Archery VR - Mesh Zerstörung

Alexander Belzer

18 März 2023

1 Einführung

Dieses Projekt wurde im Rahmen des Moduls TM038 - Prakt. Virtuelle Realität und Simulation erstellt und behandelt ein Spiel zum Bogenschießen in VR mittels der Oculus Rift S in Unity [2] mit zusätzlicher Mesh Zerstörung der getroffenen Ziele. Ein Zielobjekt welches von einem Pfeil getroffen wird, wird auf Mesh-Ebene zerstört und in etliche Teilobjekte aufgeteilt. Die Aufteilung des Meshes findet mittels eines 3D Voronoi Graph statt, welches über das Mesh gelegt wird und über die Schnittkanten das Mesh aufteilt.

2 VR - Integration

Dieses VR Spiel ist für die Oculus Rift S ausgelegt und wurde mit dem Oculus Integration Package entwickelt. Das Package bietet bereits ein Kamera Prefab mit der Darstellung und Interaktion der Controller als Hände. Diese können bereits genutzt werden, um mit weiteren Objekten zu interagieren.
Im Grunde wird hier hauptsächlich die Funktion genutzt, um Objekte zu greifen und wieder los zulassen, zusätzlich wird auch in einem Fall die Funktion zum Drücken von Buttons verwendet. Um ein Objekt greifen zu können muss es die Scripts `Grabbable.cs` und `Rigidbody.cs` angehängt haben. Ist zusätzlich das PointableUnityEventWrapper.cs Script angehängt, kann auf die Events die beim greifen und loslassen eines Objects gefeuert werden, reagiert werden. Zusätzlich werden die Scripts `HandGrabInteractable.cs` und `HandGrabPose` am entsprechenden Objekt benötigt, mit denen festgelegt werden kann, wie das Objekt angefasst werden kann, welche Finger genutzt werden, in welcher Ausrichtung es in der Hand liegt und wie die Hand am Objekt liegt. Dies muss jeweils für jede Hand mit der es gegriffen werden kann, festgelegt.
Um ein Objekt mit einem Finger zu drücken, wie einen Button, kann das `PokeInteractable.cs` Script angehängt und konfiguriert werden. Hier kann wieder das `PointableUnityEventWrapper.cs` Script angehängt werden um auf Events entsprechend zu reagieren.
Das Greifen wird im Spiel für den Bogen, zum heben des Bogens und zum ziehen der Sehne, und für die Pfeile verwendet. Das Drücken eines Buttons wird lediglich im Tutorial verwendet, um in die Hauptspielszene zu gelangen.

Abbildung 1: Game Szene

3 Spiel

Der Spieler wird zu Beginn in eine grüne Umgebung gesetzt und direkt vor ihm schwebt ein Bogen der für den Spieler mit jeder Hand am Griff und in der Mitte der Sehne greifbar ist. Neben dem Spieler ist ein Podest auf dem Pfeile spawnen, die der Spieler am Pfeil Ende greifen kann. Wird der Pfeil nah genug an die Knock-Position des Bogens gehalten (festgelegter Threshhold), wird der Pfeil automatisch in den Bogen gelegt und passt sich der Orientierung des Bogens an, bis er abgefeuert wird. Der Pfeil kann abgefeuert werden, indem die Sehne des Bogens in der Mitte gegriffen und nach hinten gezogen wird. Sobald die Sehne losgelassen wird, wird der Pfeil entsprechend der aktuellen Orientierung abgefeuert. Links neben dem Bogen wird hierbei die relative Kraft angezeigt mit der der Pfeil abgeschossen werden würde, abhängig von der Spannung der Sehne. Die Kraft wird hier zwischen 0 und 100 angezeigt. Rechts vom Bogen wird der aktuelle Höhenwinkel angezeigt, in dem der Bogen bzw. der Pfeil sich befindet. Wird der Bogen vertikal gehalten, beträgt der Winkel 0 Grad und der Pfeil geht direkt in den Sinkflug. Wird der Bogen perfekt horizontal in den Himmel gehalten, beträgt der Winkel 90 Grad und der Pfeil fliegt gerade hoch und landet auf der Ausgangsposition.
Die Bewegung des Pfeils wird durch den angehängten `Rigidbody` gesteuert und wird in Bewegung gesetzt in dem eine Kraft und ein Drehmoment in die entsprechende Richtung hinzugefügt wird.

```
1 _rigid.AddForce( projectorVecNormalized * bow.GetBowForce() * 20f,
      ForceMode.VelocityChange );
2 _rigid.AddTorque( projectorVecNormalized * 20 );
```

