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

/// <summary>
/// Circuit is the parent of every other concretized circuit, containing several predefined constructors, methods, and values.<br/><br/>
/// Each circuit must implement the abstract method <seealso cref="UpdateOutputs"/> with the logic to update its outputs.
/// </summary>
public abstract class Circuit
{
    /// <summary>
    /// The time it takes for an update call to occur. This value is measured in seconds.
    /// </summary>
    public readonly static float clockSpeed = 0.075f;

    /// <summary>
    /// The custom circuit associated with this input.<br/><br/>
    /// This value will not be null if and only if the circuit is internally within a custom circuit.
    /// </summary>
    public CustomCircuit customCircuit;

    /// <summary>
    /// Whether the circuit should have a representative in-scene mesh generated by <see cref="CircuitVisualizer"/>.
    /// </summary>
    private bool visible;

    /// <summary>
    /// The physical mesh generated by <see cref="CircuitVisualizer"/> for this circuit.<br/><br/>
    /// This value will be null if and only if the circuit is internally within a custom circuit, i.e. no mesh will be generated.
    /// </summary>
    private GameObject physicalObject;

    /// <summary>
    /// The list of input nodes belonging to the circuit.
    /// </summary>
    private Input[] inputs;

    /// <summary>
    /// The list of output nodes belonging to this circuit.
    /// </summary>
    private Output[] outputs;

```

```csharp
    /// <summary>
    /// The list of outputs belonging to this circuit whose power stauses have changed after calling <seealso cref="UpdateOutputs"/>.<br/><br/>
    /// 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.
    /// </summary>
    private List<Output> outputsToUpdate;

    /// <summary>
    /// The name of the circuit.
    /// </summary>
    private string circuitName;

    /// <summary>
    /// Input represents all required members of an input node that belong to a circuit.<br/><br/>
    /// An input can only have one connection.
    /// </summary>
    public class Input
    {
        public Input(Circuit parentCircuit) { this.parentCircuit = parentCircuit; }

        /// <summary>
        /// Whether the input is powered.
        /// </summary>
        private bool powered;

        /// <summary>
        /// The circuit the input composes.
        /// </summary>
        private Circuit parentCircuit;

        /// <summary>
        /// The connection related to the input, if any.
        /// </summary>
        private CircuitConnector.Connection connection;

        /// <summary>
        /// Contains the material that visually displays whether the input is powered or not.
        /// </summary>
        private MeshRenderer statusRenderer;

        /// <summary>
        /// The output connecting to the input, if any.
        /// </summary>
        private Output parentOutput;

        /// <summary>
        /// Transform of the GameObject representing the input, if any.
        /// </summary>
        private Transform transform;

        // Getter and setter methods
        public bool Powered { get { return powered; } set { powered = value; } }

        public Circuit ParentCircuit { get { return parentCircuit; } set { parentCircuit = value; } }

        public CircuitConnector.Connection Connection { get { return connection; } set { connection = value;  } }

        public MeshRenderer StatusRenderer { get { return statusRenderer; } set { statusRenderer = value; } }

        public Output ParentOutput { get { return parentOutput; } set { parentOutput = value; } }

        public Transform Transform { get { return transform; } set { transform = value; } }
    }

    /// <summary>
    /// Output represents all required members of an output node that belong to a circuit.<br/><br/>
    /// An output can more than one connection.
    /// </summary>
    public class Output
    {
        public Output(Circuit parentCircuit) { this.parentCircuit = parentCircuit; }

        /// <summary>
        /// Whether the output is powered.
        /// </summary>
        private bool powered;

        /// <summary>
        /// The circuit the output composes.
        /// </summary>
        private Circuit parentCircuit;

        /// <summary>
        /// The connections related to the output, if any.
        /// </summary>
        private List<CircuitConnector.Connection> connections = new List<CircuitConnector.Connection>();

        /// <summary>
        /// The inputs connecting to the output, if any.
        /// </summary>
        private List<Input> childInputs = new List<Input>();

