Entwickler Dokumentation
–
VMD Vorfahrt App

Philipp Neumann

20. August 2021

Inhaltsverzeichnis

1 Einführung **4**

2 Grundstruktur der App **5**

3 Neue Fahrzeuge hinzufügen **6**
3.1 Neues Fahrzeug-Model einbauen 6
3.1.1 Import des 3D-Modells 6
3.1.2 Import der Texturen 7
3.2 Texte und Bilder hinzufügen 8

4 Interner Programmablauf **9**
4.1 Ablauf im Editor 9
4.2 Aufbau der Scenes 10
4.3 Programmstart 11
4.4 Ladevorgang 12
4.5 Ablauf der *MainMenu*-Scene 14
4.6 Ablauf der *VehicleView*-Scene 15
4.7 Timer 17

5 Public Methods **18**
5.1 *SceneState*-Class 18
5.2 *Vehicle*-Class 19
5.3 *LanguageSwitcher*-Class 20
5.4 *ControlSwitch*-Class 20
5.5 *DisableObject*-Class 20
5.6 *SceneLoader*-Class 21
5.7 *MenuEntry*-Class 21
5.8 *MenuScene*-Class 21
5.9 *ObjectRotation*-Class 22
5.10 *AdvancedRotation*-Class 22
5.11 *SimpleRotation*-Class 22
5.12 *ZoomControl*-Class 22

6 Optimierungs Ideen **23**
6.1 Vollständig Dynamisches Laden 23
6.2 Vermeiden von Hardcoded Delays 24
6.3 Dynamisches Laden der 3D-Modelle 25

6.4 Raycast basierte Animations-Trigger 25
6.5 Import weiterer Animationen 25
6.6 Touchinput beschränken 26

1 Einführung

Im Verkehrsmuseum Dresden läuft eine Dauerausstellung unter dem Titel „Straßenverkehr“. Ein Großteil der Fahrzeuge dieser Ausstellung sind auf einer separaten Empore platziert. Diese ist jedoch für Besucher nur über vereinzelte Zugänge erreichbar und somit sind viele der Fahrzeuge inklusive ihrer Texttafeln nicht sichtbar. Um diesem Zustand entgegenzuwirken, setzt das Verkehrsmuseum im Besucherbereich der Empore zwei Tablets ein, auf denen eine Medienstationsapp installiert ist. Sie dient zur Visualisierung der schwer einsehbaren Bereiche der Ausstellung. Die App wurde ursprünglich von den Studenten Paul Wolff, der die Programmierung übernommen hat, und Johann Ludwig, der für die Erstellung der 3D-Modelle zuständig war, erstellt. Dafür wurde ein Programm auf Basis der Unity-Engine geschrieben. Es bietet eine Ansicht zur Auswahl der Fahrzeuge, Detailansichten sowie weitere Bild- und Textinformationen zum jeweils ausgewählten Fahrzeug. Das Museum stellte zu diesem Zweck eingescannte Archivaufnahmen und die Beschreibungstexte in deutscher und englischer Version zur Verfügung. In dieser Dokumentation wird die Version 2 der Medienstationsapp, die auf Basis von Echtzeitrendering arbeitet, genau erläutert. Das grundlegende Konzept und der Inhalt der Medienstationsappist dabei dem Vorgänger sehr ähnlich allerdings wurde die App und ihr Backend für Version 2 im Rahmen der Bachelor Arbeit von Philipp Neumann noch einmal von Grund auf neu geschrieben.

2 Grundstruktur der App

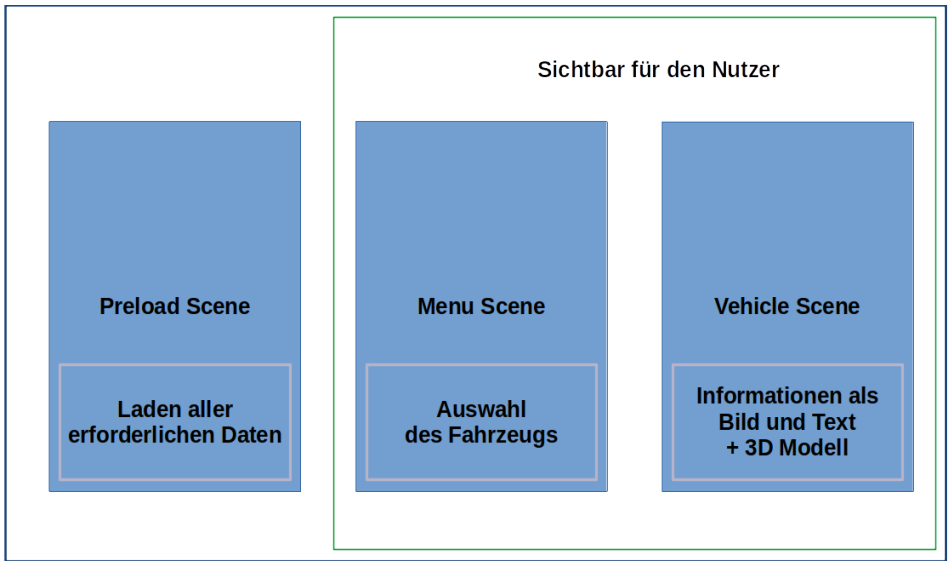

Abbildung 2.1: Grundstruktur und Szenenaufteilung der VMD Vorfahrt App

- *_preload*-Scene
  - alle erforderlichen Daten werden hier beim Start geladen
  - Entscheidung, welche Seite angezeigt werden soll
  - Lädt Bilder und Texte der gewählten Seite aus dem *StreamingAssets*-Verzeichnis
  - 3D-Modelle sind im Build der App hinterlegt, nicht von außen änderbar
- *MainMenu*-Scene
  - Auswahl aller geladenen Fahrzeuge
  - Menüeinträge werden dynamisch beim Start aus Prefab erstellt
  - jedes Titelbild ist ein Button mit Weiterleitung zur *VehicleView*-Scene
- *VehicleView*-Scene
  - inhaltlich Umfangreichste Szene
  - zeigt alle vorab geladenen Text- und Bildinformationen an
  - ermöglicht Rotation des 3D-Models entlang der vertikalen Achse
  - Alle Ansichten lassen sich in eine Vollbildansicht skalieren

