Keyboard Manager UI

Table of Contents:

C++ XAML Islands

The KBM UI is implemented as a C++ XAML Island, but all the controls are implemented in code behind rather than .xaml and .xaml.cs files. This was done as per a XAML Island Code sample and it didn't require a separate UWP project, which could be limited in terms of using hooks. There is a tech debt item for moving this to XAML. The reason it wasn't implemented in the C# Settings was because it required communication with the low level hook thread, which could be too slow if IPC is used, since the UI needs to update on every key event.

Note: For functions which take a XAML component as argument, pass it by value and not by reference. This is because `winrt` WinUI classes store their own internal references, so they are supposed to be passed by value (and internally ref counts are incremented). Passing by reference can lead to weird behavior where the object is `null`.

The windows are created as C++ windows and the window sizes are set to default by scaling them as per DPI using the `DPIAware::Convert` API from common lib. Since the UI is launched on a new thread, the window may not be in the foreground, so we call `SetForegroundWindow`.

`DesktopWindowXamlSource` has to be declared and it is initialized using the `XamlBridge`, and a second window handle is generated for the internal Xaml Island window. Most of the code was based on the Xaml Island Sample. The `XamlBridge` class contains code which handles initializing the Xaml Island containers as well as handling special messages like keyboard navigation, and focus between islands and between the C++ window and the island. It also has methods for clearing the xaml islands and closing the window.

Once the UI controls are created, the parent container is set as the content for the `DesktopWindowXamlSource` and the `XamlBridge.MessageLoop` is executed. Messages are processed by the C++ window handler like `EditKeyboardWindowProc`. The general structure we use for this is, for any `WM_PAINT` or `WM_SIZE` message we resize the Xaml Island window. For `WM_GETMINMAXINFO` we set minimum widths so that the window cannot be resized beyond a minimum height and width. This is done to prevent the WinUI elements from overlapping and getting cropped. If it is neither of these cases we send the message to the `XamlBridge.MessageHandler` which handles Destroy, Activation and Focus. If `WM_NCDESTROY` is received when the `XamlBridge` is `nullptr`, the window thread is terminated.

Note: `ContentDialog` in Xaml Islands requires manually settings a `XamlRoot`. This can generally be done by passing the XamlRoot from a component in the main window, such as the button used to open the dialog (`sender.as<Button>().XamlRoot()`). These docs have more details about this.

Debugging exceptions in XAML Islands

Sometimes if an exception occurs in XAML Islands, the stack trace may not always point to the correct code causing the exception and instead it will point to the Xaml Island message loop. In these cases the output window in VS will generally show the correct exception.

Build times

C++ Xaml Islands generally take several minutes to build because the `pch` which contains the WinUI headers takes longer to build and compiles to a file of several GBs. To minimize the build times, multi-processor compilation within the projects have been enabled (files are distributed for compilation to the processors), and references to the Xaml headers have been removed from the .h headers files as much as possible. Since several classes of ours had class members with UI controls like `StackPanel` (which requires definitions of the classes in order to compile), we worked around this by declaring them as `IInspectable` (the equivalent of an object pointer in winrt), and initializing them to the actual control like `StackPanel` in the constructor and accessing all their member functions by inline typecasting (for `IInspectable x;` we do `x = StackPanel();` and `x.as<StackPanel>().MemberFunction()`). Check this for this type of usage in `ShortcutControl`.

Setting custom backgrounds for Xaml Controls using brushes

To access the brushes available on C# Xaml, it has to be done with the `Resources.Lookup` syntax: `primaryButton.Background(Windows::UI::Xaml::Application::Current().Re`

UI Structure

The KBM UI consists of a `Grid` with several columns. Rows are added dynamically when the add button is pressed. A vector of vector of unique pointers to `SingleKeyRemapControl/ShortcutControl` is created so that references to the UI components and their data are not lost until the window is closed. `SingleKeyRemapControl` is the UI class for each row of the Remap keys table, and `ShortcutControl` is the UI class for each row of the Remap shortcuts table. `KeyDropDownControl` is used for handling the ComboBox operations. Each of these two classes have vectors of unique pointers to the `KeyDropDownControl` objects so that references to the objects are active until the control is deleted.

When the UI windows are activated the `KeyboardManagerState` object sets the `UIState` variable which is used for distinguishing if the UI is up from the keyboard hook thread. The states are also updated on opening and closing the Type window.

Clicking the Type Button opens a content dialog which registers key delays using the `KeyDelay` class for Enter and Esc keys and sets the UI states such that when a key event occurs the TextBlocks on the ContentDialog are updated accordingly. On accepting the dialog the selected keys are copied into the ComboBoxes from the TextBlocks, and on closing the window the key delays are unregistered and UI states are reset.

Since ComboBoxes are added dynamically, handlers have been added which update the accessible names for these controls, which get executed whenever a drop down is added or removed.