Sobald ein Pfeil abgefeuert wird, wird ein neuer Pfeil durch den `ArrowSpawner` im Podest erstellt und der Szene hinzugefügt.
In der Szene schweben verteilt unterschiedliche Ziel-Objekte herum, die zerstört werden können. Wird ein Ziel mit einem Pfeil getroffen, wird es zerstört und

der Spieler erhält so viele Punkte wie über dem Objekt angezeigt werden, siehe 2. Die linke Zahl stellt hierbei die Punkte dar die der Spieler für die Entfernung bekommt und die Rechte Zahl, wie viel Punkte er für die Größe und Form des Objekts erhält.
In der Szene liegt ein `TargetSpawner`, welcher eine beliebige Anzahl an Positionen enthält und an diesen eine feste Anzahl an Zielobjekte erstellt. Wird ein Ziel zerstört, wird der Reihe nach an einer anderen Position ein neues erstellt.
In der Mitte oben vor dem Spieler fliegt ein UI-Element, welches anzeigt wie viele Punkte der Spieler aktuell hat, wie viele Pfeile noch übrig sind und aus welcher Richtung, wie stark ein Wind weht, welcher dann Einfluss auf die Flugbahn des Pfeils hat, indem er die aktuelle Geschwindigkeit des Pfeils anpasst.

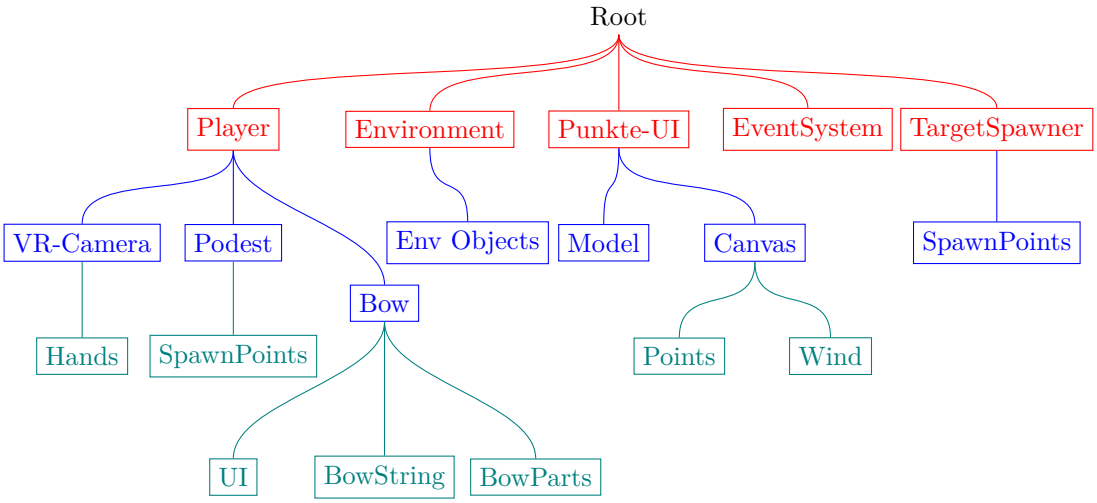
3.1 Spielprinzip

Der Spieler steht in der Szene und hat seinen Bogen in greifbarer Nähe sowie ein Podest auf dem Pfeile positioniert sind. Der Spieler kann nun solange Pfeile schießen und Punkte sammeln solange er Ziele trifft. Er hat 5 Fehlschüsse frei, danach wird ihm sein Punktestand präsentiert und er kann ein neues Spiel starten.

