```
using System;
using System.Collections.Generic;
using TMPro;
using UnityEditor;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class TaskbarManager : MonoBehaviour
{
    // Singleton state reference
    private static TaskbarManager instance;

    /// <summary>
    /// The color associated with starting and custom circuits respectively.
    /// </summary>
    [SerializeField]
    Color startingCircuitColor,
        customCircuitColor;

    /// <summary>
    /// Horizontal length of the scroll bar attached to the bookmarks menu.
    /// </summary>
    [SerializeField]
    float bookmarkScrollThickness;

    /// <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 background;

    /// <summary>
    /// The scroll bar belonging to the bookmarks menu.<br/><br/>
    /// If the size of bookmarks exceeds the view area of the bookmarks menu, this scroll bar is set to visible.
    /// </summary>
    [SerializeField]
    GameObject bookmarkScrollbar;

    /// <summary>
    /// GameObject that all bookmarks are instantiated under.
    /// </summary>
    [SerializeField]
    GameObject bookmarksPanel;

    /// <summary>
    /// Referenced when moving the bookmarks menu to the current user mouse position.
    /// </summary>
    [SerializeField]
    GameObject bookmarksScroll;

    /// <summary>
    /// List of all menus that can be active in the editor scene.
    /// </summary>
    [SerializeField]
    GameObject addMenu, // The menu in which the user can add/bookmark circuits.
        bookmarksMenu, // The menu that displays bookmarked circuits, if any.
        circuitSaveErrorMenu, // The menu indicating the reason a custom circuit has failed to create.
        guide, // The guide prefab.
        labelMenu, // The menu where the user labels empty inputs/outputs.
        notifierPanel, // An empty menu without a user-driven exit scheme; has UI indicating why (e.g. saving).
        nullState, // An empty menu without a user-driven exit scheme; does not have UI.
        saveWarning, // The menu prompting the user to save.
        sceneSaveMenu; // The menu in which the user can either save the editor scene or create a custom circuit.

    /// <summary>
    /// Prefab button for custom circuits within the add menu.
    /// </summary>
    [SerializeField]
    GameObject customBookmarkRef;

    /// <summary>
    /// Prefab button for any circuit within the bookmarks menu.
    /// </summary>
    [SerializeField]
    GameObject bookmarkRef;

    /// <summary>
    /// Exits out of <seealso cref="currentMenu"/>.<br/><br/>
    /// More often than not, an alternate input to achieve the same effect is pressing the right mouse button.
    /// </summary>
    [SerializeField]
    KeyCode cancelKey;

    /// <summary>
    /// Parent of all starting and custom circuits buttons within the add
      menu.
    /// </summary>
    [SerializeField]
    RectTransform addCustomPanel,
        addStartingPanel;

    /// <summary>
    /// Transform of the border behind the bookmarks menu.
    /// </summary>
    [SerializeField]
    RectTransform bookmarksBorder;

    /// <summary>
    /// Displays the reason for a custom circuit failing to create.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI circuitErrorText;

    /// <summary>
    /// Prompts the user to compose a label for an empty input or output.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI labelText;

    /// <summary>
    /// Utilized with <seealso cref="notifierPanel"/> to display the
      reason why a game has disabled all input to the player.<br/><br/>
    /// Its primary uses are for when the game is saving as well as when a
       custom circuit is being verified/created.
    /// </summary>
    [SerializeField]
    TextMeshProUGUI notifierText;

    /// <summary>
    /// The name field with which the user specifies the name of a
      prospective custom circuit.
    /// </summary>
    [SerializeField]
    TMP_InputField circuitNameField;

    /// <summary>
    /// Whether a custom circuit should be created.<br/>
    /// If unchecked, then the current editor scene is saved instead.
    /// </summary>
    [SerializeField]
    Toggle circuitToggle;

    /// <summary>
    /// The size of the bookmark view area and a bookmark respectively.
