Balázs Dénes Kovács

Editor Tools Development for Creating Avatar Content

The Avatar Content Editor

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

20 September 2016

Author(s) Title Number of Pages Date	Balázs Dénes Kovács Unity Editor Tools Development 34 pages + 0 appendices 20 September 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Juho Iso-Markku, Title (for example: Project Manager) Antti Laiho, Title (for example: Principal Lecturer)

The goal of this final year project was to create a more efficient replacement for a tool used for creating avatar content for a new game. This new tool had to manage clothing, color and other products in direct communication with the servers of the game to aid artists and balancers with their jobs. Additionally, the user interface of the tool had to be more advanced and easier to use than of its predecessor in order to make a successful replacement. The development of this new tool involved heavy Unity editor scripting.

This thesis focuses on the creation, implementation and functionality of the new content editor tool in the context of other game engines and their tools. Furthermore, this paper tells about the history of similar tools and engines and gives a brief comparison of what is available today for manipulation clothes and similar avatar content in recent games.

Finally, the development project of the Avatar Content Creator resulted in a finished product. Now the tool is in use by the development team. The new UI of the made it easier to browse and manage products as well as the creation of game content. Furthermore, the resulting tool reduced the development time and the difficulty of releasing game content.

Keywords	Unity3D, Unity, C#, Game Engine, Editor, Tools

Contents

1 Introduction

Nowadays in the gaming industry it is a common practice for developers to use some sort of pre-made environment, game engines, to create games with. Utilizing the modularity of game engines is a fast and inexpensive way to develop the end product. There are many game engines on the market today to choose from. However, some engines are proprietary and may never be released for commercial use. Even so, the number of game developers is on the raise. With more people developing games more and more complex game projects surface. These type of games need specific tools that the game engine itself might not provide, however the engine could support such components.

Today more and more developers use Unity as the game engine for their games. There are over 4.5 million registered developers of Unity as of now [1]. Some of the projects these developers are working on require more than only a team of skilled programmers and artists to work together using only the platform Unity provides. In order to develop their games efficiently they need to rely on tools made for Unity for special purposes. These tools can be generic to an extent and thus coded by an external personnel or they can also be highly specialized to fit specific needs of the game project. For example, a character creator tool designated to help game designers of a role playing game to customize non player characters without having to write code or tweak a complex XML file. Another example might be a tool that showcases the key numbers of an in game store to help designers balance values and product releases.

In this paper I am going to describe the development of The Avatar Content Creator (ACE) that was proposed and developed in Sulake Corporation OY for Project X. The ACE is a highly specialized tool to create content for the avatars of the game. The purpose of this project was to re-implement an already existing tool called the Cloth Editor to facilitate more and better functionalities. Furthermore, the new tool had to support a faster and more advanced art and balancing process. In addition to this, the communication with the data servers of the game had to be overhauled to enable direct editing of the database of items. Finally, the project aimed to increase productivity and improve the stability of the avatar content creation process.

2 Custom Game Engine Tools

2.1 History of Custom Tools in Game Development

In the beginning of the history of games development, games used to be made as singular entities. Developers had to make an effort to build their software from the ground up and optimize them for the hardware they were developing for. It was only in the early 1990s that the term game engine surfaced. [2, 11.]

In 1993, before the game called DOOM came out id Software introduced the term DOOM engine. This referred to a revolutionary way John Carmack3333, the lead programmer at id, organized components of their game. He made a modular separation between the creative assets and functionality in his code. There were distinct core elements of the engine such as the three-dimensional graphics rendering system, the collision detection system, and the audio system. Also, the levels, art assets and the rules governing the game were well separated. [2, 11.]

Further on, developers realized the benefit of having their own game engines where they were able to create more games, mainly in the same genre, by swapping elements inside the engine. For instance, replacing weapons and enemies or other art assets. In the late 1990s game engines like id Tech 3 and its game Quake III Arena and the first Unreal Engine were designed with reusability in mind. Furthermore, this meant that anyone could make their modifications to the games made with those engines. Thus the mod communities were born. These groups either made small modifications to the game or such big ones that yielded a completely new game using toolkits provided by the developers of the games. [2, 11.]

Early proprietary game engines and the toolkits developed for them were the first instances of high level tools being used in game development. Being able to edit levels visually in context of the game engine or visualizing rules and behaviors in the game were among the many useful features game engine tools provided in the beginning. Nowadays, there are many more tools created by the original game engine developers and independent developers alike. These tools can be generic, such as a grid creation tool, and some of them are designed to deal with a specific task.

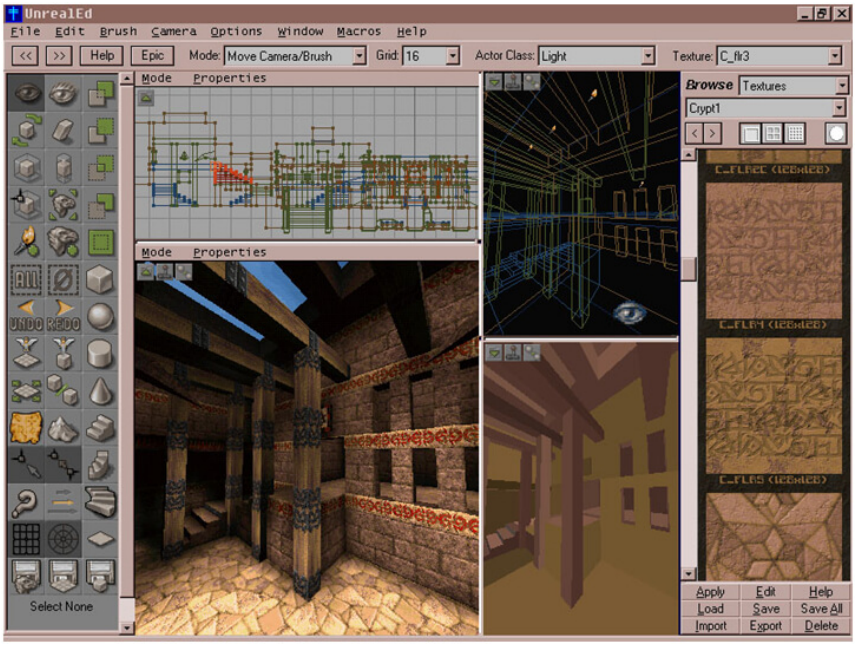

Figure 1. Example of the editor of Unreal Engine in 1998, Reprinted from www.unrealengine.com (2016) [10]

As seen on figure 1, the earliest version of the Unreal Editor used to be a simple tool for creating and editing maps for games made in Unreal Engine 1. It features a live preview, which was revolutionary in its time in 1998. Users of this editor can create geometric shapes and stairs as well as use selection and cutting tools to sculpt and compose their levels.

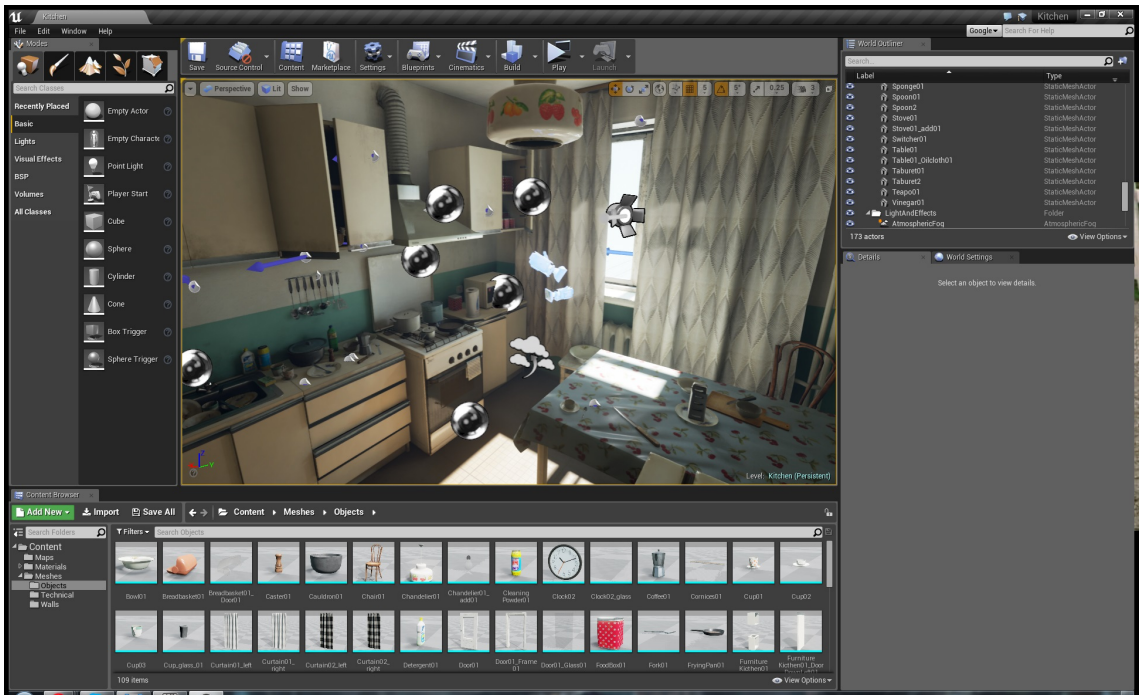

Figure 2. Example of the editor of Unreal Engine 4, Reprinted from 80.lv (2016) [11]

However, the following iterations of the editor gradually became the tool for creating separate games using the corresponding Unreal Engine version. The latest available tool is the Unreal Engine 4 Editor (UE4 Editor). Figure 2 shows an example view of the UE4 Editor. This editor has more features and more sophisticated tools to aid game developers in their work.