4 Aufbau der Szene

Es gibt zwei Szenen, die bis auf einen Fall gleich aufgebaut sind. Im Grunde gibt es sechs Grundobjekte, zum einen haben wir den Spieler, welche alle für den Spieler relevanten Objekte enthält mit denen der Spieler direkt interagieren kann. Zum einen die Camera, welche ebenfalls die Hände enthält. Außerdem ein Podest, auf dem sich die Pfeile befinden, welche aber erst in Runtime erstellt werden. Das Podest, `ArrowSpawner`, hält hierfür eine Liste von `Empty GameObjects` an deren Postionen die Pfeile gespawnt werden. Zusätzlich gehört der Bogen zum Spieler, der Aufrecht vor dem Spieler lpatziert ist. Der Bogen beinhaltet ein UI, welches die aktuelle Spannung der Sehne angibt und die aktuelle Ausrichtung des Bogens in einem Winkel zwischen 0 und 90 Grad. Zusätzlich enthält der Bogen die Sehne des Bogens, welche gesondert gehandelt wird, da der Spieler direkt mit ihr interagiert. Neben der Sehne gibt es noch weitere Elemente vom Bogen, die aber keine größere Relevanz haben.
Neben dem Spieler, gibt es die Umgebung/Environment in dem sich der Spieler befindet, diese beinhaltet sämtliche Umgebungsobjekte mit denen der Spieler nicht direkt interagieren kann und nur zur Dekoration der Szene vorhanden sind.
Im nächsten Punkt unterscheiden sich die beiden Szenen voneinander in der richtigen Spielszene haben wir hier das Punkte-UI, welches zum einen das Model hält, in dem der aktuelle Punktestand und Windrichtung festgehalten wird, und zum anderen das eigentliche UI mit den aktuellen Punkten und der Windrichtung. In der Tutorial Szene, sind diese beiden Objekte nicht von Nöten, hier ist an dieser Stelle das Tutorial-Ui, welches dem Spieler Instruktionen gibt.

Das nächste Object ist das Eventsystem zur Kommunikation und Verarbeitung der Benutzereingaben in der Szene.
Als letztes Objekt gibt es den `TargetSpawner` welches ebenfalls kein physisches Objekt darstellt, sondern an einem beliebigen Ort in der Szene liegt und wie der `ArrowSpawner` eine Liste von `Empty GameObjects` enthält, an denen dann im Spiel in Runtime Zielobjekte gespawnt werden, einmal zu Beginn und jeweils wenn ein Objekt zerstört wurde.



5 Mesh Zerstörung mittels Voronoi

Bei der Aufteilung des Meshes wird ein robuste Methode zum fragmentieren von Objekten verwendet [1] hierbei wird ein 3D Voronoi Graph über dem Mesh mit zufälligen Punkten auf dem Mesh als Input aufgebaut. Das Mesh des zu zerstörenden Objekts wird über das Mesh des Voronoi Graph gelegt, die einzelnen neuen Teile des neuen Mesh werden dann erstellt, indem die Schnittmenge des Meshes des Objekts und jeder einzelnen Voronoi Zelle genommen wird. Die einzelnen Schnittmengen bilden dann die neuen teile des ursprünglichen Objekts.
In der Grafik 2 sieht man ein zerstörbares Objekt, wie es im Spiel zu sehen ist, wenn es mit einem Pfeil getroffen wird, wird es in kleinere Teile aufgeteilt, siehe Grafik 3. Die entstehenden Meshes werden zu `Rigidbodys` und fallen zerbrochen auf den Boden.

5.1 Voronoi- Delaunay Dualität

Das Aufteilen des Meshes stützt sich auf die Dualität zwischen Voronoi und Delaunay, sodass aus der Delaunay Triangulation ein Voronoi Graph gewonnen werden kann oder auch umgekehrt aus einem Voronoi Graph eine Delaunay-Triangulation gewonnen werden kann. Für dieses Projekt wird nur der ersten Fall betrachtet im drei Dimensionalen Raum, somit wird aus der Triangulierung

Abbildung 2: zerstörbares Objekt

eine Tetraederisierung.
Hierfür werden die Zentrums Punkte der Voronoi Zellen aus den Eckpunkten der Tetraederisierung gewonnen und die Kanten einer Voronoi Zelle aus den Verbindungsvektoren zwischen zwei Mittelpunkten benachbarter Tetraeder.

5.2 Delaunay

In einer Delaunay-Triangulierung werden Dreiecke zwischen Punkten aus einer Menge in einem Raum erstellt, sodass für alle Dreiecke die Umkreisbedingung erfüllt ist, die besagt dass der Umkreis eines Dreiecks keine weiteren Punkte beinhalten darf, abgesehen von den Eckpunkten des Dreiecks. Im drei dimensionalen Raum gilt dass selbe, hierbei wird eine Sphere um das Tetraeder gelegt und es dürfen keine weiteren Punkte in der Sphere liegen, abgesehen von den vier Eckpunkten des Tetraeder.
Für die Erstellung der Tetraederisierung wird ein inkrementeller Flip-Based-Algorithmus verwendet.
Hierfür wird als Root-Element ein Tetraeder verwendet, dass größer ist alle möglichen enthaltenen Punkte und diese umschließt. Hierbei wird versichert, dass ein Punkt `p` der eingefügt wird immer in einem Tetraeder liegt.
Mit dem Algorithmus 1 können beliebig viele Punkte zur Delaunay Tetraederisierung hinzugefügt werden.

