Projektbericht – Visualisierung von Messeständen in Virtual Reality

Aleksej Konysev

Studiengang: Mobile Medien
Semester: 8
Matrikel Nr.: 31102

Dozent: Prof. Dr. Jens-Uwe Hahn

Inhaltsverzeichnis

Einleitung

Meine Involvierung bei dem Projekt entstand eher zufällig. Ich arbeite seit 2 Jahren als Werkstudent in der Softwareentwicklung bei einem Unternehmen, welches hauptsächlich CAD-Software entwickelt und bin dort in einem, so genannten, "Co-Working" Büro in Stuttgart tätig.

Dieses Gemeinschaftsbüro nutzen, außer unserer Firma, noch viele andere, unter anderem selbständige oder auch kleinere Unternehmen und "marca gráfica" ist eines dieser Unternehmen.

Sie bieten ihren Kunden an, Messestände zu organisieren und zu planen, dazu gehören auch z.B. 3D-Renderings dieser Stände. Zum Start meines Semesters, kam ihnen die Idee, als zusätzlichen Dienst und zur einfacheren Planung und stärkerem Marketing, den Kunden zu bieten, die Stände auch in Virtual Reality sehen zu können und zu verändern.

Da wir in dem "Co-Working" viel miteinander reden, wussten sie, dass ich schon Erfahrung in Augmented Reality gesammelt hatte und haben mich gefragt ob ich mir vorstellen könnte so ein Projekt zu entwickeln und da kam mir der Gedanke das ganze als Studien-Projekt im Modul Virtual Reality durchzuführen.

Da ich generell großes Interesse, jedoch sehr wenig Erfahrung in VR habe, nutzte ich somit die Chance.

Projektstart

Festsetzen der Requirements

Nachdem das Projekt genehmigt worden ist, haben wir uns direkt in dem ersten Termin an das festlegen der ersten Requirements gesetzt und setzten einige Ziele.

Wie im Laufe der Entwicklung üblich, kamen jedoch später noch einige dazu, nachdem man die ersten Prototypen testen konnte.

Die wichtigsten Requirements habe ich hier aufgelistet:

1. Milestone (Medianight)

Laden von fertigen Messeständen als eine 3D-Datei zur Laufzeit

Einzelne Objekte sollen, durch VR-Controller, frei im Raum verschiebbar sein

Bewegung im Raum durch Tracking der VR-Brille

Teleportation im Raum zur Fortbewegung

Menü zum Laden von anderen Ständen

3D Modell einer Messehalle

Objekte sollen stapelbar sein und sich zusammen bewegen lassen

2. Milestone (Juli)

Menü zum hinzufügen von einzelnen Objekten/Produkten

Gitter zum einfacheren Platzieren von Objekten

3. Milestone (August)

Genaue Kollision von Objekten

Freies verschieben von Objekten in der Höhenachse

Beschaffung der Hardware

Da das Unternehmen sich in der Hardware Branche nicht auskennt und nicht wusste, was sie genau benötigen, habe ich sie beim Kauf des Rechners und der VR-Brille beraten.

Bei der VR-Brille war die Entscheidung schwierig, da jede Ihre Vor- und Nachteile hatte:

Oculus Quest (Paketpreis: 450€)

Pro	Contra
• Kabellos • Kein Aufbau nötig • Günstig → Man kann später immernoch zu anderen Brillen greifen	• Leistung limitiert auf Smartphone → Dadurch eventuelle Änderungen am Workflow mit C4D • Komplexe Szenen nicht darstellbar (realistisches Licht, Modelle mit sehr vielen Polygonen, komplexe Materialien)

HTC Vive Pro (Paketpreis: ~ 1300€ + 350€ Wireless Adapter)

• Möglichkeit mit PC zu interagieren • Wireless Adapter bereits vorhanden	• Teuerste Option (inkl. Wifi) • Wireless hat nur einige Meter Reichweite • Aufbau der Basisstationen nötig • Guter Rechner vorrausgesetzt

Oculus Rift S (Paketpreis: 450€ + evtl. Zukünftiger Wireless Adapter 300-400€)

• Möglichkeit mit PC zu interagieren • Beste Bildqualität • Wesentlich günstiger als Vive Pro • Wahrscheinlich wird in Zukunft ein Wireless Adapter released (für die alte Rift gibt es bereits einen)	• Kabelgebunden (noch) • Guter Rechner vorrausgesetzt