3 Neue Fahrzeuge hinzufügen

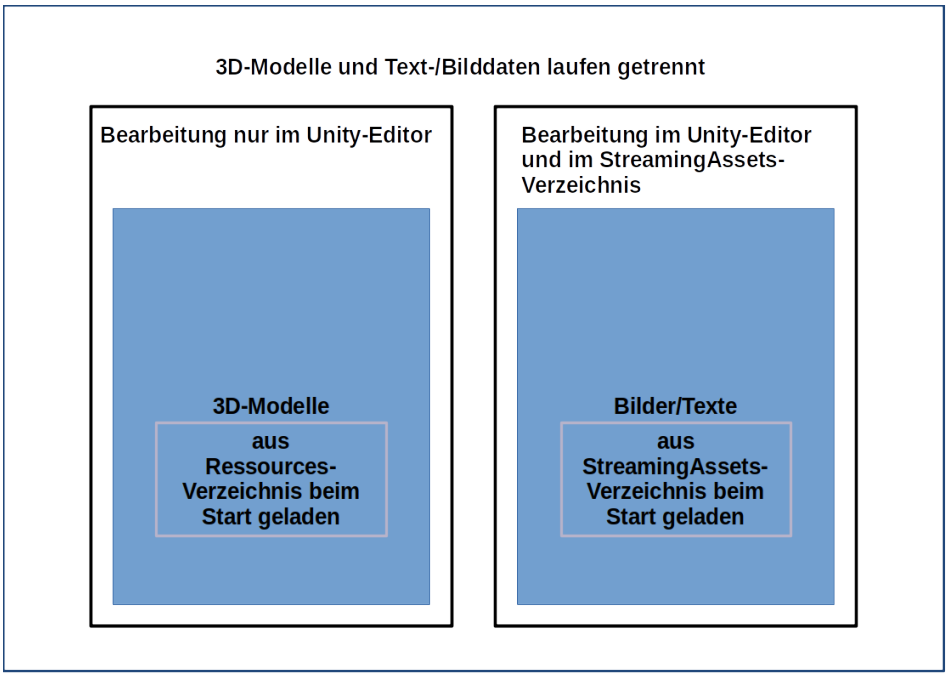

Abbildung 3.1: Trennung der Ladevorgänge nach geladenem Inhalt

3.1 Neues Fahrzeug-Model einbauen

3.1.1 Import des 3D-Modells

1. Model als .fbx Datei per Drag'n'Drop in „Models"-Verzeichnis im Project-Panel laden
2. *VehicleView*-Scene aus Scenes-Verzeichnis per Doppelklick laden
3. In Punkt 1 importierte .fbx Datei aus Models Verzeichnis per Drag'n'Drop in die Hierarchy ziehen und unter den „ModelController" einordnen
4. Im Inspector: Layer auf „Vehicle" stellen, sonst wird es ausgeblendet

3.1.2 Import der Texturen

1. Im Project-Panel im Verzeichnis „Materials“ neuen Ordner erstellen und mit Namen des Fahrzeugs benennen
2. Bilddateien (.tiff) für BaseColor-, Matelness- und Normal-Kanal in erstelltes Verzeichnis importieren
3. neues Material erstellen (RMB ⇒ Create ⇒ Material)
4. Texturen (.tiff) per Drag'n'Drop ins Project Panel laden und dem zuvor erstellen Material zuweisen
5. Material(s) dem geladenen 3D-Model zuweisen
6. Falls transparente Oberflächen (Glass) benötigt werden:
   a) Material kopieren
   b) Im Inspector Surface Type auf Transparent umstellen
   c) Materials mit „_transparent“ bzw. „_opaque“ umbenennen
   d) „_opaque“-Material dem ersten Material Slot des Modells zuweisen und „_transparent“-Material dem zweiten Slot zuweisen
7. Model in der Hierarchy auswählen und Rotations und Translations Werte auf 0 setzten
8. Modell aus Hierarchy per Drag'n'Drop in `/Resources/model_prefabs` zur entsprechenden Seite hinzufügen
9. In Prefab Abfrage „Original Prefab“ auswählen
10. Prefab mit führender Nummer umbenennen, Nummern sind nach Erscheinungsjahr sortiert
11. Nach Erstellung des Prefab geladene 3D-Modell wieder aus der Hirarchy löschen (wird später dynamisch geladen)
12. *_preload*-Sceneaus Scenes-Verzeichnis laden
13. In der Hirarchy „_app“-Object auswählen
14. Im Inspector unter *left-* bzw. *right Vehicles* Size erhöhen und den Namen des neue Fahrzeug-Prefabs aus `/Resources/model_prefabs` entsprechend des Erscheinungsjahres eintragen

