Documentation

Blend Reality

Reggie Segovia - Project Manager

Michael Termotto - Back End Developer

Guanyu Chen - Back End Developer

CIS4914 Senior Project

Advisor - Dr. Jörg Peters | jorg@cise.ufl.edu

12/03/2025

User Documentation

Deployment Instructions

1. **Environment Details:** Unity 6.x, Visual Studio/VS Code. Target platform: Oculus Quest 2.
2. **Configuration:** Ensure the Oculus XR Plugin and XR Interaction Toolkit are enabled in the Unity Package Manager. Set the build target to Android with required minimum API level.
3. **Deployment Steps:**
    - Clone the repository: git clone https://github.com/rs-dkd/BlendReality.git
    - Open the project in Unity.
    - Select File > Build Settings. Choose Android and ensure the Oculus device is connected.
    - Click Build And Run to deploy the .apk directly to the headset.

Basic Movement & Controls

1. **Movement:** Left controller - joystick
2. **Look:** Rotate headset
3. **Rotate:** Right controller - joystick
4. **Settings Panel:** Left Controller - Primary Button
5. **Obj Explorer:** Left Controller - Secondary Button
6. **Model Creation Panel:** Right Controller - Primary Button
7. **Model Edit Panel:** Right Controller - Secondary Button
8. **Select UI:** Controller - Trigger Button
9. **Select Models/Verts/Edges/Faces:** Controller - Grab Button

Workflow

The Blend Reality Workflow is designed to leverage VR's strengths:

1. **Initial Mesh:** User imports an existing .obj or starts with a primitive object (e.g., cube, cylinder).
2. **Manipulation:** User grabs controllers to manipulate vertices, edges, and faces of the control net using an intuitive, spatial approach. Tools like the Extrude or Inset operations refine the net structure.
3. **Polyhedral Spline Generation:** The user triggers the function from the menu. The control net is processed by the PnS library, and the smooth Bezier surface is instantly rendered over the net.
4. **Export:** The final control net and/or the smooth surface data is exported for use in production pipelines.

VR Key

Purpose

The VR Key prefab is utilized for the construction of the VR Keyboard.

Key Fields and Behavior

- **public enum SpecialKey:** An enumeration for storing special keys such as Space and Control.
- **public string lowerKey:** The field for the key's lowercase character.
- **public string upperKey:** The field for the key's uppercase character.
- **private VRKeyboard keyboard:** A reference to the parent VRKeyboard instance.
- **public Button button:** The UI button object used for callbacks, such as keypressed.
- **public bool isLower:** A boolean to track the key's current mode (lowercase or uppercase).
- **public TMP_Text keyText:** The TMP text component that displays the key's character.

Key Methods and Events

- **public void Setup(VRKeyboard _keyboard):** Initializes the key by storing the keyboard reference and setting the key to lowercase mode.
- **public void OnKeyPress():** The callback function invoked when the key is pressed.
- **public void Switch():** Toggles the key's character between lowercase and uppercase mode.

VR Keyboard

Purpose

To provide a virtual reality (VR) keyboard panel that can be invoked on-demand from any location within the environment to input characters into a designated binding object.

Key Fields and Behavior

- **public static VRKeyboard Instance:** The singleton instance, allowing ubiquitous access to the keyboard throughout the project.
- **public TMP_InputField inputField:** The currently bound TMP_InputField component, which receives the character input.
- **public LazyFollow lazyFollow:** The dedicated VR component responsible for the 'lazy follow' positioning behavior.
- **public GameObject fullKeyboardPanel:** The graphical user interface (GUI) object representing the complete keyboard layout.
- **public GameObject numberKeyboardPanel:** The GUI object representing the numeric-only keyboard layout.
- **public bool isNumber:** A flag indicating whether the numeric keyboard panel or the full keyboard panel is currently displayed.

Key Methods and Events

- **private void Awake():** Implements the Singleton pattern initialization.
- **public GameObject EnableKeyboard():** Deactivates the currently active keyboard layout, then activates and returns the appropriate keyboard panel based on the value of isNumber.
- **public void DisableKeyboard():** Deactivates both the full and numeric keyboard panels.
- **public void BindKeyboard(SliderUI slider):** Establishes a binding between the keyboard and a SliderUI component for numeric input.
- **public void UnbindKeyboard(SliderUI slider):** Removes the binding from the specified SliderUI component.
- **void Start():** Initializes all VRKey objects by invoking their respective initialization methods.
- **public void KeyPress(VRKey key):** Processes the key press event and inputs the corresponding character to the currently bound object based on the key's character value.
- **public void BindKeyboard(TMP_InputField fieldToBind, Transform followTarget, bool isNumeric = false):** Establishes a binding with the designated input field, setting a follow target and optionally configuring it for numeric-only input.
- **public void UnbindKeyboard():** Removes the binding from the current input field.