    /// </summary>
    [SerializeField]
    Vector2 bookmarkMaskSize,
        bookmarkSize;

    /// <summary>
    /// Whether the left mouse button is currently help down whilst in the bookmarks menu.<br/>
    /// This helps discern whether the initial left mouse button press and release occurred when hovered on UI elements.
    /// </summary>
    private bool bookmarksDown;

    /// <summary>
    /// Whether the game is currently deserializing all bookmarks belonging to the current editor scene.
    /// </summary>
    private bool currentlyRestoring;

    /// <summary>
    /// Whether the bookmarks bar can be opened.<br/>
    /// This value is false until the cooldown to open the bookmarks bar passes, enabling it again.
    /// </summary>
    private bool reopenBookmarks = true;

    /// <summary>
    /// The menu currently opened within the editor scene.
    /// </summary>
    private GameObject currentMenu;

    /// <summary>
    /// The ID list of bookmarks in the scene.<br/><br/>
    /// Helps to differentiate whether the bookmark is a starting or custom circuit, since all starting circuits have an ID of -1.
    /// </summary>
    private List<int> bookmarkIDs = new List<int>();

    /// <summary>
    /// The typed list of bookmarks in the scene.
    /// </summary>
    private List<Type> bookmarks = new List<Type>();

    // Enforces a singleton state pattern and disables frame-by-frame update calls.
    private void Awake()
    {
```

```csharp
        if (instance != null)
        {
            Destroy(this);
            throw new Exception("TaskbarManager instance already established; terminating.");
        }

        instance = this;
        enabled = false;
    }

    // Contains input listening for each user-controllable control interface.
    private void Update()
    {
        // No menu is currently opened, skips current frame
        if (currentMenu == nullState) return;

        // Bookmark control scheme
        if (currentMenu == bookmarksMenu)
        {
            // Registers left mouse button down
            if (Input.GetMouseButtonDown(0) && !EventSystem.current.IsPointerOverGameObject()) { bookmarksDown = true; }

            // Exit scheme (occurs if left mouse button is released, but not while hovered on UI.
            else if (Input.GetMouseButtonUp(0) && bookmarksDown)
            {
                if (EventSystem.current.IsPointerOverGameObject()) bookmarksDown = false; else CloseMenu();
            }

            // Moves bookmark to new mouse position
            else if (Input.GetMouseButtonDown(1) && !EventSystem.current.IsPointerOverGameObject()) UpdateBookmarkPosition();

            // Keyboard exit scheme
            else if (Input.GetKeyDown(cancelKey)) CloseMenu();
        }

        // Default control scheme for all other menus allowed to be existed by the user.
        else if (currentMenu == addMenu || currentMenu == sceneSaveMenu || currentMenu == circuitSaveErrorMenu || currentMenu == saveWarning || currentMenu == guide)
        {
            if (Input.GetKeyDown(cancelKey) || Input.GetMouseButtonDown(1))
            {
                if (currentMenu == circuitSaveErrorMenu) ConfirmError(); else CloseMenu();
            }
        }
    }

    /// <summary>
    /// Opens the <seealso cref="nullState"/> interface.<br/>
    /// Essentially disables the taskbar from functioning; a locked state that must be manually closed via script.
    /// </summary>
    public void NullState() { OpenMenu(false, nullState); }

    /// <summary>
    /// Updates <seealso cref="circuitSaveErrorMenu"/> after <seealso cref="circuitToggle"/> is pressed; called by pressing an in-scene button.
    /// </summary>
    public void UpdateSaveToggle()
    {
        bool isOn = circuitToggle.isOn;

        circuitNameField.interactable = isOn;

        if (!isOn) circuitNameField.text = "";
    }

    /// <summary>
    /// Goes back to the menu; called by pressing an in-scene button.
