = 0, outputIndex = 0;
87
88          foreach (Circuit circuit in circuits)
```

```
89          {
90              foreach (Circuit.Input input in circuit.Inputs)
91              {
92                  // If this current input exists in InputOrders, it is
                      guaranteed to be an empty input.
93                  if (previewStructure.InputOrders[inputIndex] != -1)
                      input.Transform.GetComponent<MeshRenderer>().material =
                      inputMaterial;
94
95                  inputs.Add(input);
96                  inputIndex++;
97              }
98
99              foreach (Circuit.Output output in circuit.Outputs)
100             {
101                 // If this current output exists in OutputOrders, it is
                      guaranteed to be an empty output.
102                 if (previewStructure.OutputOrders[outputIndex] != -1)
                      output.Transform.GetComponent<MeshRenderer>().material =
                       outputMaterial;
103
104                 outputs.Add(output);
105                 outputIndex++;
106             }
107         }
108     }
109
110     // Getter methods
111     public static PreviewManager Instance { get { return instance; } }
112
113     public List<Circuit> Circuits { get { return circuits; } }
114
115     public List<Circuit.Input> Inputs { get { return inputs; } }
116
117     public List<Circuit.Output> Outputs { get { return outputs; } }
118
119     public Material InputMaterial { get { return inputMaterial; } }
120
121     public Material OutputMaterial { get { return outputMaterial; } }
122 }
```

```cs
using System;
using UnityEngine;

/// <summary>
/// CircuitConnectorIdentifier serializes a connection by storing relevant index values.
/// </summary>
[Serializable]
public class CircuitConnectorIdentifier
{
    // Serialized values that are provided within the primary constructor.
    [SerializeField]
    int inputCircuitIndex,
        outputCircuitIndex,
        inputIndex,
        outputIndex;

    /// <param name="inputCircuitIndex">The ordered index of the circuit that this input is a part of.</param>
    /// <param name="outputCircuitIndex">The ordered index of the circuit that this output is a part of.</param>
    /// <param name="inputIndex">The ordered index of the circuit's inputs that this input is a part of.</param>
    /// <param name="outputIndex">The ordered index of the circuit's outputs that this output is a part of.</param>
    public CircuitConnectorIdentifier(int inputCircuitIndex, int outputCircuitIndex, int inputIndex, int outputIndex)
    {
        this.inputCircuitIndex = inputCircuitIndex;
        this.outputCircuitIndex = outputCircuitIndex;
        this.inputIndex = inputIndex;
        this.outputIndex = outputIndex;
    }

    // Getter methods
    public int InputCircuitIndex { get { return inputCircuitIndex; } }

    public int OutputCircuitIndex { get { return outputCircuitIndex; } }

    public int InputIndex { get { return inputIndex; } }

    public int OutputIndex { get { return outputIndex; } }
}
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// CircuitIdentifier provides circuit serialization and deserialization
     by storing relevant enum values.
6  /// </summary>
7  [Serializable]
8  public class CircuitIdentifier
9  {
10     /// <summary>
11     /// CircuitType is the serialized representation of any circuit
         located within a scene.
12     /// </summary>
13     public enum CircuitType { CUSTOM_CIRCUIT, INPUT_GATE, DISPLAY, BUFFER,
          AND_GATE, NAND_GATE, NOR_GATE, NOT_GATE, OR_GATE, XOR_GATE }
14
15     /// <summary>
16     /// The circuit type to be serialized by this CircuitIdentifier
         instance.
17     /// </summary>
18     [SerializeField]
19     public CircuitType circuitType;
20
21     /// <summary>
22     /// Contains a valid custom circuit ID if the referenced CircuitType
         value is <seealso cref="CircuitType.CUSTOM_CIRCUIT"/>.
23     /// </summary>
24     [SerializeField]
25     public int previewStructureID = -1;
26
27     /// <summary>
28     /// The location of the circuit within the scene.
29     /// </summary>
30     [SerializeField]
31     Vector2 circuitLocation;
32
33     /// <param name="circuit">The circuit to serialize.</param>
34     public CircuitIdentifier(Circuit circuit)
35     {
36         Vector3 pos = circuit.PhysicalObject.transform.position;
37
38         circuitType = CircuitToCircuitType(circuit);
39         circuitLocation = new Vector2(pos.x, pos.z);
40
41         // If the circuit is a custom one, store its ID
42         if (circuitType == CircuitType.CUSTOM_CIRCUIT) previewStructureID
             = ((CustomCircuit)circuit).PreviewStructure.ID;
43     }
```

```
44
45      /// <summary>
46      /// Instantiates and returns a <see cref="Circuit"/> based on the
          provided CircuitIdentifier (using default visibility settings).
47      /// </summary>
48      /// <param name="circuitIdentifier">The CircuitIdentifier to access
          and reference.</param>
49      /// <returns>The instantiated <seealso cref="Circuit"/>.</returns>
50      public static Circuit RestoreCircuit(CircuitIdentifier
          circuitIdentifier)
51      {
52          return RestoreCircuit(circuitIdentifier, true);
53      }
54
55      /// <summary>
56      /// Instantiates and returns a <see cref="Circuit"/> based on the
          provided CircuitIdentifier.
57      /// </summary>
58      /// <param name="circuitIdentifier">The CircuitIdentifier to access
          and reference.</param>
59      /// <param name="visible">False if the circuit is located inside of a
          custom circuit, otherwise true.</param>
60      /// <returns>The instantiated <seealso cref="Circuit"/>.</returns>
61      public static Circuit RestoreCircuit(CircuitIdentifier
          circuitIdentifier, bool visible)
62      {
63          Vector2 pos = visible ? circuitIdentifier.circuitLocation :
              Vector2.positiveInfinity;
64
65          switch (circuitIdentifier.circuitType)
66          {
67              case CircuitType.CUSTOM_CIRCUIT:
68                  return new CustomCircuit
                      (MenuSetupManager.Instance.PreviewStructures
                      [MenuSetupManager.Instance.PreviewStructureIDs.IndexOf
                      (circuitIdentifier.previewStructureID)], pos, visible);
69              case CircuitType.INPUT_GATE:
70                  return new InputGate(pos);
71              case CircuitType.DISPLAY:
72                  return new Display(pos);
73              case CircuitType.BUFFER:
74                  return new Buffer(pos);
75              case CircuitType.AND_GATE:
76                  return new AndGate(pos);
77              case CircuitType.NAND_GATE:
78                  return new NAndGate(pos);
79              case CircuitType.NOR_GATE:
80                  return new NOrGate(pos);
81              case CircuitType.NOT_GATE:
```

```
82                  return new NotGate(pos);
83              case CircuitType.OR_GATE:
84                  return new OrGate(pos);
85              case CircuitType.XOR_GATE:
86                  return new XOrGate(pos);
87              default:
88                  throw new Exception("Invalid circuit type.");
89          }
90      }
91
92      /// <summary>
93      /// Converts the provided <see cref="Circuit"/> to a valid <seealso cref="CircuitType"/> for serialization.
94      /// </summary>
95      /// <param name="circuit">The circuit to convert.</param>
96      /// <returns>The converted <seealso cref="CircuitType"/>.</returns>
97      private static CircuitType CircuitToCircuitType(Circuit circuit)
98      {
99          Type type = circuit.GetType();
100
101         if (type == typeof(CustomCircuit)) return CircuitType.CUSTOM_CIRCUIT;
102
103         if (type == typeof(InputGate)) return CircuitType.INPUT_GATE;
104
105         if (type == typeof(Display)) return CircuitType.DISPLAY;
106
107         if (type == typeof(Buffer)) return CircuitType.BUFFER;
108
109         if (type == typeof(AndGate)) return CircuitType.AND_GATE;
110
111         if (type == typeof(NAndGate)) return CircuitType.NAND_GATE;
112
113         if (type == typeof(NOrGate)) return CircuitType.NOR_GATE;
114
115         if (type == typeof(NotGate)) return CircuitType.NOT_GATE;
116
117         if (type == typeof(OrGate)) return CircuitType.OR_GATE;
118
119         if (type == typeof(XOrGate)) return CircuitType.XOR_GATE;
120
121         throw new Exception("Invalid circuit type.");
122     }
123 }
```

```cs
using System;
using System.IO;
using UnityEngine;

/// <summary>
/// ConnectionSerializer stores mesh and index information pertaining to the assigned connection for serialization.
/// </summary>
[Serializable]
public class ConnectionSerializer
{
    /// <summary>
    /// Whether <seealso cref="startingMesh"/> is equal to <seealso cref="endingMesh"/>.
    /// </summary>
    [SerializeField]
    bool singleWired;

    /// <summary>
    /// Contains relevant index information used to identify the connection's input and output circuit(s).
    /// </summary>
    [SerializeField]
    CircuitConnectorIdentifier circuitConnectorIdentifier;

    /// <summary>
    /// Serialized mesh data for the starting wire.
    /// </summary>
    [SerializeField]
    MeshSerializer startingMesh;

    /// <summary>
    /// Serialized mesh data for the ending wire.
    /// </summary>
    [SerializeField]
    MeshSerializer endingMesh;

    /// <summary>
    /// Serialized mesh data for the non-starting/ending wires.<br/><br/>
    /// If there is only a starting and ending wire, its value will be null.
    /// </summary>
    [SerializeField]
    MeshSerializer parentMesh;

    // Private constructor; a ConnectionSerializer can only be instantiated through its primary constructor.
    private ConnectionSerializer() { }

```

```
45      /// <summary>
46      /// Instantiates and populates a <seealso cref="ConnectionSerializer"/>
            with the assigned values; saves to the provided path.
47      /// </summary>
48      /// <param name="connection">The connection to serialize.</param>
49      /// <param name="circuitConnectorIdentifier">The obtained <seealso
            cref="CircuitConnectorIdentifier"/> representing this connection.</
            param>
50      /// <param name="path">The directory to save the serialized
            information.</param>
51      public static void SerializeConnection(CircuitConnector.Connection
            connection, CircuitConnectorIdentifier circuitConnectorIdentifier,
            string path)
52      {
53          ConnectionSerializer connectionSerializer = new
                ConnectionSerializer();
54
55          // Assigns relevant values
56          connectionSerializer.circuitConnectorIdentifier =
                circuitConnectorIdentifier;
57          connectionSerializer.startingMesh = new MeshSerializer
                (connection.StartingWire.transform.GetChild(0).GetChild
                (0).GetComponent<MeshFilter>().mesh,
                connection.StartingWire.transform);
58          connectionSerializer.endingMesh = new MeshSerializer
                (connection.EndingWire.transform.GetChild(0).GetChild
                (0).GetComponent<MeshFilter>().mesh,
                connection.EndingWire.transform);
59          connectionSerializer.singleWired = connection.StartingWire ==
                connection.EndingWire;
60
61          // If the connection has a parent mesh, serialize its information
                and save to parentMesh.
62          if (connection.GetComponent<MeshFilter>() != null)
                connectionSerializer.parentMesh = new MeshSerializer
                (connection.GetComponent<MeshFilter>().mesh,
                connection.transform);
63
64          // Writes object to directory
65          File.WriteAllText(path, JsonUtility.ToJson(connectionSerializer));
66      }
67
68      // Getter methods
69      public bool SingleWired { get { return singleWired; } }
70
71      public CircuitConnectorIdentifier CircuitConnectorIdentifier { get
            { return circuitConnectorIdentifier; } }
72
73      public MeshSerializer StartingMesh { get { return startingMesh; } }
```

```
74
75      public MeshSerializer EndingMesh { get { return endingMesh; } }
76
77      public MeshSerializer ParentMesh { get { return parentMesh; } }
78  }
```

```cs
using System;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// EditorStructure contains all serializable values to restore an editor scene.
/// </summary>
[Serializable]
public class EditorStructure
{
    /// <summary>
    /// Whether grid snapping is enabled.
    /// </summary>
    [SerializeField]
    bool inGridMode;

    /// <summary>
    /// The list of circuits determined to be powered inputs.
    /// </summary>
    [SerializeField]
    List<bool> isPoweredInput = new List<bool>();

    /// <summary>
    /// The list of circuit identifiers pertaining to each circuit within the scene.
    /// </summary>
    [SerializeField]
    List<CircuitIdentifier> circuits = new List<CircuitIdentifier>();

    /// <summary>
    /// The circuit index representation of all bookmarked circuits.<br/><br/>
    /// All indeces except that of custom circuits are unique.
    /// </summary>
    [SerializeField]
    List<int> bookmarks = new List<int>();

    /// <summary>
    /// The custom circuit ID representation of all bookmarked circuits.<br/><br/>
    /// If a bookmarked circuit is not a custom circuit, -1 is utilized as a corresponding identifier.
    /// </summary>
    [SerializeField]
    List<int> bookmarkIDs = new List<int>();

    /// <summary>
    /// Name of the editor scene assigned when a save slot is first used.
```