Model Editing Panel

Purpose

This panel serves as a settings interface for model editing, providing configurable options and functionalities for various edit modes.

Key Fields & Behavior

- **public enum EditMode:** Represents the currently active edit mode.
- **public class ScaleSnapChangedEvent : UnityEvent\<float\>:** Event invoked upon a change in the scale snap value.
- **public class RotateSnapChangedEvent : UnityEvent\<float\>:** Event invoked upon a change in the rotation snap value.
- **public class MoveSnapChangedEvent : UnityEvent\<float\>:** Event invoked upon a change in the movement snap value.
- **public class ControlPointsChangedEvent : UnityEvent:** Event invoked when the collection of control points is modified.
- **public class GizmoSpaceChangedEvent : UnityEvent:** Event invoked when the gizmo space mode is altered.
- **public class TransformTypeChangedEvent : UnityEvent:** Event invoked upon a change in the transform type.
- **public class EditModeChangedEvent : UnityEvent:** Event invoked when the edit mode is switched.
- **public static ModelEditingPanel Instance:** The singleton instance, providing project-wide accessibility.
- **public ToggleGroupUI scaleSnapToggleGroup:** Toggle groups for managing multiple setting selections.
- **public EditMode currentEditMode:** The active edit mode.
- **public GizmoSpace currentGizmoSpace:** The active GizmoSpace.
- **public TransformType currentTransformType:** The active transform type.
- **public GameObjec[] snappingOptions:** GameObjects representing the available snapping options.
- **public bool showSnap:** A flag indicating whether snapping options are currently displayed.
- **public float[] scaleSnapValues:** Constant arrays of floats and strings defining the options and values used for displaying and calculating snapping operations.
- **private float moveSnap**
- **private float rotateSnap**
- **private float scaleSnap:** The current values for each snap option.
- **public ModelData selectedModel:** The model currently selected for editing.
- **public float controlPointSize:** The visual size of the control points.
- **public List<ControlPoint> allControlPoints:** A collection of all instantiated control points.

- **public List<ControlPoint> inUseControlPoints:** The control points that are currently active.
- **public List<ControlPoint> controlPoints:** A list of the currently selected control points.

Key Methods / Events

- **private void Awake():** Implements the singleton design pattern.
- **public EditMode GetEditMode():** Provides read access to the current edit mode.
- **public void SnapToggle():** Toggles the snapping functionality on or off.
- **void Start():** Initializes the toggle group listeners and sets up the snap options.
- **public void OnEditModeToggleGroupChanged(Toggle toggle):** Methods to handle the state change for each of the configurable options.
- **public void TransformTypeChanged():** Manages the logic following a change in the transform type.
- **public void UpdateMoveSnap(float newSnap)**
- **public void UpdateRotateSnap(float newSnap)**
- **public void UpdateScaleSnap(float newSnap):** Updates the respective snap options based on newly provided snap values.
- **public float GetCurrentSnap():** Retrieves the current values for the snap options.
- **public void UpdateEditModel():** Refreshes the control points for the model being edited by deactivating outdated points, creating new ones, or reactivating existing points.
- **private void CreateControlPointForVertex(int[] vertexIndices, Vector3 position, ref int index, Vector3 _normal):** Creates a new control point or reuses an existing one for a specified vertex.
- **private ControlPoint CreateOrReuseControlPoint(ref int index):** Creates a new control point sphere or reuses an inactive one.
- **private ControlPoint CreateOrReuseControlPoint(int[] vertexIndices, EditMode type, Vector3 position, ref int index, Vector3 _normal):** Creates or reuses a control point, adds it to the list of in-use control points, and associates it with vertex indices and the edit mode.
- **public void StopEditModel(ModelData model):** Clears the currently selected model and deactivates all control points that are in use.
- **public void AddOffsetToVertsPosition(int[] _verticesIndexes, Vector3 offset):** Modifies the position of specified vertices by a given offset and updates the edited model.
- **public void UpdateNonSelectedControlPointsPositions():** Recalculates and updates the positions of control points that are not currently selected, following modifications to selected control points.
- **public List<ControlPoint> GetControlPoints():** Provides read access to the list of currently active control points.
- **public void SelectControlPoint(ControlPoint cp):** Clears any previous selection, selects the specified control point, and initiates an update.
- **public void DeSelectControlPoint(ControlPoint cp):** Deselects the specified control point.
- **public void MultiSelectControlPoint(ControlPoint cp):** Adds the control point to the

selection list for multi-selection operations.

- **public void AddOffsetToSelectedControlPoints(ControlPoint currentCP, Vector3 offset):** Applies a specified offset to all currently selected control points and updates their state.

Statistics Panel

Purpose

This panel serves to display various statistics, such as Frames Per Second (FPS) and face count of the current model, providing the user with a more comprehensive understanding of the operational status of the application and the model under modification.