HTC Vive (Paketpreis: ~ 600€ + 350€ Wireless Adapter)

• Möglichkeit mit PC zu interagieren • Wireless Adapter bereits vorhanden • Günstiger als Pro • Upgrade auf Pro möglich, da Basisstationen, Wireless Adapter, Controller nicht ersetzt werden müssen	• Schlechteste Bildqualität • Preis/Leistung im Vergleich zur Rift S schlecht (Rift S immernoch günstiger) • Wireless hat nur einige Meter Reichweite • Aufbau der Basisstationen nötig • Guter Rechner vorrausgesetzt

Die wichtigsten Punkte bei der Auswahl waren: Bewegungsfreiheit (Wireless Adapter), Bildqualität, Aufbaugeschwindigkeit und die Möglichkeit der Interaktion mit einem Rechner.

Leider konnte keiner der Brillen wirklich alle Punkte erfüllen, aber die neue Rift S schien im Punkt Preis-Leistung am besten und ist nur in der Bewegungsfreiheit eingeschränkt, was sich aber in Zukunft durch einen WiFi-Adapter ändern könnte, und erfüllt sonst alle Punkte zufriedenstellend.

Nach etwas Recherche habe ich eine Liste an PC-Komponenten aufgestellt, mit Alternativen, aus denen sie wählen konnten, wobei ich dabei stets auf möglichst hohe Preis-Leistungs Verhältnisse geachtet habe. Nach dem Kauf, habe ich sie auch selbst zusammengebaut.

Wichtigste Merkmale des Rechners:

Mainboard: MSI X470 Gaming

Prozessor: AMD Ryzen5 2600X

Grafikkarte: Radeon RX Vega56

RAM: 16GB DDR4

Preis inklusive aller Teile:

800€

Somit betrug die gesamte Investition für das VR-System nur 1250€.

Entwicklung

Recherche

Da ich bereits viel Erfahrung in der Unity-Engine hatte und mich auch aufgrund der knappen Zeit nicht in eine neue Engine einarbeiten wollte, entschied ich mich Unity weiter zu nutzen, da ich wusste, dass es eine der führenden Engines für Virtual Reality ist.

Ich habe dafür verschiedene Methoden und SDK's ausprobiert und habe mich schließlich erstmal dafür entschieden auf Basis der OpenVR, welches bereits in Unity integriert ist, zu arbeiten. Als im weiteren Verlauf der Entwicklung klar wurde, dass die Rift-S genutzt werden sollte, habe ich zusätzlich die Oculus-VR Integration eingebunden, um noch ein paar extra Features zu ermöglichen.

Dank der OpenVR Basis, ist es aber jederzeit möglich auch ein anderes Gerät zu nutzen.

Import von 3D-Modellen zur Laufzeit

Eine der Hauptaufgabe der Anwendung sollte das Laden von Modellen zur Laufzeit sein. Nach einiger Recherche zu dem Thema, bin ich auf die Open-Source Bibliothek "Assimp" gestoßen, welche die meisten 3D-Datenformate unterstützt.

Um diese in Unity zu nutzen, muss jedoch die .NET Variante der Bibliothek selbst zu einer .DLL bauen.

