User-Friendly Personality Animator on the Web

Pablo Bendiksen Gutierrez and Funda Durupinar Babur
{p.bendiksen001@umb.edu},{Funda.DurupinarBabur@umb.edu}
University of Massachusetts Boston

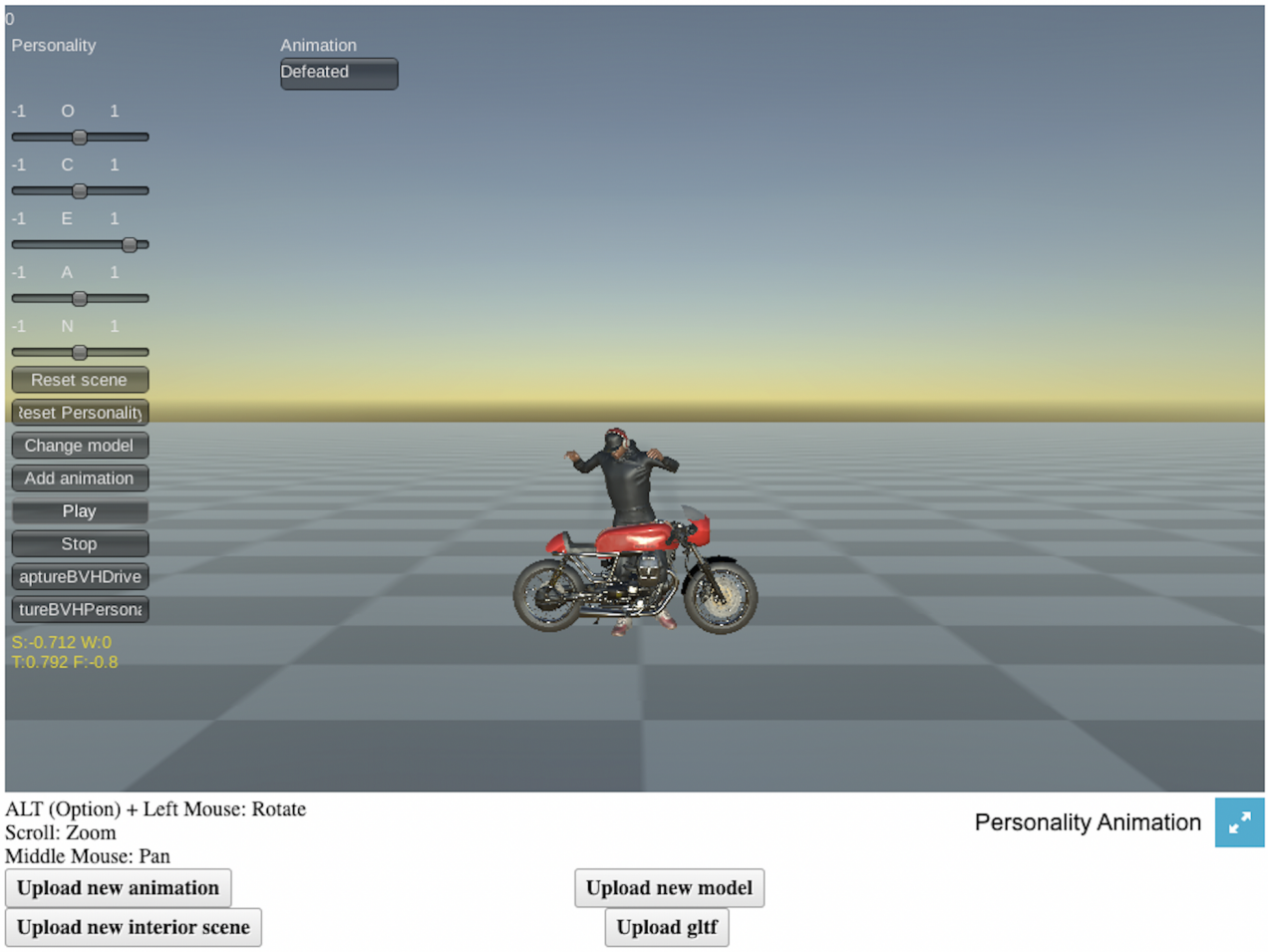

Figure 1: A Unity WebGL Build.