Key Fields & Behavior

- **public TMP_Text fpsText:** A TextMeshPro (TMP) component designated for displaying the current FPS value.
- **public float updateInterval:** The temporal duration between successive updates of the FPS calculation.
- **public float smoothing:** A factor applied to the FPS calculation to achieve a smoother numerical representation.
- **public int goodFPS:** The FPS threshold at or above which the number is displayed in green, signifying optimal performance.
- **public int warnFPS:** The FPS threshold at or above which the number is displayed in yellow, signifying a performance warning.
- **private float timeLeft:** The remaining time until the subsequent FPS update is scheduled.
- **private float accum:** The accumulated FPS values utilized for calculating the average FPS.
- **private int frames:** The count of frames processed within the current FPS update interval.
- **private float displayedFPS:** The calculated and presented average FPS value.
- **public TMP_Text objectVertsText:** TMP components dedicated to displaying the total number of models, vertices, faces, and related metrics within the scene.

Key Methods / Events

- **Start():** Initializes the text components and commences the routine for FPS updates.
- **HandleModelsChanged(List<ModelData>):** Facilitates the update of the total model count and model properties upon a change in the collection of models.
- **HandleSelectionChanged(List<ModelData>):** Facilitates the update of the currently selected model and its properties based on changes in the current selection.
- **Update():** Executes the calculation and update of the FPS at the pre-defined intervals.

OBJExporter

Purpose

This exporter is designed to facilitate the export of OBJ files at runtime, drawing inspiration from the ProBuilder editor object exporter functionality.

Key Fields and Behavior

- **public enum CoordinateHandedness:** Specifies the coordinate system handedness for the export process.
- **public class ObjExportOptions:** Encapsulates the options available for object export.
- **public class RuntimeModel:** Serves as a container for the model data at runtime.
- **public class SubMeshData:** Provides a container for a model's constituent submeshes.
- **static Dictionary<string, string> s_TextureMapKeys:** A static dictionary that maps Unity texture property names to corresponding OBJ material keywords.

Key Methods and Events

- **public bool ExportToObj(List<GameObject> meshes, string filePath, ObjExportOptions options = null):** Executes the export of multiple specified GameObjects to the provided file path, optionally applying the specified export settings.
- **public bool ExportSingle(GameObject go, string filePath, ObjExportOptions options = null):** Exports a single specified GameObject to the provided file path, optionally applying the specified export settings.
- **private bool DoExport(string path, List<RuntimeModel> models, ObjExportOptions options):** Manages the core export process, which includes directory creation, material map construction, generation of OBJ and MTL content, writing files to disk, and copying associated textures.
- **private string WriteMtlContents(List<RuntimeModel> models, Dictionary<Material, string> materialMap, out List<string> texturePaths, ObjExportOptions options):** Generates the content for the MTL (Material Template Library) file by iterating through materials, writing texture maps, and defining color properties.
- **private string WriteObjContents(List<RuntimeModel> models, string filename, Dictionary<Material, string> materialMap, ObjExportOptions options):** Generates the OBJ file content, including the writing of vertex data, merging of coincident vertices, definition of faces, and assignment of materials.
- **private Dictionary<Material, string> BuildMaterialMap(List<RuntimeModel> models):** Constructs a mapping between materials and unique, sanitized names, ensuring that duplicated material names are disambiguated.
- **private RuntimeModel CreateModelFromGameObject(GameObject go):** Extracts the necessary mesh data from a GameObject to construct the runtime model suitable for export.
- **private string SanitizeName(string name):** Cleans the provided name string by

removing invalid characters.

- **private string GetTexturePath(Texture texture, bool needsCopy):** Retrieves the file path for the specified texture, checking the project's texture directory.
- **private void CopyTextures(List<string> texturePaths, string destDir):** Copies texture files from their source paths to the designated destination directory.
- **static int AppendArrayVec2(StringBuilder sb, Vector2[] array, string prefix, bool mergeCoincident, out Dictionary<int, int> coincidentIndexMap):** Writes a Vector2 array to a StringBuilder, with an option for merging coincident vertices.
- **static int AppendArrayVec3(StringBuilder sb, Vector3[] array, string prefix, bool mergeCoincident, out Dictionary<int, int> coincidentIndexMap):** Writes a Vector3 array to a StringBuilder, with an option for merging coincident vertices.

Purpose

This panel facilitates the runtime export of OBJ files, providing users with options to configure the export settings.

Key Fields and Behavior

- **private GameObject mainSettingsContent:** References the GameObject for the settings panel.
- **private GameObject saveDialogPanel:** References the save dialog panel displayed during the export process.
- **private TMP_InputField filenameInput:** References the input field designated for entering the desired export filename.

Key Methods / Events

- **void Start():** Initializes the panel by ensuring the save dialog panel is initially concealed.
- **public void OnExportButtonClicked():** A callback invoked upon the activation of the export button, which subsequently displays the export dialog panel and binds the VR keyboard functionality to the filename input field.
- **public void ConfirmExport():** Executes the file exportation process utilizing the internal OBJExporter component.
- **public void CancelExport():** Aborts the current file exportation operation.
- **private string SanitizeForFilename(string name):** Processes the provided string to ensure it adheres to valid filename conventions.