Nach diesem Schritt war es mir jedoch Möglich schnell die ersten 3D-Daten zu laden. Ein Problem welches hinzukam war, die Materialien und Texturen aus dem "Assimp-Modell" in das "Unity-Modell" zu übersetzen. Dafür war einiges an Tests mit verschiedenen Modellen notwendig, da die Eigenschaften teilweise anders genannt wurden.

```
    mat.SetFloat("_WorkflowMode", 1.0f);
    if (assimpMaterial.Opacity < 1f)
    {
        mat.SetFloat("_Surface", 1.0f);
        mat.SetFloat("_Cull", 0.0f);
    }

    if(assimpMaterial.HasColorDiffuse) mat.SetColor("_BaseColor", new Color(color.R, color.G, color.B, assimpMaterial.Opacity));
    if(assimpMaterial.HasColorSpecular) mat.SetColor("_Specular", new Color(spec.R, spec.G, spec.B));
    if(assimpMaterial.HasReflectivity) mat.SetFloat("_Metallic", assimpMaterial.Reflectivity);
    if(assimpMaterial.HasShininess) mat.SetFloat("_Smoothness", Mathf.Clamp(smoothness, 0f, 1f));
    if(assimpMaterial.HasReflectivity) mat.SetFloat("_EnvironmentReflections", assimpMaterial.Reflectivity);
    if(assimpMaterial.HasBumpScaling) mat.SetFloat("_BumpScale", assimpMaterial.BumpScaling);
    if (assimpMaterial.HasColorEmissive)
    {
        mat.EnableKeyword("EMISSION");
        mat.SetColor("_EmissionColor", new Color(emissive.R, emissive.G, emissive.B));
    }
}

if(assimpMaterial.HasTextureDiffuse) mat.SetTexture("_BaseMap", LoadTextureFromPath(Path.Combine(_directory, assimpMaterial.TextureDiffuse.FilePath)));
if (assimpMaterial.HasTextureSpecular)
{
    mat.EnableKeyword("_METALLICSPECGLOSSMAP");
    mat.SetTexture("_MetallicGlossMap", LoadTextureFromPath(Path.Combine(_directory, assimpMaterial.TextureSpecular.FilePath)));
}

if (assimpMaterial.HasTextureNormal)
{
    mat.EnableKeyword("_NORMALMAP");
    mat.SetTexture("_BumpMap", LoadTextureFromPath(Path.Combine(_directory, assimpMaterial.TextureNormal.FilePath)));
}

if (assimpMaterial.HasTextureLightMap)
{
    mat.EnableKeyword("_METALLICSPECGLOSSMAP");
    mat.SetTexture("_MetallicGlossMap", LoadTextureFromPath(Path.Combine(_directory, assimpMaterial.TextureLightMap.FilePath)));
}

if (assimpMaterial.HasTextureEmissive)
{
    mat.EnableKeyword("EMISSION");
    mat.SetTexture("_EmissionMap", LoadTextureFromPath(Path.Combine(_directory, assimpMaterial.TextureEmissive.FilePath)));
}
```

```
1 usage
private UnityEngine.Mesh ConvertAssimpMeshToUnityMesh(Assimp.Mesh assimpMesh)
{
    Mesh unityMesh = new Mesh();

    unityMesh.vertices = assimpMesh.Vertices.Select(v => new Vector3(v.X, v.Y, v.Z)).ToArray();
    unityMesh.triangles = assimpMesh.GetIndices();
    unityMesh.normals = assimpMesh.Normals.Select(n => new Vector3(n.X, n.Y, n.Z)).ToArray();
    unityMesh.uv = assimpMesh.TextureCoordinateChannels[0].Select(uv => new Vector2(uv.X, uv.Y)).ToArray();
    unityMesh.tangents = assimpMesh.Tangents.Select(t => new Vector4(t.X, t.Y, t.Z)).ToArray();

    unityMesh.RecalculateNormals();
    unityMesh.RecalculateBounds();
    unityMesh.RecalculateTangents();

    return unityMesh;
}
```

Das gleiche galt für die Übersetzung der Meshes aus Assimp zu Unity, dabei waren die Modelle jedoch sehr ähnlich und deshalb einfach zu übersetzen.

Der gesamte Code zum Laden und Übersetzen von Assimp zu Unity befindet sich in der "ModelImporter" Klasse.

Nachdem diese Aufgaben erfüllt waren, konnte ich bereits Modelle effizient und schnell zur Laufzeit laden.

Materialien

Die Materialien stellten jedoch eine große Herausforderung dar, weil nicht jedes Format die gleichen Eigenschaften besitzen kann und vor allem jede 3D-Anwendung, wie Blender, Cinema4D oder CAD-Programme, ihre Materialien anders definiert und exportiert.

Das bedeutet, dass wir die Möglichkeit hatten uns auf eine 3D-Anwendung zu spezialisieren, was jedoch sehr eingeschränkt wäre. Deshalb suchte ich nach weiteren Alternativen und bin auf das neue 3D-Format "GLF" der Khronos-Group gestoßen.

Dieses versucht, gerade das Problem der Materialdefinitionen zu lösen, indem es einen neuen Standard entwickelt. Mittlwerweile sind die meisten 3D-Anwendungen auch dazu fähig, mit diesm Format zu arbeiten.

