The Simulating Substances App GUI 1

Communication between scenes 1

Simulating Substances scene 2

Problems in the Simulating Substance scene 2

Configuration scene 3

Problems in the Configuration scene 3

Solutions in the configuration scene 3

Delete Scene 5

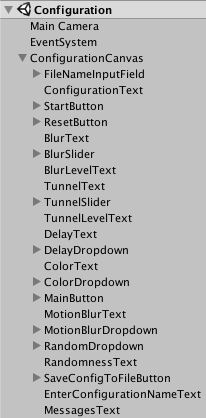
Problems in the Delete scene 5

Solutions in the Delete scene 5

Main Scene 7

Problems in the Main scene 7

## The Simulating Substances App GUI

Unity uses the metaphor „Scene“ in order to refer to a file, which contains the objects of your game. Scenes can be used to create a main menu, individual levels, and anything else. Think of each unique Scene file as a unique level. In each Scene, you will place your environments, obstacles, and decorations, essentially designing and building your game in pieces.

The Hierarchy Window in Unity is a hierarchical text representation of every object in the scene. The hierarchy reveals the structure of how objects are attached to one another and can became very complex.

Below we explain the four Simulating Substances application scenes ordered by growing complexity. Each scene has a corresponding C sharp script responsible of its control following the Model-View-Controller Pattern.

For a better understanding of the GUI it’s necessary to explain before how is data preserved between scenes.

## Communication between scenes

A scene is an independent entity with its own lifecycle. That means, that when a new scene is loaded (f. e. a new level in a game) all objects from the previous scene are destroyed.

In order to preserve data Unity provides the PlayerPrefs class, which stores and accesses player preferences between game sessions. This class uses a simple Key/Value system to save the data to a file:

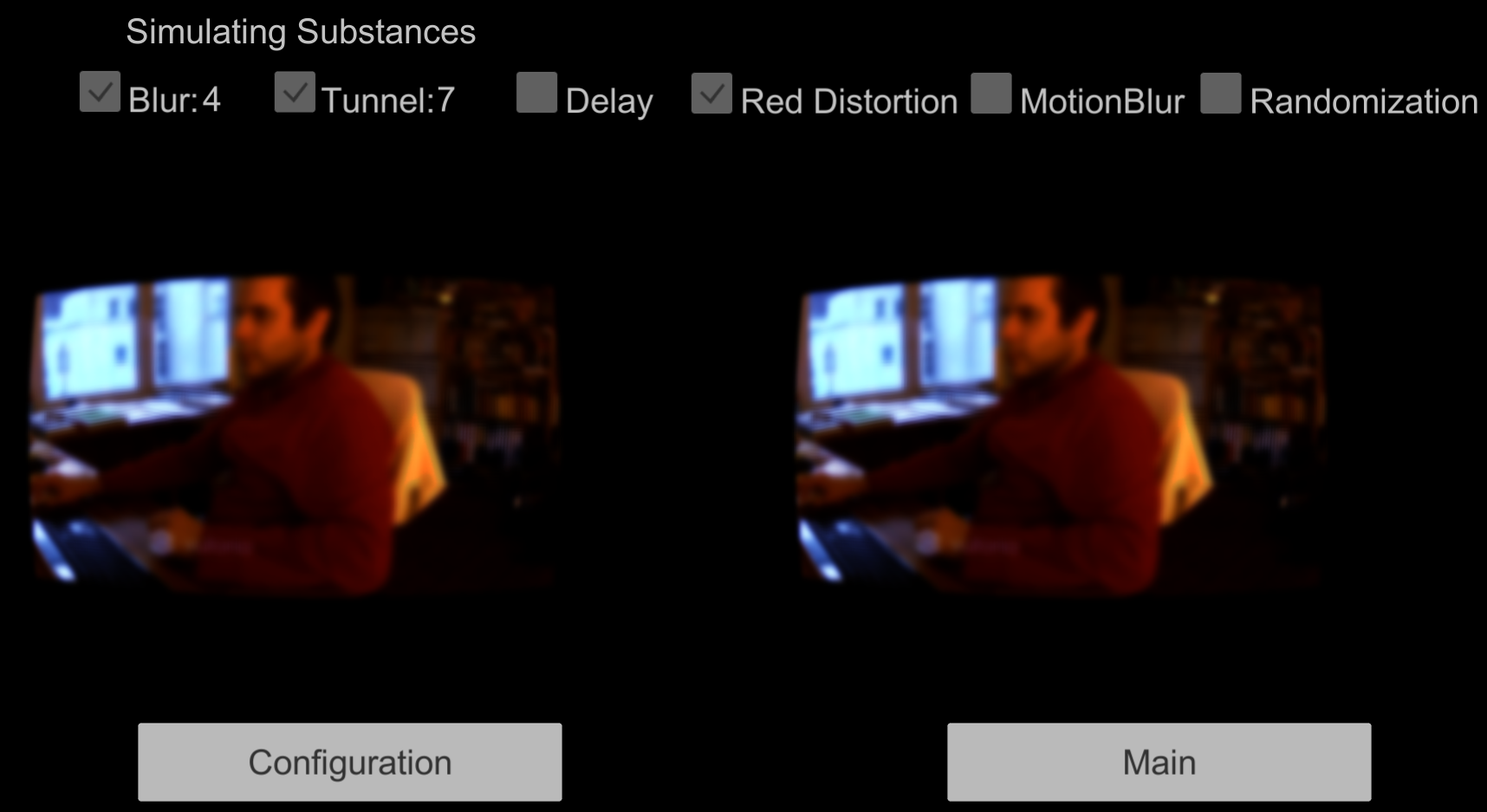
PlayerPrefs.SetInt("Player Score", 10);