ABSTRACT

Unity's WebGL Build option allows for JavaScript based applications to interface with Unity via a web browser that leverages the WebGL renderer to display Unity content to the client. This project extended prior research into personality animation by exposing a high level personality animator tool to user selected, dynamically loaded, 3D models and animation clips on the web. Moreover, glTF files (an open source format) for 3D models and assets are also supported for upload and rendering.

CS460, Fall 2022, Boston, MA

ACM ISBN 1337.
https://CS460.org

KEYWORDS

WebGL, Visualization, Unity, Animation

ACM Reference Format:
Pablo Bendiksen Gutierrez and Funda Durupinar Babur. 2022. User-Friendly Personality Animator on the Web. In *CS460: Computer Graphics at UMass Boston, Fall 2022*. Boston, MA, USA, 7 pages. https://CS460.org

1 INTRODUCTION

Imbuing personality to motion data for subsequent rendering was a focus of Professor Funda Durupinar as a post doctoral researcher. This was achieved as a high level tool that maps low level motion parameters to real number values across five dimensions of personality factors [1]. Exposing this tool as a browser based application, with additional interactive functionalities for users, naturally arose as possible next steps. The motivation being to increase visibility of the personality animator——allowing users to stylize humanoid animation with personality interactively, while allowing users the freedom of uploading elements that could be associated with a scene. Giving users the ability to upload humanoid models and animations would make personality rendering highly customizable, while allowing users to upload other elements would constitute first steps towards user-specified virtual environments. Of course, supporting open source file formats for upload would only increase the usability of the application. We thusly present a remotely hosted browser based personality animator application with functionality for personality stylization of user uploaded humanoid models and animations, as well as for user upload of other 3D models, animations, assets, and even scenes. Humanoid models and animations used for personality stylization must be uploaded as asset bundles——Unity's proprietary compression format, while anything not to be stylized can be uploaded as a GLB file.

2 RELATED WORK

Durupinar et al. (2015) developed an empirically grounded human motion generation system that can procedurally animate virtual human characters with personality [1]. This was achieved by adding an intermediary between the association of a motion to a personality. Using the guiding theory of Laban Movement Analysis via its effort component which characterizes human motion based on inner attitudes across the elements of space, weight, time and flow, a motion can be mapped to 4 effort dimension values. Since efforts can give insight into personality, these effort dimension values in turn can be mapped to a personality, as defined by a tuple of five real number values each corresponding to one of the OCEAN factors of personality. The efforts to personality associations stemmed from a user study, where subsequent data analysis determined individual effort contribution across personality dimensions, and the efforts to motion parameters mapping stemmed from a motion expert study and multivariate linear regression. The combination of the two allows for personality driven motion synthesis which represents the bedrock capabilities of this project.

3 METHOD

The base project for this endeavor already carried gigabytes of content and about 20 c/ files relating to a few Unity scenes. The first focus was to identify files associated with adding or changing game objects to a scene; files relating to this were *animationInfo.cs*, *guiController.cs*, *paramGUI.cs*, and *componentInitializer.cs*. The next focus involved identifying and modifying scripts associated with browser presentation/interactivity and browser interfacing with the Unity Engine API. The corresponding scripts were *index.htm*, *app.js*, *style.css*, and *animationInfo.cs*. The final focus was to iteratively test script changes across Unity WebGL build attempts.

3.1 Implementation

Project implementation first demanded knowledge of manual operations within Unity IDE. This included scene creation, importing and associating 3D models and animations, adding game objects to scenes and components to game objects, and creating asset bundles. It was then important to see how avatars and their associated models could be added to the project scene with the Unity Engine API. Finally it was necessary to find a means to pass content from the browser into the Unity build project that extended beyond asset bundles.

3.2 Milestones

Project implementation was divided across 4 milestones to be completed sequentially.

3.2.1 Milestone 1. Incorporate imported humanoid model and animation into project scene both via Unity IDE and Scripting API.