Metric Converter

Purpose

To provide a utility for performing metric conversion throughout the project, primarily for the slider control that denotes object size.

Key Fields and Behavior

- **public enum GridUnitSystem:** An enumeration specifying the unit system currently in use.
- **public enum ImperialDisplayMode:** An enumeration controlling the display format for imperial units (feet-inches or decimal feet).
- **private const float InchesPerMeter:** A constant float value representing the number of inches in one meter, used for conversion.
- **private const int InchesPerFoot:** A constant integer value representing the number of inches in one foot, used for conversion.

Key Methods/Events

- **public static string ToFeetAndInches(float meters):** A static method that accepts a length in meters and returns the equivalent imperial length.

Axis Indicator

Purpose

This component generates an axis indicator object to assist users in identifying directional orientation within the 3D space.

Key Fields & Behavior

- **private float axisLength:** Specifies the resultant length of the created axis.
- **private float lineWidth:** Defines the width of the generated axis line.
- **private Material xAxisMaterial / private Material yAxisMaterial / private Material zAxisMaterial:** Represents the material employed for each respective axis.

Key Methods / Events

- **private void CreateAxis(string name, Vector3 direction, Material material, Color color):** This method is responsible for constructing the axis, accepting the material and color parameters for the axis line.
- **private GameObject CreateCone(Color color):** This method generates the cone indicator situated at the termination point of the axis, using the specified color input.

Transform Gizmo

Purpose

To provide interactive visualization tools (gizmos) that enhance the manipulation of meshes within a Virtual Reality environment.

Key Fields and Behavior

- **public enum GizmoSpace:** Specifies the coordinate system used by the gizmo.
- **public enum TransformType:** Indicates the currently active transformation operation.
- **public static TransformGizmo Instance:** A singleton instance providing global access to the system.
- **public Transform axisParent:** The parent transform object that holds all axis components.
- **public Transform\[\] axes:** An array containing the transform components for the individual axes, managed by axisParent.
- **public Vector3 previousPosition:** Stores the position of the previously grasped axis to calculate movement differentials.
- **public Transform currentAxis:** A reference to the transform of the axis currently being grasped.
- **public LineRenderer\[\] axesLineRenders:** Line renderers utilized for axis visualization.
- **public CapsuleCollider\[\] axesCapsules:** Collider components facilitating VR grasping and interaction.
- **public XRGeneralGrabTransformer\[\] xrGrabTrans:** XR grab transformers used to constrain movement along the respective axis.
- **public XRGrabInteractable\[\] xrGrabInters:** XR grab interactables enabling VR grasping and interaction with the axes.
- **private Quaternion initialRotation:** Stores the initial rotation of the axis for subsequent calculations.
- **private Vector3 initialDirection:** Stores the initial direction of the grab operation for calculation purposes.

Key Methods and Events

- **private void Awake():** Initializes the singleton instance and disables the axis parent upon object awakening.
- **private void Start():** Registers event listeners for changes in transformation mode, gizmo space, and other relevant system states.
- **public void ControlPointsUpdated():** A callback method invoked when control points are modified.
- **public void UpdateGizmoGroupCenter():** Computes and updates the center position of the gizmo group.

- **public void GizmoSpaceUpdated():** A callback method triggered upon a change in the gizmo space setting.
- **public void TransformModeUpdated():** Adjusts gizmo behavior based on the updated transformation mode.
- **public void SelectAxis(Transform axis):** Invoked when an axis is grasped by the user; stores the initial position and rotation for subsequent calculation.
- **public void DeSelectAxis():** Releases the axis currently grasped by the user.
- **public void ResetAxes():** Restores the axis line renderer and capsule positions to their original state.
- **void Update():** The update loop responsible for governing gizmo behavior based on the user's grab interaction.
- **private Vector3 GetAxisVector(Transform axis):** Returns the axis vector of the input axis transform, relative to the current gizmo space.

Panel Manager

Purpose

To provide docking positions for various panels and to serve as a globally accessible singleton instance.

Key Fields & Behavior

- **public static PanelManager Instance:** The singleton instance for global access.
- **public UIPanel[] leftSidePanels:** An array of UIPanel references intended for docking on the left side of the user interface.
- **public UIPanel[] rightSidePanels:** An array of UIPanel references intended for docking on the right side of the user interface.

Key Methods / Events

- **void Awake():** Implements the Singleton Pattern.
- **private void Start():** Initializes all panels by correctly setting their side property.
- **public void OpenPanel(UIPanel panel):** Opens the specified panel and docks it on the designated side, simultaneously closing any previously docked panel.