var score = PlayerPref.GetInt("Player Score")

We employ this class to preserve the configuration values of the image effects selected by the user. So the user can switch between scenes or even shut down the application without loosing the configuration.

## Simulating Substances scene

This is the scene where all happens. The Vuforia Plugin provides de Augmented Reality functionality and the different Unity Image Filters create the desired effects as blur or tunnel view.



The effect values shown in the upper part are loaded from the PlayerPrefs when the scene starts. In the **SimulatingViewModel** class:

  UpdateBlurValue(PlayerPrefs.GetFloat(PlayerPreferences.BlurLevel));  
 UpdateTunnelValue(PlayerPrefs.GetFloat(PlayerPreferences.TunnelLevel));  
 UpdateDelay(PlayerPrefs.GetInt(PlayerPreferences.DelayLevel));

 private void UpdateBlurValue(float value)  
        {  
            if (value > 0)  
            {  
                blurValueText.text = value.ToString(); //Shows the 4 in the image

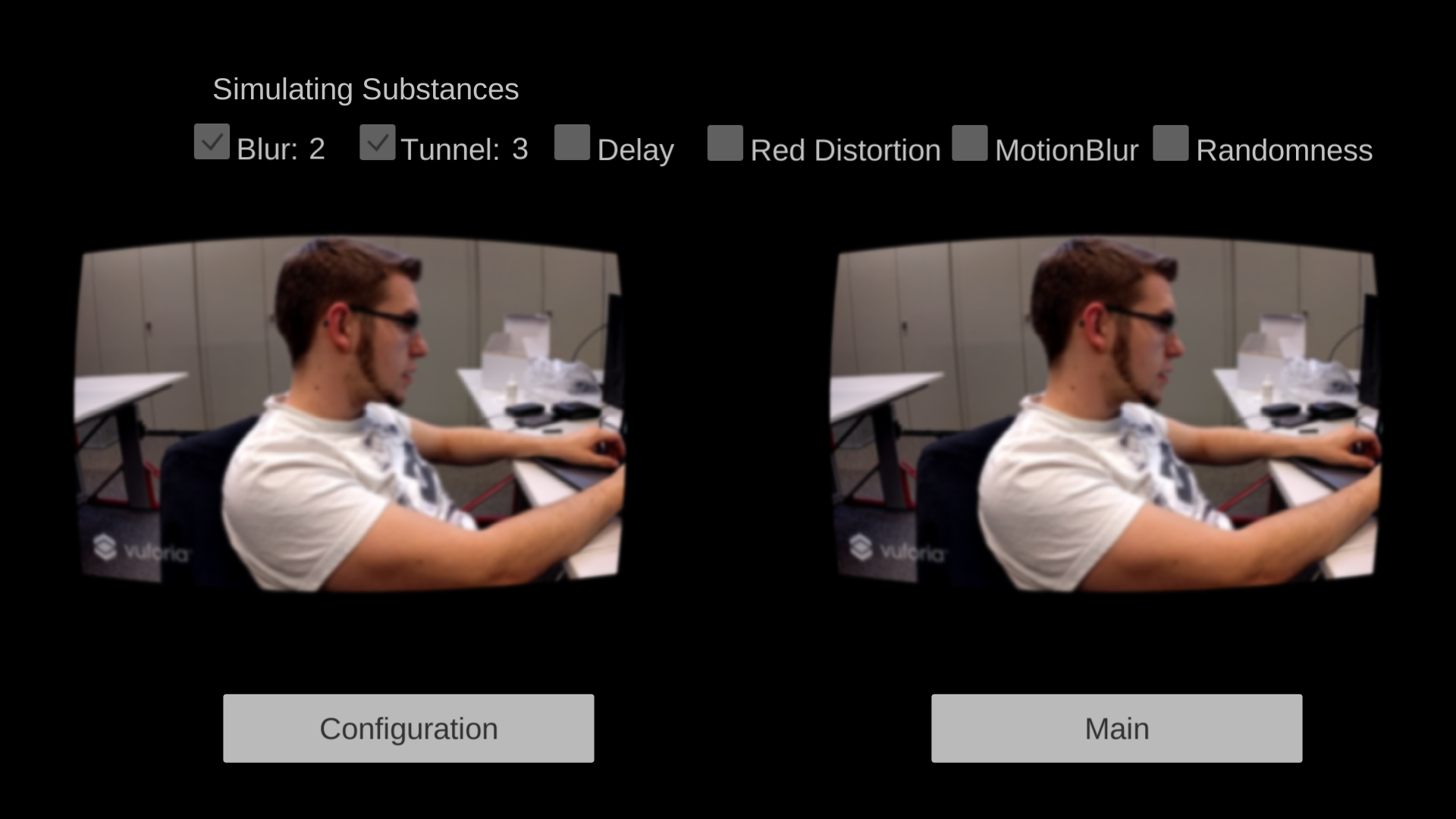
                blurToggle.isOn = true; //Shows the toggle as selected  
            }  
        }

[TODO: Ask Koni if he wants to explain here the Randomization class or in a separated section. If separated, here there should be a reference to the section]

As you can see in the image, this scene contains the application’s principal functionality; it shows the user the augmented reality.

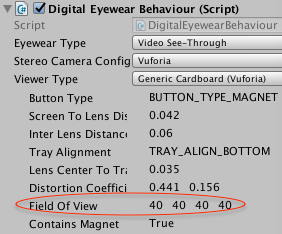
### Problems in the Simulating Substance scene

The main problem we discovered in this scene is the size of the augmented reality windows. A comparison of a Virtual Reality application (left) with the augmented reality:



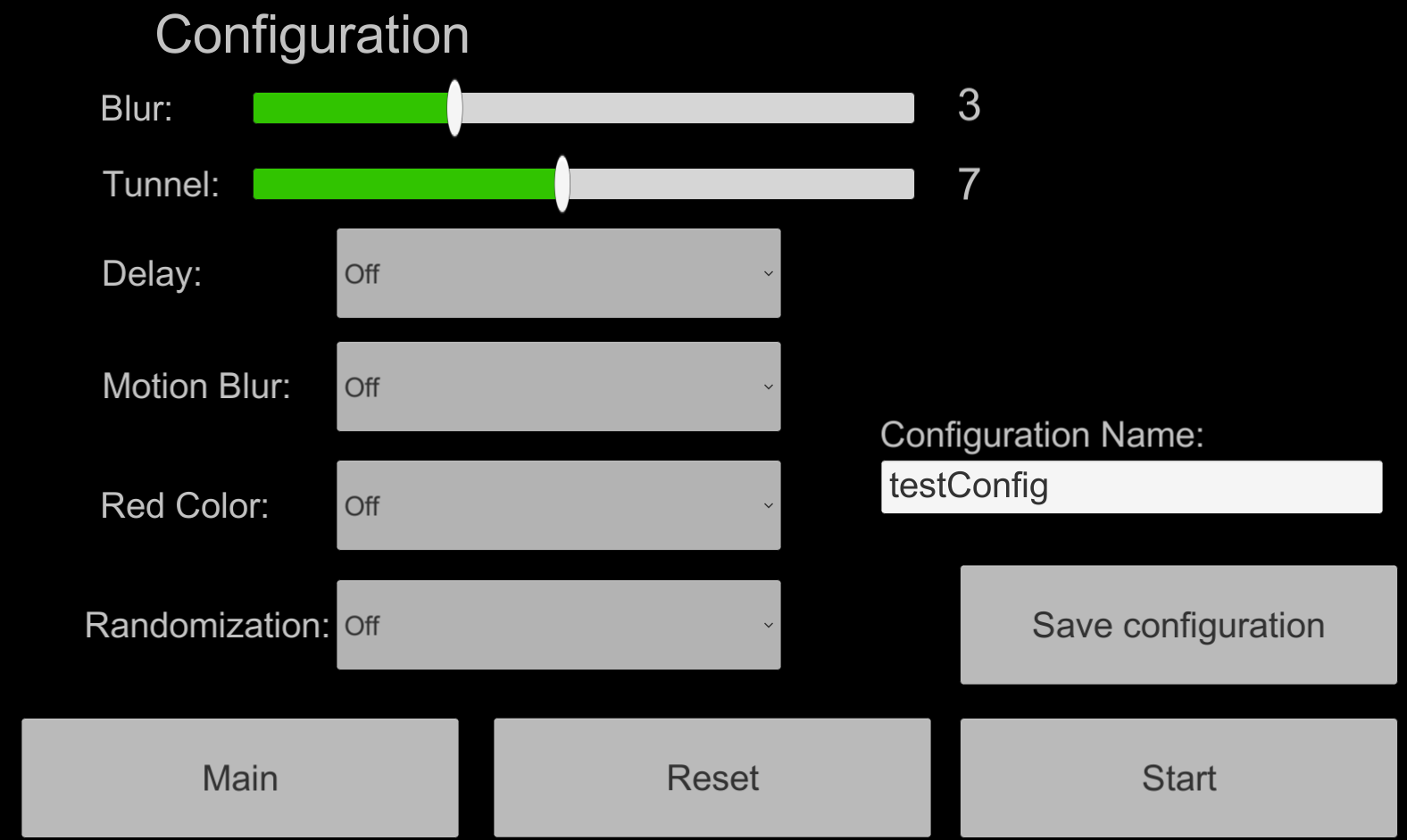
Although the images are not in scale you can see that the size of the augmented reality windows is much smaller. After long research we could determine that this is a limitation of the Vuforia Plugin used for augmented reality.

The Vuforia Augmented Reality Camera has only a 40 degrees Field of View[[1]](#footnote-1) capacity, and this conditions the windows size.



## Configuration scene

In this scene you can activate or deactivate effects or configure the effects strength. The configuration is saved automatically to the PlayerPrefs. It is also possible to save a custom configuration to a file.



### Problems in the Configuration scene

The Configuration scene has more GUI components and this increases automatically the logic complexity. The Dropdowns and sliders introduce configuration values, which have to be saved in the PlayerPrefs in order to be transmitted to the Simulating Substances scene. At the same time, when the user decides to reset these values, all GUI components must be reset to the default values.