Associating imported 3D models and animations with the project scene (i.e., *MotionSelection*) involved familiarity with the project scene hierarchy and the *Agent* game object. We demonstrate an example of this process in Figure 2. In this example a 3D model named *The Boss* was taken from mixamo.com and a walking animation was taken from the Carnegie Mellon University graphics lab motion capture database (http://mocap.cs.cmu.edu/), both in FBX format. The model was prepped——textures added, avatar created, and mapping between skeleton hierarchy and Unity generic bones ensured——and added to the scene graph under the agent node, with an instance of the Agent game object added as it's child. The motion capture file was imported as an animation clip titled *136_25* (named after the corresponding subject number and trial number) and was included within Agent as a new element under it's Animation Clips component.
The Agent game object further has the *animationInfo* and *componentInitializer* scripts included as components. Figure 3 shows the ChangeModel method of the animationInfo class that makes changing models possible via the Scripting API. The 3D model is passed as type Unity GameObject, instantiated, and set as the child of agent, as well as the parent of Agent, within the *MotionSelection* scene. Agent's *ComponentInitializer* Reset method then gets invoked: this ensures that the passed in Model contains a valid avatar and maps the model's bones to Unity's generic bones. The former model is deactivated and the current model replaces the former model within the scene graph.

3.2.2 Milestone 2. Generate a complete scene and incorporate it into the project scene.

Taking a complete scene offered for free under the Unity asset store——*Simple Garage*, the Garage Scene was created with the IDE.

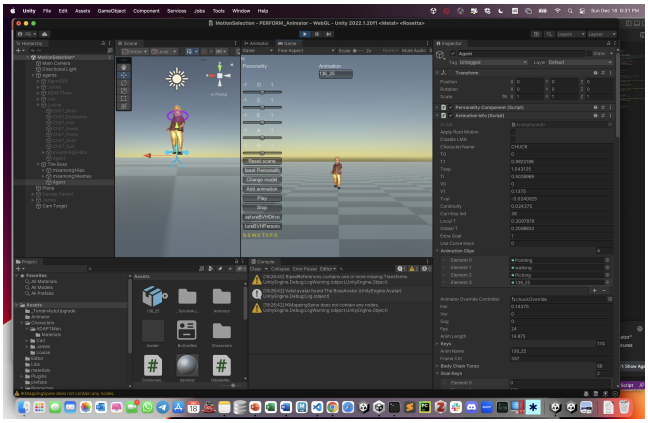

Figure 2: An example 3D model and animation clip incorporated into the Personality Animator project.

```
//ChangeModel associated with 'agent' game object as AnimationInfo is component therein
public void ChangeModel(GameObject prefab)
{
    // prefab is like a pre-built template of the game object that we then make a copy of
    GameObject newModel = Instantiate(prefab, new Vector3(0f, 0f, 0f), Quaternion.identity);

    GameObject currModel = transform.parent.gameObject;


    transform.parent = newModel.transform;
    _animator = transform.parent.GetComponent<Animator>();

    _animationClip = GetClip();
    // i.e., 'Agents'
    transform.parent.parent = GameObject.Find("agents").transform;
    currModel.SetActive(false);


    //reset components
    GetComponent<ComponentInitializer>().Reset();

    _animator.runtimeAnimatorController = _animatorOverrideController; //componentinitializer sets them back to reference controller
    _animator.Play("Idle", 0, 0); //should be called after setting a runtimeanimatorcontroller
    _animator.speed = 0f;


    GetComponent<TorsoController>().Reset();
    _torso = GetComponent<TorsoController>();
    Reset(AnimName);
    GetComponent<FlourishAnimator>().Reset();
    GetComponent<IKAnimator>().Reset();

}
```

Figure 3: Method leveraging Unity Scripting API to display new 3D model.

To test interoperability a model as well as Agent child objects included into the scene. The created Garage game object was then included as a prefab that could be added to the *MotionSelection* scene (figure 4).

Figure 4: Newly created Scene.

The created Garage game object was then included as a prefab that could be added to the *MotionSelection* scene (figure 5).