3.2 Texte und Bilder hinzufügen

1. Im Verzeichnis `/StreamingAssets/[Seite]/` ein vorhadenes Verzeichnis duplizieren und entsprechend der zuvor genutzten Prefab Bezeichnung umbennen
2. unter `/Text/` englischen und deutschen Text ersetzen dabei **Sytanx einhalten!**
3. unter `/Images/` Gallery-Bilder und *titlePic* ersetzen (Nummerierung der Bilder entspricht der Lade-Reihenfolge)
4. neues *titlePic* rendern mit Hilfe der „base.blend“-Datei (`/blend_files_for_titlePic/scene/base.blend`)
5. Mit *File* ⇒ *Append* Modell aus anderer *.blend*-Datei importieren und entsprechend der vorhandenen Fahrzeuge ausrichten
6. *F12* drücken für Still-Render und über *Image* ⇒ *Save As* als „titlePic.png“ abspeichern
7. *titlePic.png* mit den Gallery-Bildern unter `\Images\` einordnen

4 Interner Programmablauf

4.1 Ablauf im Editor

An dieser Stelle wird der Programmablauf innerhalb des Unity-Editors betrachtet, da er für zukünftige Entwicklungen die größere Rolle spielt. Zudem hat man innerhalb des Unity-Editors die Möglichkeit das Programm von verschiedenen Szenen aus zu starten während im finalen Build (Stand 03.08.21) grundsätzlich immer von der *_preload*-Scene aus gestartet wird. Da die *_preload*-Scene für das Laden sämtlicher Daten zuständig ist, muss diese zwingend zuerst ausgeführt werden, da sonst die externen Inhalte fehlen.

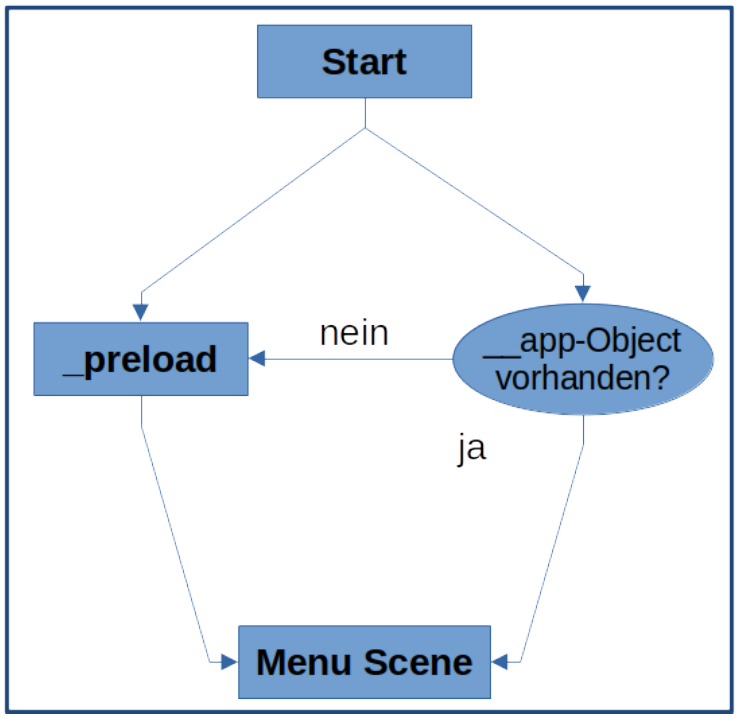

Abbildung 4.1: Ablauf des Programmstarts

4.2 Aufbau der Scenes

Jede Scene verfügt über ein oder mehrere eigene Controller Scripte, das die wichtigsten Funktionen steuern und, falls erforderlich, weitere Scripte aufrufen.

- *_preload*-Scene ⇒ *DataLoader*-Script & *SceneState*-Script
  - *DataLoader*-Script
    - stellt alle erforderlichen Dateipfade zusammen
    - lädt Einstellung aus *settings.json*
    - je nach Einstellung wird linke oder rechte Seite geladen
    - Array für zu ladende Fahrzeugeinträge wird erstellen
    - für jeden Fahrzeug-Array-Eintrag wird eine neue Instanz der *Vehicle*-Class erzeugt und bekommt die jeweils geladenen Fahrzeugdaten zugewiesen
  - *SceneState*-Class
    - Zwischenspeicher für Status-Werte der aktuellen Session
    - gespeichert werden:
      ausgewählte Sprache, gewählte Seite, Index der aktiven Scene und das gerade ausgewählte Fahrzeug
- *MainMenu*-Scene ⇒ *MenuScene*-Script
  - *MenuSceneController*-Object zugeordnet
  - Index für aktuelle Scene setzen
  - Laden aller erforderlichen Elemente (Bild/Text) für die Darstellung des Auswahlmenüs
  - Zuweisung der jeweiligen Button-Funktion
  - Bei Auswahl einer Fahrzeugs ⇒ Index des gewählten Fahrzeugs im *SceneState*-Script aktualisieren
- *VehicleView*-Scene ⇒ *VehicleScene*-Script
  - Index für aktuelle Scene setzen
  - Anzahl der verfügbaren Galerie-Bilder ermitteln
  - alle verfügbaren Text-Informationen zum Fahrzeug einfügen
  - verfügbare Galerie-Bilder laden
  - zugehöriges 3D-Model aus den Prefabs laden
  - alle zusätzlichen Button zunächst ausblenden

4.3 Programmstart

Das Ziel ist es bei jedem Programmstart, zunächst die *_preload*-Scene aufzurufen. Unabhängig davon, von welcher Scene aus gestartet wurde. Aus diesem Grund wird im *MenuScene*-Script eine *Awake*-Methode eingesetzt. Diese Methode läuft direkt nach dem „Erwachen“ des Scripts und damit noch bevor die Inhalt in die Szene geladen werden. Wie in Zeile 4 4.1 zu sehen ist, wird hier die Methode *sceneLoader.CheckPreLoadScene()* zu erst aufgerufen. Diese Methode ist Teil des SceneLoader-Scripts, von dem aus sämtliche Scene-Übergänge gesteuert werden. In diesem Fall wird geprüft, ob in der Hierarchy bereits ein Object mit dem Namen „__app“ existiert. Sollte das nicht der Fall sein, wurde die *_preload*-Scene noch nicht geladen entsprechend ruft das SceneLoader-Script in Zeile 5 4.2 die *_preload*-Scene auf.