Das heißt der Vorteil dieses Formates ist, dass 3D-Modelle nahezu identisch aussehen, egal in welcher Engine oder Anwendung sie importiert wurden, da die Materialdefinitionen einem Standard folgen.

Um also eine möglichst breite Auswahl an 3D-Anwendungen zu unterstützen, entschied ich mich dieses Format zu nutzen und Assimp nur als Notlösung zu implementieren.

GLTF-Format

Die Khronos-Group bietet eine eigene Implementierung für Unity an und macht es damit auf den ersten Blick leicht es zu implementieren. Allerdings gab es auch dabei Schwierigkeiten, denn ich nutzte in Unity die so genannte "Lightweight Rendering Pipeline", um eine möglichst hohe Leistung zu erzielen, da diese für Virtual Reality besonders wichtig ist und ich teilweise sehr komplexe 3D-Szenen darstellen muss.

Das Problem dabei war, dass die Implementierung der Khronos-Group diese noch nicht unterstützt, bzw. keine Shader dafür bereitstellen und ich somit noch einmal den Schritt der Übersetzung der Materialien auch für das GLTF-Format durchführen musste.

Jedoch hat sich der Aufwand gelohnt, denn nun kann die Anwendung jedes beliebige Modell laden und so darstellen, wie es der 3D-Designer erstellt hat.

Als guten Nebeneffekt, enthält die Implementierung auch einen Export zur Laufzeit. Dieser wird später genutzt um eine Szene, welche in Virtual Reality bearbeitet worden ist, wieder zu exportieren, um sie dann wieder in einer 3D-Anwendung zu laden.

Interaktion mit Objekten

Nutzereingabe

Zur Verarbeitung der Nutzereingaben, entschied ich mich eine zentrale Klasse zu erstellen (InputEmitter), die alle Eingaben erfasst und, wenn nötig, verarbeitet und die Möglichkeit bietet, dass sich Komponenten an "Events" binden können, durch welche die Nutzereingaben dann weitergeleitet werden.

Der Grund dafür ist, dass ich jederzeit sehr einfach neue Eingaben definieren kann, wenn ich sie brauche und auch leicht neue Eingabemöglichkeiten schaffen kann, wie z.B. Steuerung durch Maus oder andere Controller.

Dafür müssen diese einfach das jeweilige Event ausführen und die zu empfangenden Komponenten wissen nichts über die eigentliche Eingabemethode.

Ein weiterer Vorteil ist, dass ich Gesten auch an dieser Stelle genauso als Nutzereingabe erkennen und definieren kann, wie z.B. bereits implementiert für "Tap" oder "Hold".

So dient sie im Prinzip als "Wrapper" für Nutzereingaben.

Struktur

Da der Nutzer zu jeder Zeit nur mit einem Objekt gleichzeitig interagieren kann, gibt es einen so genannten "ActionController", welcher die auf die Events des InputEmitters hört, und dann bei einer Nutzereingabe entscheidet, an welches der Objekte diese weitergeleitet wird.

Das passiert durch "Raycasts" von dem Controller des Nutzers aus. Das erste Objekt, welches davon getroffen wird und das richtige Interface implementiert, ist das aktuelle interagierbare Objekt.

Es gibt ein Interface (IInteractable) welches ein Objekt implementieren muss, um als solches vom ActionController erkannt zu werden. Implementiert wird es im Moment nur von "MovableObject", was sich aber in Zukunft noch ändern kann, wenn es mehr Interaktionen und Funktionen geben soll.

Um überhaupt zu definieren, welche Teil-Objekte des 3D-Modells interagierbar sein sollen, werden sie durch den "InteractionSetup" nach ihren Gruppennamen im Modell rekursiv gefiltert. Bestimmte Namen, die vorher vom 3D-Designer definiert wurden, haben dann bestimmte Folgen.

Ein "Produkt" im Namen legt beispielsweise fest, dass es sich bei dem Objekt um ein normales verschiebbares Objekt handelt und bekommt von dem InteractionSetup dann MovableObject als Komponente angefügt.

Diese Namen können durch eine Konfigurationsdatei frei gewählt werden.

Bewegung

Wenn der Nutzer auf ein bewegbares Objekt zeigt und den rechten Trigger gedrückt hält, kann er es bewegen.