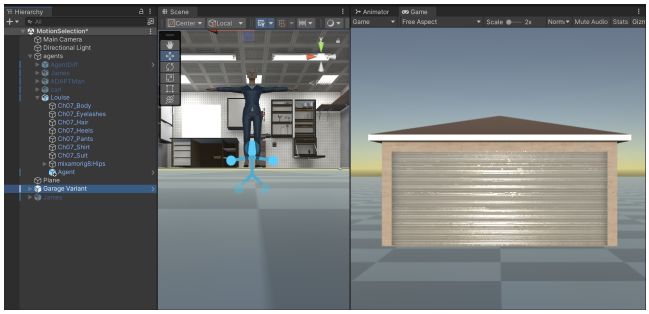

Figure 5: Garage Scene incorporated into Project Scene.

3.2.3 Milestone 3. Extend browser js interfacing with Unity to allow for multiple forms of user interactivity.

Initializing a unityInstance object is essential to running a Unity build as the runtime depends on it; this object houses the entire build module. As such it was important to first reduce the scope in which this object was accessible, despite the fact that online sources constantly demonstrated the use of a unityInstance object with global scope. Initially, a javascript globally accessible unityInstance object was accessed from another file and it's SendMessage() method invoked (figure 6). A boolean flag passed as an argument allowed for two distinct calls of the method, each only varying in their second argument: a string representation of another method to be called. The first argument references a game object that must have as a component a script with a method name that corresponds to the second argument. Recall the Agent game object and its animationInfo script component. Finally, the third argument constitutes the argument passed to the method referenced by the second argument.

```
<div id="unity-message">
    <label class="anim-file-input" for="anim-file-selector"></label>
        <input type="file" id="anim-file-selector" onchange="readAndSend(this, true)"style="visibility: hidden">
    <label class="model-file-input" for="model-file-selector"></label>
    <input type="file" id="model-file-selector" onchange="readAndSend(this,false)"style="visibility: hidden">
```

```
var script = document.createElement("script");
script.src = loaderUrl;
script.onload = () => {
    createUnityInstance(canvas, config, (progress) => {
        progressBarFull.style.width = 100 * progress + "%";
    }).then((unityInstance) => {
        myGameInstance = unityInstance;
        loadingBar.style.display = "none";
        fullscreenButton.onclick = () => {
            unityInstance.SetFullscreen(1);
        };
    }).catch((message) => {
        alert(message);
    });
};
document.body.appendChild(script);</script>
```

```
function readAndSend(event, isAnimFile) {
    console.log(event.files);
    const fileList = event.files;

    if (isAnimFile)
        myGameInstance.SendMessage("Agent", "ReceiveAnimationFromPage", URL.createObjectURL(fileList[0]));
    else
        myGameInstance.SendMessage("Agent", "ReceiveModelFromPage", URL.createObjectURL(fileList[0]));

}
```

Figure 6: Global Unity Instance and it's functionality for passing messages by their URL.

The use of SendMessage() exemplifies automated JavaScript interfacing with c sharp methods within a Unity build. Figure 7 demonstrates how, for either method invoked internally by the Unity API, the passed URL gets handled through a coroutine. Here we note the significance of the UnityWebRequestAssetBundle.GetAssetBundle() and DownloadHandlerAssetBundle.GetContent() methods. The first makes a web request asynchronously, and the second unpacks the response, assuming it was successful. In either case, only unity assetbundles are supported. Even though sending messages to the build from Javascript allows users to dynamically load assets on demand, assetbundles must be built before runtime; we therefore lacked a way for users to upload files for rendering that hadn't apriori been bundled into assetbundles.

```
void ReceiveAnimationFromPage(string url) {

    StartCoroutine(GetBundle(url, true));
}

0 references
void ReceiveModelFromPage(string url) {
    StartCoroutine(GetBundle(url, false));
}

    2 references
    IEnumerator GetBundle(string url, bool isAnim)
{
    using (UnityWebRequest uwr = UnityWebRequestAssetBundle.GetAssetBundle(url))
    {
        yield return uwr.SendWebRequest();

        if (uwr.result != UnityWebRequest.Result.Success)
        {
            Message = uwr.error;
        }
        else
        {
            // Get downloaded asset bundle
            AssetBundle bundle = DownloadHandlerAssetBundle.GetContent(uwr);

            if (isAnim)
            {
                var animations = bundle.LoadAllAssets(typeof(AnimationClip));


                int n = 0;
                foreach (var animFile in animations)
                {
                    AnimationClips.Add((AnimationClip)animFile);
                    Message = n.ToString(); //TODO for debug: delete
                    n++;
                }
            }
            else
            {

                var gameObjects = bundle.LoadAllAssets(typeof(GameObject));
                int n = 0;
                foreach (var go in gameObjects) {
                    ChangeModel((GameObject)go);
                    Message = n.ToString(); //TODO for debug: delete
                    n++;
                }
            }
```

Figure 7: Unity assetbundle restricted message passing.

