Technische Dokumentation

Nachdem wir uns darauf festgelegt haben eine virtuelle Realität zu simulieren und dafür eine 3D Umgebung in der sich der Nutzer orientieren, bewegen und interagieren kann brauchen, haben wir uns entschieden zum Erstellen dieser Umgebung Unity (Vers. 5.1.0f3) zu nutzen.

Die Argumentation dahinter ist, dass Unity von sich aus schon viele Möglichkeiten zum erstellen einer solchen, oben gennanten, Umgebung bietet, über eine einfache Script-to-Environment Integration verfügt (Diese unterstützt Java, C#, Bool und Unity Script; wir haben uns für C# entschieden), eine simple Importierung von 3D Modellen (in eingeschränkten Formaten) und es zudem möglich ist, zumindest einfaches Sound Verhalten im Editor zu managen.

Obwohl wir am Anfang des Semesters noch nie mit Unity gearbeitet hatten, war es dank einer exzellenten Dokumentation möglich, sich relativ schnell in die groben Mechaniken einzuarbeiten.

Aber natürlich ist es nicht möglich eine komplexere Software zu entwickeln ohne auf diverse Probleme zu stoßen, auch wenn die Engine auf der man aufbaut einem einiges erleichtert.

Unity nutzt beispielsweise ein Script System, dass einem objekt-orientierte Programmierung in herkömmlichem Sinne etwas erschwert.  
Erstellt man in Unity beispielsweise ein GameObject, ist dieses GameObject auch ein Objekt der Klasse GameObject. Das erleichtert einem bei einer hierarchischen Struktur eines Levels die Arbeit (Beispielsweise kann man einen Raum als Parent-Object erstellen und alles was in diesem Raum ist als Child-Object dieses Raum-Objekts und dann um etwas mit dem gesamten Raum zu machen nur das Raum-Objekt ansprechen.) zumindest, solange man nicht selber etwas scripten möchte.

Scripts in Unity nutzen, ähnlich wie in Processing, eine Start() und Update() Funktion (das Repertoire wird noch mit Awake(), LateUpdate(), etc erweitert.) Um auf Objekte, die man in seiner 3D Umgebung erstellt hat zuzugreifen, erstellt man üblicherweise eine öffentliche Variable, beziehungsweise ein Feld und kann diese dann im „Inspektor“ in Unity zu ordnen.

Das Script nur zu erstellen reicht aber nicht, man muss dieses noch in seine 3D Umgebung hinzufügen. Dazu erstellt man beispielsweise ein GameObject ohne zusätzliche Parameter („Empty GameObject“) und fügt diesem GameObject dann das spezifische Script hinzu. Nun kann man im „Inspektor“ in Unity, all seinen öffentlichen Variablen/Feldern die GameObjects zuordnen, die man möchte.

Das führt zu zwei Problemem.   
Man ist gezwungen mit öffentlichen Variablen zu arbeiten (und davon oft nicht wenige, da, wenn man beispielsweise ein Array mit der Funktion „FindGameObjectWithTags“ auffüllen möchte, also Unity von alleine alle GameObjects durchsieht und zu dem jeweiligen Array die mit dem jeweiligen Tag hinzufügt, das unglaublich viel Leistung benötigt.) und das sollte man eigentlich vermeiden, da öffentliche Variablen zwar bequem sind, aber oft zu Problemen führen. Dieses Problem lässt sich theoretisch durch die Serialisierung der Felder vermeiden, das ist aber auch nicht die eleganteste und effizienteste Methode, zu mal die Serialsierung nochmal Rechenleistung verbraucht.  
Eine optimale Lösung für dieses Problem haben wir nicht gefunden.

Das zweite Problem ist Vererbung. Denn jede child-class erbt alle öffentlichen Variablen der parent-class und das führt irgendwann zu einer endlos langen Liste von öffentlichen Variablen im Inspektor, die man entweder alle mit Platzhaltern stopfen oder mit endlos vielen NullPointerException leben muss.

Deswegen haben wir auf eine komplexere Vererbungs-Hierarchie verzichtet. Die Parameter werden als Getter oder Statisches Array übergeben und die meisten Scripts sind eigenständige Klassen.  
Andere Lösungswege, wie die Möglichkeit das Attribut [HideInInspector] zu vergeben oder die betroffenen Variablen in der Child-Class als private zu überschreiben, sind keine Lösungsoptionen, denn Unity bietet keine Möglichkeit einer reibungsfreien Override-Funktion, auch wenn diese rein theoretisch in C# möglich ist.

Die Implementierung von Assets (soll heißen: Materials/Textures, Models und Audio Files) ist im allgemeinen sehr einfach, man hat aber oft spezifische Format-Probleme; Texturen werden nicht im TrueColor Format importiert oder ein bestimmtes 3Dmodelling-Format wird nicht unterstützt.

Größere Probleme hatten wir aber nur bei den 3D Modellen. Oft hatten oder haben diese nur eine Seite, das Mesh wird nicht importiert und ein Collider dafür erstellen ist beinahe unmöglich oder die Texturen werden nur auf einer Seite des Models übernommen und von der anderen ist es dann durchsichtig.  
Manche Probleme konnten wir lösen, beispielsweise ist es besser in Unity ein leeres GameObject zu erstellen und das Model als ChildObject zu nutzen, dann hat man keine Probleme mit Rotationen um den Nullpunkt oder physikalischen Vorgängen. (Rigidbody Nutzung) Andere Probleme nur notdürftig (beispielsweise bei manchen Modellen, die nur einseitig texturierbar waren.).

Die Implementierung von Virtual Reality sollte, laut Unity, seit dem 5.1 Patch sehr einfach sein, denn mit diesem bekam der Editor 1st-Party-Support für sehr viele Virtual Reality Produkte (auch für die Oculus Rift). Das verlief aber natürlich nicht so reibungslos, wie wir uns das vorgestellt hatten. Für das Nutzen der nativen Unterstützung benötigt man auch die 0.6 Runtime für die Rift und diese ist, vorsichtig formuliert, unausgereift. Anfangs ist die Oculus bei jedem noch so kleinen Versuch auch nur irgendetwas zu tun abgestürzt. Dank der Unterstützung der Unity-Community und dem Verzicht auf körperliche Bedürfnisse, konnten wir aber alle Probleme der Reihe nach lösen. (Falscher Grafikkarten-Treiber-Zugriff, kein VGA-HMD Adapter Support, falsche Unity-Camera-Scripts, kein Headtracking, die gebuildete .exe stürzt ab, sekundärer Monitor wird gar nicht, falsch oder nicht doppelt verwendet, der direkte Zugriff auf die .exe funktioniert nicht)

Für C# haben wir uns entschieden, weil wir gerne eine weitere Sprache lernen wollten, C# eine weit verbreitete und sehr elegante Sprache ist und es sich bei Unity gleich angeboten hat.  
Zudem ist C# weitaus performanter als Unity- oder Javascript und ein guter Einstieg für komplexere C-Sprachen, wie Objective C oder C++.

Bei den Modellen haben wir uns für Maya entschieden, weil [Dina?!]