Dokumentation VR Projekt

Da mein Projekt Konzept sehr simple ist habe ich nicht sehr viel Zeit benötigt die meisten Dinge einzubauen.

Gravitation

Meine erste Aufgabe war die Kreation eines Scripts der die Gravitation zwischen zwei Massen im Raum simuliert. Die erste Variante sah so aus, dass sich das Script auf dem Mond oder auf der Erde befinden sollte und durch eine Referenz, die durch den Inspector zum anderen Himmelskörper gegeben wird, die benötigte Kraft und Richtung ausrechnet und diese dann direkt wirken lässt.

Jedoch schien eine allgemeinere Herangehensweise die Bessere zu sein. Ein GameManager Objekt übernimmt nun die Rolle der Berechnung und Zuweisung der Kraft. Dadurch benötigt die Klasse Gravitation, wie ich sie so schön getauft habe, eine Referenz auf beide jeweiligen Himmelskörper. In diesem Fall für Mond und Erde. (Siehe Abb.: 1)

```
[SerializeField]
private Transform earth = null;
[SerializeField]
private Transform moon = null;
```

Abbildung 1

Bevor man nun mit diesen beiden Objekten physikalisch berechnete Bewegungen durchführen kann brauchen beide eine Rigidbody und Collider Komponente. (Siehe Abb.: 2)

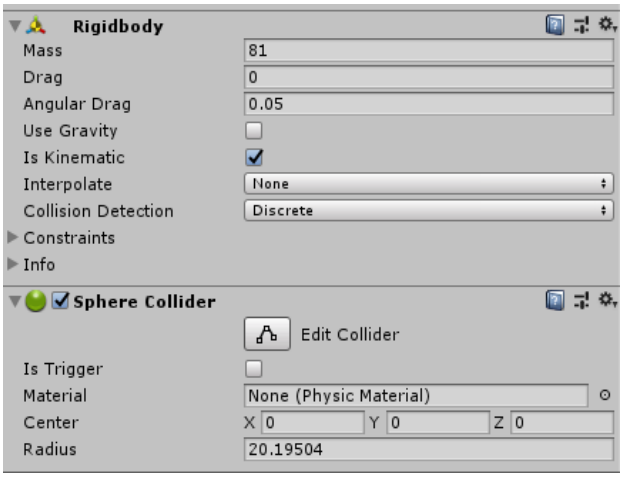

Abbildung 3 - Erde Komponenten

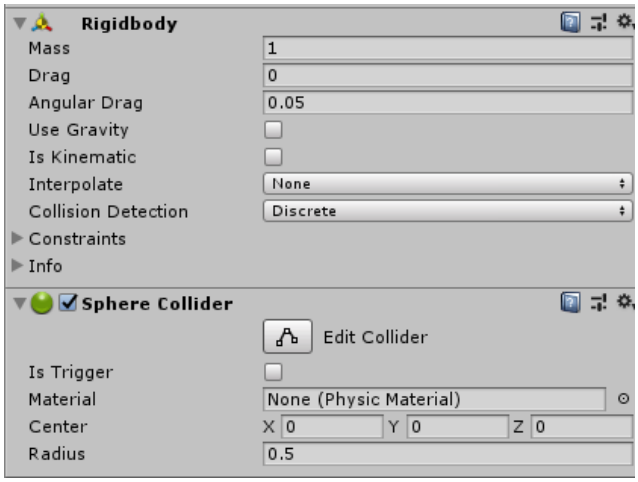

Abbildung 2 - Mond Komponenten

Wenn Beide diese wichtigen Komponenten haben müssen Diesen korrekte Parameter zugewiesen werden, um auch halbwegs akkurate Ergebnisse zu erzeugen. Da die Erde und der Mond eine enorme Masse besitzen rechnete ich mir überschlagsmäßig das Verhältnis aus um die Massen so gering wie möglich zu halten und zu vermeiden die Physik-Engine von Unity an ihre Grenzen zu treiben. Wie Sie

sehen können beträgt mein Ergebnis rund 81 zu 1. Das gleiche Schema wandte ich bei der Größe der beiden Körper an. Die Zahlen 20.1 und 0.5 (Radien der Sphären) kamen jedoch zufällig zustande. Die eigentlichen Größen schätze ich Visuell, mit Hilfe meines errechneten Verhältnisses, ab und skalierte die Objekte dementsprechend. Zusätzlich setzte ich den Rigidbody des Erdenobjekts auf „isKinematic“ da sich die Erde nicht bewegen soll.