Before addressing the assetbundle limitation, code was first added to allow for more than two interactive browser components while unfettering the global space. UnityInstance was instantiated with local scope in the body of onload(), and event listeners for each of the declared html elements that corresponded to interactive browser buttons, with a callback that ensured to pass both the unityInstance as well as the corresponding method within Unity build (in string format) as arguments (figure 8).

3.2.4 Milestone 4. Extend browser js interfacing with Unity to allow for open source file contents to be passed and rendered.

```
// Calling Unity script method from JS
// safely passes unity game instance and unity script method (must be attached to referenced object in scene)
function passGameInstance(event, gameInstance, string_method) {
    console.log('inside passGameInstance');
    readAndSend(event, gameInstance, string_method);
}

script.onload = () => {
    var localUnityInstance;
    // createUnityInstance(canvas, config, onProgress).then(onSuccess).catch(onError);
    // creates a new instance of the content; canvas object renders game, config contains the build configuration
    createUnityInstance(canvas, config, (progress) => {
        progressBarFull.style.width = 100 * progress + "%";
    }).then((unityInstance) => {
        localUnityInstance = unityInstance;
        loadingBar.style.display = "none";
        fullscreenButton.onclick = () => {
            unityInstance.SetFullscreen(1);
        };
    }).catch((message) => {
        alert(message);
    });
    // used event listeners to ensure unity game instance object has limited scope
    document.getElementById("anim-file-selector").addEventListener("change", function() { passGameInstance(this, localUnityInstance, "ReceiveAnimationFromPage"); });
    document.getElementById("model-file-selector").addEventListener("change", function() { passGameInstance(this, localUnityInstance, "ReceiveModelFromPage"); });
    document.getElementById("interior-scene-file-selector").addEventListener("change", function() { passGameInstance(this, localUnityInstance, "ReceiveSceneFromPage"); });
    document.getElementById("gltf-file-selector").addEventListener("change", function() { passGameInstance(this, localUnityInstance, "ReceiveGLTF"); });
};
```

```
// send message to build (Unity script method bound to game object) from browser's JS
// can dynamically load assets on demand as user interacts with content, but assetbundles must be built before runtime
function readAndSend(event, gameInstance, string_method) {
    console.log('readAndSend' + event.files);
    const fileList = event.files;
    gameInstance.SendMessage("Agent",string_method, URL.createObjectURL(fileList[0]));

    }
```

Figure 8: Clearing global space and increasing user interactivity.

The next objective was to determine a means by which 3D models in glTF format could be loaded during runtime. Three possible sources were considered: KhronosGroup/UnityGLTF, Unity Asset Store's Piglet glTF Importer, and Siccity/GLTFUtility. The first however was divided across two libraries that had to be built separately. Moreover, steps for updating the Unity license or version were not straightforward or clear. The second allowed importing glTF models using URLs and even file paths, but it had to be purchased. Not only does this cast doubts as to its utility for a webGL build, as its source code may be shielded from us, but it doesn't abide by the open source philosophy. We settled on the Siccity/GLTFUtility as it's source code is freely available and as it appeared easier to install than the first option. This process was difficult to undergo given unexpected build, browser display, and runtime errors: though GLTFUtility could be quickly applied in developer mode (see figure 9), it led to unexpected build errors.
These errors cascaded from the first error stating that build.js, which comes from a folder that gets automatically generated during the build process, contained an undefined symbol: DecodeDracoMesh:

```
Building Library/Bee/artifacts/WebGL/build/debug_WebGL
_wasm/build.js
failed with output: error: undefined symbol:
DecodeDracoMesh (referenced by top-level compiled
C/C++ code)
warning: Link with `-s LLD_REPORT_UNDEFINED` to get more
information on undefined symbols
warning: To disable errors for undefined symbols use `-s
ERROR_ON_UNDEFINED_SYMBOLS=0`
warning: _DecodeDracoMesh may need to be added to
EXPORTED_FUNCTIONS if it arrives from a system library
error: undefined symbol: GetAttribute (referenced by
top-level compiled C/C++ code)
```

Solving this problem was considerable and is detailed under the Challenges section.