UI Panel Documentation

Purpose

The fundamental container object for all panel elements, designed for display by docking onto the panel manager.

Key Fields and Behavior

- **public GameObject panel:** The instantiated prefab game object housed within this panel.
- **private bool isPanelOpen:** A boolean indicator reflecting the panel's current operational state (open or closed).
- **public ControllerInputType inputType:** An enumeration specifying the Virtual Reality (VR) controller responsible for initiating the panel's opening.
- **private InputAction menuAction:** The input action corresponding to the button press designated for this panel.
- **public bool isLeftSide:** A boolean indicating the intended docking position of the panel (left or right side).

Key Methods and Events

- **void Start():** Conceals the panel upon initialization and establishes the necessary input bindings.
- **public void SetSide(bool _isLeftSide):** Assigns the value for the isLeftSide property.
- **public bool GetSide():** Provides access to the current value of isLeftSide.
- **public void ToggleMenu():** Alternates the panel's state between open and closed.
- **public void OpenMenu():** Transitions the panel to the open state.
- **public void CloseMenu():** Transitions the panel to the closed state.
- **void SetupInput():** Initializes the input action and binds it to the appropriate VR controller button.
- **void OnMenuButtonPressed(InputAction.CallbackContext context):** The callback function invoked when the designated menu button is actuated.

UI Panel Controller

Purpose

This component serves as the controller for a UI panel, responsible for establishing and managing references to the requisite input actions.

Key Fields and Behavior

- **public InputActionReference menuButtonAction:** A reference to the input action configured for the activation (opening) of the controlled panel.
- **public UIPanel uiPanel:** The specific UIPanel instance managed by this controller.
- **public bool useDirectInputAction:** A flag determining whether the panel utilizes direct input action creation or relies on an existing input action reference.
- **private InputAction menuAction:** The direct input action instance utilized when useDirectInputAction is set to true.

Key Methods and Events

- **void Start():** Initializes the component by locating the associated UIPanel if not pre-assigned and setting up the input system configuration.
- **void SetupInput():** Configures the input action based on the value of the useDirectInputAction flag, either using a direct input creation or the provided reference.
- **void OnMenuButtonPressed(InputAction.CallbackContext context):** A callback function that is invoked upon the detection of a press event for the menu button.
- **void OnEnable():** Activates and enables the configured input action.
- **void OnDisable():** Deactivates and disables the configured input action.
- **void OnDestroy():** Executes necessary cleanup procedures, specifically by detaching event callbacks.

SliderUI

Purpose

This document describes the Slider UI component, which is designed for use within panels.

Key Fields and Behavior

The SliderUI component exposes the following critical fields and behaviors:

- **public SliderValueChangedEvent OnSliderValueChangedEvent:** An event that is raised whenever the slider's value is modified.
- **public TMP_Text valueText:** The TextMeshPro component responsible for displaying the current numerical value of the slider.
- **public Slider slider:** A reference to the underlying Unity UI Slider component.
- **public TMP_InputField inputField:** An input field component enabling users to manually enter a precise value.
- **public GridUnitSystem unitSystem:** Specifies the active unit system (e.g., Metric or Imperial) governing the display of the value.
- **public Transform keyboardTransLeft:** A Transform reference utilized for positioning the virtual keyboard when the slider is on the left side.
- **public Transform keyboardTransRight:** A Transform reference utilized for positioning the virtual keyboard when the slider is on the right side.
- **public Toggle editToggle:** A toggle control for switching the component between view and edit modes.
- **public TMP_Text titleText:** The TextMeshPro component that displays the title or label of the slider.
- **public string title:** The textual string assigned as the title for this instance of the slider.
- **public bool isWholeNumbers:** A boolean flag that, when true, restricts the slider input to whole number values only.
- **public GameObject editGO:** The GameObject representing the edit button.
- **public GameObject closeGO:** The GameObject representing the close button.
- **public bool followsUnitSystem:** Indicates whether the displayed value should be converted and formatted according to the current unitSystem.
- **public bool isLeftPanel:** Indicates whether the slider component is situated on the left-hand panel.

Key Methods and Events

The following methods and events govern the primary functionality of the SliderUI component:

- **public Transform GetKeyboardParent():** Retrieves the correct parent Transform for the virtual keyboard, based on the component's panel-side location.
- **public void SetTitle(string _title):** Assigns a title to the slider and updates the corresponding title text display.