```
45      /// </summary>
46      [SerializeField]
47      string name;
48
49      /// <summary>
50      /// Location of the camera within the editor scene.
51      /// </summary>
52      [SerializeField]
53      Vector3 cameraLocation;
54
55      public EditorStructure(string name) { this.name = name; }
56
57      // Getter and setter methods
58      public bool InGridMode { get { return inGridMode; } set { inGridMode =
          value; } }
59
60      public List<bool> IsPoweredInput { get { return isPoweredInput; } set
          { isPoweredInput = value; } }
61
62      public List<CircuitIdentifier> Circuits { get { return circuits; } set
          { circuits = value; } }
63
64      public List<int> Bookmarks { get { return bookmarks; } set { bookmarks
          = value; } }
65
66      public List<int> BookmarkIDs { get { return bookmarkIDs; } set
          { bookmarkIDs = value; } }
67
68      public Vector3 CameraLocation { get { return cameraLocation; } set
          { cameraLocation = value; } }
69
70      // Getter method
71      public string Name { get { return name; } }
72  }
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// InternalConnection serializes all connections occurring in a custom
     circuit.
6  /// </summary>
7  [Serializable]
8  public class InternalConnection
9  {
10     /// <summary>
11     /// The input and output indeces of the connection.<br/><br/>
12     /// These values are referenced once the <see cref="Circuit.Input"/>
         and <see cref="Circuit.Output"/> lists of the <see
         cref="CustomCircuit"/> are established.
13     /// </summary>
14     [SerializeField] int inputIndex, outputIndex;
15
16     public InternalConnection(int inputIndex, int outputIndex)
17     {
18         this.inputIndex = inputIndex;
19         this.outputIndex = outputIndex;
20     }
21
22     // Getter methods
23     public int InputIndex { get { return inputIndex; } }
24
25     public int OutputIndex { get { return outputIndex; } }
26 }
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// MeshSerializer serves as a wrapper class for containing all relevant
     mesh and transform values for serialization and deserialization of wires.
6  /// </summary>
7  [Serializable]
8  public class MeshSerializer
9  {
10     [SerializeField]
11     int[] triangles;
12
13     [SerializeField]
14     Vector2[] uv;
15
16     [SerializeField]
17     Vector3 position, rotation, scale;
18
19     [SerializeField]
20     Vector3[] normals, vertices;
21
22     /// <summary>
23     /// Extracts mesh and transform values pertaining to a wire.
24     /// </summary>
25     /// <param name="mesh">The mesh to extract relevant values from.</
         param>
26     /// <param name="parentTransform">The parent transform of the
         GameObject containing the mesh (could be itself).</param>
27     public MeshSerializer(Mesh mesh, Transform parentTransform)
28     {
29         triangles = mesh.triangles;
30         uv = mesh.uv;
31         position = parentTransform.position;
32         rotation = parentTransform.eulerAngles;
33         scale = parentTransform.localScale;
34         normals = mesh.normals;
35         vertices = mesh.vertices;
36     }
37
38     // Getter methods
39     public int[] Triangles { get { return triangles; } }
40
41     public Vector2[] UV { get { return uv; } }
42
43     public Vector3 Position { get { return position; } }
44
45     public Vector3 Rotation { get { return rotation; } }
46
```

```
47      public Vector3 Scale { get { return scale; } }
48
49      public Vector3[] Normals { get { return normals; } }
50
51      public Vector3[] Vertices { get { return vertices; } }
52  }
```

```
1  using System;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  /// <summary>
6  /// PreviewStructure contains all serializable values to restore a preview
     scene.
7  /// </summary>
8  [Serializable]
9  public class PreviewStructure
10 {
11     /// <summary>
12     /// The list of circuit identifiers pertaining to each circuit within
         the scene.
13     /// </summary>
14     [SerializeField]
15     List<CircuitIdentifier> circuits = new List<CircuitIdentifier>();
16
17     /// <summary>
18     /// The unique ID assigned to this preview structure.<br/><br/>
19     /// Functionally, this ID is utilized to access the specific folder
         under which the connection and save information of the preview
         structure is.
20     /// </summary>
21     [SerializeField]
22     int id;
23
24     /// <summary>
25     /// The order in which empty inputs and outputs were selected by the
         user.<br/><br/>
26     /// Functionally, a visualized custom circuit will output these inputs
         and outputs in their selected order (bottom to top).
27     /// </summary>
28     [SerializeField]
29     List<int> inputOrders,
30         outputOrders;
31
32     /// <summary>
33     /// Identifying list of connections that exist within the custom
         circuit.
34     /// </summary>
35     [SerializeField]
36     List<InternalConnection> connections;
37
38     /// <summary>
39     /// Name of the custom circuit.
40     /// </summary>
41     [SerializeField]
42     string name;
```

```
43
44      /// <summary>
45      /// The corresponding user-assigned label for each empty input/output.
46      /// </summary>
47      [SerializeField]
48      List<string> inputLabels,
49          outputLabels;
50
51      /// <summary>
52      /// Location of the camera within the editor scene.
53      /// </summary>
54      [SerializeField]
55      Vector3 cameraLocation;
56
57      public PreviewStructure(string name) { this.name = name; }
58
59      /// <summary>
60      /// Internal class utilized to obtain the custom circuit ID via in-
          scene raycasting.
61      /// </summary>
62      public class PreviewStructureReference : MonoBehaviour
63      {
64          private int id;
65
66          public int ID { get { return id; } set { id = value; } }
67      }
68
69      // Getter and setter methods
70      public List<CircuitIdentifier> Circuits { get { return circuits; } set
          { circuits = value; } }
71
72      public int ID { get { return id; } set { id = value; } }
73
74      public List<int> InputOrders { get { return inputOrders; } set
          { inputOrders = value; } }
75
76      public List<int> OutputOrders { get { return outputOrders; } set
          { outputOrders = value; } }
77
78      public List<InternalConnection> Connections { get { return
          connections; } set { connections = value; } }
79
80      public List<string> InputLabels { get { return inputLabels; } set
          { inputLabels = value; } }
81
82      public List<string> OutputLabels { get { return outputLabels; } set
          { outputLabels = value; } }
83
84      public Vector3 CameraLocation { get { return cameraLocation; } set
```

```
        { cameraLocation = value; } }

    // Getter method
    public string Name { get { return name; } }
}
```

```
 1  using System;
 2  using System.Collections;
 3  using System.Collections.Generic;
 4  using UnityEngine;
 5
 6  /// <summary>
 7  /// CircuitCaller handles every circuit call after a short delay defined
      in <see cref="Circuit"/>.
 8  /// </summary>
 9  public class CircuitCaller : MonoBehaviour
10  {
11      private static CircuitCaller instance; // Singleton state reference
12
13      // Enforces a singleton state pattern
14      private void Awake()
15      {
16          if (instance != null)
17          {
18              Destroy(this);
19              throw new Exception("CircuitCaller instance already
                  established; terminating.");
20          }
21
22          instance = this;
23      }
24
25      /// <summary>
26      /// Starts a coroutine that shortly accesses the list of provided
          update calls.
27      /// </summary>
28      /// <param name="updateCalls">The list of update calls to pursue.</
          param>
29      public static void InitiateUpdateCalls(List<Circuit.UpdateCall>
          updateCalls) { instance.StartCoroutine(UpdateCalls(updateCalls)); }
30
31      /// <summary>
32      /// Attempts to access the list of provided update calls.
33      /// </summary>
34      /// <param name="updateCalls">The list of update calls to call.</
          param>
35      private static IEnumerator UpdateCalls(List<Circuit.UpdateCall>
          updateCalls)
36      {
37          yield return new WaitForSeconds(Circuit.clockSpeed);
38
39          foreach (Circuit.UpdateCall updateCall in updateCalls)
40          {
41              // Sometime between the call initiation and now, the
                    referenced output was destroyed and should no longer be
```

```
                pursued.
42              if (updateCall.Input.ParentOutput == null) continue;
43
44              if (!CustomCircuitTest(updateCall)) continue;
45
46              // Otherwise, the update call is accessed to update the
                  relevant circuits.
47              Circuit.UpdateCircuit(updateCall.Powered, updateCall.Input,
                  updateCall.Output);
48          }
49      }
50
51      /// <summary>
52      /// Ensures that an update call pertaining to a custom circuit only
          runs if its custom circuit is not deleted.
53      /// </summary>
54      /// <param name="updateCall">The update call to test.</param>
55      /// <returns>Whether this update call should be utilized.</returns>
56      private static bool CustomCircuitTest(Circuit.UpdateCall updateCall)
57      {
58          // In preview scene, therefore not necessary to run the test
59          if (EditorStructureManager.Instance == null) return true;
60
61          // If the input of an update call is under a parent circuit, it is
               guaranteed that its output is as well.
62          bool isInternalConnection =
                updateCall.Input.ParentCircuit.customCircuit != null;
63
64          // Connection does not pertain to the inside of a custom circuit.
65          if (!isInternalConnection) return true;
66
67          // Otherwise, obtain the top-most custom circuit and check to see
                if it is still within the scene.
68          CustomCircuit customCircuitParent =
                updateCall.Input.ParentCircuit.customCircuit;
69
70          while (customCircuitParent.customCircuit != null)
                customCircuitParent = customCircuitParent.customCircuit;
71
72          return !customCircuitParent.ShouldDereference;
73      }
74
75      /// <summary>
76      /// Deletes the specified circuit from the scene.
77      /// </summary>
78      /// <param name="circuit">The circuit to destroy.</param>
79      public static void Destroy(Circuit circuit)
80      {
81          // First disconnects any potential input connections
```

```
82          foreach (Circuit.Input input in circuit.Inputs)
83          {
84              if (input.Connection != null)
85              {
86                  CircuitConnector.Disconnect(input.Connection);
87              }
88          }
89
90          // Then disconnects any potential output connections
91          foreach (Circuit.Output output in circuit.Outputs)
92          {
93              foreach (CircuitConnector.Connection connection in new
                  List<CircuitConnector.Connection>(output.Connections))
94              {
95                  CircuitConnector.Disconnect(connection);
96              }
97          }
98
99          // Ensures all remaining calls within the custom circuit are
              skipped
100         if (circuit.GetType() == typeof(CustomCircuit)) ((CustomCircuit)
              circuit).ShouldDereference = true;
101
102         EditorStructureManager.Instance.DisplaySavePrompt = true; //
              Destroying a circuit triggers the save prompt
103         EditorStructureManager.Instance.Circuits.Remove(circuit); //
              Removes circuit for potential serialization
104         Destroy(circuit.PhysicalObject);
105     }
106 }
```

```
1  using System;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  /// <summary>
6  /// CircuitConnector facilitates the connection process between circuits
     in the scene editor.
7  /// </summary>
8  public class CircuitConnector : MonoBehaviour
9  {
10     // Singleton state reference
11     private static CircuitConnector instance;
12
13     /// <summary>
14     /// Reference to the wire prefab.
15     /// </summary>
16     [SerializeField]
17     GameObject wireReference;
18
19     /// <summary>
20     /// The material for powered and unpowered statuses respectively.
21     /// </summary>
22     [SerializeField]
23     Material poweredMaterial, unpoweredMaterial;
24
25     private bool cancelled;
26
27     private Connection currentConnection;
28
29     private GameObject currentWire;
30
31     private int count;
32
33     private Vector3 startingWirePos, currentPos;
34
35     /// <summary>
36     /// Represents a connection from the output circuit to the input
       circuit.
37     /// </summary>
38     public class Connection : MonoBehaviour
39     {
40         /// <summary>
41         /// The input associated with the connection.
42         /// </summary>
43         private Circuit.Input input;
44
45         /// <summary>
46         /// The output associated with the connection.
47         /// </summary>
```

```
48          private Circuit.Output output;
49
50          /// <summary>
51          /// The starting and ending wires associated with the connection.
52          /// </summary>
53          private GameObject endingWire, startingWire;
54
55          // Getter and setter methods
56          public Circuit.Input Input { get { return input; } set { input =
              value; } }
57
58          public Circuit.Output Output { get { return output; } set { output
               = value; } }
59
60          public GameObject EndingWire { get { return endingWire; } set
              { endingWire = value; } }
61
62          public GameObject StartingWire { get { return startingWire; } set
              { startingWire = value; } }
63      }
64
65      // Enforces a singleton state pattern and disables update calls.
66      private void Awake()
67      {
68          if (instance != null)
69          {
70              Destroy(this);
71              throw new Exception("CircuitConnector instance already
                  established; terminating.");
72          }
73
74          instance = this;
75          enabled = false;
76      }
77
78      // While the connection has not been cancelled or completed, this
          method allows for creating pivots to organize the wire.
79      private void Update()
80      {
81          // If the connection process has been cancelled, disable update
              calls and return.
82          if (cancelled)
83          {
84              cancelled = enabled = false;
85              return;
86          }
87
88          // If the game is currently paused, skip frame.
89          if (BehaviorManager.Instance.CurrentStateType ==
```

```
            BehaviorManager.StateType.PAUSED) return;
90
91          // Whether the user is staring at a valid input or output for completing the connection.
92          bool staringAtIO = Physics.Raycast
              (CameraMovement.Instance.PlayerCamera.ScreenPointToRay
              (Input.mousePosition), out RaycastHit hitInfo) &&
              hitInfo.transform.gameObject.layer ==
              BehaviorManager.Instance.IOLayerCheck;
93
94          // The position to move the end of the wire to.
95          // If hovered onto a valid input or output for completing the connection, it will snap to its position.
96          Vector3 pos = staringAtIO ? hitInfo.transform.position :
              Coordinates.Instance.ModePos;
97
98          pos.y = GridMaintenance.Instance.GridHeight;
99          UpdatePosition(currentWire, currentPos, pos); // Updates the position of the wire.
100
101         // Creates a new pivot as long as the wire is active (has a length >= 0).
102         if (Input.GetMouseButtonDown(0) && currentWire.activeSelf)
103         {
104             count++;
105
106             if (count == 2) startingWirePos = currentPos;
107
108             currentConnection.EndingWire = currentWire;
109
110             // Places a new wire at the current pivot
111             InstantiateWire(currentConnection,
                  Coordinates.Instance.ModePos);
112         }
113     }
114
115     /// <summary>
116     /// Final step in restoring the logic of a serialized connection by initializing a <seealso cref="Connection"/> instance and assigning all of its values.
117     /// </summary>
118     /// <param name="prefab">The base GameObject of the connection.</param>
119     /// <param name="input">The input of the connection.</param>
120     /// <param name="output">The output of the connection.</param>
121     /// <param name="endingWire">The ending wire of the connection.</param>
122     /// <param name="startingWire">The starting wire of the connection.</param>
```