When the `EditKeyboardWindow/EditShortcutsWindow` is created, we iterate through the remappings stored in `KeyboardManagerState` and add rows to the UI Grid. For both the windows we have `static` buffers `singleKeyRemapBuffer` and `shortcutRemapBuffer` which store the corresponding key/shortcuts as per the selections in the UI if they are valid with no warnings.

EditKeyboardWindow/EditShortcutsWindow

OK and Cancel button

On pressing the OK button in `EditKeyboardWindow`, first the `CheckIfRemappingsAreValid` method is executed which performs basic validity checks on the current remappings in the remap buffer (`static SingleKeyRemapControl::singleKeyRemapBuffer`), such as if there are no NULL columns and none of the source keys are repeated. All other validity checks are assumed to happen while the user adds the remapping. If this is found to be invalid a ContentDialog is displayed which shows that some remappings are invalid and if the user proceeds only the valid ones will be applied. If it is valid `GetOrphanedKeys` is executed which checks if any keys are orphaned (i.e. the key has been remapped and no other key has been remapped to it, so there is no way to send that key code), and a dialog is shown for notifying the user with a list of orphaned keys. After this the settings are applied by adding it to the `KeyboardManagerState.singleKeyReMap` member and they are saved to the JSON file. `EditShortcutsWindow` differs slightly from this, as there is no orphaned keys check, and on pressing OK both the global and app-specific shortcuts are validated and updated.

The code used for updating the remapping tables in `KeyboardManagerState` can be found here. For shortcut remaps, the `sortedKeys` vectors are updated and re-sorted whenever an element is added to them (like this).

On pressing OK (after confirmation dialogs) or Cancel, the window is closed and UI states are reset.

Delete button

Since there is no single method to delete the elements in a row for a Grid, the logic we use involves decrementing the rowIndex for all the UI controls that appearing after the row to be deleted, and removing each of the items of the row from the Grid, followed by deleting that row definition. We also update the accessible names for all the rows since the indexing has changed. After this the corresponding row in the remap buffer is also deleted, and `SingleKeyRemapControl/ShortcutControl` objects are deleted from the vector.

Handling common modifiers in EditKeyboardWindow

In the SingleKeyRemap table for a remapping of the form Ctrl->X, where Ctrl is the common version and not L/R, we can't store it directly as Ctrl->X because when the hook receives the key event it only gets LCtrl or RCtrl specifically and not `VK_CONTROL`. To simplify the backend code, when single key remappings are applied, any remapping of the form Ctrl->X is split into Ctrl(L)->X and Ctrl(R)->X (i.e. both L and R versions are remapped to the same target), and when remappings are loaded in EditKeyboardWindow, we pre-process the remap table such that if the L and R versions of a modifier are remapped to the same key/shortcut we combine them. This also results in the behavior where a user adds LCtrl->X and RCtrl->X and after closing and re-opening KBM UI it appears combined as Ctrl->X.

SingleKeyRemapControl

The left drop down column uses a single ComboBox and the Type button is linked to `createDetectKeyWindow`, whereas the right column is linked to `createDetectShortcutWindow` as the column can accept a key or a shortcut (required to support key to key and key to shortcut). The `KeyDropDownControl` for the left column in the window uses a smaller key list, without `None`.

ShortcutControl

Both the columns in are linked to `createDetectShortcutWindow`, however the drop down selection handlers differ in their logic as the left column only allows shortcuts, and the drop downs do not contain the `Disable` key, whereas the right column allows you to select both shortcuts and keys (to support shortcut to shortcut and shortcut to key), and it allows selection of `Disable`.

For the app-specific shortcut target app text-box, we had to validate that the shortcut row is still valid when the target app is changed (for example, Ctrl+A is remapped for Chrome, and another remapping for Ctrl+A was remapped to Edge, but the target was changed to Chrome.). For this we didn't use the TextChanged handler as every time a letter is typed it would get executed. Instead we used the `LostFocus` handler which gets executed whenever you focus into the box (by clicking or tabbing) and then tabbing or clicking out. In the method we perform the same shortcut buffer validation used for selections in the drop down menus and update the buffers accordingly.

KeyDropDownControl

Each ComboBox has a linked flyout, which is used to show warnings to the users whenever a user selects an invalid key from the drop down. When the warning is displayed the ComboBox is also reset to -1, i.e. no selection.

For selection handlers on the ComboBoxes we couldn't use just the Selection-Changed handler directly as it gets executed even on searching for elements in the drop down. Instead we used DropDownClosed (when a user opens the drop down and searches and selects something) and SelectionChanged when the drop down is not open (for setting selections programmatically or selection made by searching with tab focus on the drop down without opening it). This was required because if we execute the selection handlers while users are searching, it could cause false positive flyout warnings if the search causes an invalid value to be selected, and flyouts cause the drop down to close leading to bad UI experience.