    /// </summary>
    public void OpenOptions()
    {
        // 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.
        if (EditorStructureManager.Instance != null && EditorStructureManager.Instance.DisplaySavePrompt) OpenMenu(true, saveWarning); else SceneManager.LoadScene(0);
    }

    /// <summary>
    /// Goes back to the game menu; called by pressing an in-scene button.
    /// </summary>
    public void OpenMenuScene() { SceneManager.LoadScene(0); }

    /// <summary>
    /// Opens <seealso cref="labelMenu"/> <see cref="IOAssigner"/> is 
      enabled and the user presses LMB on an incomplete empty input/
      output.
    /// </summary>
    /// <param name="isInput"></param>
    public void OpenLabelMenu(bool isInput)
    {
        OpenMenu(true, labelMenu);
        labelText.text = "compose a label for the selected " + (isInput ? 
          "input" : "output");
    }

    /// <summary>
    /// Opens <seealso cref="guide"/>; called by pressing an in-scene 
      button.
    /// </summary>
    public void OpenGuide() { OpenMenu(true, guide); }

    /// <summary>
    /// Opens <seealso cref="sceneSaveMenu"/>; called by pressing an in-
      scene button.
    /// </summary>
    public void OpenSave() { OpenMenu(true, sceneSaveMenu); }

    /// <summary>
    /// Displays an error message if saving a custom circuit fails.
    /// </summary>
    /// <param name="errorMessage">The error message to display.</param>
    public void CircuitSaveError(string errorMessage)
    {
        CloseMenu();
        circuitErrorText.text = errorMessage;
        OpenMenu(true, circuitSaveErrorMenu);
    }

    /// <summary>
    /// Closes <seealso cref="circuitSaveErrorMenu"/>; can be called by 
      pressing an in-scene button.
    /// </summary>
    public void ConfirmError()
    {
        CloseMenu();
        OpenSave();
    }

    /// <summary>
    /// Confirms <seealso cref="sceneSaveMenu"/> input and either saves 
      the editor scene or creates a custom circuit based on <seealso 
      cref="circuitToggle"/>.<br/>
    /// Called by pressing an in-scene button.
    /// </summary>
    public void SaveConfirm()
    {
        CloseMenu();

        // Should attempt to create custom circuit
        if (circuitToggle.isOn)
        {
            notifierText.text = "verifying...";
            OpenMenu(true, notifierPanel);
            PreviewStructureManager.Instance.VerifyPreviewStructure(circuitNameField.text.ToLower().Trim());
        }

        // Should save scene
        else
        {
            notifierText.text = "saving scene...";
            OpenMenu(true, notifierPanel);
            EditorStructureManager.Instance.Serialize();
        }
    }

    /// <summary>
    /// Opens <seealso cref="addMenu"/>; called by pressing an in-scene button.
    /// </summary>
    public void OpenAdd() { OpenMenu(true, addMenu); }

    /// <summary>
    /// Opens the bookmarks menu.
    /// </summary>
    public void OpenBookmarks() { OpenBookmarks(false); }

    /// <summary>
    /// Called when a custom circuit is successfully created.
    /// </summary>
    public void OnSuccessfulPreviewStructure()
    {
        circuitNameField.text = "";
        CloseMenu();
    }

    /// <summary>
    /// Called when a custom circuit successfully passes validation.
    /// </summary>
    public void OnSuccessfulPreviewVerification()
    {
        CloseMenu();
        OpenMenu(true, notifierPanel);
        notifierText.text = "creating...";
    }

    /// <summary>
    /// Opens the bookmarks menu.<br/>
    /// If there are no bookmarks to display, this method does nothing.
    /// </summary>
    /// <param name="showBackground"></param>
    public void OpenBookmarks(bool showBackground)
    {
        if (bookmarks.Count == 0) return;

        bookmarksDown = false;
        OpenMenu(showBackground, bookmarksMenu);
    }

    /// <summary>
    /// Deserializes all bookmarks stored in the current editor structure.