```
123     /// <param name="isEditor">Whether the connection is being restored in
            the editor.</param>
124     public static void ConnectRestoration(GameObject prefab, Circuit.Input
            input, Circuit.Output output, GameObject endingWire, GameObject
          startingWire, bool isEditor)
125     {
126         Connection connection = prefab.AddComponent<Connection>();
127
128         connection.Input = input;
129         connection.Output = output;
130         input.Connection = connection;
131         input.ParentOutput = output;
132         output.Connections.Add(connection);
133         output.ChildInputs.Add(input);
134         connection.EndingWire = endingWire;
135         connection.StartingWire = startingWire;
136
137         if (isEditor) EditorStructureManager.Instance.Connections.Add
              (connection); // Re-adds connection for potential serialization
138
139         // If the circuit is an input gate, do not pursue an update call.
140         if (output.ParentCircuit.GetType() == typeof(InputGate)) return;
141
142         Circuit.UpdateCircuit(input, output);
143     }
144
145     // Finalizes the current connection in progress.
146     public static void Connect(Circuit.Input input, Circuit.Output output)
147     {
148         Instance.count = -1;
149         Instance.currentConnection.Input = input;
150         Instance.currentConnection.Output = output;
151         Instance.currentConnection.Input.Connection =
              Instance.currentConnection;
152         Instance.currentConnection.Output.Connections.Add
              (Instance.currentConnection);
153         Instance.currentConnection.Output.ChildInputs.Add(input);
154         Instance.currentConnection.EndingWire.name = "Ending Wire";
155         Instance.OptimizeMeshes();
156         EditorStructureManager.Instance.Connections.Add
              (Instance.currentConnection); // Adds connection for potential
              serialization
157         Instance.currentConnection = null; Instance.currentWire = null;
158         Instance.cancelled = true;
159         Circuit.UpdateCircuit(input, output);
160     }
161
162     /// <summary>
163     /// Removes a connection from the editor scene.
```

```
164     /// </summary>
165     /// <param name="connection"></param>
166     public static void Disconnect(Connection connection)
167     {
168         EditorStructureManager.Instance.DisplaySavePrompt = true;
169         EditorStructureManager.Instance.Connections.Remove(connection); // Removes connection for potential serialization
170         Circuit.UpdateCircuit(false, connection.Input, null);
171         connection.Input.Connection = null;
172         connection.Output.Connections.Remove(connection);
173         connection.Output.ChildInputs.Remove(connection.Input);
174         Destroy(connection.gameObject);
175     }
176 
177     /// <summary>
178     /// Updates all wire materials pertaining to a connection, if applicable.
179     /// </summary>
180     /// <param name="connection"></param>
181     /// <param name="powered"></param>
182     public static void UpdateConnectionMaterial(Connection connection, bool powered)
183     {
184         // If there is an optimized mesh in the wire, update it.
185         if (connection.GetComponent<MeshRenderer>() != null)
              connection.GetComponent<MeshRenderer>().material = powered ? Instance.poweredMaterial : Instance.unpoweredMaterial;
186 
187         // If there is a starting mesh in the wire, update it.
188         if (connection.StartingWire != null)
              connection.StartingWire.GetComponentInChildren<MeshRenderer>().material = powered ? Instance.poweredMaterial : Instance.unpoweredMaterial;
189 
190         // If there is an ending mesh in the wire, update it.
191         if (connection.EndingWire != null)
              connection.EndingWire.GetComponentInChildren<MeshRenderer>().material = powered ? Instance.poweredMaterial : Instance.unpoweredMaterial;
192     }
193 
194     /// <summary>
195     /// Begins the connection process at the specified position.
196     /// </summary>
197     /// <param name="pos"></param>
198     public void BeginConnectionProcess(Vector3 pos)
199     {
200         count = 0;
201         cancelled = false;
```

```
202             currentConnection = InstantiateConnection();
203             InstantiateWire(currentConnection, pos);
204             currentConnection.StartingWire = currentWire;
205             currentWire.name = "Starting Wire";
206             enabled = true; // Enables frame-by-frame update calls from Unity
207         }
208
209         /// <summary>
210         /// Cancels the current connection process.
211         /// </summary>
212         public void CancelConnectionProcess()
213         {
214             count = -1;
215             cancelled = true;
216             Destroy(currentConnection.gameObject);
217         }
218
219         /// <summary>
220         /// Creates a new wire at the specified position for the given
              connection.
221         /// </summary>
222         /// <param name="connection">The connection this wire is a part of.</
              param>
223         /// <param name="a">The startring position to instantiate this wire
              at.</param>
224         private void InstantiateWire(Connection connection, Vector3 a)
225         {
226             currentWire = Instantiate(wireReference, connection.transform);
227             currentPos = new Vector3(a.x, GridMaintenance.Instance.GridHeight,
                  a.z);
228             currentWire.transform.position = currentPos;
229             currentWire.SetActive(false);
230         }
231
232         /// <summary>
233         /// Specific signature of <seealso cref="UpdatePosition(GameObject,
              Vector3, Vector3, bool)"/> under which isCentered is always false.
234         /// </summary>
235         /// <param name="wire">The wire to move.</param>
236         /// <param name="a">The starting position.</param>
237         /// <param name="b">The ending position.</param>
238         public static void UpdatePosition(GameObject wire, Vector3 a, Vector3
              b) { UpdatePosition(wire, a, b, false); }
239
240         /// <summary>
241         /// Updates the start and end positions of the specified wire.
242         /// </summary>
243         /// <param name="wire">The wire to move.</param>
244         /// <param name="a">The starting position.</param>
```

```
245     /// <param name="b">The ending position.</param>
246     /// <param name="isCentered">Whether the wire should be centered.</
          param>
247     public static void UpdatePosition(GameObject wire, Vector3 a, Vector3
          b, bool isCentered)
248     {
249         // If the wire is centered, then startingWire == endingWire and it
              must be positioned differently.
250         if (isCentered) wire.transform.position = (a + b) / 2;
251
252         wire.transform.localScale = new Vector3(1, 1, (a - b).magnitude);
253         wire.SetActive(wire.transform.localScale.z != 0);
254         wire.transform.LookAt(b);
255
256         // Ensures the height of the wire does not exceed the global grid
              height
257         Vector3 temp = wire.transform.position;
258
259         temp.y = GridMaintenance.Instance.GridHeight;
260         wire.transform.position = temp;
261     }
262
263     /// <summary>
264     /// Optimizes all non-starting and non-ending wire meshes by merging
          them together into a single mesh.
265     /// </summary>
266     private void OptimizeMeshes()
267     {
268         // There is a single wire in the connection
269         if (currentConnection.StartingWire ==
              currentConnection.EndingWire)
270         {
271             Destroy(currentWire);
272
273             // If there is a single wire in the connection, it must be
                  centered so UpdatePosition() can work properly.
274             currentConnection.StartingWire.transform.position =
                  (currentConnection.Input.Transform.position +
                  currentConnection.Output.Transform.position) / 2;
275
276             // Furthermore, the pivot must be altered.
277             currentConnection.StartingWire.transform.GetChild
                  (0).transform.localPosition = Vector3.back * 0.5f;
278
279             // Ensures the height of the wire does not exceed the global
                  grid height
280             Vector3 temp =
                  currentConnection.StartingWire.transform.position;
281
```

```
282             temp.y = GridMaintenance.Instance.GridHeight;
283             currentConnection.StartingWire.transform.position = temp;
284             return;
285         }
286 
287         // Ensures the starting wire behaves properly with the
              UpdatePosition() method
288         currentConnection.StartingWire.transform.position =
              startingWirePos;
289         currentConnection.StartingWire.transform.eulerAngles += Vector3.up
               * 180;
290 
291         // Checks to see whether there are exactly 3 wires.
292         // 2 of them are the starting and ending wires (cannot be merged
              into a mesh) and the third wire is the placement wire, which
              will be deleted regardless.
293         if (currentConnection.transform.childCount == 3)
294         {
295             Destroy(currentWire);
296             return;
297         }
298 
299         // Begins the mesh combination process
300         List<CombineInstance> combineInstances = new List<CombineInstance>
              ();
301 
302         foreach (Transform child in currentConnection.transform)
303         {
304             GameObject childObj = child.gameObject;
305 
306             if (childObj == currentConnection.StartingWire || childObj ==
                  currentConnection.EndingWire || childObj == currentWire)
                  continue;
307 
308             MeshFilter meshFilter =
                  childObj.GetComponentInChildren<MeshFilter>();
309             CombineInstance combineInstance = new CombineInstance();
310 
311             combineInstance.mesh = meshFilter.mesh;
312             combineInstance.transform =
                  meshFilter.transform.localToWorldMatrix;
313 
314             combineInstances.Add(combineInstance);
315         }
316 
317         Mesh combinedMesh = new Mesh();
318 
319         combinedMesh.CombineMeshes(combineInstances.ToArray());
320 
```

```
321         // Deletes the original instances of the unmerged meshes.
322         foreach (Transform child in currentConnection.transform)
323         {
324             GameObject childObj = child.gameObject;
325 
326             if (childObj == currentConnection.StartingWire || childObj ==
                  currentConnection.EndingWire) continue;
327 
328             Destroy(childObj);
329         }
330 
331         MeshFilter combinedMeshFilter =
              currentConnection.gameObject.AddComponent<MeshFilter>();
332 
333         currentConnection.gameObject.AddComponent<MeshRenderer>();
334         combinedMeshFilter.mesh = combinedMesh;
335         currentConnection.gameObject.layer = 11;
336         currentConnection.gameObject.AddComponent<MeshCollider>();
337     }
338 
339     /// <summary>
340     /// Creates a new connection GameObject.
341     /// </summary>
342     /// <returns>The connection component of a newly instantiated
        GameObject.</returns>
343     private Connection InstantiateConnection() { return new GameObject
        ("Connection").AddComponent<Connection>(); }
344 
345     // Getter methods
346     public static CircuitConnector Instance { get { return instance; } }
347 
348     public Connection CurrentConnection { get { return
        currentConnection; } }
349 }
```

```
1  using UnityEngine;
2
3  /// <summary>
4  /// CircuitReference is a <see cref="Circuit"/> wrapper class attached to
     all instantiated circuit GameObjects.<br/><br/>
5  /// This script is utilized to obtain the target circuit for any given
     GameObject through raycasts.
6  /// </summary>
7  public class CircuitReference : MonoBehaviour
8  {
9      /// <summary>
10     /// The circuit that this
11     /// </summary>
12     private Circuit circuit;
13
14     // Getter and setter method
15     public Circuit Circuit { get { return circuit; } set { circuit =
         value; } }
16 }
```

```
1  using System;
2  using TMPro;
3  using UnityEngine;
4
5  /// <summary>
6  /// CircuitVisualizer generates meshes for both circuits and custom
     circuits.
7  /// </summary>
8  public class CircuitVisualizer : MonoBehaviour
9  {
10     // Singleton state reference
11     private static CircuitVisualizer instance;
12
13     /// <summary>
14     /// The color associated with starting and custom circuits.
15     /// </summary>
16     [SerializeField]
17     Color startingCircuitColor,
18         customCircuitColor;
19
20     /// <summary>
21     /// Thickness of the border surrounding the base of a circuit.
22     /// </summary>
23     [SerializeField]
24     float borderThickness; // Border surrounding the base of the circuit
25
26     /// <summary>
27     /// Square dimensions of an input node.
28     /// </summary>
29     [SerializeField]
30     float inputSize;
31
32     /// <summary>
33     /// Square dimensions of an output node.
34     /// </summary>
35     [SerializeField]
36     float outputSize;
37
38     /// <summary>
39     /// Square dimensions of the power indicator on input and output
         nodes.
40     /// </summary>
41     [SerializeField]
42     float powerSize;
43
44     /// <summary>
45     /// The distance between each input and output node.
46     /// </summary>
47     [SerializeField]
```

```
48      float heightMargins;
49
50      /// <summary>
51      /// The width of all circuits.
52      /// </summary>
53      [SerializeField]
54      float width;
55
56      /// <summary>
57      /// Reference to the display prefab.
58      /// </summary>
59      [SerializeField]
60      GameObject displayRef;
61
62      /// <summary>
63      /// Various materials utilized in circuit creation.
64      /// </summary>
65      [SerializeField]
66      Material baseMaterial, borderMaterial, inputMaterial, outputMaterial,
          powerOffMaterial, powerOnMaterial;
67
68      /// <summary>
69      /// The font to utilize for circuit names.
70      /// </summary>
71      [SerializeField]
72      TMP_FontAsset font;
73
74      /// <summary>
75      /// The padding that should be applied to the text component of a
          visualized circuit.
76      /// </summary>
77      [SerializeField]
78      Vector2 textPadding;
79
80      /// <summary>
81      /// Refers to the triangles of any generated quad.
82      /// </summary>
83      private readonly int[] triangles = new int[] { 0, 1, 3, 3, 1, 2 };
84
85      /// <summary>
86      /// Refers to the UV of any generated quad.
87      /// </summary>
88      private readonly Vector2[] uv = new Vector2[] { new Vector2(0, 0), new
          Vector2(0, 1), new Vector2(1, 1), new Vector2(1, 0) };
89
90      /// <summary>
91      /// Refers to the normals of any generated quad.
92      /// </summary>
93      private readonly Vector3[] normals = new Vector3[] { Vector3.up,
```

```
        Vector3.up, Vector3.up, Vector3.up };

    /// <summary>
    /// Wrapper class allowing for the discovery of an <see cref="Circuit.Input"/> through raycasting.
    /// </summary>
    public class InputReference : MonoBehaviour
    {
        /// <summary>
        /// The wrapped input.
        /// </summary>
        private Circuit.Input input;

        // Getter and setter method
        public Circuit.Input Input { get { return input; } set { input = value; } }
    }

    /// <summary>
    /// Wrapper class allowing for the discovery of an <see cref="Circuit.Output"/> through raycasting.
    /// </summary>
    public class OutputReference : MonoBehaviour
    {
        /// <summary>
        /// The wrapped output.
        /// </summary>
        private Circuit.Output output;