Beim Einfügen eines neuen Punkts wird über die bestehenden Tetraeder iteriert und jenes ausgewählt, welches den Punkt enthält. Das ausgewählte Tetraeder wird in vier neue Tetraeder aufgeteilt, welche dann einem Stack hinzugefügt werden. Dieser Stack wird abgearbeitet und solange durchlaufen bis er leer ist. Ist der Stack leer bedeutet dies, dass alle Tetraeder Delaunay sind und müssen

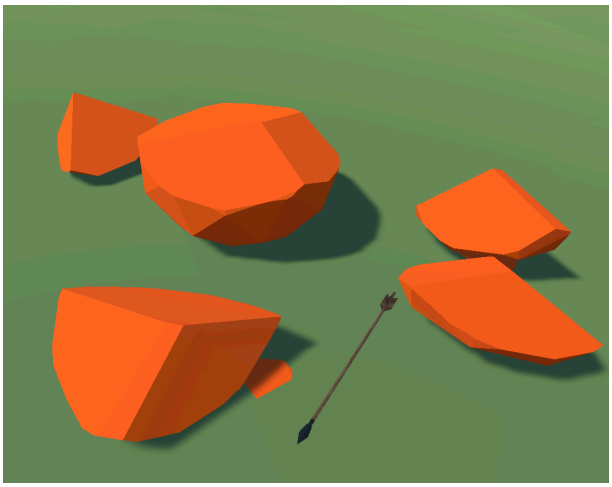


Abbildung 3: zerstörtes Objekt mittels Voronoi

nicht mehr geflipped werden.
Es wird das Tetraeder `(a, b, c, p)` vom Stack genommen, wobei p der neu hinzugefügte Punkt ist, finde das benachbarte Tetraeder `(a, b, c, d)`, welche sich die Fläche `(a, b, c)` ohne den neuen Punkt `p`, mit dem vom Stack genommen Tetraeder teilt. Teste ob der übrig gebliebene Punkt `d`, in der Sphere des Tetraeders `(a, b, c, p)` liegt. Ist dass der Fall ist das Tetraeder noch nicht Delaunay und es wird ein Flip durchgeführt, die teilnehmenden Tetraeder werden aus der Liste der Tetraeder entfernt und die aus dem Flip entstehenden Tetraeders werden der Liste hinzugefügt und ebenfalls auf den Stack gepusht um ein weiteres mal überprüft zu werden. Liegt aber der Punkt `d` nicht in der Sphere des Tetraeders `(a, b, c, p)` ist das Tetraeder Delaunay und das nächste kann getestet werden.

Beim Ausführen des Flips wird noch einmal zwischen 2 Flips unterschieden, dem 23Flip und dem 32Flip. Entweder sind vom neu hinzugefügten Punkt, zwei Seitenflächen des Tetraeders `(a, b, c, d)` zu sehen oder nur eine. Ist nur eine sichtbar, bedeutet dies, dass die Vereinigung der beiden Tetraeder eine konvexes Polygon ergibt und es wird der Flip23 angewendet. Sind aber zwei Seitenflächen zu sehen, bedeutet dies, dass die Vereingung konkave Polygon ergibt und es wird untersucht ob ein weiteres Tetraeder `(a, b, p, d)` existiert, welche die Kante `(a, b)` mit den anderen beiden Tetraedern teilt, ist das der Fall kann ein Flip32 ausgeführt werden, ansonsten muss kein Flip durchgeführt werden und das Tetraeder ist Delaunay. Sind 3 Seitenflächen des Tetraeders zu sehen, wird ebenfalls keine Aktion ausgeführt.
Die Überprüfung wie viele Seitenflächen gesehen werden, wird nicht darüber bestimmt ob die Vereinigung der Tetraeder konvex oder konkav ist, sie kann bestimmt werden, in dem überprüft wird die der Vektor zwischen `p` und `d` das Dreieck `(a, b, c)` schneidet. Also ob die Kante zwischen den beiden Punkten,