On the other hand, the Unity engine did not have such a primordial stage of a level editor. This engine and its editor was made to be a generic tool for creating games. The first version of unity was released in June 2005 [12]. The first version of the editor, as seen on figure 3, looks very similar to the latest versions of Unity. From the beginning Unity enabled the creation of 3D games on both Mac and Windows systems.

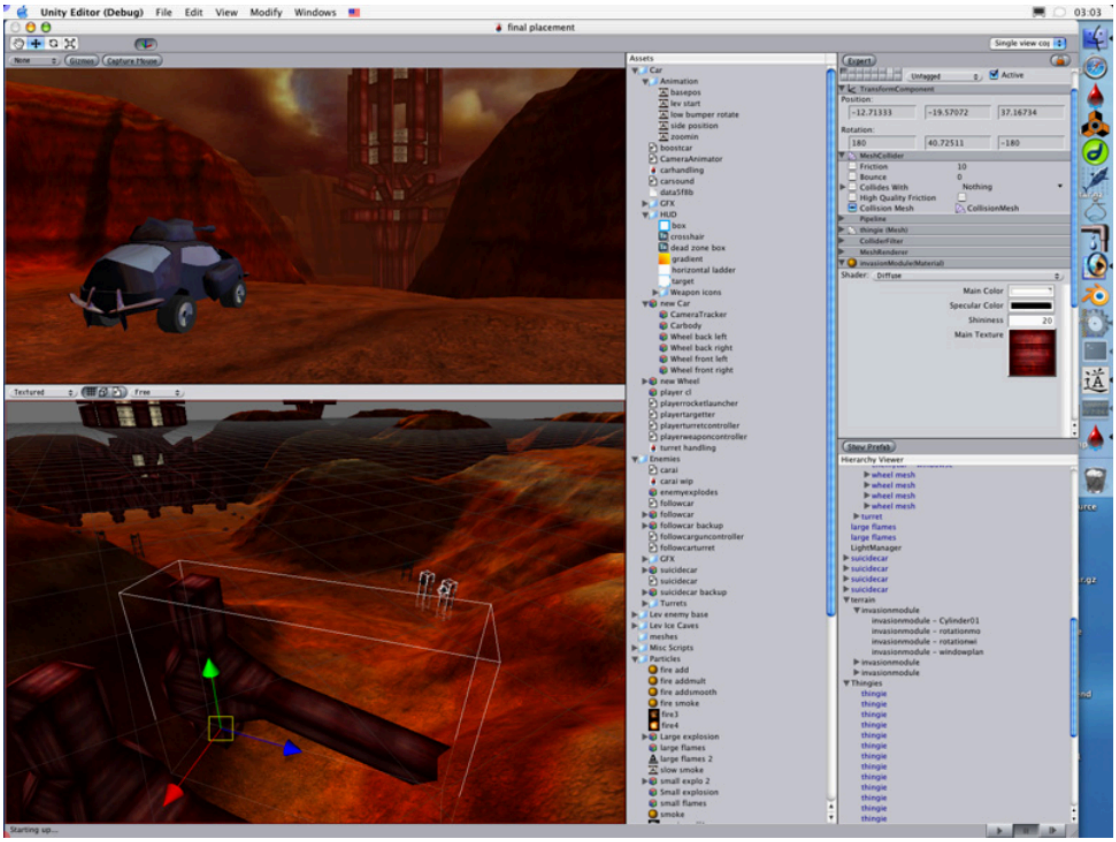

Figure 3. Editor Window of the Unity 1.0, Reprinted from Haas (2014) [12]

The later versions of Unity gradually introduced more and more cross-platform compatibility as well as many other game engine improving features. Among many features different and updated shaders, a better animation system and support for 2D games were introduced. [12.]

2.2 Avatar Customization

The demand for the ability to customize the in-game representation of players in recent games is high. Many RPG and MMORPG games offer such functionalities. Players may alter the sex, ethnicity and age of their avatars. Furthermore, hairstyles and clothes as well as their colors can usually be changed. These are an important elements of games where players are not supposed to play an existing character with a set personality. This also engages them to become more involved in these games. Players feel more attachment to the characters which they are allowed to customize. This makes them feel that they are part of the game to some extent. [3, 70-71.]

There are several examples of avatar customization between games. One of the best examples is the Mii editors made by Nintendo. It has many features through which players can make lookalikes of themselves, family members, celebrities or even cartoon characters.

Figure 4. Wii U Mii Studio, Reprinted from Player Essence.com (2012) [4]

The Mii Studio for the Wii U is one of the editors for Miis made by Nintendo as seen in figure 4. Miis are a generic character many games on Nintendo consoles use since the release of the Wii in 2006. Players may race with their Miis in Mario Kart Wii or fight other Nintendo characters in the Super Smash Bros. franchise.

On the other hand, there are character customization features in games that do not perform well. For example, the customization screen of the game Kendall & Kylie. The characters do not change much no matter what the player picks to customize it. The clothing styles are very plain and the variety does not let players choose freely a set of clothes they would feel comfortable wearing.

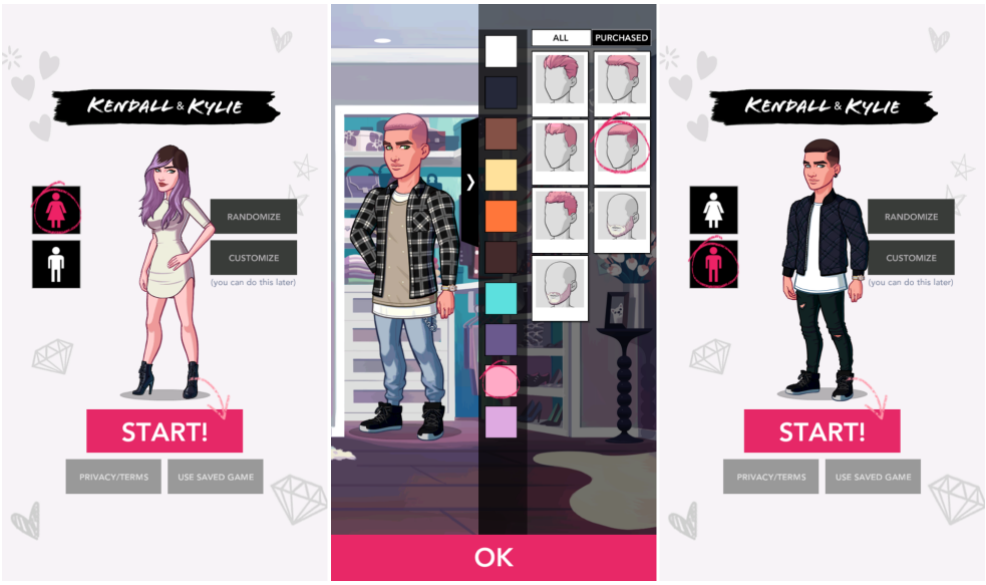

Figure 5. Screenshots of Kendall & Kylie, Reprinted from Cosmopolitan (2016) [5]

As seen on figure 5 the offered colors for customization are not matching the colora on the items. Additionally, the behavior of these characters in the game are not representative of the actions or emotions of the players. The characters of this game are only customizable to a limited extent. Styles and colors only represent the narrow subculture the game is catering to and other cultures portrayed from the perspective of this audience. This essentially does not allow for the freedom of expression a customizable avatar should provide.

2.3 Unity Editor Scripting

Editor scripting in Unity is a useful practice for large projects. It allows the developers to build tools and automation which are not implemented by the original developers of Unity. The purpose of this practice is to speed up, simplify and aid game creation processes. [6, 2.] Custom tools made by editor scripting can range from custom inspectors to any form of internal utilities such as custom windows, wizards or useful scripts executed from the menus of the editor.

Unity editor tools might range from generic purpose to highly specialized ones. Using editor scripts one may create custom inspectors. Unmodified inspector views show all the basic properties of the object selected in the hierarchy. Similarly, the inspector reveals all the public fields of the script and draw the default inspector for them. Once the scripted game object gets too many properties and becomes a burden to handle, making a custom inspector might be a good idea. With the help of the editor scripts developers can organize the fields of the objects into logical groups and even implement methods to handle the data from the inspector in a specific way. [6, 49.] Furthermore, the default inspector might be drawn if needed through editor scripts. A simple use case for editor scripts is to display the level of a character object based on a calculation, where the character script itself only stores information about experience points it gathered.

Additionally, the Unity editor features classes such as GUIStyle and GUISkin to allow further modification of the default look of the editor window elements. With the help of the GUIStyle class the font, the color, even the size of the elements can be changed among many other options. [6, 147-150.] Using the GUISkin class the developers may define new skin packages for the default editor window components [6, 156]. Applying a GUISkin rather than using GUIStyles with each element is generally a better approach.