Da nun beide Objekte bereit sind durch Physik manipuliert zu werden können wir uns wieder dem Script zuwenden. Zunächst brauchen wir zwei Variablen (Siehe Abb.: 3), um die Rigidbody Komponenten zu speichern, dass wir diese nicht bei jedem Schritt erneut finden müssen. Diese Variablen müssen nicht

```
private Rigidbody rbEarth;
private Rigidbody rbMoon;
```

Abbildung 5

außerhalb der Klasse oder im Inspector zugänglich sein und können daher einfach als „private“ deklariert werden. Nachdem beide Variablen deklariert sind müssen sie noch Zugewiesen werden. Dies werden wir zu Beginn des Spiels machen. In der Start Methode holen wir uns die Komponenten von den schon gegebenen Objekten. (Siehe Abb.: 4)

```
0 references
private void Start()
{
    rbEarth = earth.GetComponent<Rigidbody>();
    rbMoon = moon.GetComponent<Rigidbody>();
}
```

Abbildung 4

Jetzt kommen wir zum bedeutensten Teil dieser Klasse: Der eigentlichen Berechnung der Anziehungskraft.

```
1 reference
private float AttractionForce (float mass1, float mass2, float distance)
{
    return mass1 * mass2 / distance * GRAVITATION * factor;
}
```

Abbildung 6

Die eleganteste Lösung war es eine eigene Methode zu erstellen welche ihre benötigten Informationen durch ihre Parameter Liste bekommt und mit diesen die Kraft errechnet. Die Formel selbst ist überall im Internet und in Physikbüchern zu finden und ist keine Herausforderung. Die beiden Massen und die Distanz ,zwischen den Massen, nimmt die Funktion also als Parameter auf (Siehe Abb.: 6). Nun fehlen nur noch die Gravitationskonstante und ein Faktor. Die gravitationale Beschleunigung kann man aus der

„Physics“ Klasse von Unity heraus verwenden und sie auch danach in den Projekteinstellungen ändern. Wir wollen sie jedoch direkt in unserer Klasse kontrollieren (wenn dies nötig wird) und deklarieren sie deshalb als eine Konstante bei den anderen Variablen (Siehe Abb.: 7). Zusätzlich deklarieren wir noch den Faktor welchen ich in die Formel eingefügt habe (Siehe Abb.: 7). Dieser ermöglich es die Kraft

```
2 references
public void EarthAttract(bool mode)
{
    factor = mode ? 1 : 0;
}
```

Abbildung 8 - Methode zur Akti- und Deaktivierung der Gravitation

```
private int factor = 1;

private const float GRAVITATION = 9.81f;
```

Abbildung 7

einfach zu akti- und deaktivieren, indem man ihn entweder auf eins oder null setzt (Siehe Abb.: 8).

Um die Kraft nun zu berechnen und dem Mond zuzuweisen benötigen wir die von Unitys MonoBehaviour Klasse zur Verfügung gestellte FixedUpdate Methode. Diese wird alle 0.02 Sekunden aufgerufen.

Die Richtung der Kraft lässt sich mit einfacher Vektorrechung berechen und die Distanz, zwischen den beiden Himmelskörpern, wird uns durch eine Methode der Vector3 Klasse gegeben (Siehe Abb.: 9).

```
0 references
private void FixedUpdate()
{
    // get directions vector3 toward earth and distance
    Vector3 direction = (earth.position - moon.position).normalized;
    float distance = Vector3.Distance(a: earth.position, b: moon.position);

    // apply the gravitation to the moon
    rbMoon.AddForce(direction * AttractionForce(rbEarth.mass, rbMoon.mass, distance));
}
```

Abbildung 9

Die kalkulierten Werte können wir nun benutzten um die Kraft, mit unserer vorbereiteten Methode zu berechnen und sie dann auf dem Mond wirken lassen.

Mit diesem Script allein konnte man die Gravitation schon ausprobieren. Wieß man alle Objekte den richtigen Variablen im Inspector zu und positionierte die Erde und den Mond in der Scene konnte man, wenn man das Spiel startete, beobachten wie der Mond sich in seine Umlaufbahn begibt.