Algorithm 1 Insert Point into Delaunay Triangulation

```
1: procedure InsertPoint(T, p)
2:     τ ← Walk(T, p)                          ▷ Obtain tetrahedron containing p
3:     insertPointWithFlip14(τ, p)
4:     stack ← newStack                        ▷ Initialize an empty stack
5:     pushFourNewTetrahedra( stack, τ)
6:     while stack is non-empty do
7:         τ ← pop(stack)
8:         τ_a ← getAdjacentTetrahedron(τ, abc)   ▷ Find adjacent tetrahedron
   with facet abc
9:         if d is inside circumsphere of τ then
10:            Flip(τ, τ_a)
11:        end if
12:    end while
13: end procedure
```

den Innenraum der gemeinsamen Fläche beider Tetraeder kreuzt.
1 berechnet die Normale des Dreiecks mit `a, b, c` als Eckpunkten. Mittels dem Skalarprodukt von dem KantenVektor zwischen `p` und `d` und der Normale des Dreiecks wird zuerst überprüft, ob die Kante parallel zum Dreieck ist, ist dass der Fall werden sie sich nie kreuzen, siehe 2. Mit 3 und 4 kann der Punkt berechnet werden an dem die Fläche und die Gerade sich treffen.

$$\text{triangleNormal} = \frac{(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})}{\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|} \tag{1}$$

$$\text{triangleNormal} \cdot \text{lineDirection} = 0 \tag{2}$$

$$t = \frac{\text{triangleNormal} \cdot (\mathbf{a} - \text{startPoint})}{\text{triangleNormal} \cdot \text{lineDirection}} \tag{3}$$

$$\text{intersectionPoint} = \text{linePoint} + \text{lineDirection} \cdot t \tag{4}$$

Da mit dem IntersectionPoint nun der Punkt auf der Fläche gegeben ist, kann mittels 8 berechnet werden, ob der Punkt sich im Dreieck befindet. Der Punkt befindet sich im Dreieck, wenn die Normalenvektoren aus 5, 6 und 7 in die gleiche Richtung zeigen.

$$\mathbf{n}_1 = (\mathbf{b}-\mathbf{a}) \times (\text{point}-\mathbf{a}) \tag{5}$$

$$\mathbf{n}_2 = (\mathbf{c}-\mathbf{b}) \times (\text{point}-\mathbf{b}) \tag{6}$$

$$\mathbf{n}_3 = (\mathbf{a}-\mathbf{c}) \times (\text{point}-\mathbf{c}) \tag{7}$$

$$\mathbf{n1} \cdot \mathbf{n2} \geq 0 \quad \text{and} \quad \mathbf{n1} \cdot \mathbf{n3} \geq 0 \tag{8}$$

5.3 Voronoi

Sei `S` eine Menge von Punkten in einem dreidimensionalen Raum R. Die Voronoi Zelle eines Punktes `p` in `S` beschreibt die Menge von zufälligen Punkten `x` in `R`,

die zu `p` näher liegen als zu jedem anderen Punkt in `S`. Die Kombination der Voronoi Zellen aller Punkte `p` in `S` bildet das Voronoi-Diagramm von `S`. In einem 3D - Raum hat das Voronoi Diagramm die Form eines konvexen Polyeders.

Für die Erstellung der neuen Meshes werden CSG-Bäume von dem zerstörbaren Objekt und den einzelnen Voronoi Zellen erstellt.
Ein CSG-Baum (Constructive Solid Geometry) ist eine Datenstruktur, die verwendet wird, um komplexe 3D-Objekte zu erstellen und zu manipulieren. Er basiert auf logischen Operationen wie Vereinigung, Schnitt und Differenz, die auf einfache geometrische Formen angewendet werden. Die neuen Meshes werden dann aus dem Schnitt des CSG Baums des Objekts und aus den einzelnen CSG-Bäumen der Voronoi Zellen.

6 Technologie:

Die verwendete Unity version ist `2021.3.15F1`.
Als Hardware wird die Okkulus Rift S mit zwei HandControllern verwendet.
Github: `https://github.com/Bezes13/ArcheryVR`

Literatur

[1] Zhang, Shao-Xiong, et al. "Physical Solid Fracture Simulation Based on Random Voronoi Tessallation." 2016 International Conference on Computer Engineering and Information Systems, Nov. 2016, https://doi.org/10.2991/ceis-16.2016.40. pdf

[2] *https://unity.com/de*