Furthermore, saving data into scriptable objects persistently is also possible with editor scripts. Scriptable objects always exist in the project without being attached to a gameobject. This is why they are widely used for saving changes during play mode. Moreover, saving information into scriptable objects have some benefits over using XML or JSON files for the same purpose. Finally, allowing persistent changes to certain elements of the game gives liberty to game designers when they have to adjust certain values to make the game a better experience for players.

3 Creating Unity Editor Tools

3.1 Custom Inspector

A custom inspector is a view of a specifically modified script in the inspector view of the Unity editor. In any default inspector of a MonoBehaviour each public variable of the script is exposed in a corresponding field.
using UnityEngine;

```
public class InspectorExample : MonoBehaviour {
public int variableA = 100;
[SerializeField]
private int variableB = 200;
// This variable won't be exposed in the inspector
[HideInInspector]
public int variableC = 300;
}
```

Listing 1. Hiding public fields in the inspector, Copied from Tadres (2015) [6, 54]

Any value in the default inspector is changeable and are also serializable. Hiding an exposed public field is possible by attaching a HideInInspector attribute to it as seen in listing 1 and the result in the inspector in figure 6.

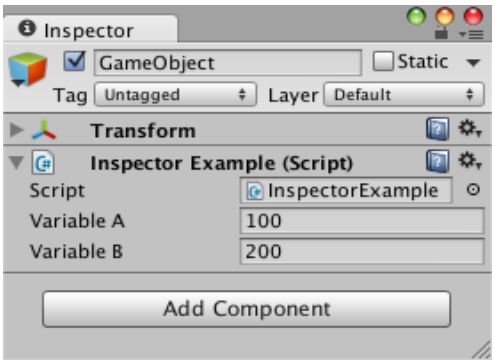

Figure 6. Hidden public fields in the inspector, Reprinted from Tadres (2015) [6, 54]

On the other hand, developers might not satisfied with the default inspector and variable hiding or the user friendliness of the resulting interface. They may implement other interfaces utilizing editor scripting.

In order to create a custom inspector for a class the CustomEditor attribute must be used on a class that inherits from the Editor class. The attribute also needs the type of script it is going to be used to create the custom inspector for as shown in listing 2.

```
[CustomEditor(typeof(Level))]
public class LevelInspector : Editor {
}
```

Listing 2. The usage of the CustomEditor attribute. Reprinted from Tadres [6, 55]

This inspector class will cause the Unity editor to use the OnInspectorGUI function of the class whenever inspecting the script specified in the type of the CustomEditor attribute. A Class structure in the previously specified way has a variable called target. This variable stores the inspected element as a Unity object. Therefore, if developers want to access the script they need to cast this object to the appropriate type. [6, 55.]

Furthermore, in cases when the default inspector should be part of the custom inspector, one may add the DrawDefaultInspector function inside the OnInspectorGUI to draw it [6, 58]. The placement of this function is completely arbitrary and will only have an impact on the layout of the inspector. However, drawing custom elements in the inspector can be done by using the functions of the EditorGUI and the EditorGUILayout functions. Additionally, some more elements can be found in the GUILayout class as well. Classes with Layout in their names refer to elements that can be organized into a layout. Whereas the ones without the word have drawing functions that needs to have a Rect specifying their position and size on the display area.

Layouts in the Editor can flow from top to bottom or from left to right. The inspector is drawn with a top to bottom layout. These layouts can be changed by the script and arranged like tables in HTML scripting. The functions from the EditorGUILayout class are used for a vertical layout are BeginVertical() and EndVertical(). Also for the horizontal flow BeginHorizontal() and EndHorizontal() are used. [6, 66.] An example of the usage of these layout functions is shown in listing 3.

```
EditorGUILayout.BeginVertical("box");
EditorGUILayout.LabelField("Name: " + _itemInspected.name);
Editor.CreateEditor(_itemInspected.inspectedScript).OnInspectorGUI();
EditorGUILayout.EndVertical();
```

Listing 3. Example of the Usage of the EditorGUILayout elements, Copied from Tadres (2015) [6, 134]

Finally, organized into layouts are the fields and controls. Layout elements can be drawn in the custom inspector with the help of the GUILayout and the EditorGUILayout classes functions. There are several type specific fields developers can choose from. All of these fields have several overloads to accommodate different needs.

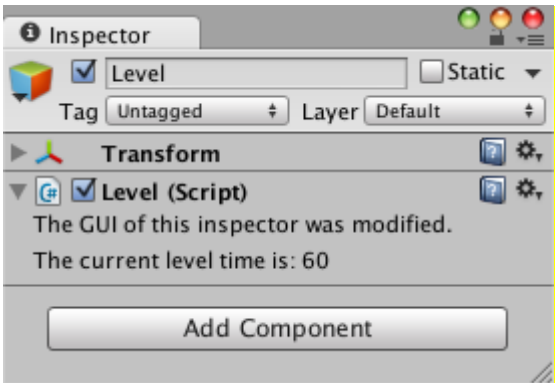

Figure 7. Messages drawn by the LabelField function, Reprinted from Tadres (2015) [6, 58]

For example, some fields need a label and some do not. The overloads of the field functions allow for both type of displays as seen in figure 7. Also these fields and controls after all the initial properties they take as arguments, they take more arguments in their overloads such as GUIStyle and GUILayoutOptions. The properties allow to change the general look and feel of the field as well as the fixed and dynamic size of the GUI elements. [6, 66-67.]

3.2 Property Drawers

Property drawers gives developers control over how a Serializable class or property is drawn in the Inspector GUI. Using these elements of Unity, it is possible to customize the inspector without writing code for an entire custom inspector. Furthermore, using the property drawers coders can plug in custom Serializable classes into their default inspector to be displayed. [6, 70-71.]

There are several built-in property drawers in Unity. These are attributes that can be applied to all the default serializable fields that Unity recognizes without further instructions. For instance, integers can be alternatively displayed with the Range property drawer attribute.

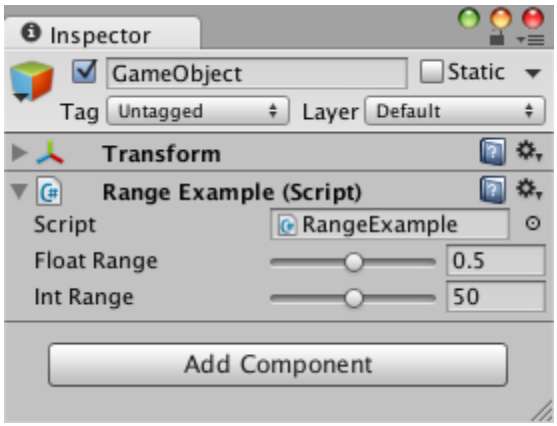

Figure 8. Example of the Range Attribute, Reprinted from Tadres (2015) [6, 72]

This attribute will draw a slider next to the default number input field in the Inspector as shown in figure 8. Additionally, other property drawers, such as the TextArea and the Multiline attributes, can make the default single line text fields to display as multiple lines with and without scrolling options.

```
[ContextMenuItem("Reset this value", "Reset")]
public int intReset = 100;
public void Reset() { intReset = 0; }
```

Listing 4. Usage of the ContextMenuItem Attribute, Reprinted from Tadres (2015) [6, 74]

Furthermore, functions can be added into the right click context menu of the inspector by using the ContextMenu attribute. Also with the help of the ContextMenuItem, as seen in listing 4, attribute functions can be called specified in its the second argument. [6, 71-74.]

Finally, Unity also features decorator drawers. The purpose of these attributes is to give developers a way to organize their custom property drawers. The built-in decorator drawer attributes are Header, Space and Tooltip. The Header attribute is used to add a bold label field atop of a property in the inspector. For adding a specified height space

between elements the Space attribute is used. Lastly, the Tooltip attribute enables developers to attach a message for the users of the inspector to read when they hover the property this attribute is attached to. [6, 75-76.]

3.3 Custom Editor Windows

Certain custom features in the Unity editor are not suitable to be implemented as a custom inspector. For these features Unity allows developers to create their own editor windows derived from the EditorWindow class, similar to the built-in windows of Unity. Ton instantiate a window from a class inheriting from the EditorWindow the GetWindow function has to be called as seen in listing 5.

```
public class PaletteWindow : EditorWindow {
    public static PaletteWindow instance;
    public static void ShowPalette () {
        instance = (PaletteWindow) EditorWindow.GetWindow
            (typeof(PaletteWindow));
        instance.titleContent = new GUIContent("Palette");
    }
}
```

Listing 5. The Basics of a Custom Editor Window, Reprinted from Tadres (2015) [6, 88]

The ShowPalette function can also be given a MenuItem attribute so the window could be opened from the menus of Unity. Also, windows added in the menus can be called using keyboard shortcuts. These shortcuts are special characters written inside the string of the MenuItem attribute separated from the menu path at the end of the string with a white space. Furthermore, when the window is drawn, Unity will display the contents specified in the OnGUI function of the custom window class. Also for asynchronous data manipulation developers can utilize the Update function which runs 100 times every second when the window is shown. Finally, it is a common practice to use events to communicate with the scene. Developers can use these events do detect virtually anything they desire and show the necessary information in their custom windows. [6, 86-92.]