Listing 4.1: Jede Scene führt als erstes, CheckPreloadScene() aus.

```
1     private void Awake()
2     {
3         //all operations in here depent on DontDestroyOnLoad-Feature
              ()SceneState.cs --> Awake-Methode
4         sceneLoader.CheckPreloadScene();
5         state = FindObjectOfType<SceneState>(); //find state-script
6         loader = FindObjectOfType<DataLoader>(); //find loader-
              script
```

Listing 4.2: Sollte das *__app*-Object in der aktuellen Scene nicht existieren, wird anschließend die Scene „0“ bzw. die *_preload*-Scene geladen.

```
1     public void CheckPreloadScene()
2     {
3         GameObject check = GameObject.Find("__app");
4         if (check == null)
5         { UnityEngine.SceneManagement.SceneManager.LoadScene(0); }
6     }
```

4.4 Ladevorgang

Das zuvor erwähnte „__app“-Object beinhaltet das *DataLoader*-Script und das *SceneState*-Script, beide werden während des Starts der *_preload*-Scene ausgeführt. Das *DataLoader*-Script lädt innerhalb seiner *Awake*-Methode zunächst die Settings-Datei (Zeile 5) 4.3 und erzeugt anschließend ein Array der zu ladenden Fahrzeuge (Zeile 7). Im nächsten Schritt wird zusätzlich eine Liste der 3D-Modelle erstellt (Zeile 8). Um einen Error bei Fehlenden Daten beim Start zu vermeiden, wird anschließend noch geprüft, ob mit den geladenen Daten überhaupt eine Fahrzeugliste erstellt werden kann (Zeile 9). Die Bilder werden zunächst unsortiert geladen und erst nachträglich sortiert. Für jeden Ladevorgang kommt eine Methode vom Typ *IEnumerator* zum Einsatz, die die Verwendung des *yield return*-Befehls ermöglicht. Auf diese Weise können die Bilder über mehrere CPU-Threads verteilt und damit schneller geladen werden. Der Nachteil besteht darin, dass die Reihenfolge an dieser Stelle nicht erhalten bleibt und nachträglich wiederhergestellt werden muss. Die *Start*-Methode (Wird direkt nach der *Awake*-Methode ausgeführt) startet deshalb als erstes einen Sortieralgorithmus, um die unsortiert geladenen Galeriebilder aufsteigend nummeriert neu zu ordnen (Zeile 5) 4.5. Danach wird eine kurze Ladepause ausgeführt. An dieser Stelle wäre es deutlich sinnvoller eine Methode zu verwenden, die auf das tatsächliche Ende einer *yield return*-Befehls warten kann. Leider war diesem Zeitpunkt (Stand 04.08.21) keine Methode bekannt, die diese Funktion mitbringen würde. Zum Schluss wird das SceneLoader-Script aufgerufen und die nächste Scene, hier die *MainMenu*-Scene geladen (Zeile 11).

Listing 4.3: Im *DataLoader*-Script werden zunächst Arrays und Listen für die zu ladenenden Fahrzeugdaten erstellt und anschließend für jedes Fahrzeug instanziiert.

```
1     private void Awake()
2     {
3         uwrLocalPath = "file://";
4         LoadStreamingAssetsDir();
5         LoadSettingsFile();
6         LoadSide();
7         CreateVehicleArray();
8         CreateModelList();
9         if (availableVehicles.Length != 0)
10        {
11            LoopThroughAvailableVehicles();
12        }
```

Listing 4.4: Durch den *yield return*-Befehl kann der Ladevorgang über mehrere CPU-Threads verteilt werden und läuft damit schneller ab.

```
1     private IEnumerator LoadGalImg(string file, int index)
2     {
3         string searchForMag = magUWRPath + file;
4
5         using (UnityWebRequest uwr = UnityWebRequestTexture.
              GetTexture(uwrLocalPath + searchForMag))
6         {
7             yield return uwr.SendWebRequest();
8             if (uwr.isNetworkError || uwr.isHttpError)
9             {
10                Debug.Log(uwr.error);
11            }
12            else
13            {
14                //ads image files in random order to list
15                vehicles[index].SetGallery(DownloadHandlerTexture.
                      GetContent(uwr), Path.GetFileNameWithoutExtension
                      (file));
16            }
17        }
18    }
```

Listing 4.5: In der *Start*-Methode werden alle Galleriebilderlisten geordnet und nach einer kurzen ladepause in die nächste Scene gewechselt.

```
1     private IEnumerator Start()
2     {
3         for (int i = 0; i <= availableVehicles.Length - 1; i++)
4         {
5             SortGalList(i);
6         }
7
8         if (moveon)
9         {
10            yield return new WaitForSeconds(1);
11            sceneLoader.LoadNextScene();
12        }
13    }
```

4.5 Ablauf der *MainMenu*-Scene

Wie unter 4.2 bereits erwähnt, wird innerhalb der *Start*-Methode zunächst der Scene-Index aktualisiert (Zeile 3) 4.7. Dann werden die Menü-Einträge der Fahrzeuge auf Basis der in 4.4 geladenen Daten erstellt (Zeile 4). Innerhalb der *CreateMenuEntries*-Methode wird dann für jedes geladenen Fahrzeug ein Button mit dem zugehören *titlePic* des jeweiligen Fahrzeugs instanziiert (Zeile 6). Dann erhält der erstellte Button seine zugehörige *Listener*-Funktion, die bei einem Klick ausgeführt werden sollen (Zeile 7). Anschließend werden noch Titel des Fahrzeugs und das jeweilige Erscheinungsjahr zugeordnet. Die Button werden an dieser Stelle auf Basis eines *Prefabs* erzeugt, welches das Layout der Elemente vorgibt. Dieses *Prefab* ist im `/Prefab/`-Verzeichnis unter dem Namen „MenuEntry“ zu finden.