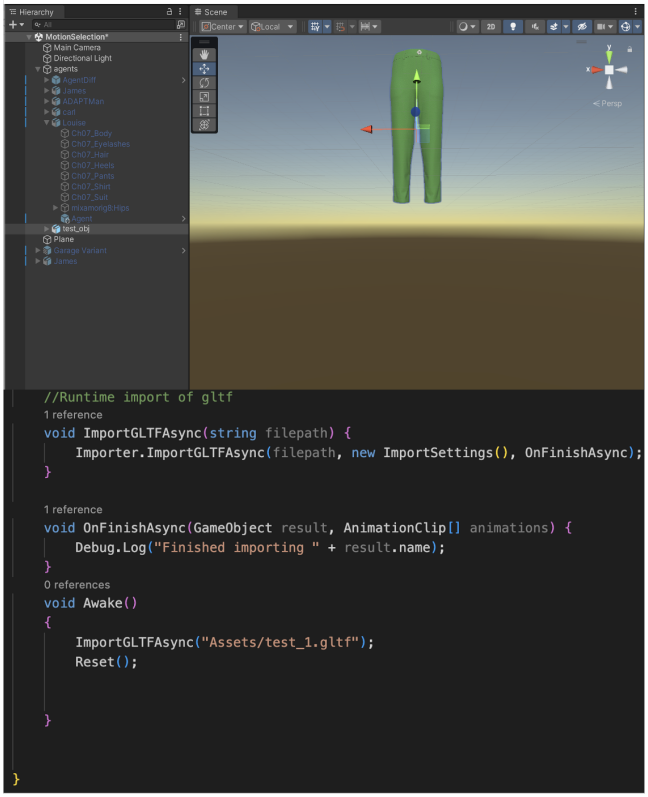

Figure 9: Example gLTF model loaded into Unity Developer Mode.

3.3 Challenges

Describe the challenges you faced.

- Challenge 1: Learning and applying concepts within Unity IDE

  As a first time user of Unity, familiarity with the software's interface elements and their relations took time to develop. Related to this was the need to understand concepts that are essential to game design and graphics at large. All of this was a pre-requisite to anything else project related.

- Challenge 2: Relating learned concepts between Unity IDE and Unity API

  It was important that effects observed on the IDE be mirrored on the programmatic front. We provided a 'change model' example in this paper, but this coupling was also needed for manipulations of the scene graph and any given child object's components, as well as the loading of animation files, large game objects that nested other objects, and gLTF files.

- Challenge 3: Importing content into Unity project during runtime.

  Loading content into Unity Build during runtime was first restricted to Unity assetbundles (indeed an achievement in itself), as previously mentioned. Across numerous trial-and-error attempts, a few key steps were determined that achieved the install and use of Siccity's GLTFUtility in the context of a WebGL build. First we added four GLTFUtility-specific shaders into Unity's "Always Included Shaders" section of the Project settings. These shaders had to be found within the plugin folder and they were as follows: Standard Transparent (Metallic), Standard (Metallic), Standard Transparent (Specular), and Standard (Specular).

  Next, we removed the version of Draco and installed an updated version as it was clear the errors (at least in part) stemmed from the Draco mesh compression tool. When this did not solve the problem a search online revealed a GLTFUtility pull request by github username rasoulMrz, claiming that aside from the updated Draco version being necessary for WebGL builds on Unity 2021 and 2022, GLTFUtilityDracoLoader.cs had to be adapted to the new Draco API. In other words, the GLTFUtility was attempting to use methods whose signature had changed. Simply swapping the plugin folder for the one with the adapted version of GLTFUtility-DracoLoader.cs was unsuccessful; Unity complained about unsafe code use, and changing Unity's handling of unsafe code as permissible, either programmatically or through the IDE, could resolve the problem. The solution was to find and swap the GLTFUtilityDracoLoader.cs file, after having deleted the old Draco plugin folder from source and installed the new version. Though project builds could be completed now, running the browser application quickly informed us that the project would not run (figure 10).