        // Getter and setter method
        public Circuit.Output Output { get { return output; } set { output = value; } }
    }

    // Enforces a singleton state pattern
    private void Awake()
    {
        if (instance != null)
        {
            Destroy(this);
            throw new Exception("CircuitVisualizer instance already established; terminating.");
        }

        instance = this;
    }

    /// <summary>
    /// Restores a serialized connection back to the scene.
```

```
137     /// </summary>
138     /// <param name="connection">The serialized connection to restore.</param>
139     /// <returns>The instantiated GameObjects in the form of a <see cref="ConnectionSerializerRestorer"/>.</returns>
140     public ConnectionSerializerRestorer VisualizeConnection(ConnectionSerializer connection)
141     {
142         GameObject parentObj = new GameObject("Connection");
143         Vector3 normalOffset = new Vector3(0, -0.125f, 0.5f);
144         Vector3 pivotOffset = new Vector3(0, 0, -0.5f);
145 
146         parentObj.transform.position = Vector3.zero;
147 
148         // If there is an optimized parent mesh, create it.
149         if (connection.ParentMesh != null) CreateMesh(parentObj, connection.ParentMesh);
150 
151         // Runs if the starting wire and ending wire point to the same GameObject.
152         if (connection.SingleWired)
153         {
154             // Creates the single wire parent
155             GameObject singleWire = new GameObject("Ending Wire");
156 
157             singleWire.transform.parent = parentObj.transform;
158             singleWire.transform.position = connection.EndingMesh.Position;
159             singleWire.transform.eulerAngles = connection.EndingMesh.Rotation;
160             singleWire.transform.localScale = connection.EndingMesh.Scale;
161 
162             // Creates the single wire pivot
163             GameObject pivot = new GameObject("Pivot");
164 
165             pivot.transform.parent = singleWire.transform;
166             pivot.transform.localPosition = pivotOffset;
167             pivot.transform.localEulerAngles = Vector3.zero;
168             pivot.transform.localScale = Vector3.one;
169 
170             // Creates the single wire mesh
171             GameObject actual = new GameObject("GameObject");
172 
173             actual.transform.parent = pivot.transform;
174             actual.transform.localPosition = normalOffset;
175             actual.transform.localEulerAngles = Vector3.zero;
176             actual.transform.localScale = Vector3.one;
177             CreateMesh(actual, connection.EndingMesh);
178 
```

```
179             return new ConnectionSerializerRestorer
                  (connection.CircuitConnectorIdentifier, singleWire,
                  singleWire, parentObj);
180         }
181
182         else
183         {
184             // Creates the starting wire parent
185             GameObject startingWire = new GameObject("Starting Wire");
186
187             startingWire.transform.parent = parentObj.transform;
188             startingWire.transform.position =
                  connection.StartingMesh.Position;
189             startingWire.transform.eulerAngles =
                  connection.StartingMesh.Rotation;
190             startingWire.transform.localScale =
                  connection.StartingMesh.Scale;
191
192             // Creates the starting wire pivot
193             GameObject pivot = new GameObject("Pivot");
194
195             pivot.transform.parent = startingWire.transform;
196             pivot.transform.localPosition = Vector3.zero;
197             pivot.transform.localEulerAngles = Vector3.zero;
198             pivot.transform.localScale = Vector3.one;
199
200             // Creates the starting wire mesh
201             GameObject actual = new GameObject("GameObject");
202
203             actual.transform.parent = pivot.transform;
204             actual.transform.localPosition = normalOffset;
205             actual.transform.localEulerAngles = Vector3.zero;
206             actual.transform.localScale = Vector3.one;
207             CreateMesh(actual, connection.StartingMesh);
208
209             // Creates the ending wire parent
210             GameObject endingWire = new GameObject("Ending Wire");
211
212             endingWire.transform.parent = parentObj.transform;
213             endingWire.transform.position =
                  connection.EndingMesh.Position;
214             endingWire.transform.eulerAngles =
                  connection.EndingMesh.Rotation;
215             endingWire.transform.localScale = connection.EndingMesh.Scale;
216
217             // Creates the ending wire pivot
218             pivot = new GameObject("Pivot");
219             pivot.transform.parent = endingWire.transform;
220             pivot.transform.localPosition = Vector3.zero;
```

```
221             pivot.transform.localEulerAngles = Vector3.zero;
222             pivot.transform.localScale = Vector3.one;
223
224             // Creates the ending wire mesh
225             actual = new GameObject("GameObject");
226             actual.transform.parent = pivot.transform;
227             actual.transform.localPosition = normalOffset;
228             actual.transform.localEulerAngles = Vector3.zero;
229             actual.transform.localScale = Vector3.one;
230             CreateMesh(actual, connection.EndingMesh);
231
232             return new ConnectionSerializerRestorer
                  (connection.CircuitConnectorIdentifier, startingWire,
                  endingWire, parentObj);
233         }
234     }
235
236     /// <summary>
237     /// Generates a circuit GameObject corresponding to its specific
        properties.
238     /// </summary>
239     /// <param name="circuit">The circuit to reference.</param>
240     /// <param name="startingPosition">The starting position of the
        circuit.</param>
241     public void VisualizeCircuit(Circuit circuit, Vector2
        startingPosition)
242     {
243         // Target circuit is a display; run alternate code
244         if (circuit.GetType() == typeof(Display))
245         {
246             Display display = (Display)circuit;
247             GameObject displayObj = Instantiate(displayRef);
248             DisplayReference displayVals =
                  displayObj.GetComponent<DisplayReference>();
249             Circuit.Input[] inputs = display.Inputs;
250
251             displayObj.name = display.CircuitName;
252             displayObj.transform.position = new Vector3
                  (startingPosition.x, GridMaintenance.Instance.GridHeight,
                  startingPosition.y);
253             display.PhysicalObject = displayObj;
254             display.Pins = displayVals.Pins;
255             display.PreviewPins = displayVals.PreviewPins;
256
257             for (int i = 0; i < 8; i++)
258             {
259                 GameObject currentInput = displayVals.Inputs[i];
260                 InputReference inputReference =
                      currentInput.AddComponent<InputReference>();
```

```
261
262                 inputReference.Input = inputs[i];
263                 inputs[i].Transform = currentInput.transform;
264                 inputs[i].StatusRenderer = displayVals.InputStatuses[i];
265             }
266
267             Destroy(displayVals); // Reference script no longer needed
                  after extracing relevant values
268
269             CircuitReference circuitRef =
                  displayObj.AddComponent<CircuitReference>();
270
271             circuitRef.Circuit = circuit;
272             return;
273         }
274
275         // Setting dimensions
276         int numInputMargins = circuit.Inputs.Length + 1, numOutputMargins
              = circuit.Outputs.Length + 1;
277         float inputHeight = numInputMargins * heightMargins +
              circuit.Inputs.Length * inputSize;
278         float outputHeight = numOutputMargins * heightMargins +
              circuit.Outputs.Length * outputSize;
279         Vector2 dimensions = new Vector2(width, Mathf.Max(inputHeight,
              outputHeight));
280
281         // Creating circuit base
282         GameObject physicalObject = new GameObject("\"" +
              circuit.CircuitName + "\" Gate");
283         GameObject baseQuad = new GameObject("Base");
284
285         physicalObject.transform.position = new Vector3
              (startingPosition.x, GridMaintenance.Instance.GridHeight,
              startingPosition.y);
286         baseQuad.layer = 8;
287         baseQuad.transform.parent = physicalObject.transform;
288         baseQuad.transform.localPosition = Vector3.up * 0.005f;
289
290         Vector3[] vertices = new Vector3[]
291         {
292             new Vector3(-dimensions.x / 2, 0, -dimensions.y / 2),
293             new Vector3(-dimensions.x / 2, 0, dimensions.y / 2),
294             new Vector3(dimensions.x / 2, 0, dimensions.y / 2),
295             new Vector3(dimensions.x / 2, 0, -dimensions.y / 2)
296         };
297
298         CreateQuad(baseQuad, vertices, baseMaterial);
299
300         // Creating circuit border
```

```
301         GameObject borderQuad = new GameObject("Border");
302
303         borderQuad.layer = 13;
304         borderQuad.transform.parent = physicalObject.transform;
305         borderQuad.transform.localPosition = Vector3.zero;
306         vertices = new Vector3[]
307         {
308             new Vector3(-dimensions.x / 2 - borderThickness, 0, -
                dimensions.y / 2 - borderThickness),
309             new Vector3(-dimensions.x / 2 - borderThickness, 0,
                dimensions.y / 2 + borderThickness),
310             new Vector3(dimensions.x / 2 + borderThickness, 0,
                dimensions.y / 2 + borderThickness),
311             new Vector3(dimensions.x / 2 + borderThickness, 0, -
                dimensions.y / 2 - borderThickness)
312         };
313         CreateQuad(borderQuad, vertices, borderMaterial, false);
314
315         // Power on/off vertices
316         Vector3[] powerVertices = new Vector3[]
317         {
318             new Vector3(-powerSize / 2, 0, -powerSize / 2),
319             new Vector3(-powerSize / 2, 0, powerSize / 2),
320             new Vector3(powerSize / 2, 0, powerSize / 2),
321             new Vector3(powerSize / 2, 0, -powerSize / 2)
322         };
323
324         // Creating input nodes
325         float inputStepSize = (dimensions.y - circuit.Inputs.Length *
              inputSize) / numInputMargins;
326         int index = 0;
327
328         vertices = new Vector3[]
329         {
330                 new Vector3(-inputSize / 2, 0, -inputSize / 2),
331                 new Vector3(-inputSize / 2, 0, inputSize / 2),
332                 new Vector3(inputSize / 2, 0, inputSize / 2),
333                 new Vector3(inputSize / 2, 0, -inputSize / 2)
334         };
335
336         for (float currentHeight = inputStepSize + inputSize / 2; index <
              circuit.Inputs.Length; currentHeight += inputStepSize +
              inputSize)
337         {
338             GameObject inputQuad = new GameObject("Input " + (index + 1));
339             GameObject inputQuadPower = new GameObject("Input Status " +
                (index + 1));
340             Vector3 pos = new Vector3(-dimensions.x / 2, 0.01f,
                currentHeight - dimensions.y / 2);
```

```
341
342             inputQuad.layer = 9;
343             inputQuad.transform.parent = inputQuadPower.transform.parent =
                    physicalObject.transform;
344             inputQuad.transform.localPosition =
                  inputQuadPower.transform.localPosition = pos;
345             CreateQuad(inputQuad, vertices, inputMaterial);
346             CreateQuad(inputQuadPower, powerVertices, powerOffMaterial,
                  false);
347
348             Vector3 temp = inputQuadPower.transform.localPosition;
349
350             temp.y = 0.015f;
351             inputQuadPower.transform.localPosition = temp;
352             circuit.Inputs[index].Transform = inputQuad.transform;
353             circuit.Inputs[index].StatusRenderer =
                  inputQuadPower.GetComponent<MeshRenderer>();
354
355             InputReference inputReference =
                  inputQuad.AddComponent<InputReference>();
356
357             inputReference.Input = circuit.Inputs[index];
358             index++;
359         }
360
361         // Creating output nodes
362         float outputStepSize = (dimensions.y - circuit.Outputs.Length *
              outputSize) / numOutputMargins;
363
364         index = 0;
365         vertices = new Vector3[]
366         {
367                 new Vector3(-outputSize / 2, 0, -outputSize / 2),
368                 new Vector3(-outputSize / 2, 0, outputSize / 2),
369                 new Vector3(outputSize / 2, 0, outputSize / 2),
370                 new Vector3(outputSize / 2, 0, -outputSize / 2)
371         };
372
373         for (float currentHeight = outputStepSize + outputSize / 2; index
              < circuit.Outputs.Length; currentHeight += outputStepSize +
              outputSize)
374         {
375             GameObject outputQuad = new GameObject("Output " + (index +
                  1));
376             GameObject outputQuadPower = new GameObject("Output Status " +
                    (index + 1));
377             Vector3 pos = new Vector3(dimensions.x / 2, 0.01f,
                  currentHeight - dimensions.y / 2);
378
```

```
379             outputQuad.layer = 10;
380             outputQuad.transform.parent = outputQuadPower.transform.parent
                    = physicalObject.transform;
381             outputQuad.transform.localPosition =
                  outputQuadPower.transform.localPosition = pos;
382             CreateQuad(outputQuad, vertices, outputMaterial);
383             CreateQuad(outputQuadPower, powerVertices, powerOffMaterial,
                  false);
384
385             Vector3 temp = outputQuadPower.transform.localPosition;
386
387             temp.y = 0.015f;
388             outputQuadPower.transform.localPosition = temp;
389             circuit.Outputs[index].Transform = outputQuad.transform;
390             circuit.Outputs[index].StatusRenderer =
                  outputQuadPower.GetComponent<MeshRenderer>();
391
392             OutputReference outputReference =
                  outputQuad.AddComponent<OutputReference>();
393
394             outputReference.Output = circuit.Outputs[index];
395             index++;
396         }
397
398         // Adding text component
399         GameObject name = new GameObject("Name");
400
401         name.transform.parent = physicalObject.transform;
402         name.transform.localPosition = Vector3.up * 0.01f + Vector3.right
              * (inputSize - outputSize) / 4;
403         name.transform.eulerAngles = Vector3.right * 90;
404
405         Vector2 nameDimensions = new Vector2(dimensions.x - (inputSize +
              outputSize) / 2 - 2 * textPadding.x, dimensions.y - 2 *
              textPadding.y);
406         TextMeshPro text = name.AddComponent<TextMeshPro>();
407
408         text.text = circuit.CircuitName;
409         text.rectTransform.sizeDelta = nameDimensions;
410         text.alignment = TextAlignmentOptions.Center;
411         text.enableAutoSizing = true;
412         text.fontSizeMin = 0;
413         text.font = font;
414         text.color = startingCircuitColor;
415
416         circuit.PhysicalObject = physicalObject; // Connects new
              GameObject to its circuit for future reference.
417
418         CircuitReference circuitReference =
```