    /// </summary>
    /// <param name="circuitIndeces">The serialized integer to circuit identifiers.</param>
    /// <param name="circuitIDs">The preview structure IDs of each circuit (-1 if non-custom).</param>
    public void RestoreBookmarks(List<int> circuitIndeces, List<int> circuitIDs)
    {
        int index = 0;

        currentlyRestoring = true;

        foreach (int circuitIndex in new List<int>(circuitIndeces))
        {
            // Is a custom circuit
            if (circuitIndex != -1)
            {
                Toggle toggle = addStartingPanel.GetChild(circuitIndex).GetComponentInChildren<Toggle>();

                toggle.isOn = true;
                UpdateBookmarkAll(toggle.gameObject);
            }

            // Is a starting circuit
            else AddCustomCircuitPanel(circuitIDs[index], true);

            index++;
        }
```

```csharp

            currentlyRestoring = false;
        }

        /// <summary>
        /// Adds all non-bookmarked custom circuits belonging to the current preview structure back to <seealso cref="addMenu"/>.
        /// </summary>
        public void RestoreCustomCircuits()
        {
            foreach (PreviewStructure previewStructure in MenuSetupManager.Instance.PreviewStructures)
            {
                // Is bookmarked, continue.
                if (bookmarkIDs.Contains(previewStructure.ID)) continue;

                AddCustomCircuitPanel(previewStructure.ID, false);
            }
        }

        /// <summary>
        /// Adds a custom circuit to <seealso cref="addCustomPanel"/>.
        /// </summary>
        /// <param name="circuitID">The custom circuit ID.</param>
        /// <param name="bookmarked">Whether this circuit is bookmarked in the current editor scene.</param>
        public void AddCustomCircuitPanel(int circuitID, bool bookmarked)
        {
            GameObject current = Instantiate(customBookmarkRef, addCustomPanel.transform); // Instantiates a prefab copy
            Toggle toggle = current.GetComponentInChildren<Toggle>();
            PreviewStructure.PreviewStructureReference reference = current.AddComponent<PreviewStructure.PreviewStructureReference>();

            current.GetComponentInChildren<TextMeshProUGUI>().text = MenuSetupManager.Instance.PreviewStructures[MenuSetupManager.Instance.PreviewStructureIDs.IndexOf(circuitID)].Name;
            reference.ID = circuitID;

            // Adds listeners required to add and bookmark the custom circuit.
            current.GetComponentInChildren<Button>().onClick.AddListener(delegate { AddBookmarkCircuit(-1, reference.ID); });
            toggle.onValueChanged.AddListener(delegate { UpdateBookmark(reference); });

            if (bookmarked)
            {
```

```
            toggle.isOn = true;
            UpdateBookmarkCustom(reference);
        }
    }

    /// <summary>
    /// Identical to <seealso cref="UpdateBookmarkAll(GameObject)"/> except reserved specifically for <see cref="Toggle"/> calls.<br/><br/>
    /// This is due to boolean changes within scripting also triggering <seealso cref="Toggle.onValueChanged"/> events.
    /// </summary>
    /// <param name="obj">The bookmark to update.</param>
    public void UpdateBookmark(GameObject obj)
    {
        // If toggles are being adjusted within the script, do not continue.
        if (currentlyRestoring) return;

        UpdateBookmarkAll(obj);
    }

    /// <summary>
    /// Identical to <seealso cref="UpdateBookmarkCustom(PreviewStructure.PreviewStructureReference)"/> except reserved specifically for <see cref="Toggle"/> calls.<br/><br/>
    /// This is due to boolean changes within scripting also triggering <seealso cref="Toggle.onValueChanged"/> events.
    /// </summary>
    /// <param name="obj">The bookmark to update.</param>
    public void UpdateBookmark(PreviewStructure.PreviewStructureReference previewStructureReference)
    {
        // If toggles are being adjusted within the script, do not continue.
        if (currentlyRestoring) return;

        UpdateBookmarkCustom(previewStructureReference);
    }

    /// <summary>
    /// Updates a starting bookmark after it has been bookmarked or unbookmarked.