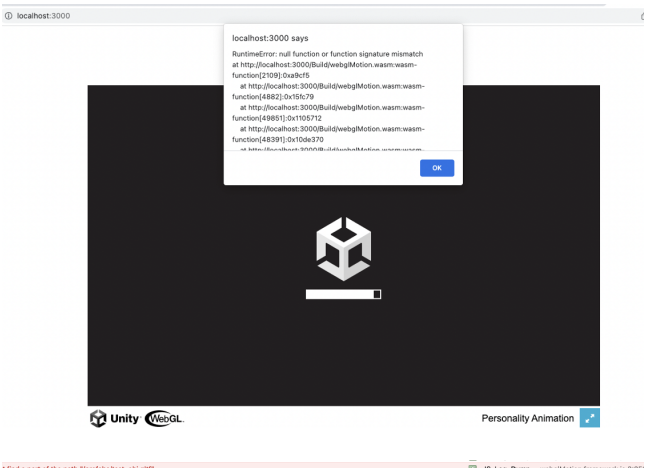

Figure 10: Runtime Errors after diagnosing GLTFUtility issues

A workaround here was to enable error exceptions, within Unity's settings for WebGL, as "Full Without Stacktrace",

effectively setting the most liberal error exception policy. Finally the build both compiled and ran; yet, the browser no longer revealed the interactive buttons defined in index.html and styled in style.css (figure 11), even after clearing browser cache. We found this very strange as, during this process, no changes were made to any scripts associated with the browser presentation and functionality/communication.

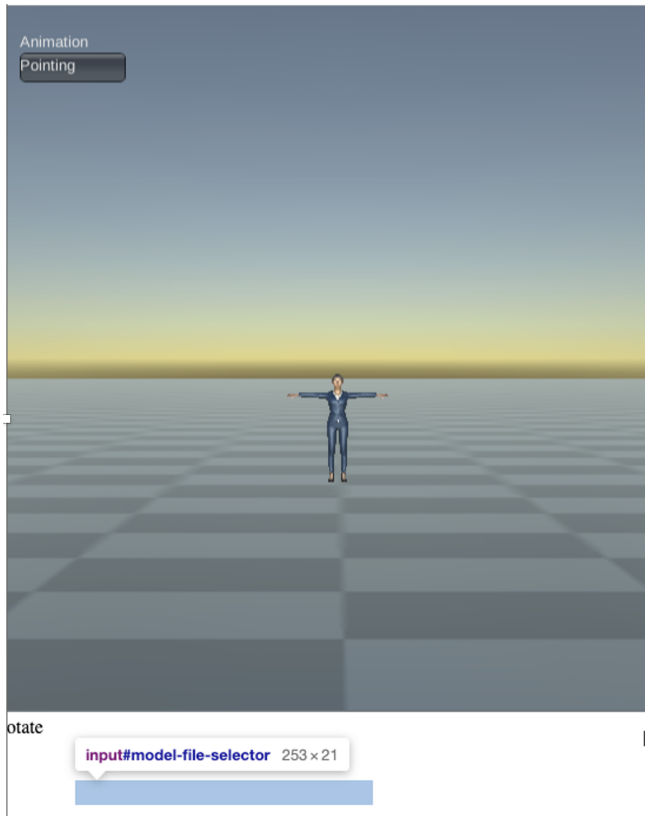

Figure 11: html button elements mysteriously disappear after laborious GLTFUtility debugging

Changing browsers from Chrome to Safari solved this problem. Curiously, returning to Chrome in the future, after resetting Unity's error exception policy, no longer repeated the problem. Regardless, the GlTF import to Unity Build problem remained unsolved; passing glTF files by means of a URL is not supported by the importer, and 'part of path' not found errors were being reported by the JavaScript console. The solution was to handle a Unity Web response for the passed URL as bytes: first converting the response content to bytes, then using a LoadFromBytes() method of the Importer (figure 12).