- **public void SetMinMax(float min, float max):** Establishes the minimum and maximum permissible values for the slider.
- **public void Show():** Activates and displays the slider GameObject.
- **public void Hide():** Deactivates and conceals the slider GameObject.
- **public float GetValue():** Returns the slider's current numerical value.
- **public void SetValue(float val):** Programmatically sets the slider to a specified numerical value.
- **void Start():** Performs initial component setup, including setting the title, subscribing to unit system changes, configuring whole number restrictions, and initializing the display.
- **public void OnEditToggle():** Manages the state change of the edit toggle, triggering the activation or deactivation of edit mode.
- **public void DeactivateEdit():** Transitions the component into view mode by making the title text visible, concealing the input field, and adjusting button visibility.
- **public void ActivateEdit():** Transitions the component into edit mode by concealing the title text, making the input field visible, and adjusting button visibility.
- **public void OnUnitSystemChanged(GridUnitSystem _unitSystem):** Updates the component to utilize the newly provided unit system and refreshes the displayed value.
- **public void OnSliderUpdated():** Refreshes the display of the value text and triggers the OnSliderValueChangedEvent.
- **public void UpdateValueText():** Responsible for updating the value text, applying necessary conversions for the Imperial unit system (feet/inches) or standard decimal formatting for the Metric unit system.
- **public void OnInputFieldUpdated():** Processes the text input from the manual input field, parses the numerical value, and, if the input is valid, invokes the value changed event.
- **public bool IsInputNumber():** Returns a boolean value indicating whether the input field's content type is configured to accept decimal or integer numbers.

ToggleGroupUI

Purpose

A Unity UI element toggle group component designed to enforce mutual exclusivity, ensuring that only one option within the group can be active simultaneously. This component is typically utilized within interface panels.

Key Fields and Behavior

- **public class ToggleGroupChangedEvent : UnityEvent<Toggle>:** A UnityEvent that transmits a Toggle reference upon its invocation.
- **public string options:** An array comprising the names of the options for which individual toggles will be generated.
- **public ToggleGroup toggleGroup:** The mandatory Unity ToggleGroup component responsible for managing the collection of toggles.
- **public Transform optionsParent:** The designated parent Transform under which instances of the toggles will be instantiated.
- **public GameObject togglePrefab:** The prefab object employed for the creation of individual toggle items.
- **public ToggleGroupChangedEvent OnToggleGroupChanged:** The event that is invoked when a toggle within the group is selected or deselected.
- **public bool allowOff:** A boolean indicator specifying whether it is permissible for all toggles to be simultaneously deselected.
- **public bool createOnAwake:** A boolean indicator determining whether the automatic creation of toggles should occur during the Awake lifecycle stage.

Key Methods and Events

- **public void OnToggleSelected(Toggle toggle):** Invokes the OnToggleGroupChanged event when a specific toggle's state transitions to 'on'.
- **private void Awake():** Executes the Setup method if the createOnAwake property is set to true.
- **public void Setup(string options = null):** Constructs the toggle instances based on the options array, configures the associated ToggleGroup component, establishes necessary listeners, and ensures that precisely one toggle is active if the allowOff property is set to false.

ToggleItem

Purpose

This class encapsulates the toggle object along with an associated TextMeshPro (TMP) text component.

Key Fields and Behavior

public TMP_Text text: Represents the text component associated with this toggle.

public Toggle toggle: Represents the standard Unity UI Toggle element responsible for the toggling functionality.

Control Point

Purpose

Serves as a runtime handle for editing a ProBuilder mesh within a Virtual Reality (VR) environment. Each control point is associated with a defined set of vertices (or other editable targets) and facilitates the selection or manipulation of the geometry through XR interaction, specifically grabbing and moving. It also provides requisite visual feedback and dynamically scales in accordance with a global control-point size setting.

Key Fields & Behavior

1. **EditMode type, int[] vertices, Vector3 normal**: These members define the specific elements of the mesh that the handle controls.
2. **XR Integration**: The component incorporates XRGrabInteractable and XRGeneralGrabTransformer, establishes a kinematic Rigidbody (RB), disables throw functionality, and toggles position/rotation tracking based on the current transform mode.
3. **Visual Management**: Manages the swapping of materials to indicate selection or deselection status and registers as a listener for global size modifications broadcast by the ViewManager.
4. **Selection and Movement**: Integrates with the ModelEditingPanel to switch between selection-only and free-move modes. During active movement, it continuously calculates and applies offsets to the underlying associated vertices.

Key Methods / Events

1. **Initialize(int[] _vertices, Vector3 _normal, EditMode _type)**: This method is responsible for attaching the associated data and subsequently invoking the Setup() method.
2. **Setup()**: Configures the control point by adding necessary XR components, wiring event listeners, and establishing initial material assignments.
3. **TransformTypeChanged() / SetAsSelectedAble() / SetAsMoveAble()**: These methods facilitate the switching of the control point's operational state between selection-only and movable interaction modes.
4. **OnGrabStart(...) / OnGrabEnd(...)**: Handlers that manage the start and end of the grab interaction, either toggling selection state or initiating/terminating free movement.
5. **SetVertexPosAsCPCurrentPos() / AddOffsetToControlPointAndVertsPosition(...)**: Methods used to propagate the motion of the control handle to the positions of the controlled mesh vertices.
6. **ControlPointSizeChanged(float)**: Rescales the size of the control handle in response to global size updates.
7. **Deactivate()**: Executes necessary cleanup procedures and conceals the control point from view.