Listing 4.6: In der *Start*-Methode des *MenuScene*-Script wird als erstes der Scene-Index aktualisiert und anschließend werden die Menü-Einträge der Fahrzeuge erstellt.

```
1     private void Start()
2     {
3         state.SetCurScene(); //update scene index
4         CreateMenuEntries();
5     }
```

Listing 4.7: An dieser Stelle wird für jedes geladenen Fahrzeug ein Button in der *MainMenu*-Scene erstellt.

```
1     private void CreateMenuEntries()
2     {
3         menuEntries = new GameObject[loader.vehicles.Length];
4         for (int index = 0; index <= menuEntries.Length - 1; index
              ++)
5         {
6             InstanceButtonsWithImage(index);
7             AddListenerToButton(index);
8             LoadTitle(index);
9             LoadYear(index);
10        }
11    }
```

4.6 Ablauf der *VehicleView*-Scene

Die *Start*-Methode beginnt hier ebenfalls mit einer Aktualisierung des Scene-Index (Zeile 3) 4.8. Dann wird die Anzahl der Galeriebilder ermittelt (Zeile 7) und die Standard-Werte eingetragen (Zeile 8). Anschließend werden die Texte, Bilder und das 3D-Model den jeweiligen Elementen der GUI zugewiesen (Zeile 10-12). Außerdem werden die Booleans für das Starten der UI-Animationen auf den default-Wert *false* gesetzt (Zeile 4-6). Dasselbe gilt für die Button, die nur im Zusammenhang mit animierten UI-Elementen angezeigt werden sollen (Zeile 13, 14). Die UI Animationen arbeiten auf Basis des Verschiebens der Objekte innerhalb der Hierarchy. Das bedeutet, dass immer die Elemente, die im Vordergrund stehen zuletzt gerendert werden bzw. unten in der Hierarchy stehen müssen, da diese von oben nach unten arbeitet. Diese Methode konnte bisher (Stand 04.08.21) noch nicht ausgiebig getestet werden und bringt das Risiko mit sich, dass teilweise nicht sichtbare Objekte über einem Button liegen könnten und ihn somit blockieren. Bisher (Stand 04.08.21) ist dieses Problem nur wenige mal aufgetreten und konnte jeweils auf einen Bug im Code zurückgeführt werden. Eine Besonderheit in der *VehicleView*-Scene stellt das 3D-Model des Phänomen 4RL dar. Es ist bisher (Stand 04.08.21) das einzige 3D-Model, das über Animationen verfügt und benötigt deshalb einige zusätzliche Methoden und Abfragen. So werden beispielsweise die Abfragen zum Triggern der Animationen nur geladen, wenn die linke Seite aktiv ist und der Index des gewählten Fahrzeugs dem Wert in *animModelArrayPos* entspricht. Eine ähnliche Abfrage findet sich außerdem im *ObjectRotation*-Script (Zeile 3) 4.10 um zu prüfen, ob nach dem animierbaren 3D-Modellen gesucht werden soll oder nicht.

Listing 4.8: Die *MainMenu*-Scene weist die Texte, Bilder und das 3D-Modell des ausgewählten Fahzeugs den jweiligen Layout-Elementen zu.

```
1     private void Start()
2     {
3         state.SetCurScene();
4         show3D = false;
5         showGal = false;
6         showText = false;
7         SetSlides();
8         PrepopSlideNum();
9         //CheckVehicleID();
10        InserText();
11        InsertGallery();
12        Insert3DModel();
13        prevSlideBtn.SetActive(showGal);
14        nextSlideBtn.SetActive(showGal);
15    }
```

Listing 4.9: Die Button zum Toggeln der Animationen werden nur gezeigt, wenn das Model des Phänomen 4RL auch ausgewählt ist.

```
            show3D = false;
        }
        else
        {
            show3D = true;
            whiteBG.SetActive(show3D);
            exitBtn3D.SetActive(show3D);
            zoomSlider.SetActive(show3D);
        }

        ReArrangeHirarchy3DView();
        Start3DSceneAnimations();
    }

    private void ReArrangeHirarchy3DView()
    {
        whiteBG.transform.SetAsLastSibling();
        modelTexture.transform.SetAsLastSibling();
        exitBtn3D.transform.SetAsLastSibling();
    }
```

Listing 4.10: Hier wird im *ObjectRotation*-Script geprüft, ob aktuell das Model des Phänomen 4RL geladen ist und somit anch den animierbaren Objekten gesucht werden soll oder nicht.

```
    }
    private void Start()
    {
        if (vehicleCtl.CheckVehicle() == vehicleCtl.
            GetAnimModelArrayPos() && vehicleCtl.CheckSide() == 0) //
            needs cleanup
        {
            engineCover = GameObject.Find("paenomen_engine_cover");
            driverDoor = GameObject.Find("paenomen_driver_door");
            coDriverDoor = GameObject.Find("paenomen_co_driver_door"
                );

            coverAnimator = engineCover.GetComponent<Animator>();
            driverDoorAnimator = driverDoor.GetComponent<Animator>()
                ;
            coDoorAnimator = coDriverDoor.GetComponent<Animator>();
        }
```