3.4 GUI Skins and GUI Styles

To be able to change the look and feel of the custom editor GUI developers need to utilize the classes GUIStyle and GUISkin. Before the 4.6 release of Unity these classes were solely used for game UI customization. As mentioned in the previous section GUI components accept an optional GUIStyle to override their default settings.

```
EditorGUILayout.LabelField("MyTitle", EditorStyles.boldLabel);
```

Listing 6. The usage of EditorStyles in GUI components, Reprinted from Tadres [6, 148]

A set of static styles used in the default editor of Unity can be found inside the EditorStyles class. Listing 6 illustrates the usage of such styles on a label field element.

Alternatively, developers may create custom GUIStyles to fit their needs. These styles are initialized with default values such as black font color. Although, the default values do not mirror the styles used in the Unity editor. Because of this, developers need to tweak several attributes to achieve the style they aim for. [6, 148.]

Each GUIStyle defines specific states required by the Editor GUI. These states are stored inside the GUIStyle in GUIStyleState classes storing specialized values. The default state for each element is called normal. The several other states store information about the looks of the controls when the mouse is interacting with them in different ways. [6, 152-153.]

Furthermore, GUIStyles can be organized into the GUISkin class which allows to customize the whole UI instead of separate elements. This class must be created as an asset. It is extending the ScriptableObject class.

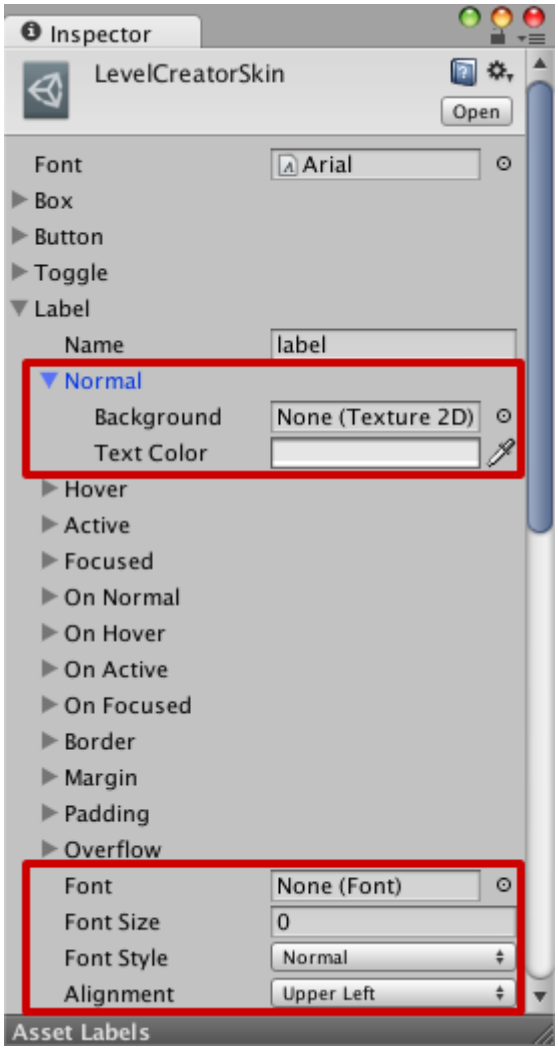

Figure 9. GUISkin custom inspector. Reprinted from Tadres (2015) [6, 159]

Additionally, the pros of using GUISkins is their reusability in other projects. Finally, these assets generate their own custom inspector to facilitate the graphical editing of GUIStyles as seen in figure 9. [6, 156-161.]

3.5 Scriptable Objects

Unity has a special object type called scriptable objects. Scripts inheriting from the ScriptableObject class do not need to be attached to an instance of a game object inside the scene in order to exist. This is due to these elements are being saved as assets in the project. Using scriptable object have the benefit of automated data handling and parsing in contrast to using plain text, XML or JSON formats. Furthermore, with the use of scriptable objects developers can create custom editors for game designers where they can

test certain aspects of the game with different values and make the changes persist. [6, 163-164.]

An example for using Scriptable Objects is to make object templates. These templates are easily modified without coding and can be added to the game. Such templates could be applied for in-game character creation. For instance, creating enemies which might be slight variations of each other. The process of creating such enemies is the following. The game developer makes the first template for an enemy. Then he or she saves the first instance into an asset file. This asset file can be duplicated and saved as another type of enemy. At this point, the enemies are identical. However the files can now be independently change, thus easily creating similar enemies with assets utilizing Scriptable Objects.

3.6 AssetPostprocessor Scripts

The AssetPostprocessor class allows developers to specify specialized settings for importing different assets. Rules for importing can range from a simple name filters making changes based on the name of the files to a sophisticated file content based check modifying the incoming assets. Gathering all these rules into a DLL is a good practice to avoid compilation errors inside the Unity project or undesired import settings. [6, 178-180.]

```
using UnityEngine;
using UnityEditor;
namespace RunAndJump.ImportPipeline {
  public class TexturePipeline : AssetPostprocessor {
    private void OnPreprocessTexture () {
      Debug.LogFormat("OnPreprocessTexture, The path is {0}", assetPath);
    }
    private void OnPostprocessTexture (Texture2D texture) {
      Debug.LogFormat("OnPostprocessTexture, The path is {0}", assetPath);
    }
  }
}
```

Listing 7. Example of Inheriting from the AssetPostprocessor class, Copied from Tadres (2015) [6, 179]

There are many events available inside the AssetPostprocessor class for detecting changes in the project assets. Some of these events detect only one type of assets

changing and some detect different sort of changes across all assets. For example, capturing all the textures being imported for a 2D game and modifying their attributes to become sprites can be done using the OnPostprocessTexture event. [6, 181-184.] Listing 7 shows the usage of the AssetPostprocessor class.

Additionally, using the properties of the AssetPostprocessor in a child class gives developers access to the specific import settings of each asset type. This can be achieved by typecasting the assetImporter property into the correct type. Once this cast is in a variable, it is freely configurable for the needs of the project. [6, 181-184.]

4 The Avatar Content Editor

4.1 Functionality

In General

The ACE is a tool that facilitates the editing and creation of avatar content such as clothes, set decorations, color, skin colors, clothing sets and gestures with the help of ACE you can modify data of the products and items listed. Furthermore, as figure 11 illustrates the ACE interface is a custom inspector of a singleton class. This class, as the UML graph in figure 10 illustrates, interacts with a number of other manager classes in order to provide all its functionality.

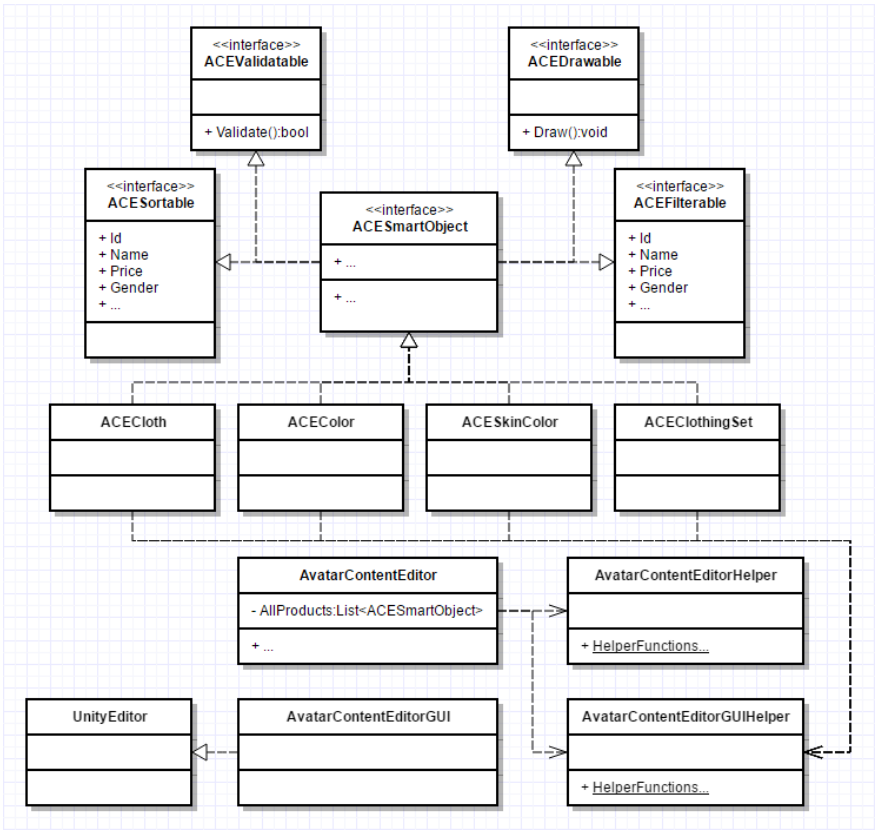

Figure 10. Rudimentary UML Representation of the Classes of the ACE

The AvatarContentEditor class hold onto all of its data using a list of ACESmartObjects. The ACESmartObject interface is used as a mean to unify similar data to be easier to draw, manipulate and save. The ACE user interface consists of two main parts. The inspector view provides users with control over all the game content via the custom inspector if ACE. Also, to preview items in the context of the game view of the Unity editor is used.