Dabei wird im Hinterhrund eine unsichtbare Linie vom Controller aus erstellt, welche den Boden trifft. Um nun die Position des Objektes zu berechnen, wird die zurückgelegte Strecke des Zeigers auf dem Boden 1 zu 1 auf das Objekt übertragen und man hat das Gefühl als würde man es direkt bewegen.

Diese Methode führte jedoch immer zu Problemen, sobald der Winkel zwischen Controller und Boden sehr steil war, oder man einfach den Controller Richtung Decke zeigen lassen hat, da dann kein Boden von der Linie getroffen wird.

Deshalb erweiterte ich diese Methode später um eine "Parabel" anstatt der Linie zum Boden. Die Berechnung war die gleiche, jedoch trifft eine Parabel immer den Boden, egal in welchem Winkel man den Controller hält.

Die richtigen Werte für die Geschwindigkeit und Empfindlichkeit zu finden erforderte viele Iterationen und Tests und selbst heute noch ist das Ergebnis

durchaus verbesserungsfähig, da die Tests zeigen, dass es immernoch Nutzer gibt, die mit dieser Methode nicht zurechtkommen.

Die Idee für die Parabel kam mir, da die meisten Virtual Reality Anwendungen die gleiche Methode nutzen, um das Teleportieren im Raum zu implementieren, nur, dass dabei die Parabel meistens sichtbar ist.

Rotation

Während der Nutzer das Objekt bewegt, kann er es durch den rechten Stick auch um die Y-Achse rotieren.

Diese Rotation war am anfang noch Linear, abhängig zum Stick, wurde aber später zum weicheren und genaueren rotieren durch eine quadratische Methode ersetzt.

Höhe

Um es dem Nutzer möglichst einfach zu machen, entschieden wir uns, dass die Position in der Y-Achse eines Objektes nicht per Hand veränderbar sein sollte und sich die Objekte möglichst intelligent selbst auf Oberflächen positionieren, wenn sich eine über ihnen befand.

Dafür gab es die unterscheidung in dem InteractionSetup zwischen Objekten die nur auf dem Boden Sinn machten (Podeste) und Objekten, die überall stehen konnten (Produkte, Dekoration).

Diese Methode funktionierte bis zur Messeversion auch noch gut, jedoch musste ich sie entfernen, da es einige Probleme mit den späteren "Mesh-Collidern" und dem neuen Verfahren in der Kollisionserkennung gab (siehe Punkt Kollision).

Aktuell ist es demnach möglich, die Objekte manuell in der Höhe durch die Tasten A und B rechts am Kontroller zu verändern, was es z.B. auch ermöglicht Regale an der Wand zu platzieren.

Kollision

Eine sehr große Herausforderung stellte die Physik-Engine von Unity und das Kollisionssystem dar.

Um Objekte überhaupt interagierbar zu machen bzw. "sichtbar" für Raycasts zu machen, brauchen sie eine Art von Kollisionsbox. Bis zur Messeversion nutzte ich einfache "BoxCollider" für diesen Zweck, also einfach Quader, welche als Grenzen das ganze Objekt umschlossen hatten. Die Größe bzw. "Bounds" dieser Quader wurde jeweils in "MovableObject" Klasse ausgerechnet.

Die Methode funktionierte gut, bis auf die Tatsache, dass es keine genaue Platzierung von Objekten ermöglicht, wie z.B. das verschieben von Stühlen unter einen Tisch, da der BoxCollider den ganzen Tisch umfasste und somit den Stühlen nicht hineinverschoben werden konnten.

Man könnte natürlich die Kollision zwischen Objekten einfach deaktivieren, aber das verhindert dann widerum das komfortable aneinander platzieren von Objekten, da man dafür dann sehr präzise steuern müsste.

Deshalb nutzte ich von nun an so genannte konvexe "MeshCollider" anstatt der simplen BoxCollider. Diese sind sehr viel genauer, brauchen dafür aber mehr Rechenleistung und brachten eigene Probleme mit sich.