```
            physicalObject.AddComponent<CircuitReference>();
419
420         circuitReference.Circuit = circuit;
421         circuit.Update();
422     }
423
424     /// <summary>
425     /// Generates a custom circuit GameObject corresponding to its specific properties.
426     /// </summary>
427     /// <param name="customCircuit">The custom circuit to reference.</param>
428     /// <param name="startingPosition">The starting position of the custom circuit.</param>
429     public void VisualizeCustomCircuit(CustomCircuit customCircuit, Vector2 startingPosition)
430     {
431         // Setting dimensions
432         int numInputMargins = customCircuit.Inputs.Length + 1, numOutputMargins = customCircuit.Outputs.Length + 1;
433         float inputHeight = numInputMargins * heightMargins + customCircuit.Inputs.Length * inputSize;
434         float outputHeight = numOutputMargins * heightMargins + customCircuit.Outputs.Length * outputSize;
435         Vector2 dimensions = new Vector2(width, Mathf.Max(inputHeight, outputHeight));
436
437         // Creating circuit base
438         GameObject physicalObject = new GameObject("\"" + customCircuit.CircuitName + "\"");
439         GameObject baseQuad = new GameObject("Base");
440
441         physicalObject.transform.position = new Vector3 (startingPosition.x, GridMaintenance.Instance.GridHeight, startingPosition.y);
442         baseQuad.layer = 8;
443         baseQuad.transform.parent = physicalObject.transform;
444         baseQuad.transform.localPosition = Vector3.up * 0.005f;
445
446         Vector3[] vertices = new Vector3[]
447         {
448             new Vector3(-dimensions.x / 2, 0, -dimensions.y / 2),
449             new Vector3(-dimensions.x / 2, 0, dimensions.y / 2),
450             new Vector3(dimensions.x / 2, 0, dimensions.y / 2),
451             new Vector3(dimensions.x / 2, 0, -dimensions.y / 2)
452         };
453
454         CreateQuad(baseQuad, vertices, baseMaterial);
455
```

```
456         // Creating circuit border
457         GameObject borderQuad = new GameObject("Border");
458
459         borderQuad.layer = 13;
460         borderQuad.transform.parent = physicalObject.transform;
461         borderQuad.transform.localPosition = Vector3.zero;
462         vertices = new Vector3[]
463         {
464             new Vector3(-dimensions.x / 2 - borderThickness, 0, -
                dimensions.y / 2 - borderThickness),
465             new Vector3(-dimensions.x / 2 - borderThickness, 0,
                dimensions.y / 2 + borderThickness),
466             new Vector3(dimensions.x / 2 + borderThickness, 0,
                dimensions.y / 2 + borderThickness),
467             new Vector3(dimensions.x / 2 + borderThickness, 0, -
                dimensions.y / 2 - borderThickness)
468         };
469         CreateQuad(borderQuad, vertices, borderMaterial, false);
470
471         // Power on/off vertices
472         Vector3[] powerVertices = new Vector3[]
473         {
474             new Vector3(-powerSize / 2, 0, -powerSize / 2),
475             new Vector3(-powerSize / 2, 0, powerSize / 2),
476             new Vector3(powerSize / 2, 0, powerSize / 2),
477             new Vector3(powerSize / 2, 0, -powerSize / 2)
478         };
479
480         // Creating input nodes
481         float inputStepSize = (dimensions.y - customCircuit.Inputs.Length
              * inputSize) / numInputMargins;
482         int index = 0;
483
484         vertices = new Vector3[]
485         {
486                 new Vector3(-inputSize / 2, 0, -inputSize / 2),
487                 new Vector3(-inputSize / 2, 0, inputSize / 2),
488                 new Vector3(inputSize / 2, 0, inputSize / 2),
489                 new Vector3(inputSize / 2, 0, -inputSize / 2)
490         };
491
492         for (float currentHeight = inputStepSize + inputSize / 2; index <
              customCircuit.Inputs.Length; currentHeight += inputStepSize +
              inputSize)
493         {
494             GameObject inputQuad = new GameObject("Input " + (index + 1));
495             GameObject inputQuadPower = new GameObject("Input Status " +
                (index + 1));
496             Vector3 pos = new Vector3(-dimensions.x / 2, 0.01f,
```

```
                currentHeight - dimensions.y / 2);
497
498             inputQuad.layer = 9;
499             inputQuad.transform.parent = inputQuadPower.transform.parent =
                    physicalObject.transform;
500             inputQuad.transform.localPosition =
                  inputQuadPower.transform.localPosition = pos;
501             CreateQuad(inputQuad, vertices, inputMaterial);
502             CreateQuad(inputQuadPower, powerVertices, powerOffMaterial,
                  false);
503
504             Vector3 temp = inputQuadPower.transform.localPosition;
505
506             temp.y = 0.015f;
507             inputQuadPower.transform.localPosition = temp;
508             customCircuit.Inputs[index].Transform = inputQuad.transform;
509             customCircuit.Inputs[index].StatusRenderer =
                  inputQuadPower.GetComponent<MeshRenderer>();
510
511             InputReference inputReference =
                  inputQuad.AddComponent<InputReference>();
512
513             inputReference.Input = customCircuit.Inputs[index];
514             index++;
515         }
516
517         // Creating output nodes
518         float outputStepSize = (dimensions.y -
              customCircuit.Outputs.Length * outputSize) / numOutputMargins;
519
520         index = 0;
521         vertices = new Vector3[]
522         {
523                 new Vector3(-outputSize / 2, 0, -outputSize / 2),
524                 new Vector3(-outputSize / 2, 0, outputSize / 2),
525                 new Vector3(outputSize / 2, 0, outputSize / 2),
526                 new Vector3(outputSize / 2, 0, -outputSize / 2)
527         };
528
529         for (float currentHeight = outputStepSize + outputSize / 2; index
              < customCircuit.Outputs.Length; currentHeight += outputStepSize
              + outputSize)
530         {
531             GameObject outputQuad = new GameObject("Output " + (index +
                  1));
532             GameObject outputQuadPower = new GameObject("Output Status " +
                    (index + 1));
533             Vector3 pos = new Vector3(dimensions.x / 2, 0.01f,
                  currentHeight - dimensions.y / 2);
```

```
534
535             outputQuad.layer = 10;
536             outputQuad.transform.parent = outputQuadPower.transform.parent
                    = physicalObject.transform;
537             outputQuad.transform.localPosition =
                  outputQuadPower.transform.localPosition = pos;
538             CreateQuad(outputQuad, vertices, outputMaterial);
539             CreateQuad(outputQuadPower, powerVertices, powerOffMaterial,
                  false);
540
541             Vector3 temp = outputQuadPower.transform.localPosition;
542
543             temp.y = 0.015f;
544             outputQuadPower.transform.localPosition = temp;
545             customCircuit.Outputs[index].Transform = outputQuad.transform;
546             customCircuit.Outputs[index].StatusRenderer =
                  outputQuadPower.GetComponent<MeshRenderer>();
547
548             OutputReference outputReference =
                  outputQuad.AddComponent<OutputReference>();
549
550             outputReference.Output = customCircuit.Outputs[index];
551             index++;
552         }
553
554         // Adding text component
555         GameObject name = new GameObject("Name");
556
557         name.transform.parent = physicalObject.transform;
558         name.transform.localPosition = Vector3.up * 0.01f + Vector3.right
              * (inputSize - outputSize) / 4;
559         name.transform.eulerAngles = Vector3.right * 90;
560
561         Vector2 nameDimensions = new Vector2(dimensions.x - (inputSize +
              outputSize) / 2 - 2 * textPadding.x, dimensions.y - 2 *
              textPadding.y);
562         TextMeshPro text = name.AddComponent<TextMeshPro>();
563
564         text.text = customCircuit.CircuitName.ToUpper();
565         text.rectTransform.sizeDelta = nameDimensions;
566         text.alignment = TextAlignmentOptions.Center;
567         text.enableAutoSizing = true;
568         text.fontSizeMin = 0;
569         text.font = font;
570         text.color = customCircuitColor;
571
572         customCircuit.PhysicalObject = physicalObject; // Connects new
              GameObject to its circuit for future reference.
573
```

```
574         CircuitReference circuitReference =
              physicalObject.AddComponent<CircuitReference>();
575
576         circuitReference.Circuit = customCircuit;
577         customCircuit.Connections.transform.parent =
              customCircuit.PhysicalObject.transform;
578     }
579
580     /// <summary>
581     /// Special signature of <seealso cref="CreateQuad(GameObject, Vector3
          [], Material, bool)"/> that always adds a mesh collider.
582     /// </summary>
583     /// <param name="quad"></param>
584     /// <param name="vertices"></param>
585     /// <param name="material"></param>
586     private void CreateQuad(GameObject quad, Vector3[] vertices, Material
          material) { CreateQuad(quad, vertices, material, true); }
587
588     /// <summary>
589     /// Creates a quad from the given mesh data.
590     /// </summary>
591     /// <param name="quad">The GameObject to save the mesh to.</param>
592     /// <param name="vertices">The vertices of the mesh.</param>
593     /// <param name="material">The material of the mesh.</param>
594     /// <param name="addMeshCollider">Whether the mesh should have a mesh
          collider for raycasting.</param>
595     private void CreateQuad(GameObject quad, Vector3[] vertices, Material
          material, bool addMeshCollider)
596     {
597         Mesh mesh = new Mesh();
598         MeshFilter meshFilter = quad.AddComponent<MeshFilter>();
599         MeshRenderer meshRenderer = quad.AddComponent<MeshRenderer>();
600
601         mesh.vertices = vertices;
602         mesh.triangles = triangles;
603         mesh.uv = uv;
604         mesh.normals = normals;
605         meshFilter.mesh = mesh;
606         meshRenderer.material = material;
607
608         if (addMeshCollider) quad.AddComponent<MeshCollider>();
609     }
610
611     /// <summary>
612     /// Creates a mesh from a given mesh serializer.
613     /// </summary>
614     /// <param name="obj">The GameObject to add the mesh to.</param>
615     /// <param name="ms">The serialized mesh data.</param>
616     private void CreateMesh(GameObject obj, MeshSerializer ms)
```

```
617     {
618         Mesh mesh = new Mesh();
619         MeshFilter meshFilter = obj.AddComponent<MeshFilter>();
620         MeshRenderer meshRenderer = obj.AddComponent<MeshRenderer>();
621 
622         // Restores mesh values and GameObject layer
623         meshFilter.mesh = mesh;
624         mesh.vertices = ms.Vertices;
625         mesh.triangles = ms.Triangles;
626         mesh.uv = ms.UV;
627         mesh.normals = ms.Normals;
628         mesh.RecalculateBounds();
629         meshRenderer.material = powerOffMaterial;
630         obj.AddComponent<MeshCollider>();
631         obj.layer = 11;
632     }
633 
634     // Getter methods
635     public static CircuitVisualizer Instance { get { return instance; } }
636 
637     public Material InputMaterial { get { return inputMaterial; ; } }
638 
639     public Material OutputMaterial { get { return outputMaterial; } }
640 
641     public Material PowerOffMaterial { get { return powerOffMaterial; } }
642 
643     public Material PowerOnMaterial { get { return powerOnMaterial; } }
644 }
```

```
1  using UnityEngine;
2
3  /// <summary>
4  /// ConnectionSerializerRestorer stores deserialized GameObject values from an external <see cref="ConnectionSerializer"/>.<br/><br/>
5  /// These stored values are then referenced by <see cref="CircuitVisualizer"/> as the assigned connection is restored to the scene.
6  /// </summary>
7  public class ConnectionSerializerRestorer
8  {
9      /// <summary>
10     /// The <see cref="CircuitConnectorIdentifier"/> pertaining to this connection.
11     /// </summary>
12     public CircuitConnectorIdentifier circuitConnectorIdentifier { get; private set; }
13
14     /// <summary>
15     /// The GameObject that has been created from the assigned <see cref="ConnectionSerializer.startingMesh"/>.
16     /// </summary>
17     public GameObject startingWire { get; private set; }
18
19     /// <summary>
20     /// The GameObject that has been created from the assigned <see cref="ConnectionSerializer.endingMesh"/>.
21     /// </summary>
22     public GameObject endingWire { get; private set; }
23
24     /// <summary>
25     /// The GameObject that has been created from the assigned <see cref="ConnectionSerializer.parentMesh"/>.
26     /// </summary>
27     public GameObject parentObject { get; private set; }
28
29     /// <summary>
30     /// Instantiates and assigns all relevant values extracted from a <see cref="ConnectionSerializer"/>.
31     /// </summary>
32     public ConnectionSerializerRestorer(CircuitConnectorIdentifier circuitConnectorIdentifier, GameObject startingWire, GameObject endingWire, GameObject parentObject)
33     {
34         this.circuitConnectorIdentifier= circuitConnectorIdentifier;
35         this.startingWire = startingWire;
36         this.endingWire = endingWire;
37         this.parentObject = parentObject;
38     }
```