        /// <summary>
        /// Contains the material that visually displays whether the output is powered or not.
        /// </summary>
        private MeshRenderer statusRenderer;

        /// <summary>
        /// Transform of the GameObject representing the output, if any.
        /// </summary>
        private Transform transform;

        // Getter and setter methods
        public bool Powered { get { return powered; } set { powered = value; } }

        public Circuit ParentCircuit { get { return parentCircuit; } set { parentCircuit = value; } }

        public List<CircuitConnector.Connection> Connections { get { return connections; } set { connections = value; } }

        public List<Input> ChildInputs { get { return childInputs; } set { childInputs = value; } }

        public MeshRenderer StatusRenderer { get { return statusRenderer; } set { statusRenderer = value; } }

        public Transform Transform { get { return transform; } set { transform = value; } }
    }

    /// <summary>
    /// UpdateCall represents an attempt to alter an input node from a given output node.<br/><br/>
    /// An update call does not occur instantly, rather after <seealso cref="clockSpeed"/> seconds have passed.<br/>
    /// This prevents any potential stack overflows caused by loops within circuits.
    /// </summary>
    public class UpdateCall
    {
        /// <summary>
        /// Whether the input should be powered.
        /// </summary>
        private bool powered;

        /// <summary>
        /// The input pertaining to this update call.
        /// </summary>
        private Input input;

        /// <summary>
        ///  The output pertaining to this update call.
        /// </summary>
        private Output output;

        public UpdateCall(bool powered, Input input, Output output)
        {
            this.powered = powered;
            this.input = input;
            this.output = output;
        }

        // Getter methods
        public bool Powered { get { return powered; } }

        public Input Input { get { return input; } }

        public Output Output { get { return output; } }
    }

    /// <summary>
    /// Utilized by custom circuits to initialize a circuit with a variable number of input and output nodes.<br/><br/>
    /// With this constructor, it is expected that <seealso cref="Inputs"/> and <seealso cref="Outputs"/> will be overriden within <see cref="CustomCircuit"/>.
    /// </summary>
    /// <param name="circuitName">Name of the circuit.</param>
    /// <param name="startingPosition">Starting position of the circuit.</param>
    public Circuit(string circuitName, Vector2 startingPosition) : this(circuitName, 0, 0, startingPosition, false) { }

    /// <summary>
    /// Utilized by inherited circuits to determine the specific number of input and output nodes.
    /// </summary>
    /// <param name="circuitName">Name of the circuit.</param>
    /// <param name="numInputs">Number of inputs associated with the circuit.</param>
    /// <param name="numOutputs">Number of outputs associated with the circuit.</param>
    /// <param name="startingPosition">Starting position of the circuit.</param>
    public Circuit(string circuitName, int numInputs, int numOutputs, Vector2 startingPosition) : this(circuitName, numInputs, numOutputs, startingPosition, true) { }

    /// <summary>
    /// Primary constructor that all other constructors reference.
    /// </summary>
    /// <param name="circuitName">Name of the circuit.</param>
    /// <param name="numInputs">Number of inputs associated with the circuit.</param>
    /// <param name="numOutputs">Number of outputs associated with the circuit.</param>
    /// <param name="startingPosition">Starting position of the circuit.</param>
    /// <param name="createIO">Whether each input and output should be initialized.</param>
    public Circuit(string circuitName, int numInputs, int numOutputs, Vector2 startingPosition, bool createIO)
    {
        this.circuitName = circuitName;

        // Initializes inputs and outputs if specified
        if (createIO)
        {
            inputs = new Input[numInputs];
            outputs = new Output[numOutputs];

            for (int i = 0; i < numInputs; i++) { inputs[i] = new Input(this); }

            for (int i = 0; i < numOutputs; i++) { outputs[i] = new Output(this); }
        }

        /* Determines whether this circuit is meant to be visible.
         * Within this project, Vector2.PositiveInfinity implicitly defines an invisible circuit.
         * The only circuits that are invisible are ones that are part of custom circuits.
         */
        visible = startingPosition.x != float.PositiveInfinity && startingPosition.y != float.PositiveInfinity;