Another responsibility of the Configuration Scene is saving the configuration values to a file, in order to allow the user to have different configurations, which can be later loaded or deleted as needed.

### Solutions in the configuration scene

For the two different problems we divided the responsibility into two different classes, the **ConfigurationViewModel** and the **SaveConfigurationSceneViewModel** classes.

When the user f. e. decides to activate the Red Color distortion, it selects “On” in the Red Color dropdown. This action generates an event captured by the Unity’s Event System and calls an “OnValueChanged(int32 value)” method.

The Inspector in Unity allows binding this method call to a script. In this case the **ConfigurationViewModel** class:

RobertoHD:Users:roberto:Documents:studium:SA:hsr-semester-arbeit-doc:images:onValueChanged.png

The OnValueChanged(int32 value) method calls the SetRedColorDistortion method of our **ConfigurationViewModel** class:

 public void SetRedColorDistortion(int value)  
        {  
            PlayerPrefs.SetInt(PlayerPreferences.RedColorDistortion, value);  
        }

In this method we save the value (it just can be a 0 for off or a 1 for on) to the PlayerPrefs. And so on with all the GUI components.

When resetting the GUI components, first we must get these components from the scene hierarchy and then reset its values to the default values (usually 0).

We get the component from the hierarchy with:

Dropdown colorDropdown =

GameObject.Find(ConfigurationControls.DelayDropdown).GetComponent<Dropdown>();

And then we reset its value to default: colorDropdown.value = 0;

This reset action is generated by the user when it presses the Reset button.

With the **SaveConfigurationSceneViewModel** class we address the file saving problem. The user selects the different configuration values in the Configuration scene, enters the configuration’s name in the input field and presses the Save configuration button.

This action invokes the SaveConfiguration method of the **SaveConfigurationSceneViewModel** class:

public void SaveConfiguration()  
{  
var fileNameInputField =

GameObject.Find(ConfigurationControls.FileNameInput).GetComponent<InputField>();  
SaveToFile(ConfigurationHelper.GenerateConfigurationByPlayerPrefs(fileNameInputField.text);

}

The ConfigurationHelper.GenerateConfigurationByPlayerPrefs(fileNameInputField.text)method is a helper method, which creates a Configuration object from the already saved data in the PlayerPrefs and pass it to the SaveToFile Method.

This SaveToFile method just calls again another Save helper method responsible for the file saving and takes care of the exception when given:

public static void Save(Configuration configuration)  
{  
   var bf = new BinaryFormatter();  
 var file = File.Create(Application.persistentDataPath + "/" + configuration.Name)