- Challenge 4: WebGL build option makes for increasingly slower build times.
  We found the build time for our project to be too high for comfort, with later builds taking even longer than prior builds.

```
void OnFinishAsync(GameObject result, AnimationClip[] animations) {
    Debug.Log("Finished importing " + result.name);
}

0 references
void OnProgressAsync(float progress) {
    Debug.Log("Importing " + progress);
}



2 references
void ImportGLBAsync(byte[] fileContent) {
   // Importer.ImportGLBAsync(fileContent, new ImportSettings(), OnFinishAsync, OnProgressAsync);

    AnimationClip[] animations;
    GameObject go = Importer.LoadFromBytes(fileContent,  new ImportSettings(), out animations);

    Debug.Log("import called");
}

0 references
public void ReceiveGLTF(string url) {


    StartCoroutine(ReadFileAsync(url));

}

2 references
IEnumerator ReadFileAsync(string url) {

    UnityWebRequest www = UnityWebRequest.Get(url);
    yield return www.SendWebRequest();


    Debug.Log(www.downloadHandler.data.Length);

    ImportGLBAsync(www.downloadHandler.data);
}
```

Figure 12: GLTFUtility import solution

There was no immediate solution to this and deployment testing suffered as a result. Any minor change to be tested via the final product lead to minutes of downtime due to project compilation alone.

4 RESULTS

Despite unforeseen challenges and unexplained behavior, we nevertheless achieved our proposed objectives as delineated by our milestones (figure 13). Our browser application allows users to:

a load 3D models, as assetbundles, that automatically replace the prior or default model (e.g., *jamesbundle_2*, *louise_model*, *the_boss*)
b load animation files, as asset bundles, that automatically populate the Animation tab (e.g., *chuckmovement* populates Animation tab with the 136_25 animation file as well as another titled defeated.)
c interact with personality stylizer GUI for any model and animation combination pre-loaded or loaded.
d load interior scene, as asset bundle, that automatically renders in the scene (e.g., *garage_scene*)
e load any gLTF file as a glb format (e.g., *mottoguzzicustom.glb*).

We present our remote application as accessible via the following URL: http://54.250.239.217/

Note that initial loading time is slow, as stated in a message when first accessing the page. This results from the lack of proper header response configuration within the Apache web server that hosts the project and, moreover, from the fact that the project is stored on a free tier Amazon AWS virtual machine instance (t2.micro) that

only carries a single virtual CPU.

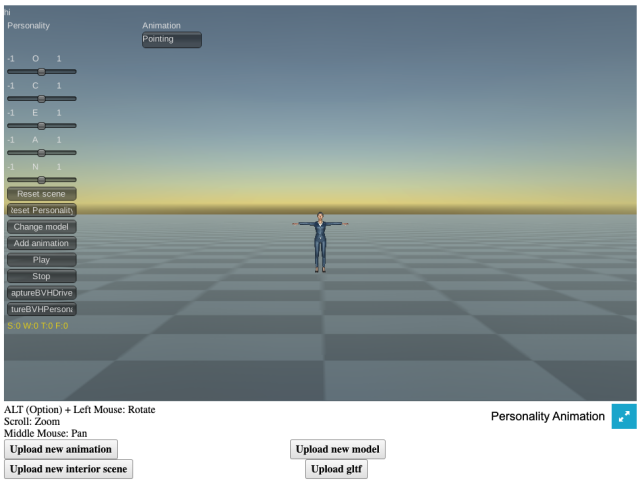

Figure 13: Personality Animator web application

5 CONCLUSION

This project represented an opportunity to learn varied concepts across the technologies of Unity, Unity Scripting API, C#, and Unity WebGL Build, as well as JavaScript, HTML, and CSS for browser side implementation; which itself was divided between client and server side code. Making large projects wieldy via browser based applications is both a growing incentive as well as challenge given the rise of complex programmatic systems and Big Data. We hope our project provides one example of managing complexity and increasing accessibility to make for a user-friendly application.

REFERENCES

[1] Funda Durupinar, Mubbasir Kapadia, Susan Deutsch, Micheal Ellen Neff, and Badler Norman. 2017. PERFORM: Perceptual Approach for Adding OCEAN Personality to Human Motion Using Laban Movement Analysis. *ACM Transactions on Graphics* 36, 1 (2017), 1:16.