```
39  }
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// CursorManager handles the switching of mouse textures within each
     scene.
6  /// </summary>
7  public class CursorManager : MonoBehaviour
8  {
9      // Singleton state reference
10     private static CursorManager instance;
11
12     /// <summary>
13     /// Texture that is utilized when the cursor is in use.
14     /// </summary>
15     [SerializeField]
16     Texture2D cursorTexture;
17
18     /// <summary>
19     /// Texture that is utilized when the link is in use.
20     /// </summary>
21     [SerializeField]
22     Texture2D linkTexture;
23
24     /// <summary>
25     /// The pixel position the cursor texture is centered around.
26     /// </summary>
27     private static Vector2 cursorPosition = Vector2.zero;
28
29     /// <summary>
30     /// The pixel position the link texture is centered around.
31     /// </summary>
32     private static Vector2 linkPosition = new Vector2(12, 0);
33
34     // Enforces a singleton state pattern
35     private void Awake()
36     {
37         if (instance != null)
38         {
39             Destroy(this);
40             throw new Exception("CircuitManager instance already
                 established; terminating.");
41         }
42
43         instance = this;
44     }
45
46     /// <summary>
47     /// Sets a new mouse texture based on the specified value.
```

```
48      /// </summary>
49      /// <param name="useCursorTexture">Whether the cursor texture should be
            used.</param>
50      public static void SetMouseTexture(bool useCursorTexture)
51      {
52          if (useCursorTexture) Cursor.SetCursor(instance.cursorTexture,
              cursorPosition, CursorMode.ForceSoftware);
53
54          else Cursor.SetCursor(instance.linkTexture, linkPosition,
              CursorMode.ForceSoftware);
55      }
56  }
```

```cs
using TMPro;
using UnityEngine;

/// <summary>
/// FPSCounter displays the current frames-per-second to an assigned text UI.
/// </summary>
public class FPSCounter : MonoBehaviour
{
    /// <summary>
    /// The text that displays the FPS.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI fpsText;

    /// <summary>
    /// The frames-per-second.
    /// </summary>
    private float fps;

    // Calculates the FPS and writes it to fpsText.
    private void Update()
    {
        fps = (int)(Time.timeScale / Time.smoothDeltaTime);
        fpsText.text = "FPS: " + fps;
    }
}
```

```
1  using System;
2  using UnityEngine;
3
4  /// <summary>
5  /// GridMaintenance instantiates and maintains the visible grid in the
     editor and preview scenes.
6  /// </summary>
7  public class GridMaintenance : MonoBehaviour
8  {
9      // Singleton state reference
10     private static GridMaintenance instance;
11
12     /// <summary>
13     /// The global height that all in-scene placements are placed at.<br/
         ><br/>
14     /// The game calculates the x and z positions by raycasting to an x-z
         plane located at this height.
15     /// </summary>
16     [SerializeField]
17     float gridHeight;
18
19     /// <summary>
20     /// The prefab that displays the in-scene grid.
21     /// </summary>
22     [SerializeField]
23     GameObject gridReference;
24
25     /// <summary>
26     /// The instantiated grid; a copy of <seealso cref="gridReference"/>.
27     /// </summary>
28     private GameObject grid;
29
30     /// <summary>
31     /// The material of the grid.
32     /// </summary>
33     private Material gridMaterial;
34
35     /// <summary>
36     /// The material texture offset of the grid.<br/><br/>
37     /// Because the grid follows the camera, its material must move
         opposite to the direction of camera movement to create an illusion of
          it standing still.
38     /// </summary>
39     private Vector2 materialOffset;
40
41     /// <summary>
42     /// Utilized to calculate the delta position between frames.
43     /// </summary>
44     private Vector3 currentPos;
```

```
45
46      // Enforces a singleton state pattern.
47      private void Awake()
48      {
49          if (instance != null)
50          {
51              Destroy(this);
52              throw new Exception("GridMaintenance instance already
                  established; terminating.");
53          }
54
55          instance = this;
56      }
57
58      // Instantiates the grid and its respective values.
59      private void Start()
60      {
61          grid = Instantiate(gridReference);
62          grid.name = "Grid";
63          grid.transform.position = new Vector3(transform.position.x,
              gridHeight, transform.position.z);
64          grid.transform.eulerAngles = Vector3.zero;
65          gridMaterial = grid.GetComponent<MeshRenderer>().material;
66          currentPos = transform.position;
67          materialOffset = gridMaterial.GetTextureOffset("_MainTex");
68      }
69
70      // Obtains the change in position from the last frame and alters
          materialOffset by an opposite value.
71      private void Update()
72      {
73          grid.transform.position = new Vector3(transform.position.x,
              gridHeight, transform.position.z);
74
75          Vector3 oldPos = currentPos;
76
77          currentPos = transform.position;
78
79          Vector3 deltaPos = currentPos - oldPos;
80          Vector2 realDeltaPos = new Vector2(deltaPos.x, deltaPos.z);
81
82          materialOffset += realDeltaPos;
83          materialOffset = new Vector2(materialOffset.x % 1, materialOffset.y
                % 1); // The grid is 1x1, therefore it can be clamped.
84          gridMaterial.SetTextureOffset("_MainTex", materialOffset);
85      }
86
87      // Getter methods
88      public static GridMaintenance Instance { get { return instance; } }
```

```
89
90      public float GridHeight { get { return gridHeight; } }
91  }
```

```
1  using System;
2  using UnityEngine;
3  using UnityEngine.UI;
4
5  /// <summary>
6  /// GuideHandler is assigned to its respective prefab, allowing for
     transitions between windows and tabs.
7  /// </summary>
8  public class GuideHandler : MonoBehaviour
9  {
10     /// <summary>
11     /// The button color of a selected and unselected tab respectively.
12     /// </summary>
13     [SerializeField]
14     Color selectedTabColor, unselectedTabColor;
15
16     /// <summary>
17     /// The button backgrounds of each window.
18     /// </summary>
19     [SerializeField]
20     Image welcomeWindow, logicGatesWindow, controlsWindow;
21
22     /// <summary>
23     /// The button backgrounds of the tabs within each window.
24     /// </summary>
25     [SerializeField]
26     Image[] welcomeTabs, logicGatesTabs, controlsTabs;
27
28     /// <summary>
29     /// The GameObjects that contains all tabs of each window
         respectively.
30     /// </summary>
31     [SerializeField]
32     GameObject welcomeTabsParent, logicGatesTabsParent,
         controlsTabsParent;
33
34     /// <summary>
35     /// The view areas of the tabs within each window.
36     /// </summary>
37     [SerializeField]
38     GameObject[] welcomeTabsViews, logicGatesTabsViews, controlsTabsViews;
39
40     /// <summary>
41     /// The background button of the current window that is viewable.
42     /// </summary>
43     private Image currentWindow;
44
45     /// <summary>
46     /// Displays the index of the currently opened tab for each window.
```

```
47      /// </summary>
48      private int currentWelcomeTab = 0, currentLogicGatesTab = 0,
          currentControlsTab = 0;
49
50      // Initializes the default window as the welcome window.
51      private void Start() { currentWindow = welcomeWindow; }
52
53      /// <summary>
54      /// Switches to a new tab for the current window in use.
55      /// </summary>
56      /// <param name="newTab">The index of the tab to switch to.</param>
57      public void UpdateTab(int newTab)
58      {
59          GameObject currentView, newView; // The view areas to turn off and
              on respectively.
60          Image currentButton, newButton; // The button backgrounds to color
              unselected and selected respectively.
61
62          // Populates initialized variables based on current window.
63          if (currentWindow == welcomeWindow)
64          {
65              if (newTab == currentWelcomeTab) return;
66
67              currentButton = welcomeTabs[currentWelcomeTab];
68              newButton = welcomeTabs[newTab];
69              currentView = welcomeTabsViews[currentWelcomeTab];
70              newView = welcomeTabsViews[newTab];
71              currentWelcomeTab = newTab;
72          }
73
74          else if (currentWindow == logicGatesWindow)
75          {
76              if (newTab == currentLogicGatesTab) return;
77
78              currentButton = logicGatesTabs[currentLogicGatesTab];
79              newButton = logicGatesTabs[newTab];
80              currentView = logicGatesTabsViews[currentLogicGatesTab];
81              newView = logicGatesTabsViews[newTab];
82              currentLogicGatesTab = newTab;
83          }
84
85          else if (currentWindow == controlsWindow)
86          {
87              if (newTab == currentControlsTab) return;
88
89              currentButton = controlsTabs[currentControlsTab];
90              newButton = controlsTabs[newTab];
91              currentView = controlsTabsViews[currentControlsTab];
92              newView = controlsTabsViews[newTab];
```

```
93              currentControlsTab = newTab;
94          }
95
96          // Incorrent window currently in use
97          else throw new Exception("Invalid current window.");
98
99          // Updates the obtained values
100         currentButton.color = unselectedTabColor;
101         newButton.color = selectedTabColor;
102         currentView.SetActive(false);
103         newView.SetActive(true);
104     }
105
106     /// <summary>
107     /// Switches to a new window.
108     /// </summary>
109     /// <param name="newWindow">The new window to switch to.</param>
110     public void UpdateWindow(Image newWindow)
111     {
112         if (currentWindow == newWindow) return;
113
114         // Updates the button colors.
115         currentWindow.color = unselectedTabColor;
116         newWindow.color = selectedTabColor;
117
118         // Makes the current window invisible.
119         if (currentWindow == welcomeWindow) welcomeTabsParent.SetActive
              (false);
120
121         else if (currentWindow == logicGatesWindow)
              logicGatesTabsParent.SetActive(false);
122
123         else if (currentWindow == controlsWindow)
              controlsTabsParent.SetActive(false);
124
125         else throw new Exception("Invalid current window.");
126
127         // Makes the new window visible.
128         if (newWindow == welcomeWindow) welcomeTabsParent.SetActive(true);
129
130         else if (newWindow == logicGatesWindow)
              logicGatesTabsParent.SetActive(true);
131
132         else if (newWindow == controlsWindow) controlsTabsParent.SetActive
              (true);
133
134         else throw new Exception("Invalid new window.");
135
136         currentWindow = newWindow;
```

```
137     }
138 }
```

```
1  using System;
2  using System.Collections.Generic;
3  using TMPro;
4  using UnityEditor;
5  using UnityEngine;
6  using UnityEngine.EventSystems;
7  using UnityEngine.SceneManagement;
8  using UnityEngine.UI;
9
10 public class TaskbarManager : MonoBehaviour
11 {
12     // Singleton state reference
13     private static TaskbarManager instance;
14
15     /// <summary>
16     /// The color associated with starting and custom circuits
         respectively.
17     /// </summary>
18     [SerializeField]
19     Color startingCircuitColor,
20         customCircuitColor;
21
22     /// <summary>
23     /// Horizontal length of the scroll bar attached to the bookmarks
         menu.
24     /// </summary>
25     [SerializeField]
26     float bookmarkScrollThickness;
27
28     /// <summary>
29     /// Set to visible when an interface is opened.<br/><br/>
30     /// Within the scene, this should be a semi-transparent background
         overlayed onto everything except the currently opened interface.
31     /// </summary>
32     [SerializeField]
33     GameObject background;
34
35     /// <summary>
36     /// The scroll bar belonging to the bookmarks menu.<br/><br/>
37     /// If the size of bookmarks exceeds the view area of the bookmarks
         menu, this scroll bar is set to visible.
38     /// </summary>
39     [SerializeField]
40     GameObject bookmarkScrollbar;
41
42     /// <summary>
43     /// GameObject that all bookmarks are instantiated under.
44     /// </summary>
45     [SerializeField]
```

```
46      GameObject bookmarksPanel;
47
48      /// <summary>
49      /// Referenced when moving the bookmarks menu to the current user
          mouse position.
50      /// </summary>
51      [SerializeField]
52      GameObject bookmarksScroll;
53
54      /// <summary>
55      /// List of all menus that can be active in the editor scene.
56      /// </summary>
57      [SerializeField]
58      GameObject addMenu, // The menu in which the user can add/bookmark
          circuits.
59          bookmarksMenu, // The menu that displays bookmarked circuits, if
            any.
60          circuitSaveErrorMenu, // The menu indicating the reason a custom
            circuit has failed to create.
61          guide, // The guide prefab.
62          labelMenu, // The menu where the user labels empty inputs/outputs.
63          notifierPanel, // An empty menu without a user-driven exit scheme;
              has UI indicating why (e.g. saving).
64          nullState, // An empty menu without a user-driven exit scheme;
            does not have UI.
65          saveWarning, // The menu prompting the user to save.
66          sceneSaveMenu; // The menu in which the user can either save the
            editor scene or create a custom circuit.
67
68      /// <summary>
69      /// Prefab button for custom circuits within the add menu.
70      /// </summary>
71      [SerializeField]
72      GameObject customBookmarkRef;
73
74      /// <summary>
75      /// Prefab button for any circuit within the bookmarks menu.
76      /// </summary>
77      [SerializeField]
78      GameObject bookmarkRef;
79
80      /// <summary>
81      /// Exits out of <seealso cref="currentMenu"/>.<br/><br/>
82      /// More often than not, an alternate input to achieve the same effect
          is pressing the right mouse button.
83      /// </summary>
84      [SerializeField]
85      KeyCode cancelKey;
86
```

```
87      /// <summary>
88      /// Parent of all starting and custom circuits buttons within the add menu.
89      /// </summary>
90      [SerializeField]
91      RectTransform addCustomPanel,
92          addStartingPanel;
93
94      /// <summary>
95      /// Transform of the border behind the bookmarks menu.
96      /// </summary>
97      [SerializeField]
98      RectTransform bookmarksBorder;
99
100     /// <summary>
101     /// Displays the reason for a custom circuit failing to create.
102     /// </summary>
103     [SerializeField]
104     TextMeshProUGUI circuitErrorText;
105
106     /// <summary>
107     /// Prompts the user to compose a label for an empty input or output.
108     /// </summary>
109     [SerializeField]
110     TextMeshProUGUI labelText;
111
112     /// <summary>
113     /// Utilized with <seealso cref="notifierPanel"/> to display the reason why a game has disabled all input to the player.<br/><br/>
114     /// Its primary uses are for when the game is saving as well as when a custom circuit is being verified/created.
115     /// </summary>
116     [SerializeField]
117     TextMeshProUGUI notifierText;
118
119     /// <summary>
120     /// The name field with which the user specifies the name of a prospective custom circuit.
121     /// </summary>
122     [SerializeField]
123     TMP_InputField circuitNameField;
124
125     /// <summary>
126     /// Whether a custom circuit should be created.<br/>
127     /// If unchecked, then the current editor scene is saved instead.
128     /// </summary>
129     [SerializeField]
130     Toggle circuitToggle;
131
```