Model Creator

Purpose

Provides a user interface-driven factory for the runtime creation and regeneration of parametric ProBuilder shapes (including cube, sphere, cone, pipe, cylinder, and plane variants). It features controls for both uniform and non-uniform sizing, as well as subdivision adjustments. This component is responsible for managing the current ModelData instance.

Key Behavior

1. Manages toggle and slider groups to facilitate the selection of shape types and parameters, dynamically updating UI visibility and slider ranges according to the chosen shape.
2. Creates a new mesh of the selected type, attaches the necessary ModelData, instantiates it at a designated spawn point, and ensures its selection.
3. Rebuilds the mesh topology upon changes to subdivision parameters, preserving the current transform data.
4. Incorporates custom generation routines for specialized meshes, including a subdivided cube, a wavy plane, and a dome plane.

Key Methods / Events

5. **ModelTypeSelected(Toggle):** Responds to UI events by resetting sliders, updating UI elements and ranges, and either creating a new model or updating the existing one.
6. **CreateOrUpdateModel():** Destroys any previous preview, constructs a new ProBuilderMesh, encapsulates it within ModelData, handles positioning and scaling, and selects the resultant object.
7. **CreateModelByType(ModelType):** Serves as a dispatcher to select between built-in ProBuilder functions and custom generation methods.
8. **CreateSubdividedCube(...), GenerateWavyPlane(...), GenerateDomePlane(...):** Implement the custom mesh construction logic.
9. **SizeSliderUpdated(float):** Applies scaling, which may be uniform or per-axis, based on user input.
10. **SubDSliderUpdated(float):** Triggers a rebuild of the mesh topology and updates the ModelData via ModelData.UpdateMeshCreation(...).
11. **FinalizeLastModel():** Confirms the current model and clears the temporary preview state.

Model Data

Purpose

A runtime wrapper that encapsulates a ProBuilder editing mesh and associated scene components (MeshRenderer/Filter/Collider, XR grab, snapping). It is responsible for tracking selection state, shading, edit/transform modes, and provides accessors and update APIs for mesh, face, and vertex data.

Key Behavior

1. It registers with the ModelsManager, subscribes to ViewManager shading and ModelEditingPanel mode changes, and manages XR grab selection when in object or select mode.
2. It facilitates the conversion between the "editing" ProBuilderMesh and a standard Mesh, and executes refreshes following editing operations.
3. It maintains the model's name/ID and manages the visual representation of its selection state.

Key Methods / Events

- **SetupModel(ProBuilderMesh) :** Initializes components, configures the XR interactable and snapping transformer, registers necessary events, and sets the material and shading.
- **UpdateMeshCreation(ProBuilderMesh):** Replaces vertex positions and faces using data from a newly generated mesh.
- **UpdateMeshEdit()** / **FinalizeEditModel():** Refreshes the editing mesh or finalizes the mesh data to a standard Mesh object.
- **SelectModel()** / **UnSelectModel() :** Updates the model's selection state and corresponding materials.
- **OnEditModeChanged()** / **OnTransformTypeChanged() :** Enables or disables XR tracking functionality based on the current edit and transform modes.
- **OnGrab(SelectEnterEventArgs):** Toggles the selection state when the model is grabbed in object or select mode.
- Accessors: **GetVerts()**, **GetFaces()**, **GetFacesCount()**, and others.

Model Manager

Purpose

Functions as a singleton registry for all ModelData within the scene, providing a change event for observers (e.g., UI elements). It is also responsible for assigning unique identifiers to newly tracked models.

Key Methods and Events

- **OnModelsChanged: UnityEvent<List<ModelData>>:** An event emitted whenever the collection of tracked models is modified.
- **TrackModel(ModelData):** Assigns an index/ID to the ModelData and adds it to the internal list.
- **UnTrackModel(ModelData):** Removes the specified ModelData from the internal list.
- **GetAllModelsInScene():** Returns the currently tracked collection of models.
- Instance: Provides global access to the registry (persisting across scene loads).

Selection Manager

Purpose

Serves as the singleton controller for the active selection set of models, responsible for emitting selection-changed events and providing essential utility operations, including clearing, adding, removing, and executing single-selection procedures.

Key Methods and Events

- **OnSelectionChanged : UnityEvent<List<ModelData>> :** Notifies registered listeners upon updates to the selection set.
- **SelectModel(ModelData) :** Executes a single-selection operation: clears the current selection, selects the specified model, and triggers a notification.
- **AddModelToSelection(ModelData)** / **RemoveModelFromSelection(ModelData) :**Utility methods facilitating multi-selection management.
- **ClearSelection():** Deselects all active models and issues a selection-changed notification.
- **GetFirstSelected():** Provides expedited access to the primary selected item.
- **Instance :** Offers global access to the persistent manager instance.