4.7 Timer

Die *Timer*-Class hat die Aufgabe die *VehicleView*-Scene in die *MainMenu*-Scene zurückzusetzen unter der Bedingung, dass über eine bestimmte Zeit keinerlei Eingaben erfolgt sind. Der Wert wird in die *Settings*-Datei (`/StreamingAssets/Settings.json`) geschrieben und setzt sich aus dem *invisibleCountdown* und dem *visibleCountdown* Wert zusammen. Der *invisibleCountdown* ist die Anzahl an Sekunden, die vergehen müssen, bevor auf dem Display der Reset-Timer angezeigt wird. sobald diese „nicht sichtbaren" Sekunden abgelaufen sind, werden die verbleiben „sichtbaren Sekunden angezeigt" (Zeile 9ff.) 4.11. Dieser Wert entspricht der Angabe unter *visibleCountdown* in der *Settings.json* Datei. Der Countdown läuft damit im Hintergrund des *Vehicle*-Class ab und wird nur unter den angegebenen Bedingungen sichtbar. Um den Countdown bei jeder Interaktion mit der Oberfläche zurückzusetzen, steht *ResetTimer* als einzige *public methode* dieser Klasse zur Verfügung. Sie wird von jedem Button, jeder Zoom-/Scroll-Bar, den Rotationseingaben in der 3D-Vollbildansicht sowie den 3D-Animationen aufgerufen, um den Timer bei jeder Eingabe zurückzusetzen (Zeile 3) 4.12.

Listing 4.11: Sobald der im Hintergrund laufende *displayTimeValue* den Rückgabewert von *GetStateVisibleCountdown()* unterschritten hat, werden die Zahl des Countdowns auf dem Display angezeigt.

```
1   void Update()
2   {
3       if (displayTimeValue > vehicleScene.GetStateVisibleCountdown
            () + 1)
4       {
5           displayTimeValue -= Time.deltaTime;
6           timerField.transform.SetAsFirstSibling();
7           timerField.SetActive(false);
8       }
9       else if (displayTimeValue > 0)
10      {
11          displayTimeValue -= Time.deltaTime;
12          DisplayTime();
13      }
14      else
15      {
16          displayTimeValue = 0;
17          sceneLoader.LoadMenuScene();
18      }
19  }
```

Listing 4.12: *ResetTimer()* setzt den Timer auf seinen urspünglich geladenen Wert zurück.

```
1   public void ResetTimer()
2   {
3       displayTimeValue = vehicleScene.GetStateInvisibleCountdown()
            + vehicleScene.GetStateVisibleCountdown();
4   }
```

5 Public Methods

In diesem Kapitel werden alle *public methods* der wichtigsten Klassen genannt und ihre Funktion erläutert.

5.1 *SceneState*-Class

public string GetLanguage()
gibt die aktuell gewählte Sprache zurück

public void SetLanguage(string language)
aktualisiert die ausgewählte Sprache

public void SetCurScene()
ermittelt die aktuelle Scene auf Basis des Unitys Scene Index

public int GetSelectedVehicle()
gibt den Listen Index des aktuell ausgewählten Fahrzeugs zurück

public void SetSelectedVehicle(int VehicleNumber)
aktualisiert den Index des ausgewählten Fahrzeugs

public void SetSelectedSide(string jsonData)
setzt die ausgewählte Seite abhängig von settings.json im *StreamingAssets*-Verzeichnis
0 = left | 1 = right

public int GetLoadedSide()
gibt die ausgewählte Seite als Integer zurück
0 = left | 1 = right

public string GetSelectedSide()
gibt die ausgewählte Seite als String (left | right) zurück

5.2 *Vehicle*-Class

public Vehicle(string name)
Constructor der *Vehicle*-Class:
String Parameter wird als „name“-Attribute an die neu erzeugte *Vehicle*-Instanz übergeben
ABER: nicht verwechseln mit dem Titel des Fahrzeugs

public void LoadText(string jsonData)
bekommt einen rohen json-String übergeben und ordnet die json-Attribute den identischen Variablen der Klasse zu

public string GetName()
gibt den Namen des Fahrzeugs zurück
ABER: nicht verwechseln mit dem Titel des Fahrzeugs

public string GetGerTitle()
gibt den vollständigen Titel des Fahrzeugs auf deutsch zurück

public string GetEngTitle()
gibt den vollständigen Titel des Fahrzeugs auf englisch zurück

public string GetYear()
gibt das Erscheinungsjahr des Fahrzeugs zurück

public string GetGerHeader()
gibt den Header (Titel + Jahr kombiniert) des Fahrzeugs auf deutsch zurück

public string GetEngHeader()
gibt den Header (Titel + Jahr kombiniert) des Fahrzeugs auf englisch zurück

public string GetGerPreDescr()
gibt die technischen Eckdaten des Fahrzeugs auf deutsch zurück

public string GetEngPreDescr()
gibt die technischen Eckdaten des Fahrzeugs auf englisch zurück

public string GetGerDescr()
gibt die Beschreibung des Fahrzeugs auf deutsch zurück

public string GetEngDescr()
gibt die Beschreibung des Fahrzeugs auf englisch zurück

public Texture2D GetTitlePic()
gibt das Titelbild des Fahrzeugs als Texture2D zurück

public void SetTitlePic(Texture2D titlePic, string titlePicName)
setzt bzw. überschreibt das Titelbild und dessen Bezeichnung

public void SetGallery(Texture2D loadedGallery, string vehicleName)
fügt der Galeriebilder-Liste einen Eintrag hinzu und weist diesem einen Namen zu

public List<Texture2D> GetGallery()
gibt die Galeriebilder als Liste vom Typ Texture2D zurück

public Texture2D GetGallery(int index)
gibt ein einzelnes Galeriebild auf Basis das übergebenen Index zurück

public void Set3DModel(GameObject vehiclePrefFab)
fügt das 3D-Model des Fahrzeugs der Instanz hinzu

public GameObject Get3DModel()
gibt das 3D-Model als GameObject zurück

5.3 *LanguageSwitcher*-Class

public void SwitchLanguage()
wechselt die aktuelle Sprache von deutsch zu englisch bzw. von englisch zu deutsch