        // Creates a corresponding mesh if the circuit is visible.
        if (visible) CircuitVisualizer.Instance.VisualizeCircuit(this, startingPosition);
    }

    /// <summary>
    /// Alternate signature of UpdateCircuit() that assumes the specified output is not null.
    /// </summary>
    /// <param name="input">The input to update.</param>
    /// <param name="output">The output that caused the update.</param>
    public static void UpdateCircuit(Input input, Output output) { UpdateCircuit(output.Powered, input, output); }

    /// <summary>
    /// Updates the circuit belonging to the specified input based on the given power status.<br/><br/>
    /// Afterward, the circuit belonging to the specified input will update all circuits connected to its output(s).
    /// </summary>
    /// <param name="powered">Whether the specified input should be powered.</param>
    /// <param name="input">The input to update.</param>
    /// <param name="output">The output that caused the update.</param>
    public static void UpdateCircuit(bool powered, Input input, Output output)
    {
        input.Powered = powered;

        // Updates input power status material, if applicable
        if (input.StatusRenderer != null) input.StatusRenderer.material = powered ? CircuitVisualizer.Instance.PowerOnMaterial : CircuitVisualizer.Instance.PowerOffMaterial;

        // Updates the connection wire material associated with the input, if applicable
        if (input.Connection != null) CircuitConnector.UpdateConnectionMaterial(input.Connection, powered);

        input.ParentOutput = output;
        input.ParentCircuit.Update();
        input.ParentCircuit.UpdateChildren();
    }

    /// <summary>
    /// Obtains the outputs that should be accessed by <seealso cref="UpdateChildren"/> as well as updating their <seealso cref="Output.statusRenderer"/> materials.
```

```csharp
    /// </summary>
    public void Update()
    {
        // If all outputs should be checked, disregard any potential short circuiting optimization.
        bool shouldCheckAllOutputs = customCircuit != null && customCircuit.finalOutputs.Count > 0;

        outputsToUpdate = UpdateOutputs();

        if (shouldCheckAllOutputs) { outputsToUpdate = Outputs.ToList(); }

        UpdateStatuses();
    }

    /// <summary>
    /// Calls and updates all connections associated to each valid output.<br/><br/>
    /// This method can be called recursively, i.e. trigger a chain reaction.
    /// </summary>
    public void UpdateChildren()
    {
        List<UpdateCall> updateList = new List<UpdateCall>();

        foreach (Output output in outputsToUpdate)
        {
            if (customCircuit != null && customCircuit.finalOutputs.Contains(output)) customCircuit.finalOutputs.Remove(output);

            foreach (Input input in output.ChildInputs) updateList.Add(new UpdateCall(output.Powered, input, output));
        }

        CircuitCaller.InitiateUpdateCalls(updateList);
    }

    /// <summary>
    /// Updates the materials of each valid output.
    /// </summary>
    private void UpdateStatuses()
    {
        foreach (Output output in outputsToUpdate)
        {
            if (output.StatusRenderer == null) continue;

            output.StatusRenderer.material = output.Powered ? CircuitVisualizer.Instance.PowerOnMaterial :
                  CircuitVisualizer.Instance.PowerOffMaterial;
        }
    }

    /// <summary>
    /// Abstract implementation representing the input to output logic of a circuit.<br/>
    /// Utilizes all inputs to recalculate the power status of each output.
    /// </summary>
    /// <returns>The list of outputs that have changed before and during this method.</returns>
    protected abstract List<Output> UpdateOutputs();

    // Getter and setter methods
    public bool Visible { get { return visible; } set { visible = value; } }

    public GameObject PhysicalObject { get { return physicalObject; } set { physicalObject = value; } }

    public Input[] Inputs { get { return inputs; } set { inputs = value; } }

    public Output[] Outputs { get { return outputs; } set { outputs = value; } }

    public string CircuitName { get { return circuitName; } set { circuitName = value; } }
}
```