Figure 11. The basic look of the Avatar Content Editor

Clothes, colors, skin colors and clothing sets are stored on the servers of H4K, while set decorations and gestures are stored on the hard drive of the users. Because of this duality in storage methods, the ACE does not make changes to any of the data until the users sync and save their changes. With the help of the ACE users can also take screenshots of dressed up and animated avatars, also known as mannequins.

Dialogs and Visual Feedback

Feedback to the users of the ACE is very important. In order to determine whether something is done wrong or the tool needs time to load this tool uses a combination of dialogs, progress bars and various colors to explain certain situations clearly to its users. For instance, as the ACE is started up, the entire view in the custom inspector becomes disabled and several progress bars appear to notify the users of the loading process and the steps involved.

Figure 12. Browser Button Background Colors on Clothing Sets

There are several colors used to tell artists and developers about certain properties of items and their attributes. Red is used to highlight fatal errors, which would prevent data from being processed. The button of each item having an error will be colored red as well as the fields in their attributes holding erroneous data.

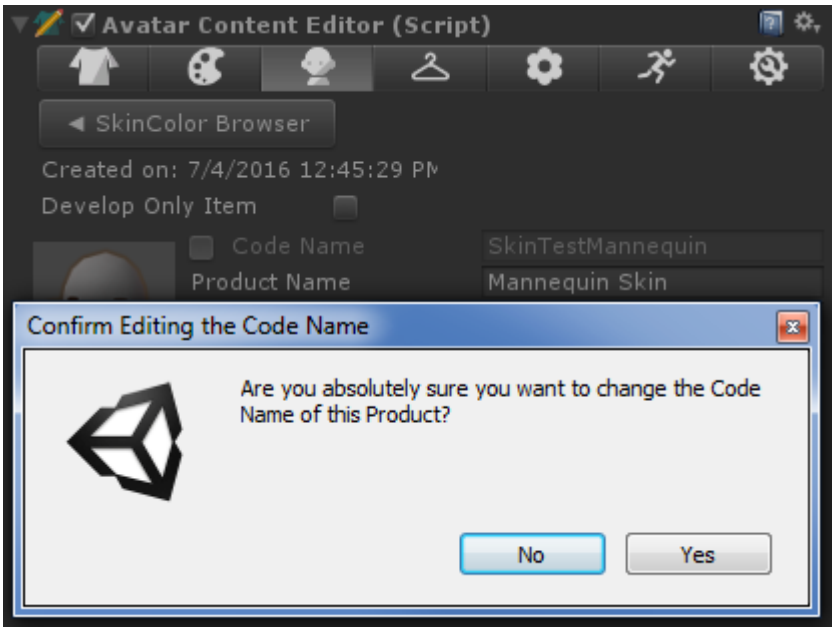

Figure 13. Code Name Changing Dialog

Furthermore, yellow is used in two ways. It refers to fields in the attributes which contain some sort of warning. In the browser views, yellow buttons mean the item has been modified and not yet saved. Finally, blue is used on the browser view buttons to show that that item is new and is not yet saved. All of these colors are shown on browser buttons in figure 12.

Additionally, several dialogs will also be shown to the users following a regular workflow. These dialogs are set to dismiss the action of the user by having their default selection being a negative answer to prevent accidental data entries. Actions like changing the code name of items or syncing to a server will prompt users in such manner as seen in figure 13. Furthermore, an even more sensitive action has to pass a double dialog check. This action is the migration between servers which overrides all item information on the destination server. Thus, this needs to be well kept from accidental clicks.

Validation

The ACE features a validation functionality which provides helpful feedback on suspicious or erroneous properties of certain items. Suspicious items can be products which are set to be purchasable, although they are also either free or are not assigned to be available in any shop. This would mean that the item is either for free or not available at all in-game. Fields with these type of warnings are colored yellow and show a description bellow the attributes of the items. These descriptions are marked with an exclamation point inside a yellow triangle as seen in figure 14. Additionally, fields with errors are marked with red. This directs the attention of the user to the data fields where they must make changes. As seen on figure 14. these fields also have a description bellow the attributes. Furthermore, the descriptive error and warning messages appear in the same order as the fields flow on the UI. However, the errors have a higher priority to fix, so those will always be displayed before the warning.

Error and warning flags are the indicators of the results of any validation process in the ACE. Each validation function will take one or both of these flags as references and set the appropriate flags according to the validation rules set in the function. It is a function of each class that implement the IACESmartObject interface. These validate functions are doing type specific checks against certain rules on each of their properties. The validation is a function called for each and every change on the attribute fields.

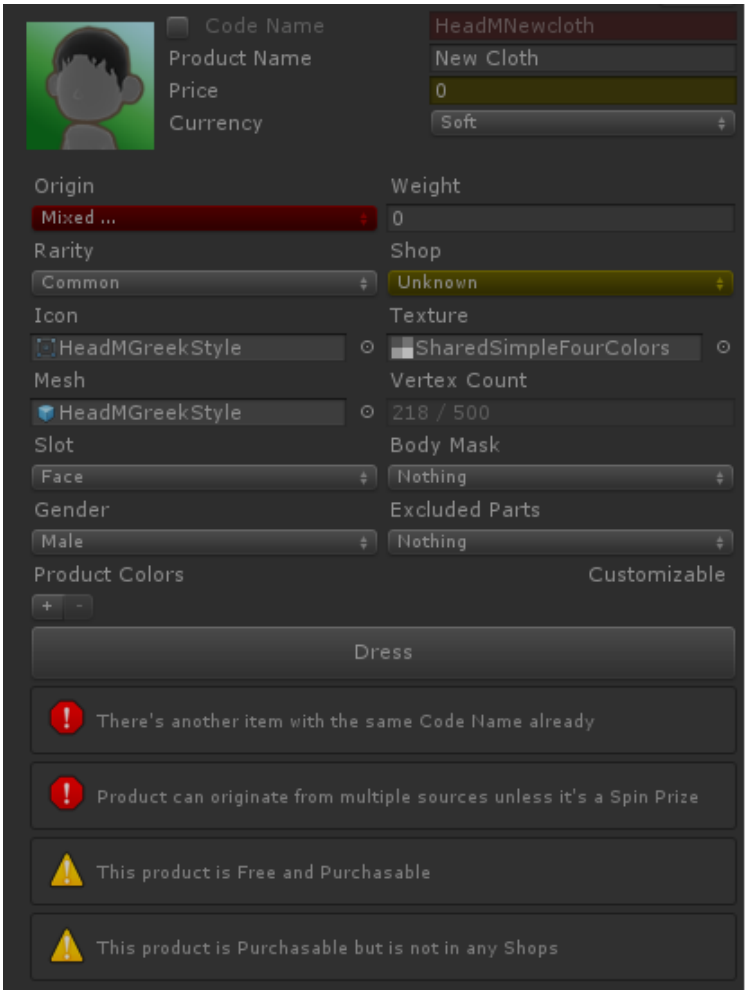

Figure 14. Example of Validation in the ACE

This means that even if a single letter is typed into a text field, the validation will be run. There are many rules this process is based on. For instance, the code name of any given item must not match another. Also, another example, clothing items must have all of their meshes, textures and icons assigned to them. Each of these rules are stored in static functions in the ACEHelper class which return a boolean value and set the error and warning flags of the given item. This way different items can call the same function for the same rules to validate fields with.

```
public static bool ValidateCodeName(string codeName, ref ItemErrorFlags er-
rorFlags)
{
    if (!string.IsNullOrEmpty(codeName))
    {
        errorFlags &= ~ItemErrorFlags.CodeNameIsEmpty;
        if (!Regex.IsMatch(codeName, @"^[a-zA-Z0-9]+$"))
```

```
        {
            errorFlags |= ItemErrorFlags.InvalidCodeNameChars;
            return false;
        }
        errorFlags &= ~ItemErrorFlags.InvalidCodeNameChars;
        return true;
    }
    errorFlags |= ItemErrorFlags.CodeNameIsEmpty;
    return false;
}
```

Listing 8. The Function that Validate Code Names in the ACE

An example of a validation function is shown in listing 8. This function will check whether the code name passed has any character and in case it has it will check if the code name is alphanumeric. After each check the appropriate flag is set on the referenced error flag variable.

Network Communication

The ACE has an active connection to the server the users are currently working on. Although, this does not mean that the tool is making constant and direct updates to the database. Instead, the ACE has to be manually synced to the connected server once the users are done with their current workflow. Also, the ACE does not support multiple person editing server data at the same time.

Once the ACE connects to a server it will stay connected and ping frequently to maintain the connection. In case the tool loses connection, it will automatically reconnect on any user initiated network action. This is to prevent data loss from artist and designers working on items locally in case they did not manage to save.

When a connection is established between the client and the server, the ACE sends a message to the server to request all the items stored in the database. The tool waits until both set of data, products and clothing sets arrive and gets parsed then enables the custom inspector and allows users to interact with the items. Additionally, when users hit the reset button, as illustrated in figure 15, in the ACE settings, all the item data is cleared from the local session and will be downloaded again from the server.