    bf.Serialize(file, configuration);  
    file.Close();  
}

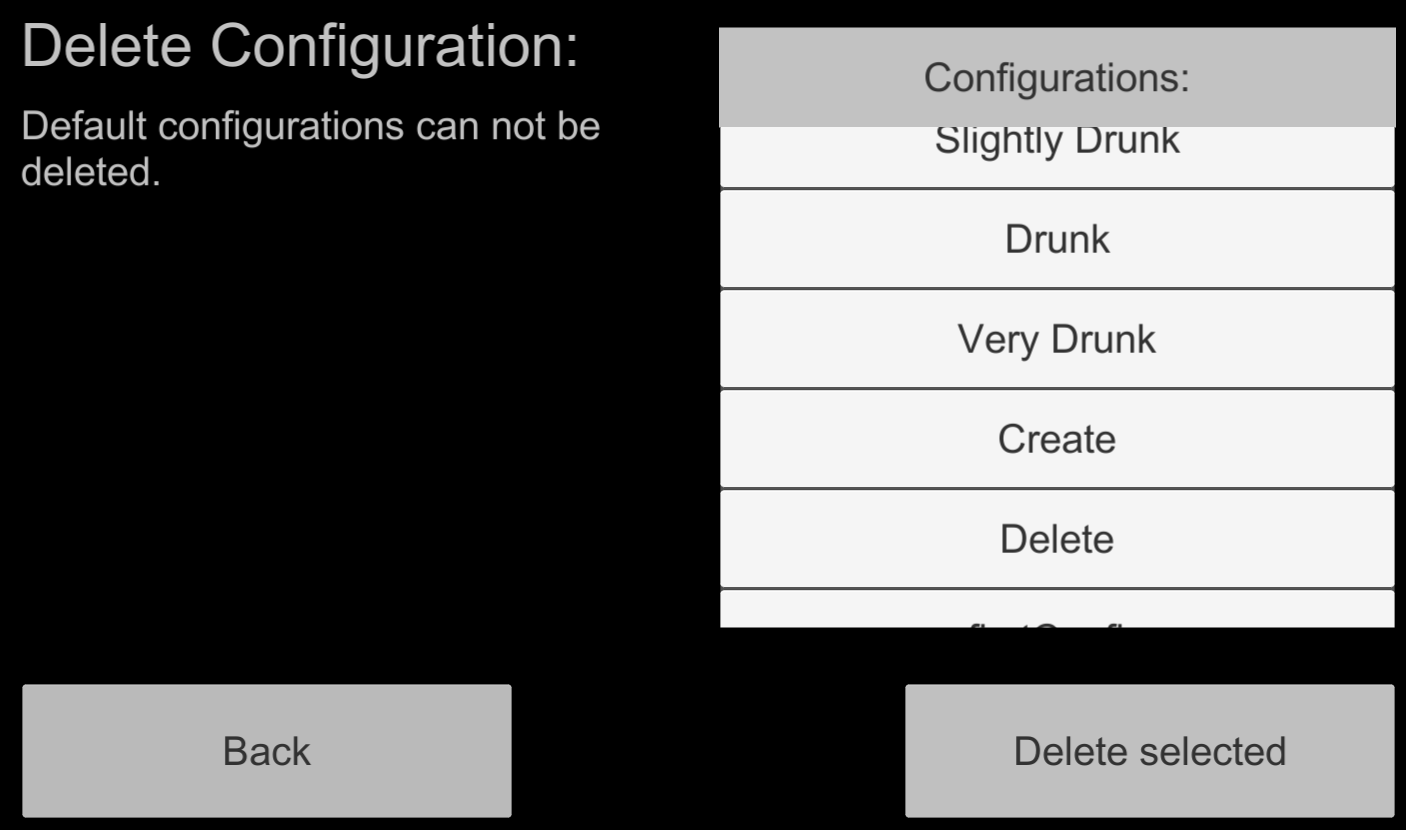
## Delete Scene

In this scene the user is able to delete the custom configurations saved before in the Configuration scene. He can also see the different values of the configurations he created and the values of the default configurations (Sober, Drunk, Slightly Drunk, Very Drunk).



### Problems in the Delete scene

In this scene is necessary a scrollable list with all the configurations, both the default and custom configurations. From this list the user selects a configuration item and sees its values and deletes it if desired. If the selected item is a default configuration an error message is shown:



The problem is that Unity does not have standard GUI Elements like scrollable lists with selectable items. They have to be built from the given Elements (Buttons, Fields, Panels, etc.) and filled dynamically at run time.

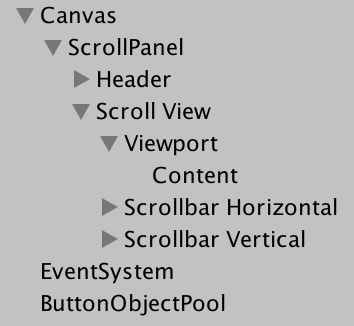
### Solutions in the Delete scene

After an Internet research we found several tutorials, which address this problematic, being the Unity tutorial[[2]](#footnote-2) the most appropriate for our needs.

This tutorial makes a scrollable list with buttons as list items starting from a panel with following features:

* The scrollable panel automatically scales to fit the content
* The content (buttons in this case) have their preferred sizes
* The content is clipped/masked using an image mask
* Vertical Scroll

The object’s hierarchy tree looks like:



#### Scrollable List Details

Here’s what each object does and the components it needs:

**Canvas[[3]](#footnote-3)**

The Canvas is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.