5.4 *ControlSwitch*-Class

public void ControlSwitcher()
wechselt die aktiven Steuerungs-Scripte zwischen der Vehicle-Ansicht und der 3D-Vollbildansicht

5.5 *DisableObject*-Class

public void DisableText()
deaktiviert nicht länger benötigte Objekte der Text-Vollbildansicht
wird in diesem Fall am Ende einer Animation gestartet

public void Disable3D()
deaktiviert nicht länger benötigte Objekte der 3D-Vollbildansicht
wird in diesem Fall am Ende einer Animation gestartet

5.6 *SceneLoader*-Class

public void LoadNextScene()
erhöht den Scene Index um 1 und wechselt damit zur nächsten Scene

public void LoadNextScene(int sceneID)
hier wird der Scene Index direkt übergeben, um eine bestimmte Scene zu laden

public void CheckPreloadScene()
prüft anhand des *__app*-GameObject ob die *_preload*-Scene bereits gestartet wurde oder nicht
nur während der Entwicklung relevant, das fertige Build startet immer von der *_preload*-Scene

public void LoadMenuScene()
lädt direkt in die *MainMenu*-Scene

public void LoadVehicleScene()
lädt direkt in die *VehicleView*-Scene

public void quit()
schließt das Programm

5.7 *MenuEntry*-Class

public void SetYear(string importedYear)
weist dem entsprechenden Textfeld im MenuEntry-Prefab das Erscheinungsjahr zu
wird von der *MainMenu*-Scene 4.5 aufgerufen, um die Einträge zuzuweisen

public void Setname(string importedName)
weist dem entsprechenden Textfeld im MenuEntry-Prefab den Titel zu
wird von der *MainMenu*-Scene 4.5 aufgerufen, um die Einträge zuzuweisen

5.8 *MenuScene*-Class

public void UpdateSaenfte()
aktualisiert den Titel der Sänfte in der *MainMenu*-Scene auf deutsch bzw. englisch

public void UpdatePennyFarth()
aktualisiert den Titel des Hochrads in der *MainMenu*-Scene auf deutsch bzw. englisch

public void SwitchVehicle(int num)
aktualisiert das ausgewählte Fahrzeug im *SceneState*-Class

5.9 *ObjectRotation*-Class

public void ToggleCover()
toggelt die Animation der Motorhaube des Phänomen 4RL Modells

public void ToggleDriverDoor()
toggelt die Animation der Fahrertür des Phänomen 4RL Modells

public void ToggleCoDoor()
toggelt die Animation der Beifahrertür des Phänomen 4RL Modells

5.10 *AdvancedRotation*-Class

public void disableInput()
deaktiviert die *AdvancedRotation*-Class

public void enableInput()
aktiviert die *AdvancedRotation*-Class

5.11 *SimpleRotation*-Class

public void disableInput()
deaktiviert die *SimpleRotation*-Class

public void enableInput()
aktiviert die *SimpleRotation*-Class

5.12 *ZoomControl*-Class

public void SliderZoom(float zoomValue)
verschiebt die Kameraposition entlang ihrer lokalen Z-Achse und den FOV-Wert um den übergebenen Float-Wert
kommt beim Zoom-Slider zum Einsatz

public void ResetCam()
setzt die Kameraposition und den FOV-Wert auf den Standard-Wert zurück

6 Optimierungs Ideen

Hier werden noch einige Optimierungs-Ideen aufgezählt, die während der Entwicklung nicht implementiert werden konnten. Diese Features waren entweder nicht zwingend erforderlich oder hätten zu viel Zeit für eine (aus Nutzer Sicht) geringe Verbesserung in Anspruch genommen.

6.1 Vollständig Dynamisches Laden

Für den Ladeprozess verwendet das *DataLoader*-Script aktuell (Stand 05.08.21) noch eine zuvor vom Entwickler erstellte Liste, um die einzelnen Ordner des *StreamingAssets*-Verzeichnis zu durchsuchen. Hier wäre eine Methode interessant, die es dem Script ermöglicht die Ordner unter `/StreamingAssets/left/`bzw. `/StreamingAssets/rechts/` selbstständig in eine Liste zu laden, ähnlich, wie es in Zeile 3 6.1 zu sehen ist. Die *getFiles()*-Methode der *FileInfo*-Class kann eine Liste gefundener Ordner in einem Verzeichnis zurückgeben. Da diese Methode allerdings eine Systemeigene C#- und keine Unity-Methode ist, steht sie im finalen Build nicht mehr zur Verfügung, wodurch der gesamte Ladevorgang außerhalb des Unity-Editors nicht mehr funktioniert. Es müsste also an dieser Stelle eine Unity-Methode gefunden werden, die diese Funktion ebenfalls mitbringt und als Unity eigene Funktion auch im finalen Build noch funktioniert. Außerdem müsste die Unity-Methode mit dem *UnityWebRequest*-Ladesystem kompatibel sein, da dieses für das Laden aus dem Streaming-Assets Ordner genutzt wird.