```
132     /// <summary>
133     /// The size of the bookmark view area and a bookmark respectively.
134     /// </summary>
135     [SerializeField]
136     Vector2 bookmarkMaskSize,
137         bookmarkSize;
138
139     /// <summary>
140     /// Whether the left mouse button is currently help down whilst in the bookmarks menu.<br/>
141     /// This helps discern whether the initial left mouse button press and release occurred when hovered on UI elements.
142     /// </summary>
143     private bool bookmarksDown;
144
145     /// <summary>
146     /// Whether the game is currently deserializing all bookmarks belonging to the current editor scene.
147     /// </summary>
148     private bool currentlyRestoring;
149
150     /// <summary>
151     /// Whether the bookmarks bar can be opened.<br/>
152     /// This value is false until the cooldown to open the bookmarks bar passes, enabling it again.
153     /// </summary>
154     private bool reopenBookmarks = true;
155
156     /// <summary>
157     /// The menu currently opened within the editor scene.
158     /// </summary>
159     private GameObject currentMenu;
160
161     /// <summary>
162     /// The ID list of bookmarks in the scene.<br/><br/>
163     /// Helps to differentiate whether the bookmark is a starting or custom circuit, since all starting circuits have an ID of -1.
164     /// </summary>
165     private List<int> bookmarkIDs = new List<int>();
166
167     /// <summary>
168     /// The typed list of bookmarks in the scene.
169     /// </summary>
170     private List<Type> bookmarks = new List<Type>();
171
172     // Enforces a singleton state pattern and disables frame-by-frame update calls.
173     private void Awake()
174     {
```

```
175         if (instance != null)
176         {
177             Destroy(this);
178             throw new Exception("TaskbarManager instance already established; terminating.");
179         }
180
181         instance = this;
182         enabled = false;
183     }
184
185     // Contains input listening for each user-controllable control interface.
186     private void Update()
187     {
188         // No menu is currently opened, skips current frame
189         if (currentMenu == nullState) return;
190
191         // Bookmark control scheme
192         if (currentMenu == bookmarksMenu)
193         {
194             // Registers left mouse button down
195             if (Input.GetMouseButtonDown(0) && !EventSystem.current.IsPointerOverGameObject())
196                 { bookmarksDown = true; }
197             // Exit scheme (occurs if left mouse button is released, but not while hovered on UI.
198             else if (Input.GetMouseButtonUp(0) && bookmarksDown)
199             {
200                 if (EventSystem.current.IsPointerOverGameObject())
                        bookmarksDown = false; else CloseMenu();
201             }
202
203             // Moves bookmark to new mouse position
204             else if (Input.GetMouseButtonDown(1) && !EventSystem.current.IsPointerOverGameObject())
                    UpdateBookmarkPosition();
205
206             // Keyboard exit scheme
207             else if (Input.GetKeyDown(cancelKey)) CloseMenu();
208         }
209
210         // Default control scheme for all other menus allowed to be existed by the user.
211         else if (currentMenu == addMenu || currentMenu == sceneSaveMenu || currentMenu == circuitSaveErrorMenu || currentMenu == saveWarning || currentMenu == guide)
212         {
```

```
213             if (Input.GetKeyDown(cancelKey) || Input.GetMouseButtonDown
                  (1))
214             {
215                 if (currentMenu == circuitSaveErrorMenu) ConfirmError();
                      else CloseMenu();
216             }
217         }
218     }
219
220     /// <summary>
221     /// Opens the <seealso cref="nullState"/> interface.<br/>
222     /// Essentially disables the taskbar from functioning; a locked state
          that must be manually closed via script.
223     /// </summary>
224     public void NullState() { OpenMenu(false, nullState); }
225
226     /// <summary>
227     /// Updates <seealso cref="circuitSaveErrorMenu"/> after <seealso
          cref="circuitToggle"/> is pressed; called by pressing an in-scene
          button.
228     /// </summary>
229     public void UpdateSaveToggle()
230     {
231         bool isOn = circuitToggle.isOn;
232
233         circuitNameField.interactable = isOn;
234
235         if (!isOn) circuitNameField.text = "";
236     }
237
238     /// <summary>
239     /// Goes back to the menu; called by pressing an in-scene button.
240     /// </summary>
241     public void OpenOptions()
242     {
243         // If the current scene is in the editor, check if the save prompt
              should first be displayed. Otherwise (including if in a preview
              scene), go back to the menu.
244         if (EditorStructureManager.Instance != null &&
              EditorStructureManager.Instance.DisplaySavePrompt) OpenMenu
              (true, saveWarning); else SceneManager.LoadScene(0);
245     }
246
247     /// <summary>
248     /// Goes back to the game menu; called by pressing an in-scene button.
249     /// </summary>
250     public void OpenMenuScene() { SceneManager.LoadScene(0); }
251
252     /// <summary>
```

```
253     /// Opens <seealso cref="labelMenu"/> <see cref="IOAssigner"/> is enabled and the user presses LMB on an incomplete empty input/output.
254     /// </summary>
255     /// <param name="isInput"></param>
256     public void OpenLabelMenu(bool isInput)
257     {
258         OpenMenu(true, labelMenu);
259         labelText.text = "compose a label for the selected " + (isInput ? "input" : "output");
260     }
261
262     /// <summary>
263     /// Opens <seealso cref="guide"/>; called by pressing an in-scene button.
264     /// </summary>
265     public void OpenGuide() { OpenMenu(true, guide); }
266
267     /// <summary>
268     /// Opens <seealso cref="sceneSaveMenu"/>; called by pressing an in-scene button.
269     /// </summary>
270     public void OpenSave() { OpenMenu(true, sceneSaveMenu); }
271
272     /// <summary>
273     /// Displays an error message if saving a custom circuit fails.
274     /// </summary>
275     /// <param name="errorMessage">The error message to display.</param>
276     public void CircuitSaveError(string errorMessage)
277     {
278         CloseMenu();
279         circuitErrorText.text = errorMessage;
280         OpenMenu(true, circuitSaveErrorMenu);
281     }
282
283     /// <summary>
284     /// Closes <seealso cref="circuitSaveErrorMenu"/>; can be called by pressing an in-scene button.
285     /// </summary>
286     public void ConfirmError()
287     {
288         CloseMenu();
289         OpenSave();
290     }
291
292     /// <summary>
293     /// Confirms <seealso cref="sceneSaveMenu"/> input and either saves the editor scene or creates a custom circuit based on <seealso cref="circuitToggle"/>.<br/>
```

```
294     /// Called by pressing an in-scene button.
295     /// </summary>
296     public void SaveConfirm()
297     {
298         CloseMenu();
299 
300         // Should attempt to create custom circuit
301         if (circuitToggle.isOn)
302         {
303             notifierText.text = "verifying...";
304             OpenMenu(true, notifierPanel);
305             PreviewStructureManager.Instance.VerifyPreviewStructure
                  (circuitNameField.text.ToLower().Trim());
306         }
307 
308         // Should save scene
309         else
310         {
311             notifierText.text = "saving scene...";
312             OpenMenu(true, notifierPanel);
313             EditorStructureManager.Instance.Serialize();
314         }
315     }
316 
317     /// <summary>
318     /// Opens <seealso cref="addMenu"/>; called by pressing an in-scene
        button.
319     /// </summary>
320     public void OpenAdd() { OpenMenu(true, addMenu); }
321 
322     /// <summary>
323     /// Opens the bookmarks menu.
324     /// </summary>
325     public void OpenBookmarks() { OpenBookmarks(false); }
326 
327     /// <summary>
328     /// Called when a custom circuit is successfully created.
329     /// </summary>
330     public void OnSuccessfulPreviewStructure()
331     {
332         circuitNameField.text = "";
333         CloseMenu();
334     }
335 
336     /// <summary>
337     /// Called when a custom circuit successfully passes validation.
338     /// </summary>
339     public void OnSuccessfulPreviewVerification()
340     {
```

```
341         CloseMenu();
342         OpenMenu(true, notifierPanel);
343         notifierText.text = "creating...";
344     }
345 
346     /// <summary>
347     /// Opens the bookmarks menu.<br/>
348     /// If there are no bookmarks to display, this method does nothing.
349     /// </summary>
350     /// <param name="showBackground"></param>
351     public void OpenBookmarks(bool showBackground)
352     {
353         if (bookmarks.Count == 0) return;
354 
355         bookmarksDown = false;
356         OpenMenu(showBackground, bookmarksMenu);
357     }
358 
359     /// <summary>
360     /// Deserializes all bookmarks stored in the current editor structure.
361     /// </summary>
362     /// <param name="circuitIndeces">The serialized integer to circuit identifiers.</param>
363     /// <param name="circuitIDs">The preview structure IDs of each circuit (-1 if non-custom).</param>
364     public void RestoreBookmarks(List<int> circuitIndeces, List<int> circuitIDs)
365     {
366         int index = 0;
367 
368         currentlyRestoring = true;
369 
370         foreach (int circuitIndex in new List<int>(circuitIndeces))
371         {
372             // Is a custom circuit
373             if (circuitIndex != -1)
374             {
375                 Toggle toggle = addStartingPanel.GetChild(circuitIndex).GetComponentInChildren<Toggle>();
376 
377                 toggle.isOn = true;
378                 UpdateBookmarkAll(toggle.gameObject);
379             }
380 
381             // Is a starting circuit
382             else AddCustomCircuitPanel(circuitIDs[index], true);
383 
384             index++;
385         }
```

```
386
387         currentlyRestoring = false;
388     }
389
390     /// <summary>
391     /// Adds all non-bookmarked custom circuits belonging to the current
      preview structure back to <seealso cref="addMenu"/>.
392     /// </summary>
393     public void RestoreCustomCircuits()
394     {
395         foreach (PreviewStructure previewStructure in
          MenuSetupManager.Instance.PreviewStructures)
396         {
397             // Is bookmarked, continue.
398             if (bookmarkIDs.Contains(previewStructure.ID)) continue;
399
400             AddCustomCircuitPanel(previewStructure.ID, false);
401         }
402     }
403
404     /// <summary>
405     /// Adds a custom circuit to <seealso cref="addCustomPanel"/>.
406     /// </summary>
407     /// <param name="circuitID">The custom circuit ID.</param>
408     /// <param name="bookmarked">Whether this circuit is bookmarked in the
       current editor scene.</param>
409     public void AddCustomCircuitPanel(int circuitID, bool bookmarked)
410     {
411         GameObject current = Instantiate(customBookmarkRef,
          addCustomPanel.transform); // Instantiates a prefab copy
412         Toggle toggle = current.GetComponentInChildren<Toggle>();
413         PreviewStructure.PreviewStructureReference reference =
          current.AddComponent<PreviewStructure.PreviewStructureReference>
          ();
414
415         current.GetComponentInChildren<TextMeshProUGUI>().text =
          MenuSetupManager.Instance.PreviewStructures
          [MenuSetupManager.Instance.PreviewStructureIDs.IndexOf
          (circuitID)].Name;
416         reference.ID = circuitID;
417
418         // Adds listeners required to add and bookmark the custom circuit.
419         current.GetComponentInChildren<Button>().onClick.AddListener
          (delegate { AddBookmarkCircuit(-1, reference.ID); });
420         toggle.onValueChanged.AddListener(delegate { UpdateBookmark
          (reference); });
421
422         if (bookmarked)
423         {
```

```
424             toggle.isOn = true;
425             UpdateBookmarkCustom(reference);
426         }
427     }
428 
429     /// <summary>
430     /// Identical to <seealso cref="UpdateBookmarkAll(GameObject)"/> except reserved specifically for <see cref="Toggle"/> calls.<br/><br/>
431     /// This is due to boolean changes within scripting also triggering <seealso cref="Toggle.onValueChanged"/> events.
432     /// </summary>
433     /// <param name="obj">The bookmark to update.</param>
434     public void UpdateBookmark(GameObject obj)
435     {
436         // If toggles are being adjusted within the script, do not continue.
437         if (currentlyRestoring) return;
438 
439         UpdateBookmarkAll(obj);
440     }
441 
442     /// <summary>
443     /// Identical to <seealso cref="UpdateBookmarkCustom(PreviewStructure.PreviewStructureReference)"/> except reserved specifically for <see cref="Toggle"/> calls.<br/><br/>
444     /// This is due to boolean changes within scripting also triggering <seealso cref="Toggle.onValueChanged"/> events.
445     /// </summary>
446     /// <param name="obj">The bookmark to update.</param>
447     public void UpdateBookmark(PreviewStructure.PreviewStructureReference previewStructureReference)
448     {
449         // If toggles are being adjusted within the script, do not continue.
450         if (currentlyRestoring) return;
451 
452         UpdateBookmarkCustom(previewStructureReference);
453     }
454 
455     /// <summary>
456     /// Updates a starting bookmark after it has been bookmarked or unbookmarked.
457     /// </summary>
458     /// <param name="obj">The starting bookmark to update.</param>
459     public void UpdateBookmarkAll(GameObject obj)
460     {
461         bool newStatus = obj.GetComponent<Toggle>().isOn;
462         Type type = CircuitType(obj.transform.parent.GetSiblingIndex());
```