    /// </summary>
    /// <param name="obj">The starting bookmark to update.</param>
    public void UpdateBookmarkAll(GameObject obj)
    {
        bool newStatus = obj.GetComponent<Toggle>().isOn;
        Type type = CircuitType(obj.transform.parent.GetSiblingIndex());
```

```csharp

        // Adds bookmark
        if (newStatus && !bookmarks.Contains(type))
        {
            if (!currentlyRestoring) EditorStructureManager.Instance.DisplaySavePrompt = true;

            EditorStructureManager.Instance.Bookmarks.Add(StartingCircuitIndex(type));
            bookmarks.Add(type);
            bookmarkIDs.Add(-1);

            GameObject bookmark = Instantiate(bookmarkRef, bookmarksPanel.transform);
            Button button = bookmark.GetComponentInChildren<Button>();
            TextMeshProUGUI text = bookmark.GetComponentInChildren<TextMeshProUGUI>();

            bookmark.name = text.text = obj.transform.parent.name;
            text.color = startingCircuitColor;

            // Ensures pressing on the bookmark will add its representative circuit.
            button.onClick.AddListener(delegate { AddBookmarkCircuit(StartingCircuitIndex(type), -1); });
        }

        // Deletes bookmark
        else if (!newStatus && bookmarks.Contains(type))
        {
            if (!currentlyRestoring) EditorStructureManager.Instance.DisplaySavePrompt = true;

            int index = bookmarks.IndexOf(type);

            EditorStructureManager.Instance.Bookmarks.Remove(StartingCircuitIndex(type));
            bookmarks.Remove(type);
            bookmarkIDs.RemoveAt(index);
            Destroy(bookmarksPanel.transform.GetChild(index).gameObject);
        }
    }

    /// <summary>
    /// Updates a custom bookmark after it has been bookmarked or unbookmarked.
    /// </summary>
    /// <param name="obj">The custom bookmark to update.</param>
    public void UpdateBookmarkCustom
```

```
        (PreviewStructure.PreviewStructureReference reference)
    {
        bool newStatus = reference.GetComponentInChildren<Toggle>().isOn;
        int id = reference.GetComponentInChildren<PreviewStructure.PreviewStructureReference>().ID;

        // Adds bookmark
        if (newStatus && !bookmarkIDs.Contains(id))
        {
            if (!currentlyRestoring) EditorStructureManager.Instance.DisplaySavePrompt = true;

            EditorStructureManager.Instance.Bookmarks.Add(-1);
            bookmarks.Add(typeof(CustomCircuit));
            bookmarkIDs.Add(id);

            GameObject bookmark = Instantiate(bookmarkRef, bookmarksPanel.transform);
            Button button = bookmark.GetComponentInChildren<Button>();
            TextMeshProUGUI text = bookmark.GetComponentInChildren<TextMeshProUGUI>();

            bookmark.name = text.text = MenuSetupManager.Instance.PreviewStructures[MenuSetupManager.Instance.PreviewStructureIDs.IndexOf(id)].Name;
            text.color = customCircuitColor;

            // Ensures pressing on the bookmark will add its representative circuit.
            button.onClick.AddListener(delegate { AddBookmarkCircuit(-1, id); });
        }

        // Deletes bookmark
        else if (!newStatus && bookmarkIDs.Contains(id))
        {
            if (!currentlyRestoring) EditorStructureManager.Instance.DisplaySavePrompt = true;

            int index = bookmarkIDs.IndexOf(id);

            EditorStructureManager.Instance.Bookmarks.RemoveAt(index);
            bookmarks.RemoveAt(index);
            bookmarkIDs.Remove(id);
            Destroy(bookmarksPanel.transform.GetChild(index).gameObject);
        }
    }

    /// <summary>
    /// Adds a bookmarked circuit based on its circuit type and custom circuit ID (if applicable).