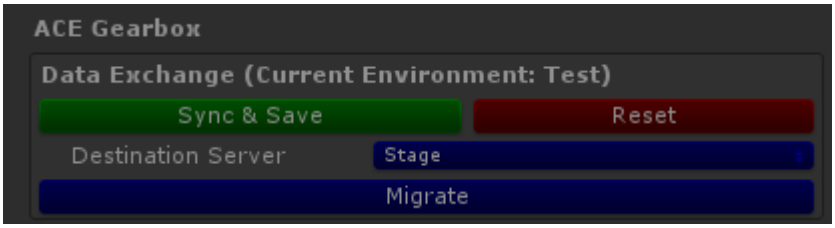

Figure 15. Network Action Buttons in the ACE Gearbox

Also, clicking Sync & Save will serialize the data of all the modified items relevant to the server and will upload them. The server deserializes and stores the data after it goes through another validation process. Finally, the migration process is essentially a composite action which will first initiate a reset from the current server to ensure data validity. Then, it will connect to the selected destination server and start the sync process. Once it is done, the ACE disconnects from the destination server and will connect to the origin server again and pull the data again so users can continue editing on their selected server.

4.2 Items and Products

With the Avatar Content Editor artists and developers can create and edit many types of products and items. There are several types of items represented in the ACE. On the other hand, not all of these items are products as well. Clothing items and colors are purchasable in-game thus they can be referred to as products. The rest of the items such as skin colors, clothing sets, set decorations and gestures are not products. Clothing sets are a group of clothes which provide prizes on acquisition of all items in-game as well as special set decorations once all parts of the set are worn by players.

Each of these items are represented using a common interface called the IACESmartObject. This is an umbrella interface that implements many other interfaces in order to bring several functionalities into one as shown in listing 9. The IACESmartObjects are validatable, which means they have a function that returns a boolean value which determines if the item is valid or not.

```
public interface IACESmartObject : IACEValidatable, IACEFilterable, IACESorta-
ble, IACEDrawable, IACEUISpriteSavable, IRawProductConvertable
{
    EditorViews BelongsToView { get; }
```

```
    ACEItemFlags Flags { get; set; }
}
```

Listing 9. The Implementation of the IACESmartObject

Also, implementing the IACEFilterable and the IACESortable interfaces respectively mean these items will have fields by which they can be put into groups when users filter their browser view and ordered in specific ways. Furthermore, the IACEDrawable interface allows these smart object to be drawn in the custom editor of the ACE. Functions implementing this interface are drawing the browser buttons and the attributes of each item. Finally, the IACEUISpriteSavable and the IRawProductConvertable are responsible for ensuring the smart object has the functionality to save UI sprites whenever necessary and provides them with the ability to be saved as a RawProduct on demand. Once a class implements the IACESmartObject interface, it can be displayed and manipulated using the ACE. Thus, this interface allows for the addition of further items whenever needed.

4.3 Visual Components

The entire UI of the ACE is a custom inspector of the AvatarContentEditor class. Some elements are scalable when the size of the inspector is changed, but some elements need extra code to work with these changes. For example, the buttons of the browser views are coded similarly to what users can see in a generic file browser window of an operating system. The icons of browser view in the ACE are getting rearranged and spaced out dynamically as the size of the view changes. This allows more customizability and more flexible work space.

The ACE uses tabs to divide the UI into item categories and a setting section. The tabs are marked with an icon representing the item type they hold as shown in figure 16. Also, clicking Sync & Save will serialize the data of all the modified items relevant to the server and will upload them.

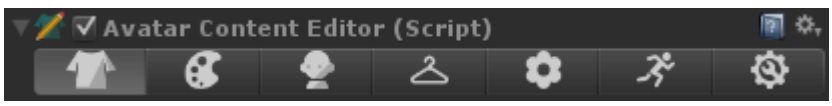

Figure 16. Tabs of the ACE

The icons on the tabs in order represent: clothes, colors, skin color, clothing sets, set decorations, gestures and settings. Clicking any of these buttons will show the corresponding sections. Because the browser views are generic each of these button presses will result in the ACE doing an initial type filtering to show the correct type of items.

Browsers and pickers are an essential part of the ACE. Through these views users can see and access the attributes or select items. Item browsers are a generic view where any IACESmartObject can be displayed. The functions of the browsers and pickers in the ACE are called to look at a filtered and ordered list that has every item that needs to be displayed. The browser differs from the picker in look and functionality as shown in figure 17. The picker is a cancellable view with less sorting and filtering options than the regular browser view.

Figure 17. A comparison between the Item Browser and the Item Picker

The filters and the sorting can be modified on top of the scroll view of the browser. These fields are from top to bottom: the search bar, flag filter, sorting options, shop filter, the origin filter, gender filter, clothing slot filter and rarity filter. These filters might not be present on every type. However, the IACESortable aad the IACEFilterable interfaces, as seen in listing 10, make sure that sorting and filtering on each IACESmartObject is possible.

```
public interface IACEFilterable
{
    ShopName Shop { get; }
    ProductOriginFlag Origin { get; }
    Gender Gender { get; }
```

```
    AvatarPart ClothingSlot { get; }
    ColorSlot ColorSlot { get; }
}

public interface IACESortable
{
    string Code { get; }
    string Name { get; }
    int Price { get; }
    CurrencyType Currency { get; }
    Rarity Rarity { get; }
    DateTime CreationDate { get; }
    int VertexCount { get; }
    ColorValue ColorValue { get; }
}
```

Listing 10. The Implementation of IACEFilterable and IACESortable

The search bar overrides all other filters and will show any item that matches the name or the code name entered depending on the state of the drop down menu on the right side. This means that users who know what they are looking for can this way find anything with ease. Additionally, sorting helps to arrange many items in an order specified in the drop down. Users can sort their items by the alphabet, by hard or soft currency value, by vertex count on items with meshes or they can arrange the colors specifically by their color values. Furthermore, filtering by origin means that the items can be filtered by how they are acquired in-game. Also, the shop filter is used to refine which shops the items are purchasable from. Finally, the slot and rarity filters reduce the list of items by showing only the selected rarities of items and slots of clothes or colors respectively.

Buttons shown in the browsers and pickers consist of many elements. Also, for different profiles different information is shown on each button. All buttons have an icon. Depending on the type of the item the icon will be either the in-game icon or a representative icons of the type. Figure 12 shows clothing set buttons, while figure 17 displays a clothing item button of the browser view. Generally, most items have a rarity attribute. These rarities are shown on top of the browser button with their in-game representation, stars. For the balancing, admin and developer profiles the same bars can be seen on the bottom of the icons with yellow and blue colors.

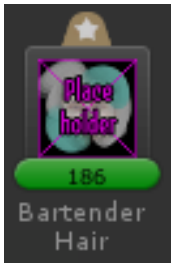

Figure 18. Vertex Count on a Browser Button

These represent the price of the items. Yellow for soft and blue for hard currency. Also, for these three profiles a shopping cart icon is displayed overlaying the icon to tell balancing user that that item is purchasable. This makes it easier for these users to pinpoint items sold in shops in-game from the mass of icons inside the browser view. However, for artist this bar represents the number of vertices on items which have meshes assigned to them. Also the bars representing vertex numbers are colored green as seen on figure 18.

Additionally, attributes of items are the most important component of the ACE. Through the attributes view artists and designers are able to change several settings. These settings are responsible for the correct representation of the items in-game.

```
private void ItemAttributes()
{
  if (EditorInstance.SelectedItem != null)
  {
    EditorInstance.SelectedItem.DrawAttributes(EditorInstance);

    EditorInstance.SelectedItem.DrawValidationMessages();
  }
  else
  {
    GUILayout.FlexibleSpace();
    EditorGUILayout.HelpBox("Select an item to see its attributes", Mes-
sageType.Info);
    GUILayout.FlexibleSpace();
   }
}
```

Listing 11. The Method of the ACE to Draw Item Attributes

Listing 11 shows the way the ACE is rendering the attributes of items. With the help of the IACESmartObject interface items can be rendered as specified in the class implementation of each type. There are many attributes the items share across types. However, there are many that are specific to each kind.