Planeten Geschoss

Das Grundprinzip des Spiels ist es den Mond in die Umlaufbahn der Erden zu schießen. Da die Erde nun eine Umlaufbahn besaß, brauchte es noch einen Script der den Mond schoss.

Die Idee war es durch das Drücken einer Taste den Mond an der rechten Hand des Spielers zu positionieren. Durch das gedrückt halten der gleichen Taste soll sich die Kraft, mit der der Mond am Ende geschossen wird, langsam aufbauen. Für eine gute Kontrolle über diese Kraft, muss diese natürlich auch dem Spieler visuell dargestellt werden.

Die Referenzen, die man dafür braucht, sind: Der Planet/Objekt, das geschossen wird, die Gravitationsklasse und der Rigidbody des Geschossobjektes.

```
[SerializeField] private Transform planet = null;
[SerializeField] private Gravitation gravitation = null;
[SerializeField] private float factorIncreace = 0.2f;
[SerializeField] private float forceMax = 5000;
[SerializeField] private TMP_Text forceText;

private Rigidbody planetRb;
private bool isReset;
private readonly KeyCode actionKey = KeyCode.Space;
private float force = 1f;
```

Abbildung 10 - Alle Variablen, die für den Planeten Schuss Script benötigt werden

Zunächst muss man erkennen ob der Spieler diese Taste, welche wir auch im Vorhinein als Variable definieren (Siehe Abb.: 10), drückt oder nicht. Dies machen wir in der Update Methode, welche auch von der Engine automatisch ausgeführt wird und das jeden abgespielten Frames des Spiels (Siehe Abb.: 11).

```
private void Update()
{
    forceText.text = force.ToString(format: "F");

    if (Input.GetKeyDown(actionKey) && !isReset)
    {
        ResetPlanetPosition();
    }

    if (Input.GetKey(actionKey) && isReset)
    {
        force *= factorIncreace;
        Debug.Log(force);
    }
    else if (Input.GetKeyUp(actionKey))
    {
        Shoot(force);
    }
}
```

Abbildung 11

Hier hakte es and meiner Taktik oder an meinem Verständnis von Unitys API. In der Theorie sollte der Code folgendermaßen laufen: Wenn die Taste einmal gedrückt wird und der geschossene Planet sich

nicht gerade in der Hand des Spielers befindet wird dieser zurückgesetzt und in die Hand des Spielers befördert. Wenn das Geschoss jedoch schon zurückgesetzt ist und die Taste wird gedrückt, soll sich die Kraftvariable um einen bestimmten Faktor vermehren. Dies wird solange gemacht bis die Taste wieder losgelassen wird. Geschieht dies wird die der Planet mit der aufgebauten Kraft geschossen. In Realität wird jedoch oft der Planet nach einem Tastendruck zurückgesetzt aber danach sofort wieder weg katapultiert, weil Tastendruck vermutlich (aus welchen Gründen auch immer) beide Methoden gleichzeitig triggert.

Die Methode die den Planeten wieder zurücksetzt bereitete auch Schwierigkeiten.

```
public void ResetPlanetPosition()
{
    // disable gravity of earth
    gravitation.EarthAttract( mode: false);

    // set the planets position and rotation to the guns
    planet.parent = transform;
    planet.localPosition = Vector3.zero;
    planet.localRotation = Quaternion.identity;

    // reset its velocity and the force factor
    planetRb.velocity = Vector3.zero;
    isReset = true;
    force = 1;
}
```

Abbildung 13

```
planet.SetPositionAndRotation(transform.position, transform.rotation);
```

Abbildung 12

Genauer gesagt die Positionierung des Planeten an der korrekten Position bei der rechten Hand. Ob man den Planeten ein Child Objekt des Handobjektes macht und die lokale Position und Rotation auf null setzt (Siehe Abb.: 13) oder die globale Position des Planten der der Hand gleich setzt (Siehe Abb.: 12) änderte nichts an dem Ergebnis, dass die Position des Planeten immer ein bisschen falsch war. Unterschiedlich genug um das Spielerlebnis zu beeinträchtigen und gering genug um überhaupt keinen Sinn zu ergeben. (Siehe Abb.: 14)