```
463
464         // Adds bookmark
465         if (newStatus && !bookmarks.Contains(type))
466         {
467             if (!currentlyRestoring)
                  EditorStructureManager.Instance.DisplaySavePrompt = true;
468
469             EditorStructureManager.Instance.Bookmarks.Add
                  (StartingCircuitIndex(type));
470             bookmarks.Add(type);
471             bookmarkIDs.Add(-1);
472
473             GameObject bookmark = Instantiate(bookmarkRef,
                  bookmarksPanel.transform);
474             Button button = bookmark.GetComponentInChildren<Button>();
475             TextMeshProUGUI text =
                  bookmark.GetComponentInChildren<TextMeshProUGUI>();
476
477             bookmark.name = text.text = obj.transform.parent.name;
478             text.color = startingCircuitColor;
479
480             // Ensures pressing on the bookmark will add its
                  representative circuit.
481             button.onClick.AddListener(delegate { AddBookmarkCircuit
                  (StartingCircuitIndex(type), -1); });
482         }
483
484         // Deletes bookmark
485         else if (!newStatus && bookmarks.Contains(type))
486         {
487             if (!currentlyRestoring)
                  EditorStructureManager.Instance.DisplaySavePrompt = true;
488
489             int index = bookmarks.IndexOf(type);
490
491             EditorStructureManager.Instance.Bookmarks.Remove
                  (StartingCircuitIndex(type));
492             bookmarks.Remove(type);
493             bookmarkIDs.RemoveAt(index);
494             Destroy(bookmarksPanel.transform.GetChild(index).gameObject);
495         }
496     }
497
498     /// <summary>
499     /// Updates a custom bookmark after it has been bookmarked or
          unbookmarked.
500     /// </summary>
501     /// <param name="obj">The custom bookmark to update.</param>
502     public void UpdateBookmarkCustom
```

```
          (PreviewStructure.PreviewStructureReference reference)
503     {
504         bool newStatus = reference.GetComponentInChildren<Toggle>().isOn;
505         int id =
              reference.GetComponentInChildren<PreviewStructure.PreviewStructu
              reReference>().ID;
506
507         // Adds bookmark
508         if (newStatus && !bookmarkIDs.Contains(id))
509         {
510             if (!currentlyRestoring)
                  EditorStructureManager.Instance.DisplaySavePrompt = true;
511
512             EditorStructureManager.Instance.Bookmarks.Add(-1);
513             bookmarks.Add(typeof(CustomCircuit));
514             bookmarkIDs.Add(id);
515
516             GameObject bookmark = Instantiate(bookmarkRef,
                  bookmarksPanel.transform);
517             Button button = bookmark.GetComponentInChildren<Button>();
518             TextMeshProUGUI text =
                  bookmark.GetComponentInChildren<TextMeshProUGUI>();
519
520             bookmark.name = text.text =
                  MenuSetupManager.Instance.PreviewStructures
                  [MenuSetupManager.Instance.PreviewStructureIDs.IndexOf
                  (id)].Name;
521             text.color = customCircuitColor;
522
523             // Ensures pressing on the bookmark will add its
                  representative circuit.
524             button.onClick.AddListener(delegate { AddBookmarkCircuit(-1,
                  id); });
525         }
526
527         // Deletes bookmark
528         else if (!newStatus && bookmarkIDs.Contains(id))
529         {
530             if (!currentlyRestoring)
                  EditorStructureManager.Instance.DisplaySavePrompt = true;
531
532             int index = bookmarkIDs.IndexOf(id);
533
534             EditorStructureManager.Instance.Bookmarks.RemoveAt(index);
535             bookmarks.RemoveAt(index);
536             bookmarkIDs.Remove(id);
537             Destroy(bookmarksPanel.transform.GetChild(index).gameObject);
538         }
539     }
```

```
540
541     /// <summary>
542     /// Adds a bookmarked circuit based on its circuit type and custom circuit ID (if applicable).
543     /// </summary>
544     /// <param name="circuitType">Index representing the circuit type.</param>
545     /// <param name="circuitID">The custom circuit ID (-1 if custom).</param>
546     private void AddBookmarkCircuit(int circuitType, int circuitID)
547     {
548         switch (circuitType)
549         {
550             // Custom circuit
551             case -1:
552                 AddCircuit(new CustomCircuit
                      (MenuSetupManager.Instance.PreviewStructures
                      [MenuSetupManager.Instance.PreviewStructureIDs.IndexOf
                      (circuitID)]));
553                 return;
554             // Input
555             case 0:
556                 AddCircuit(new InputGate());
557                 return;
558             // Display
559             case 1:
560                 AddCircuit(new Display());
561                 return;
562             // Buffer
563             case 2:
564                 AddCircuit(new Buffer());
565                 return;
566             // And gate
567             case 3:
568                 AddCircuit(new AndGate());
569                 return;
570             // NAnd gate
571             case 4:
572                 AddCircuit(new NAndGate());
573                 return;
574             // NOr gate
575             case 5:
576                 AddCircuit(new NOrGate());
577                 return;
578             // Not gate
579             case 6:
580                 AddCircuit(new NotGate());
581                 return;
582             // Or gate
```

```
583             case 7:
584                 AddCircuit(new OrGate());
585                 return;
586             // XOr gate
587             case 8:
588                 AddCircuit(new XOrGate());
589                 return;
590         }
591     }
592 
593     /// <summary>
594     /// Adds a starting circuit to the scene; called by pressing an in-scene button.
595     /// </summary>
596     /// <param name="startingCircuitIndex">Representative index of the starting circuit.</param>
597     public void AddStartingCircuit(int startingCircuitIndex) { AddCircuit(GetStartingCircuit(startingCircuitIndex)); }
598 
599     /// <summary>
600     /// Adds a circuit to the scene.
601     /// </summary>
602     /// <param name="newCircuit">The circuit to add.</param>
603     private void AddCircuit(Circuit newCircuit)
604     {
605         // Cancels any modes that would obstruct the placement process.
606         switch (BehaviorManager.Instance.UnpausedGameState)
607         {
608             case BehaviorManager.GameState.CIRCUIT_MOVEMENT:
609                 BehaviorManager.Instance.CancelCircuitMovement();
610                 break;
611 
612             case BehaviorManager.GameState.IO_PRESS:
613                 BehaviorManager.Instance.CancelWirePlacement();
614                 break;
615         }
616 
617         // Switches to placement mode
618         BehaviorManager.Instance.UnpausedGameState = BehaviorManager.GameState.CIRCUIT_PLACEMENT;
619         BehaviorManager.Instance.UnpausedStateType = BehaviorManager.StateType.LOCKED;
620         BehaviorManager.Instance.CircuitPlacement(newCircuit);
621         CloseMenu();
622     }
623 
624     /// <summary>
625     /// Obtains the circuit type based on its circuit index.
626     /// </summary>
```

```
627     /// <param name="circuitIndex">The index of the circuit.</param>
628     /// <returns>The type of the circuit.</returns>
629     private Type CircuitType(int circuitIndex)
630     {
631         switch (circuitIndex)
632         {
633             case -1:
634                 return typeof(CustomCircuit);
635             case 0:
636                 return typeof(InputGate);
637             case 1:
638                 return typeof(Display);
639             case 2:
640                 return typeof(Buffer);
641             case 3:
642                 return typeof(AndGate);
643             case 4:
644                 return typeof(NAndGate);
645             case 5:
646                 return typeof(NOrGate);
647             case 6:
648                 return typeof(NotGate);
649             case 7:
650                 return typeof(OrGate);
651             case 8:
652                 return typeof(XOrGate);
653             default:
654                 throw new Exception("Invalid starting circuit index.");
655         }
656     }
657
658     /// <summary>
659     /// Obtains the index representation of a starting circuit.
660     /// </summary>
661     /// <param name="circuitType">The type of the starting circuit.</
        param>
662     /// <returns>The index representation of the circuit.</returns>
663     private int StartingCircuitIndex(Type circuitType)
664     {
665         if (circuitType == typeof(InputGate)) return 0;
666
667         else if (circuitType == typeof(Display)) return 1;
668
669         else if (circuitType == typeof(Buffer)) return 2;
670
671         else if (circuitType == typeof(AndGate)) return 3;
672
673         else if (circuitType == typeof(NAndGate)) return 4;
674
```

```
675         else if (circuitType == typeof(NOrGate)) return 5;
676
677         else if (circuitType == typeof(NotGate)) return 6;
678
679         else if (circuitType == typeof(OrGate)) return 7;
680
681         else if (circuitType == typeof(XOrGate)) return 8;
682
683         else throw new Exception("Invalid starting circuit type.");
684     }
685
686     /// <summary>
687     /// Opens a menu.
688     /// </summary>
689     /// <param name="showBackground">Whether <seealso cref="background"/>
          should be visible.</param>
690     /// <param name="newMenu">The menu to open.</param>
691     private void OpenMenu(bool showBackground, GameObject newMenu)
692     {
693         // If another menu is open, do nothing.
694         if (currentMenu != null && currentMenu != bookmarksMenu) return;
695
696         // Close the bookmarks menu if another menu is opened.
697         if (currentMenu == bookmarksMenu) CloseMenu();
698
699         currentMenu = newMenu;
700
701         // If applicable, the bookmarks menu should open around the user's
              cursor.
702         if (newMenu == bookmarksMenu)
703         {
704             UpdateBookmarkPosition();
705             UpdateBookmarkScroll();
706         }
707
708         BehaviorManager.Instance.LockUI = true;
709         background.SetActive(showBackground); currentMenu.SetActive(true);
710         enabled = true; // Enables the frame-by-frame listener.
711     }
712
713     /// <summary>
714     /// Updates the size of the bookmarks menu and enables/disables the
          scroll bar.
715     /// </summary>
716     private void UpdateBookmarkScroll()
717     {
718         // If the bookmarks menu does not show all bookmarked circuts, the
              vertical scroll bar should appear.
719         bool exceededViewport = bookmarkSize.y * bookmarks.Count >
```

```
            bookmarkMaskSize.y;

        if (exceededViewport)
        {
            // If large enough to scroll, always starts at top of options list
            bookmarksPanel.GetComponent<RectTransform>().anchoredPosition *= Vector2.right;
            bookmarksPanel.GetComponent<RectTransform>().sizeDelta = Vector2.right * (bookmarkSize.x + bookmarkScrollThickness);
            bookmarksBorder.sizeDelta = new Vector2(bookmarkSize.x + bookmarkScrollThickness, Mathf.Clamp(bookmarks.Count * bookmarkSize.y, 0, bookmarkMaskSize.y));
        }

        // Do not show scroll bar, all bookmarks are visible in the view area.
        else
        {
            bookmarksPanel.GetComponent<RectTransform>().sizeDelta = Vector2.right * bookmarkSize.x;
            bookmarksBorder.sizeDelta = new Vector2(bookmarkSize.x, Mathf.Clamp(bookmarks.Count * bookmarkSize.y, 0, bookmarkMaskSize.y));
        }

        bookmarkScrollbar.SetActive(exceededViewport);
    }

    /// <summary>
    /// Moves the bookmarks menu to the current position of the mouse.
    /// </summary>
    private void UpdateBookmarkPosition()
    {
        RectTransform bottomLeftPos = bookmarksScroll.GetComponent<RectTransform>();
        Vector2 currentPosition = Input.mousePosition;

        currentPosition.x -= bookmarkSize.x / 2;

        float downVal = bookmarkMaskSize.y - (bookmarkSize.y / 2 * bookmarks.Count);

        currentPosition.y -= Mathf.Clamp(downVal, bookmarkMaskSize.y / 2, bookmarkMaskSize.y);
        bottomLeftPos.anchoredPosition = currentPosition; // Moves all bookmarks to the new position

        Vector2 borderPosition = currentPosition;
```

```
755
756         borderPosition.y += Mathf.Clamp(0, downVal - bookmarks.Count *
              bookmarkSize.y / 2, bookmarkMaskSize.y);
757         bookmarksBorder.anchoredPosition = borderPosition; // Moves the
              bookmarks border to the new position
758     }
759
760     /// <summary>
761     /// Closes the currently opened menu.
762     /// </summary>
763     public void CloseMenu()
764     {
765         BehaviorManager.Instance.LockUI = false;
766         reopenBookmarks = false;
767         Invoke("UnlockUI", 0.1f);
768         background.SetActive(false); currentMenu.SetActive(false);
769
770         if (currentMenu == addMenu) addStartingPanel.anchoredPosition =
              addCustomPanel.anchoredPosition = Vector2.zero;
771
772         currentMenu = null;
773         enabled = false;
774     }
775
776     /// <summary>
777     /// Allows the bookmarks menu to be opened; called by invokement
          within this script.
778     /// </summary>
779     private void UnlockUI() { reopenBookmarks = true; }
780
781     /// <summary>
782     /// Creates a starting circuit from its index representation.
783     /// </summary>
784     /// <param name="startingCircuitIndex">Index of the starting
          circuit.</param>
785     /// <returns>The newly created circuit.</returns>
786     private Circuit GetStartingCircuit(int startingCircuitIndex)
787     {
788         switch (startingCircuitIndex)
789         {
790             case 0:
791                 return new InputGate();
792             case 1:
793                 return new Display();
794             case 2:
795                 return new Buffer();
796             case 3:
797                 return new AndGate();
798             case 4:
```

```
799                 return new NAndGate();
800             case 5:
801                 return new NOrGate();
802             case 6:
803                 return new NotGate();
804             case 7:
805                 return new OrGate();
806             case 8:
807                 return new XOrGate();
808             default:
809                 throw new Exception("Invalid starting circuit index.");
810         }
811     }
812
813     /// <summary>
814     /// Serializes the current editor scene; called by pressing an in-scene button.
815     /// </summary>
816     public void Serialize() { EditorStructureManager.Instance.Serialize(); }
817
818     // Getter methods
819     public static TaskbarManager Instance { get { return instance; } }
820
821     public bool ReopenBookmarks { get { return reopenBookmarks; } }
822
823     public GameObject CurrentMenu { get { return currentMenu; } }
824
825     public List<int> BookmarkIDs { get { return bookmarkIDs; } }
826 }
```