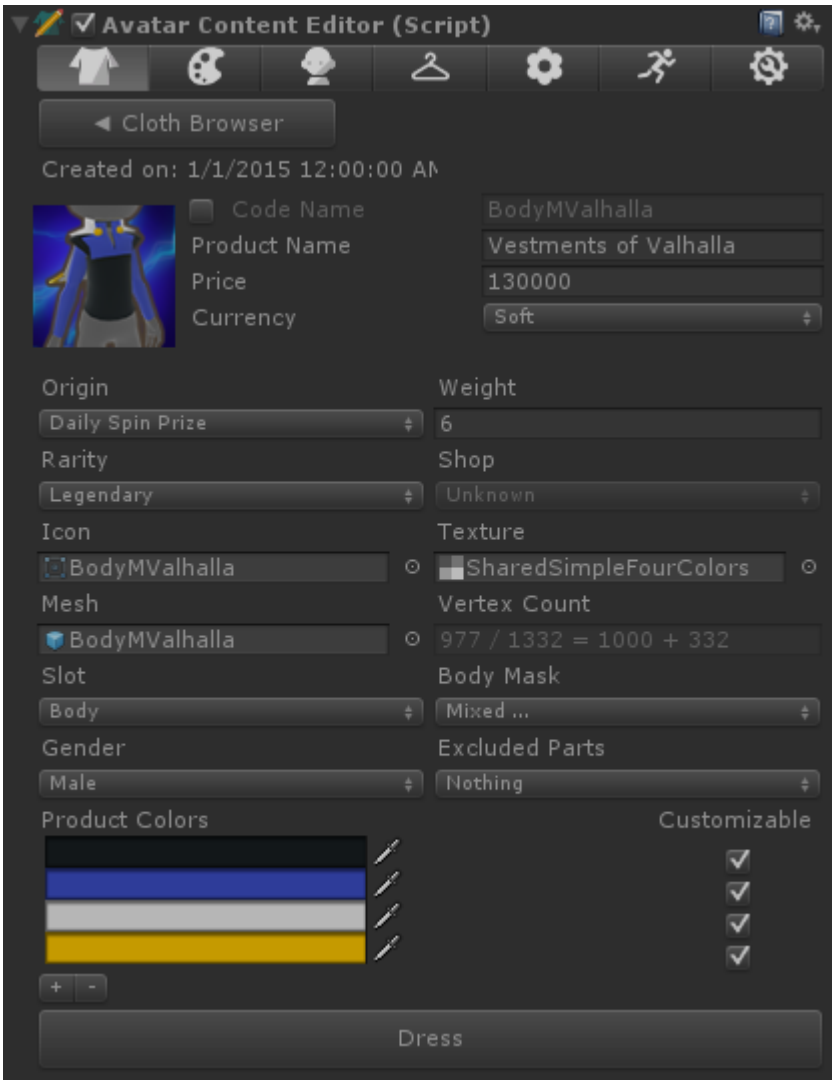

Figure 19. Clothing Attributes in the ACE

For example, as seen on figure 19, the attributes of the item named Vestments of Valhalla contain many fields. It contains common fields such as the code name, the name, the price and the currency fields. Furthermore, it contains fields that specify what assets the clothing item will use when drawn in the game. Also, a very important part of the item creation process where meshes are involved is the vertex count field. This field shows a calculation of how many vertices in a mesh are allowed for the specified item. It also

shows the number of vertices gained from masking specific parts of the avatar mesh. The faux equation consists of the following parts:

$$Mesh\ Vertices\ /\ Total\ Vertices\ Available = Max\ Vertices\ on\ Slot + Masked\ Vertices.$$

This is non-mathematical formula is just an indicator for the artists to show them information about how they need to optimize their new mesh assets for the game. Furthermore, the last section of the clothing attributes, shown in figure 18, is the color fields. These fields represent the default colors of a clothing item. These colors are used mainly to give a nice look to the clothes for creating their thumbnail for the in-game inventory. Users can add or delete colors by clicking the + or - buttons below the fields. Additionally, these colors can be set to customizable, which means that players can change the color of that slot freely. Clicking a color field will call a color picker view so artists can select from the range of colors already added to the game. This will prevent creating arbitrary colors for clothes and will make coloring consistent throughout the game. Finally, the placement of each color is decided by the texture and the UV map of the mesh, which is not editable from the ACE.

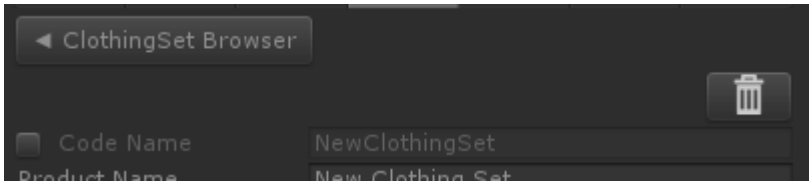

Figure 20. The Delete Button on a New Clothing Set

Furthermore, new items are always created locally. This means, they are not synced to the server immediately. In some cases, users might need to delete these items. As seen in figure 20, users need to click the button with the trash can icon in order to delete the current item.

Figure 21 shows the attributes of another different item type, the gestures. Gestures do not contain information about the animations associated to them. Instead, the animator and other gesture related events in the scripts make sure that the gestures are played correctly.

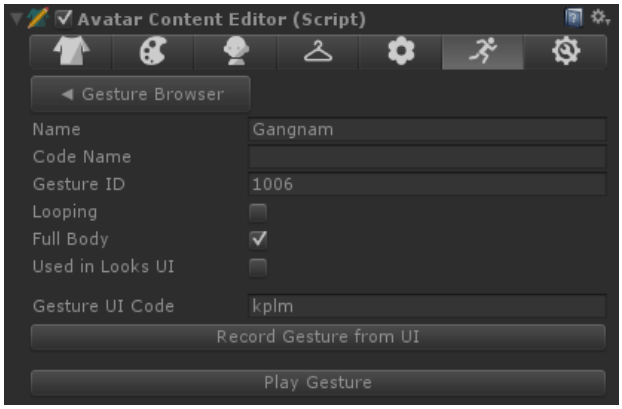

Figure 21. A comparison between the Item Browser and the Item Picker

The properties of this view only define the attributes which players are concerned about. Here artists and designers can set up what the gesture is called, type of an animation is the it and also change the UI gesture code which triggers the gesture from the in-game UI. Finally, all gesture can be previewed on the mannequin with the Play Gesture button.

However, most of the attributes of the items are very similar to each other the properties of clothing sets are fundamentally different. This difference can be clearly seen between the left and right hand side of figure 22. Clothing sets only share the code name and name as basic attributes with other items and the preview button as similar functionality.

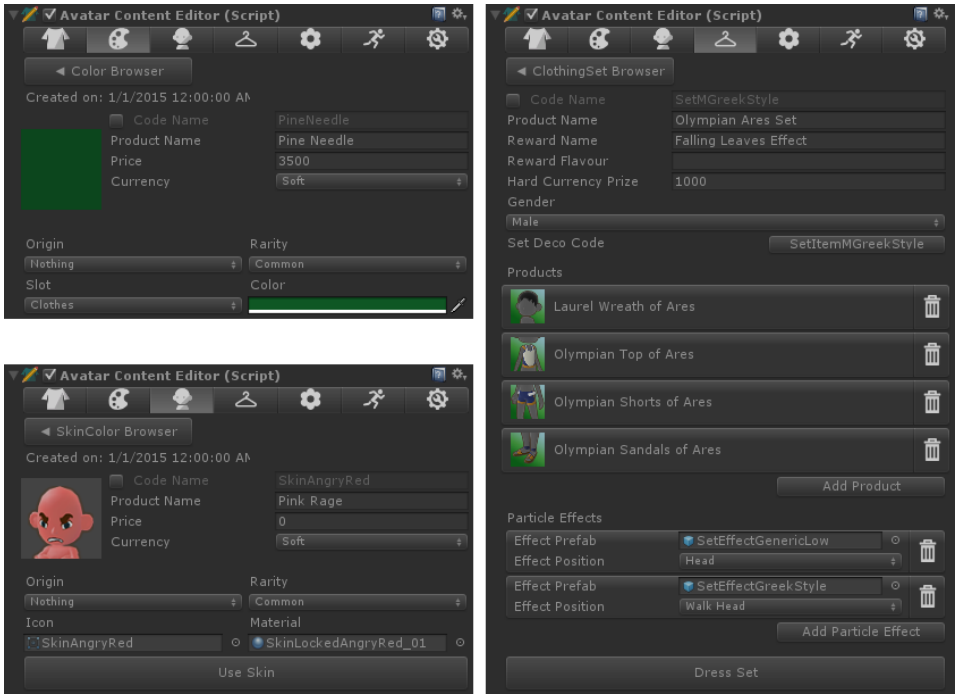

Figure 22. The Difference Between Color, Skin Color and Clothing Set Attributes

Clothing sets have many specific fields. These include a Reward Name and Reward Flavor fields which hold data for the in-game achievement system as well as the Hard Currency Prize which determines how much currency the players will get for acquiring the whole set. A set decoration can also be assigned to the clothing set by clicking the button labelled with Set Deco Code. This will launch a set decoration picker where users can select from all the available decorations. Furthermore, in the next section, the products section, users can assign new clothes to the clothing set by clicking the Add Product button at the bottom of the section. All the added clothes will produce a new button with an icon and the name of the clothes. Also, a trash bin icon is provided on the right side of the button to remove the items from the clothing set. Finally, particle effect can also be assigned to each clothing set which would be shown on the avatars when the full set is worn by players. These effects consist of the prefab of the particle effect and the position on the avatar it will be displayed.

The ACE also features a settings view. This view is visible outside play mode. However, there is a separated view during play mode available containing mannequin and animation controls. In the non-play mode version users of the tool can tweak many settings. Camera speed and preferences, dependencies and folders can be set up using this view.