Localized key names

For getting localized key names and symbols for each virtual key code, whenever the key lists are accessed, i.e. whenever the drop down is opened or when `GetKeyName` is called in the Type window, the current `KeyboardLayout` is retrieved to ensure that the displayed key names are always updated. Since the `WM_INPUTLANGCHANGED` event was having some issues with XAML islands we weren't able to use this to update the keyboard layout. In addition to this we do not refresh the UI, so the key lists get updated only on opening/interacting with them.

Single Key ComboBox Selection Handler

On making a selection in the drop down, the selection handler validates the input with the buffer from the other column and other rows. Error messages are shown using flyouts if the selection is not considered valid and the drop down and buffer for that entry are reset to empty selection. The errors that can occur on the single key ComboBox are: - Remap to same key (A->A) - Same key previously remapped (A->B and A->C) - Conflicting modifier previously remapped (Ctrl->A and Ctrl(left)->B, since Ctrl also includes Ctrl(left)) If the selection is found to be valid, the `singleKeyRemapBuffer` is updated accordingly. For handling `Shortcut` and key in the remap buffer for the right column, we use `std::variant`, which allows us to store either of the two types and check which one of them is present in the buffer by using the `index` method. `ValidateAndUpdateKeyBufferElement` does not reference any UI components and instead takes all the relevant data as arguments. This method has tests which covers all the cases that could arise from making selections on the UI.

Shortcut ComboBox Selection Handler

On making a selection in the drop down, the selection handler validates the input with the buffer from the other column and other rows. Error messages are shown using flyouts if the selection is not considered valid and the drop down and buffer for that entry are reset to empty selection. This differs from the Single Key ComboBox handler in the sense that after validating the current selection we may perform a set of actions such as: - Adding a drop down (if a modifier is selected) - Removing a drop down (if None is selected) - Clearing terminal empty drop downs (if an action key is selected in a non-last drop down and the remaining ones are empty)

After performing the corresponding action, if any, we check if the drop down resulted in an error, in which case we do a second level of validation on all the drop downs in that list of drop downs. This is done because there can be cases where an error in one drop down results us setting it to empty, and the remaining selection is also invalid. For example, you have Ctrl+A -> Ctrl+Shift+A, and you change Shift to Ctrl. This would show a warning for having two Ctrls in the shortcut being invalid, after which setting that to empty would result in Ctrl+A->Ctrl+Empty+A, which is a case of remapping a shortcut to itself.

Once this second level of validation is done, we proceed with updating the buffer. Depending on the number of drop downs with valid values, this could be either a key or a shortcut (for the right columns). We also set the buffer value for the target app while doing this.

Unlike the Single Key handler, there is a different set of errors that can occur here which are related to making a selection that is considered as a valid shortcut. The `isHybridControl` argument is used to distinguish between the differing behaviors for the two types of columns (shortcut only or shortcut/key column). The errors that can occur for this handler are: - Shortcut must start with modifier (selecting A on the first drop for the left column is invalid) - Shortcut can't have a repeated modifier (Ctrl+Ctrl(left)+A is not a shortcut) - Shortcut can only have upto 2 modifiers (Ctrl+Shift+Alt is not supported as we have enforced a 3 key constraint (**not a backend limitation, there is an issue requesting to**

remove this)) - Shortcut must contain an action key (Ctrl+A and change A to None, only for left column) - Shortcut must have at least two keys (Ctrl+A and change Ctrl to None, only for left column) - Disable can't be a modifier or action key (Ctrl+Disable is invalid) - Shortcut can't have more than one action key (Ctrl+Shift+A, change Shift to B) - Remap to same shortcut(Ctrl+A->Ctrl+A) - Same shortcut previously remapped for same target app (Ctrl+A->B and Ctrl+A->C) - Conflicting shortcut previously remapped for same target app (Ctrl+A->B and Ctrl(left)+A->C, since Ctrl also includes Ctrl(left)) - Illegal shortcut remaps like Win+L or Ctrl+Alt+Del (since these cannot be remapped using LL hooks)

`ValidateShortcutBufferElement` does not reference any UI components and instead takes all the relevant data as arguments. This method has tests which covers all the cases that could arise from making selections on the UI.

Note: After updating the buffer we have code to handle a special case, which was required to prevent scenarios where a drop down can get deleted but the corresponding `KeyDropDownControl` object isn't deleted. The code checks if the drop down is still linked to the parent and accordingly deletes the `KeyDropDownControl` object from the vector.

IgnoreKeyToShortcutWarning special case: An additional check was added to ignore the Map to Same key error when an existing remapping is loaded. This was because a remapping like Ctrl->Ctrl+A has an intermediate step of Ctrl->Ctrl, which could lead to an error of invalid input, even though Ctrl+A is valid. The only way to actually add this is from the Type button or by adding them in a different order (like typing Shift+A and then changing Shift to Ctrl). Since the intermediate check could fail, this was causing the app to crash since the Xaml Island wouldn't be completely loaded at that point and the Flyout can't be displayed. This is the linked issue which describes the repro scenario.