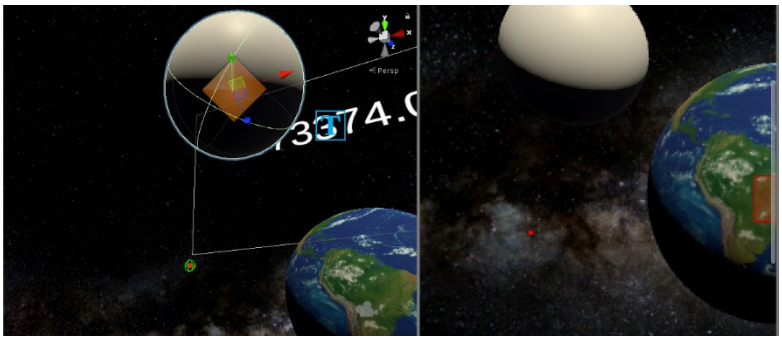

Abbildung 14 – Hand Objekt in Rot, Mond befindet sich in zurück gesetzter Position

Die Funktion die den Planeten schlussendlich Schoss funktionierte wie vorgesehen.

```
private void Shoot(float zForce)
{
    // unparent planet
    planet.parent = null;

    // enable gravity
    gravitation.EarthAttract(mode:true);

    // set min and max force
    zForce = zForce > forceMax ? forceMax : zForce;

    // shoot the planet
    planetRb.AddForce(transform.forward.normalized * zForce);
    Debug.Log(message:"Shot with " + zForce + " Force!");
    isReset = false;
}
```

Abbildung 15

Die Maximale Kraft führte ich zu einem späteren Zeitpunkt hinzu, da diese oft zu groß wurde. Normalerweise würde man so ein Limit mit der Clamp Funktion der Mathematikklasse von Unity machen. Diese Weise funktionierte in der Anwendung jedoch nicht. Meine eigene Lösung ist eine simple Abfrage ob die Kraft größer als ein Maximum ist, wenn das zutrifft wird die Kraft einfach auf dieses Maximum gesetzt.

Game Over

Das Spiel kann zwar schon gespielt werden und macht zu diesem Zeitpunkt auch schon Spaß, man jedoch noch nicht gewinnen oder verlieren. Das Verlieren wird zuerst gestrichen. Man kann in diesem Spiel nur gewinnen, es sei denn man gibt auf und hört auf zu spielen. Wie kann man also gewinnen?

Das Ziel ist es den Mond in die richtige Umlaufbahn zu schießen sodass er die gewollten Punkte im Raum passiert.

In Unity setzte ich das mit Hilfe von Trigger Boxen um. Es gibt insgesamt zwei davon. Wenn der Mond durch beide einmal hindurch geflogen ist hat man gewonnen. Erster Schritt zur Erstellung dieser Boxen ist die Objekte zu kreieren und ihnen einen Box Collider zuzuweisen. Wenn man sie dann skaliert und positioniert hat kann sich der Visualisierung widmen. Dafür habe ich mir aus dem Assetstore einen Script für eine Outline, welche durch andere Objekte sichtbar ist, heruntergeladen und den Boxen, die anderenfalls nicht sichtbar gewesen wären, zugewiesen.

Nun zur Funktionalität der Boxen. Dafür benötigte ich neue Klassen: Einen GameManager, welcher kontrolliert ob wir das Spiel gewonnen haben oder nicht (Siehe Abb.: 17) und eine TriggerDetektion für die Boxen, welche dem GameManager die Information gibt, ob sich das Planetengeschoss durch sie hindurchbewegt hat. (Siehe Abb.: 16)

```
public int triggersHit = 0;

0 references
private void Update()
{
    if (triggersHit == 2)
    {
        EndGame();
    }
}
```

Abbildung 17

```
private void OnTriggerEnter(Collider col)
{
    if (col.CompareTag(PLANET_TAG) && !hitBool)
    {
        gameManager.triggersHit += 1;
        Debug.Log(gameManager.triggersHit);
        hitBool = true;
    }
}
```

Abbildung 16

Fazit

Trotzt des sehr simplen Konzepts traten einige Probleme auf, die man nicht vorhersehen konnte. Das Spiel ist nicht vollendet, es bedarf noch einer Implementation der VR Button Steuerung.