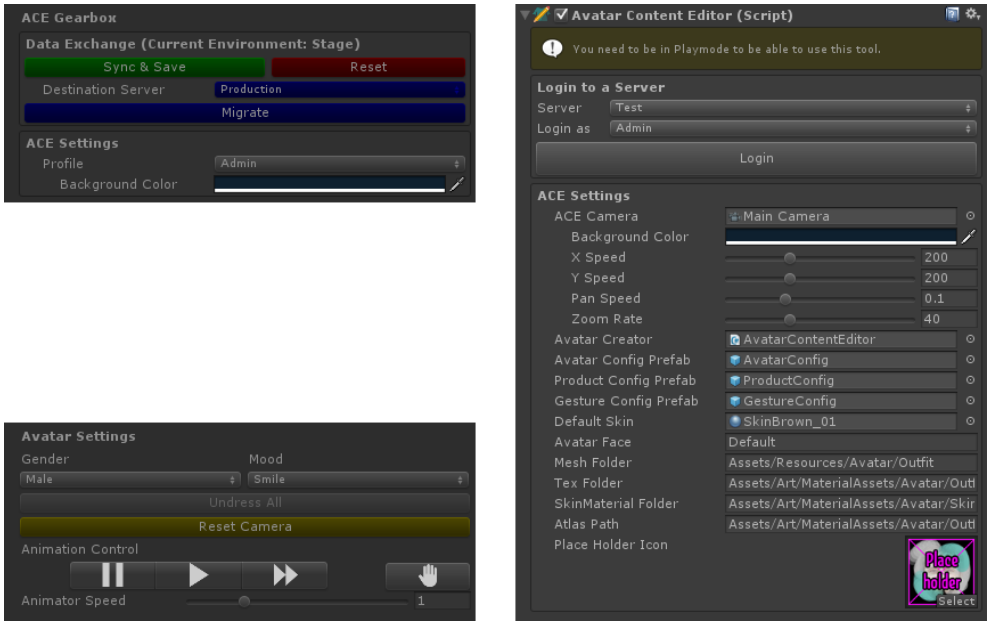

Figure 23. The ACE Settings in Play Mode and Outside of Play Mode

Also, a universal placeholder icon can be set for unfinished items, which would require an icon to be made later using the corresponding field as shown on the right hand side of figure 23. Additionally, the play mode version of this view consists of two parts. Shown on the top left of figure 23, one of the parts, the ACE Gearbox, contains the network action buttons, the profile changing drop down menu and the camera background color field. The other part, the Avatar Settings, contains mannequin related fields such as the gender and mood of it. This section also has buttons to undress all clothes from the mannequin as well as a yellow button to reset the camera to its starting position if needed. Furthermore, the animation controls are responsible for setting the animation speed of the gesture which can be previewed from the ACE. The animation speed can be controlled more precisely with the help of the Animator Speed slider. Finally, animations can be stopped completely with the help of the stop button, represented with a palm icon as seen on figure 23.

As previously seen on figure 11, the game view does not only contain the preview mannequin, but it also has a special UI element drawn for gesture control. This hexagonal UI element is responsible for reading drag gestures across its buttons and play the corresponding gestures on the mannequin. This element is a part of Project X and is made according to the MVC model. Each button has a unique identifier which the UI reads in the same order as the drag gesture was made. Through this element artist working on new gesture do not have to remember the button identifiers to compile a new drag gesture codes. They simply can hit the Record Gesture From UI button in the gesture attributes view and using the mouse drag a new gesture on the UI element.

5 Project Outcome and Accomplishments

This project was completed over the course of 5 months. During this time, I have learned many skills and accomplished several goals. In the beginning of the project I started my work with a basic knowledge of Unity and C#. My job in the project was to develop, code and deliver a tool to replace the old one, the Cloth Editor. During the first couple of months I have learned to use the Unity editor and the editor scripting more extensively. This allowed me to speed up my progress with the creation of the ACE. Furthermore, as research was also part of my job, I learned many useful skills and absolved many guidelines of the Unity development process. Also, I have learned more generic IT skills, such as using git and also how the release flow of a software is.

The first working iteration of the ACE was anticipated by the art team since the beginning of the project. The first version was finished in June and was tested with live data soon after. Many bugs were found. Among these bugs were not saving assets properly, not regenerating the texture atlas for the game and an identifier mismatch between server and local data. However not long after the discovery of these, the bugs were fixed and some extra features were added per request.

The ACE now enables the artists and the game designers of Project X to create new avatar content and edit them faster and easier. With the many visual features, validation and direct network connection to the servers the ACE allows for a less cumbersome experience than its predecessor. Using this tool, it would take approximately 1 to 2 minutes for an experienced user to create a single product. On the other hand, less trained persons might take up to 3 to 4 minutes to add their first products and learn how the ACE works. Creating a line of clothing with the ready artistic presets could take as little as 20 minutes. Also, the ACE was not made initially to be more than just a content editor tool.

However, the ACE turned out to be also a great tool as a photo studio for capturing dressed up and posed avatars in front of a simple background. This helps the artists to create static images for various purposes such as a background or a showcase picture of clothes. In about a few clicks a dressed and posed avatar can be captured by the tool. Additionally, the modularity of the ACE and the core system of the game allowed for new features, such as rarities of colors to be added with ease on top of the already existing data.

The ACE has most of the features implemented which the game developer team was wishing for. However, there are still some feature that could be implemented in the future to make the tool a better piece of technology. Among many of these functionalities is the ability to manipulate the Animator component of Unity to allow the editing of the animation state machine to ease the creation of gestures. Another one of these features is a statistical report on the products on key points such as gender or rarity distribution in certain shops or the ratio of cheap and expensive clothes in certain stores to name a few examples.

During the implementation of the project several issues arose. One of the main issues was to come up with the way the ACE and the servers are going to communicate. Designing these methods and implementing them once was not enough. After the third design and implementation the network connection was working the way we planned. The first design included sending items first in the same message to the server for syncing. However, this was changed to separate messages and then merged back again. Also batching of this information was introduced in the beginning which was later not used.

Currently, the ACE is being used in the release cycle of the game as a replacement of the old Cloth Editor. With the help of the tool the arts team can develop and release clothes faster and easier than before. Also it provides an easy to browse platform for the balancers who decide which shop sells which products and for how much. Now, the ACE is fulfilling its purpose of being a useful and quick solution for product creation and deployment.

6 Conclusion

The goal of this project was to create a replacement tool for the Clothes Editor. During the project the Avatar Content Editor was developed. The ACE features many graphical features to aid the development process of avatar content in Project X. With the use of this new tool, artists and designers of the game can do their job faster and with more ease.

During the first phase of the project I was learning many important Unity developer skills as well as made a rudimentary UI sketch as a reference for later. This sketch was used and mutated during the project into the current UI. During the following months I implemented the basic functionalities of the tool such as the item browser and the attributes view. After these features I coded the tool so it would work with all of the product types and also managed to communicate with the servers for data. However, the first code review of the ACE revealed that the structure I coded was not ideal, so I spent a week rewriting the class structure of the tool. This being done I continued to polish the ACE and fix several bugs. Also many small features made it into the final version, such as screen capturing and live texture reloading on file changes. In the end, the tool finally worked as intended. After a meeting and a discussion, the team came up with the best strategy to use the tool in their release cycle.

During the project I acquired many skills relevant in the game development field and was aided by professionals. Even though the project was carried out on time, more time would be required to make the project polished and work flawlessly. In the end the project could be improved with multiple user editing and more connectivity to the actual animations while editing gestures. Finally, the project was a success and generally improved the workflow of the art and design team.

References

1. Company Facts. [Online]. San Francisco, USA: Unity Technologies. URL: https://unity3d.com/public-relations. Accessed 8 July 2016.
2. Gregory J. Game Engine Architecture. Boca Raton, USA: CRC Press; 2009
3. Sloan R J S. Virtual Character Design. Boca Raton, USA: CRC Press; 2015
4. First Look at the Wii U Mii Studio. [Online]. USA: Player Essence.com; 15 November 2012. URL: http://playeressence.com/first-look-at-the-wii-u-mii-studio-screenshots/. Accessed 8 July 2016.
5. Rees A. Here's What Happens When You Play Kendall and Kylie Jenner's New App, Which Is Delightful. [Online]. USA: Cosmopolitan, Hearst Communications, Inc.; 17 February 2016. URL: http://www.cosmopolitan.com/entertainment/celebs/news/g5425/kendall-kylie-jenner-app-game-review-walkthrough/. Accessed 8 July 2016.
6. Tadres A. Extending Unity with Editor Scripting. Birmingham, UK: Packt Publishing Ltd.; 2015.
7. Smith M, Queiroz C. Unity 4.x Cookbook. Birmingham, UK: Packt Publishing Ltd.; 2013.
8. Henson C, Henson R. Unity 4.x Game Development by Example Beginner's Guide. Birmingham, UK: Packt Publishing Ltd; 2013.
9. Navarro A, Pradilla JV, Rio O. Open Source 3D Game Engines for Serious Games Modelling. Alexandru C, editor. Rijeka, Croatia: InTech; 2012.
10. Sweeney T. Build for VR in VR. [Online]. NC USA, Cary: Epic Games Inc.; 4 February 2016. URL: http://www.unrealengine.com/blog/build-for-vr-in-vr/. Accessed 15 July 2016.

11. Masalcev D, Goryachev A. Soviet Kitchen in Unreal Engine 4. [Online]. USA, 80 level; 1 February 2016. URL: http://80.lv/articles/scene-creation-in-unreal-engine-4/. Accessed 15 July 2016.
12. Haas J. A History of the Unity Game Engine. Worcester, UK: Worcester Polytechnic Institute; 2014.

Title of the Appendix

Content of the appendix is placed here.

Title of the Appendix

Content of the appendix is placed here.