    /// </summary>
    /// <param name="circuitType">Index representing the circuit type.</param>
    /// <param name="circuitID">The custom circuit ID (-1 if custom).</param>
    private void AddBookmarkCircuit(int circuitType, int circuitID)
    {
        switch (circuitType)
        {
            // Custom circuit
            case -1:
                AddCircuit(new CustomCircuit(MenuSetupManager.Instance.PreviewStructures[MenuSetupManager.Instance.PreviewStructureIDs.IndexOf(circuitID)]));
                return;
            // Input
            case 0:
                AddCircuit(new InputGate());
                return;
            // Display
            case 1:
                AddCircuit(new Display());
                return;
            // Buffer
            case 2:
                AddCircuit(new Buffer());
                return;
            // And gate
            case 3:
                AddCircuit(new AndGate());
                return;
            // NAnd gate
            case 4:
                AddCircuit(new NAndGate());
                return;
            // NOr gate
            case 5:
                AddCircuit(new NOrGate());
                return;
            // Not gate
            case 6:
                AddCircuit(new NotGate());
                return;
            // Or gate
            case 7:
                AddCircuit(new OrGate());
                return;
            // XOr gate
            case 8:
                AddCircuit(new XOrGate());
                return;
        }
    }

    /// <summary>
    /// Adds a starting circuit to the scene; called by pressing an in-scene button.
    /// </summary>
    /// <param name="startingCircuitIndex">Representative index of the starting circuit.</param>
    public void AddStartingCircuit(int startingCircuitIndex) { AddCircuit(GetStartingCircuit(startingCircuitIndex)); }

    /// <summary>
    /// Adds a circuit to the scene.
    /// </summary>
    /// <param name="newCircuit">The circuit to add.</param>
    private void AddCircuit(Circuit newCircuit)
    {
        // Cancels any modes that would obstruct the placement process.
        switch (BehaviorManager.Instance.UnpausedGameState)
        {
            case BehaviorManager.GameState.CIRCUIT_MOVEMENT:
                BehaviorManager.Instance.CancelCircuitMovement();
                break;

            case BehaviorManager.GameState.IO_PRESS:
                BehaviorManager.Instance.CancelWirePlacement();
                break;
        }

        // Switches to placement mode
        BehaviorManager.Instance.UnpausedGameState = BehaviorManager.GameState.CIRCUIT_PLACEMENT;
        BehaviorManager.Instance.UnpausedStateType = BehaviorManager.StateType.LOCKED;
        BehaviorManager.Instance.CircuitPlacement(newCircuit);
        CloseMenu();
    }

    /// <summary>
    /// Obtains the circuit type based on its circuit index.
    /// </summary>
    /// <param name="circuitIndex">The index of the circuit.</param>
    /// <returns>The type of the circuit.</returns>
    private Type CircuitType(int circuitIndex)
    {
        switch (circuitIndex)
        {
            case -1:
                return typeof(CustomCircuit);
            case 0:
                return typeof(InputGate);
            case 1:
                return typeof(Display);
            case 2:
                return typeof(Buffer);
            case 3:
                return typeof(AndGate);
            case 4:
                return typeof(NAndGate);
            case 5:
                return typeof(NOrGate);
            case 6:
                return typeof(NotGate);
            case 7:
                return typeof(OrGate);
            case 8:
                return typeof(XOrGate);
            default:
                throw new Exception("Invalid starting circuit index.");
        }
    }

    /// <summary>
    /// Obtains the index representation of a starting circuit.