Listing 6.1: Die *FileInfo*-Class ist dazu in der Lage Verzeichnisse zu durchsuchen und alle gefunden Ordner als Liste zurückzugeben.

```
1     private void CreateVehicleArray()
2     {
3         FileInfo[] Vehicles = streamingAssetsDir.GetFiles("*.*"); //
              wont work outside editor
4         vehicles = new Vehicle[availableVehicles.Length]; //try to
              adjust to work dynamic like it did in line above
5     }
```

6.2 Vermeiden von Hardcoded Delays

Die Ladevorgänge im *DataLoader*-Script nutzen an vielen Stellen den Befehl *yield return*, um den darauf folgenden Code auf mehrere Threads aufzuteilen (Zeile 7) 6.2. Das hat den Vorteil, dass diese Teile des Codes, besonders das Laden der Bilder, parallel verarbeitet werden kann. Allerdings hat das Script anschließend keine Information mehr darüber, wann der ausgelagerte Ladevorgang abgeschlossen ist. Der aktuell genutzte „quick-fix“ für dieses Problem ist ein hardcoded Delay, das während der *Start*-Methode ausgeführt wird (Zeile 10) 6.3. Das ist aber eher ein Workaround als eine tatsächliche Lösung. Hier bräuchte es eine Funktion, die prüft, ob aktuell noch ausgelagerte Funktionen arbeiten oder alle Ladevorgänge abgeschlossen sind.

Listing 6.2: Code Teile, die auf *yield return* folgen, können parallel verarbeitet werden.

```
1     private IEnumerator LoadGalImg(string file, int index)
2     {
3         string searchForMag = magUWRPath + file;
4
5         using (UnityWebRequest uwr = UnityWebRequestTexture.
              GetTexture(uwrLocalPath + searchForMag))
6         {
7             yield return uwr.SendWebRequest();
8             if (uwr.isNetworkError || uwr.isHttpError)
9             {
10                Debug.Log(uwr.error);
11            }
12            else
13            {
14                //ads image files in random order to list
15                vehicles[index].SetGallery(DownloadHandlerTexture.
                      GetContent(uwr), Path.GetFileNameWithoutExtension
                      (file));
16            }
17        }
18    }
```

Listing 6.3: Das hardcoded Delay fon einer Sekunde stellt vor dem Szenenwechsel sicher, dass alle Ladevorgänge abgeschlossen sind.

```
1     private IEnumerator Start()
2     {
3         for (int i = 0; i <= availableVehicles.Length - 1; i++)
4         {
5             SortGalList(i);
6         }
7
8         if (moveon)
9         {
10            yield return new WaitForSeconds(1);
11            sceneLoader.LoadNextScene();
12        }
```

6.3 Dynamisches Laden der 3D-Modelle

Das *DataLoader*-Script greift aktuell auf die Daten (Texte und Bilder) im *StreamingAssets*-Verzeichnis zu, um diese bei jedem Programmstart neu einzulesen. Somit kann man theoretisch neue Fahrzeug-Einträge erstellen, ohne dafür einen neuen Build erstellen zu müssen. Wie bereits in 6.1 erwähnt, ist das bisher durch die Limitierung im *DataLoader*-Script nicht möglich. Ein weiteres Problem stellt das Laden der 3D-Modelle dar. Die verwendeten .fbx Dateien lassen sich nicht mit Hilfe des *UnityWebRequest*-Systems zur Laufzeit laden. Eine möglich Methode, die zunächst getestet werden müsste, wäre das Laden von .GLTF-Dateien, wie es in diesem Artikel beschrieben ist:
`https://theslidefactory.com/loading-3d-models-from-the-web-at-runtime-in-unity/`.

6.4 Raycast basierte Animations-Trigger

Die hervorgehobenen Teile des *Phänomen 4RL* können mit einem Klick/Berührung animiert werden. Für diese unmittelbare Interaktion kommt ein Raycast-System zum Einsatz. Vom Klick/Berührungspunkt des Displays aus, wird ein Strahl in die Szene geschickt. Sollte dieser Strahl auf ein animierbares Objekt treffen, wird die Animation getriggert und entweder vorwärts oder rückwärts abgespielt. Jedoch funktioniert dieses System derzeit (Stand 17.08.21) nur auf einer 1080p Auflösung präzise. sobald sich die Auflösung ändert, kommt ein Offset hinzu dessen Ursache noch nicht gefunden wurde. Da der Einsatz der App aber zu diesem Zeitpunkt lediglich auf Geräten geplant ist, die über eine 1080p Auflösung verfügen, sollte diese Einschränkung kein Problem darstellen.

6.5 Import weiterer Animationen

Das Phänomen 4RL ist zum jetzigen Stand (05.08.21) das einzige Fahrzeug, das über Animationen verfügt. Diese Animationen werden über Unitys internes Animationssystem gesteuert. Dieses System bringt allerdings für jegliche neue Animationen einen sehr hohen Setup-Overhead mit sich und sollte deshalb für weitere Animationen vermieden werden. Das 3D-Model selbst bringt beim Import in Unity aktuell keinerlei Informationen zu den Animationen mit. Es müsste also innerhalb des 3D-Programms ein Rig und entsprechende Animationen für jedes Fahrzeug erstellt werden. Auf diese Weise könnten die voreingestellten Animationen direkt beim Import in Unity ausgelesen werden und über entsprechende Trigger abgespielt werden.

6.6 Touchinput beschränken

Der Touchinput und sämtlich Scroll-Fields verwenden von Unity aus die gleichen Input Befehle. Das führt dazu, dass beim Hoch-bzw. Runterscrollen eines Textfelds oft auch das 3D-Modell mit hin und her rotiert wird. In der aktuellen Version der Software (Stand 05.08.21) wird dieses Problem teilweise gelöst, in dem die Rotationssteuerung deaktiviert wird, sobald ein Scroll-Field betätigt wird und damit neue Werte liefert. das sorgt allerdings dafür, dass bei stillstehendem Scroll-Field weiterhin durch leichte links-rechts-Bewegungen eine Rotation des Modells getriggert werden kann. Eine deutlich sauberere Variante wäre es, die Touchinputs nur dann an die Rotationssteuerung zu senden, wenn diese innerhalb eines bestimmten Bereichs erfolgen.