View Manager

Purpose

Manages the global view and UI state, encompassing shading, the appearance of control points, and a dynamic measurement grid (supporting metric/imperial systems, adaptive scaling, and backface visibility). It exposes events to notify listeners of changes to shading, control-point size, and unit systems.

Key Functionality

- **Shading:** Manages paired materials for selected and unselected states across various modes (Standard, Wireframe, Unlit, Clay, HiddenLine). It provides functionality to toggle backface culling or "show backfaces" flags and broadcasts notifications of shading changes.
- **Control Points:** Stores the global size parameter and broadcasts updates, allowing the **ControlPoint** component to dynamically rescale its visual representation.
- **Units & Grid:** Supports both metric and imperial modes with adaptive spacing. It calculates the appropriate grid spacing based on the camera's height and updates shader parameters for major/minor lines and the snapping center.
- **Grid API:** Offers a straightforward interface for toggling the visibility of the grid.

Key Methods and Events

- **OnShadingChanged**, **OnControlPointSizeChanged(float)**, **OnUnitSystemSizeChanged(GridUnitSystem)**: Global notification events for system-wide changes.
- **BackfaceUpdated()**: Applies backface culling or visibility flags consistently across all active materials.
- **GetCurrentShading(bool isSelected)**: Retrieves the correct set of materials corresponding to the current shading mode and selection state.
- **ControlPointSizeUpdated(float)**: Broadcasts the updated handle size value.
- **OnShadingToggleSelected(Toggle)**, **OnUnitSystemToggleSelected(Toggle)**: Methods primarily utilized for binding with UI elements.
- **ToggleGrid()**: Controls the visibility state of the grid.
- **Update()**: Recomputes the grid spacing and center position every frame to maintain adaptive functionality.

Operation Tool

Purpose

Serves as an abstract base class for all operational tools. It provides essential components, including a standardized user interface (UI) panel, a formal designation, and methods for capability assessment. This foundation allows derived tools to formally advertise their availability and manage the visibility of their associated UI.

Key Members & Methods

- **string toolName, GameObject panel :** These members represent the display name of the tool and the bound UI panel, respectively.
- **GetToolName() :** A method that returns the value of **toolName**.
- **ShowTool()** / **HideTool() :** Methods used to control the visibility of the tool’s UI panel.
- **virtual bool CanShowTool() :** A virtual method that determines whether the tool's UI is appropriate for display within the current operational context.
- **virtual bool CanPerformTool() :** A virtual method that assesses whether the intended operation is currently valid and ready for execution.

Align Tool

Purpose

This tool facilitates the alignment of selected control points to a common coordinate along a designated axis by simultaneously offsetting both the control handle and the underlying mesh vertices. The functionality is visible and executable exclusively within the Vertex, Edge, or Face edit modes, contingent upon the selection of at least one control point.

Key Methods

- **AlignToX() / AlignToY() / AlignToZ():** These methods calculate the average position of the selected control points along the chosen axis. Subsequently, each point is offset such that all selected points share this common coordinate. The underlying mesh is updated through each point's execution of the AddOffsetToControlPointAndVertsPosition(...) method.
- **override CanShowTool():** This method returns true when the application is operating in Vertex, Edge, or Face edit modes; the tool is concealed in Object and Pivot modes.
- **override CanPerformTool():** Execution of the tool is contingent upon the selection of one or more control points and the application being in a non-Object and non-Pivot edit mode.

Bevel Tool

Purpose

This tool is intended to bevel selected faces. The tool is only available and executable when the application is in Face mode and a selection has been made;

Key methods

- **override CanShowTool():** Restricts visibility to Face mode only.
- **override CanPerformTool() :** Requires selection of one or more control points and operation in Face mode.

Delete Tool

Purpose

Deletes currently selected targets. In Object mode, it removes selected **ModelData** instances.

Key Methods

- **override CanShowTool()**: Visible in all modes except Pivot mode.
- **override CanPerformTool()**: **Currently returns false in all scenarios**; requires correction to return true when conditions are met.
- **DeleteSelectedObjects()**: In Object mode with selected models, this method invokes **model.DeleteModel()** for each selected instance.
-

Extrude Tool

Purpose

This tool is designed to perform the extrusion and optional inset of selected faces. Its availability is correctly restricted to Face mode with a selection.

Key Methods

- **override CanShowTool() :** Available exclusively in Face mode.
- **override CanPerformTool() :** Requires the selection of one or more face control points.

Weld Tool

Purpose

The tool is designed to weld selected control points or vertices, such as merging them to a common position.

Key Methods

- **override CanShowTool():** Determines tool visibility; it is visible in Vertex, Edge, and Face modes, and hidden in Object and Pivot modes.
- **override CanPerformTool():** Defines the execution criteria; it requires the selection of at least one control point and a mode other than Object or Pivot.
- **Operation:** Weld selected control points or vertices, such as merging them to a common position.