    /// </summary>
    /// <param name="circuitType">The type of the starting circuit.</
      param>
    /// <returns>The index representation of the circuit.</returns>
    private int StartingCircuitIndex(Type circuitType)
    {
        if (circuitType == typeof(InputGate)) return 0;

        else if (circuitType == typeof(Display)) return 1;

        else if (circuitType == typeof(Buffer)) return 2;

        else if (circuitType == typeof(AndGate)) return 3;

        else if (circuitType == typeof(NAndGate)) return 4;

        else if (circuitType == typeof(NOrGate)) return 5;

        else if (circuitType == typeof(NotGate)) return 6;

        else if (circuitType == typeof(OrGate)) return 7;

        else if (circuitType == typeof(XOrGate)) return 8;

        else throw new Exception("Invalid starting circuit type.");
    }

    /// <summary>
    /// Opens a menu.
    /// </summary>
    /// <param name="showBackground">Whether <seealso cref="background"/> should be visible.</param>
    /// <param name="newMenu">The menu to open.</param>
    private void OpenMenu(bool showBackground, GameObject newMenu)
    {
        // If another menu is open, do nothing.
        if (currentMenu != null && currentMenu != bookmarksMenu) return;

        // Close the bookmarks menu if another menu is opened.
        if (currentMenu == bookmarksMenu) CloseMenu();

        currentMenu = newMenu;

        // If applicable, the bookmarks menu should open around the user's cursor.
        if (newMenu == bookmarksMenu)
        {
            UpdateBookmarkPosition();
            UpdateBookmarkScroll();
        }

        BehaviorManager.Instance.LockUI = true;
        background.SetActive(showBackground); currentMenu.SetActive(true);
        enabled = true; // Enables the frame-by-frame listener.
    }

    /// <summary>
    /// Updates the size of the bookmarks menu and enables/disables the scroll bar.
    /// </summary>
    private void UpdateBookmarkScroll()
    {
        // If the bookmarks menu does not show all bookmarked circuts, the vertical scroll bar should appear.
        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;
        borderPosition.y += Mathf.Clamp(0, downVal - bookmarks.Count * bookmarkSize.y / 2, bookmarkMaskSize.y);
        bookmarksBorder.anchoredPosition = borderPosition; // Moves the bookmarks border to the new position
    }

    /// <summary>
    /// Closes the currently opened menu.
    /// </summary>
    public void CloseMenu()
    {
        BehaviorManager.Instance.LockUI = false;
        reopenBookmarks = false;
        Invoke("UnlockUI", 0.1f);
        background.SetActive(false); currentMenu.SetActive(false);

        if (currentMenu == addMenu) addStartingPanel.anchoredPosition = addCustomPanel.anchoredPosition = Vector2.zero;

        currentMenu = null;
        enabled = false;
    }

    /// <summary>
    /// Allows the bookmarks menu to be opened; called by invokement within this script.
    /// </summary>
    private void UnlockUI() { reopenBookmarks = true; }

    /// <summary>
    /// Creates a starting circuit from its index representation.
    /// </summary>
    /// <param name="startingCircuitIndex">Index of the starting circuit.</param>
    /// <returns>The newly created circuit.</returns>
    private Circuit GetStartingCircuit(int startingCircuitIndex)
    {
        switch (startingCircuitIndex)
        {
            case 0:
                return new InputGate();
            case 1:
                return new Display();
            case 2:
                return new Buffer();
            case 3:
                return new AndGate();
            case 4:
                return new NAndGate();
            case 5:
                return new NOrGate();
            case 6:
                return new NotGate();
            case 7:
                return new OrGate();
            case 8:
                return new XOrGate();
            default:
                throw new Exception("Invalid starting circuit index.");
        }
    }

    /// <summary>
    /// Serializes the current editor scene; called by pressing an in-scene button.
    /// </summary>
    public void Serialize() { EditorStructureManager.Instance.Serialize(); }

    // Getter methods
    public static TaskbarManager Instance { get { return instance; } }

    public bool ReopenBookmarks { get { return reopenBookmarks; } }

    public GameObject CurrentMenu { get { return currentMenu; } }

    public List<int> BookmarkIDs { get { return bookmarkIDs; } }
}
```