Unity berechnet die der MeshCollider selbst anhand des 3D-Modells. Das Problm ist, dass bei der Berechnung viele Fehler auftreten können. Ich muss deshalb die Modelle vorher immer manuell einer Prüfung unterziehen um zu garantieren, dass sie:

- nicht planar sind (nur aus einer Fläche bestehen)

- nicht nur aus Punkten ohne Flächen bestehen

- nicht zu dünn sind

- keine negative Skalierung haben

In diesen Fällen nutze ich einfach weiterhin einen BoxCollider, oder versuche sie zu reparieren, z.B. durch das erstellen eines leeren "Container-Objektes", um die negative Skalierung auszugleichen.

Fortbewegung im Raum

Der Nutzer kann sich, dank Room-Tracking, selbst einfach im Raum bewegen oder, falls der Platz nicht ausreicht, durch den linken Stick am Controller mit Hilfe von Teleportation bewegen.

Diese Funktion war bereits in der Oculus-VR Integration implementiert und ich konnte sie für meine Zwecke anpassen und nutzen.

Als weitere Möglichkeit der Fortbewegung sind noch "Kamerapunkte" geplant, die mit aus dem 3D-Modell importiert werden und dazu dienen bestimmte Ansichten festzulegen, durch die sich der Nutzer dann mit einem Klick bewegen kann.

Sonstige Features

Platzierungshilfe durch Gitter

An der Medianight wünschten sich viele Tester eine Form von Platzierungshilfe für die Objekte. Eine Idee war, dass sich Objekte gegenseitig erkennen und aneinander "snappen" können. Da diese Methode jedoch sehr komplex ist, weil jedes Objekt eine beliebige Größe haben kann, entschieden wir uns zur Hilfe ein Gitter unter dem aktiven Objekt darzustellen. Dieses wird nur um einen bestimmten Radius herum angezeigt und verschwindet, falls der Nutzer kein Objekt verschiebt, um die Blick auf den Stand nicht zu stören.

Dieses Gitter wird durch einen, selbst programmierten, Shader berechnet und angezeigt, wodurch ich die Chance hatte, etwas über Shaderprogrammierung zu lernen.

User Interface

Bisher gibt es nur ein sehr einfaches, funktionales, Fenster, welches die aktuellen Dateien in einem bestimmten Ordner anzeigt, um sie laden zu können.

In zukünftigen Versionen ist ein richtiges UI mit File-Explorer geplant und einem weiteren Fenster, dass es ermöglichen soll, zusätzliche Produkte und Dekorationen zu der Szene hinzuzufügen.

Außerdem sind noch kleine Hilfstexte an den Controllern selbst geplant, damit der Nutzer die Funktionen auf den einzelnen Tasten immer nachvollziehen kann.

Interaktion mit Maus

Ein weiteres Features, welches in Zufunft implementiert werden soll, ist die Interaktion mit der Szene durch Tastatur und Maus.

Falls ein Kunde beispielsweise nicht mit der Steuerung umgehen kann, oder einfach nicht will, sondern sich nur die Szene in Virtual Reality angucken möchte, könnte der Nutzer am Rechner die Szene weiterhin zur Laufzeit, je nach Kundenwünschen, bearbeiten.

Medianight

Mein Auftritt auf der Medianight war, wie bei dem Thema erwartet, nicht besonders attraktiv für Studenten, dennoch kamen einige Interessenten aus der Branche vorbei mit denen ich mich austauschen konnte und noch viele Ideen und Verbesserungsvorschläge sammeln konnte.

Außerdem hat sich gerade dort gezeigt, dass frühe Tests sehr wichtig sind, da die eigene Perspektive oft nicht auf jeden zutrifft.

Fazit

Trotz dass es so ein kleines Unternehmen ist und wir viel direkten Kontakt hatten, ist es immernoch manchmal passiert, dass es Unklarheiten oder ungeklärte Dinge. Deshalb nehme ich mit, dass ich in Zukunft noch enger und genauer mit Kunden oder Unternehmen zusammenarbeite, um eben diese zu vermeiden.

Ich hatte zwar bereits einiges an Erfahrung in der Unity-Engine gesammelt, aber wenn man ein an einem Projekt in einem ganz anderen Bereich und einer anderen Branche arbeitet, lernt man immernoch sehr viele neue Aspekte und Seiten der Engine bzw. allgemein für alle Anwendungen in diesem Bereich kennen.

Alles in allem konnte ich, gerade auch durch die Zusammenarbeit mit einem Unternehmen, sehr viel Erfahrung sammeln und meinen ersten guten Einstieg in der Virtual Reality Szene machen.