**Scroll Panel**

This is just the parent Game Object containing the other elements

**Header**

A button without functionality

**Scroll View**

Contains elements that set up how the content is viewed. The **Viewport** is the component that actually contains the list elements. The both **Horizontal** and **Vertical scrollbars** define the list’s behavior when the user scrolls up or down the list. Logically only vertical scroll is allowed.

**Viewport Content: Button Prefabs**

The list will be filled on runtime with default configuration and custom configurations. That means that the length and size of the list is unknown until the scene is loaded and that we have to generate the list items and add them to the list dynamically.

For this we use Unity Prefabs[[4]](#footnote-4), a Game Object template from which you can create new object instances in the scene.

We built a Button Prefab with our desired features, and we instantiate them like:

public GameObject listButton; *// Button Prefab*

[…]

private void FillListInGui()  
{  
 foreach (var file in configFilesList)  
  {  
    var newButton = Instantiate(listButton); *// Instantiates Button Prefab*  
    var button = newButton.GetComponent<ListButton>();*// Button contains a C# class*  
    button.nameLabel.text = file.FileName; *// New button shows the file name*   
    var index = configFilesList.IndexOf(file);  
    button.button.onClick.AddListener(() => { OnButtonClicked(index); });  
    newButton.transform.SetParent(contentPanel); *// Adds new Button to parent panel in*

*// order to keep the hierarchy tree*  
     }  
}

The advantages of prefabs are obvious. You are able to reuse the prefab in different scenes in combination with different Game Objects and settings.

For more detailed information about the scrollable list with selectable items please refer to the above-mentioned Unity tutorial.

## Main Scene

The main scene is the entry point to the Simulating Substances application. From this scene the user can create or delete configurations, start from a default configuration or start from a custom configuration.



### Problems in the Main scene

Same as in the Delete scene, the first problem again was to implement a scrollable list with selectable items. Please refer to the Solutions in the Delete scene section to see how we solved it.

Another unexpected problem in this scene we discovered is that the application’s event processing doesn’t work properly when the application is running on a mobile phone.

When the user tries to select a list item by clicking on it sometimes the first click get lost and the logically selection doesn’t work. This behaviour is not deterministic, sometimes it works and other times it doesn’t.

A long research revealed no documentation about this issue and the only preliminary conclusion we could come to is that it is a still not documented Unity bug.

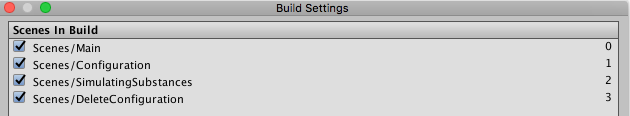
## Navigation between scenes

As our application has a simple structure with only 4 scenes we looked for a simple navigation solution too.

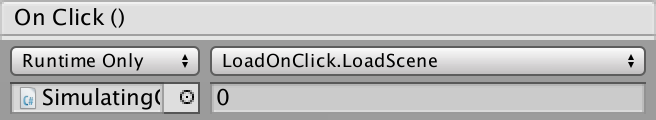
Unity offers the SceneManager[[5]](#footnote-5), which manages the scenes at run-time and has the method:

  private void LoadScene(int sceneIndex)  
        {  
            SceneManager.LoadScene(sceneIndex);  
        }

Each scene has an index in the Unity Editor:



Then when we need to navigate to another scene we call the LoadScene method from a GUI component, f. e. a button, with the corresponding index:



1. https://docs.unity3d.com/Manual/class-Camera.html [↑](#footnote-ref-1)
2. https://unity3d.com/learn/tutorials/topics/user-interface-ui/intro-and-setup?playlist=17111 [↑](#footnote-ref-2)
3. https://docs.unity3d.com/Manual/UICanvas.html [↑](#footnote-ref-3)
4. https://docs.unity3d.com/Manual/Prefabs.html [↑](#footnote-ref-4)
5. https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html [↑](#footnote-ref-5)