ODDITY

4AHITM | MEDT | 2023/24

Buchinger
4AHITM

Bunea
(Projektleiter)
4AHITM

Langeneder
4AHITM

Inhaltsverzeichnis

Grundlagen und Methoden

Projektziel

Das Ziel des Projekts „Oddity“ besteht darin, ein einfaches low-poly[1] Raumschiffspiel zu entwickeln, welches am Infotag der IT-HTL Ybbs präsentiert wird. Der Spieler übernimmt die Kontrolle über ein Raumschiff und hat die Möglichkeit, damit zu fliegen und eine vorgegebene Rennstrecke zu absolvieren.

Methoden

Für die Erstellung eines Spiels wird eine Game Engine[2] benötigt. Obwohl zahlreiche Optionen zur Verfügung stehen, sind Unity und Unreal Engine die bekanntesten. Ursprünglich wurde Unity als erste Wahl in Betracht gezogen, da das Team bereits umfangreiche Erfahrung mit dieser Engine gesammelt hat. Aufgrund einer drastischen Änderung in der Preisstruktur von Unity und aus moralischen Erwägungen entschied man sich jedoch für Godot. Godot ist Open Source[3], und mit der Veröffentlichung von Godot 4 präsentiert es sich als eine attraktive Alternative zu Unity.

[1] Low Poly bezeichnet eine 3D-Modellierungstechnik mit minimalen Polygonen, die oft für stilisierte Grafiken oder zur Reduzierung der Rechenbelastung verwendet wird.
[2] Eine Software oder ein Toolkit, das speziell für die Entwicklung und Ausführung von Videospielen entwickelt wurde.
[3] Software, deren Quellcode öffentlich zugänglich ist und die oft kostenlos genutzt, modifiziert und verteilt werden kann.

Realisierung und Ergebnisse

Idee

Vor vier Jahren wurde mit Unity ein kleines Raumschiffspiel entwickelt. Das Genre hat stets Begeisterung ausgelöst, und es gab den Wunsch, ein solches Spiel selbst zu erstellen. Doch damals waren die technischen Kenntnisse und die Größe der Idee nicht ausbalanciert, sodass eine realistische Umsetzung schwierig war.

Anfang 2023 entstand die Idee für eine Kombination aus einem Raumschiffspiel und "Initial D", einem Anime über Straßenrennen. Das Hauptziel dieses Projekts ist es, eine einfache Demo dieser neuen Idee zu präsentieren.

Durchführung

Raumschiff-Modellierung

Das Raumschiff, welches der Spieler steuern wird, stellt ein zentrales Element des Spiels dar. Sein Design orientiert sich an früheren Spielkonzepten sowie an populären Raumschiffen der Science-Fiction, wie dem X-Wing aus "Star Wars" oder dem RSI Scorpius aus "Star Citizen". Das Modell bietet genügend Flexibilität, um später eventuell ein Landegestell oder Klappanimationen für die Flügel zu integrieren. Zudem wurden die Haupt-, Retro- und RCS-Triebwerke detailliert modelliert.

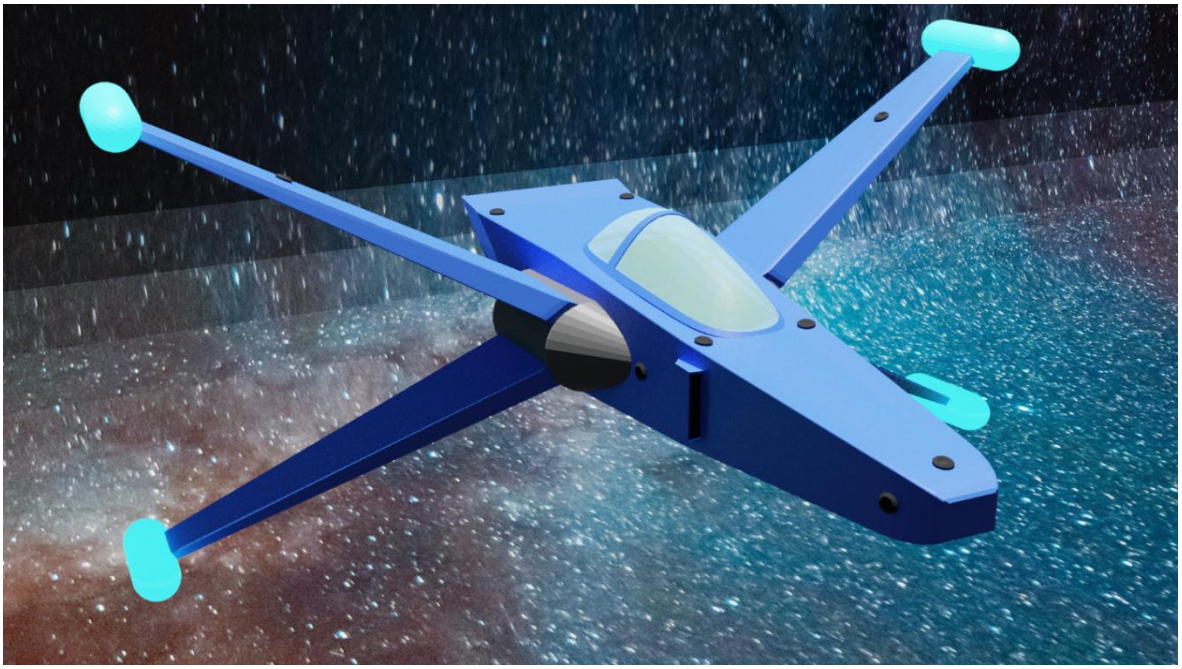

Abbildung 1: Raumschiff Render mit Blender

Raumschiff in Godot umsetzen

Das Raumschiff weist in Godot folgende Struktur auf:

Abbildung 2: Raumschiffstruktur in Godot

Die Haupt-Node ist als Rigidbody3D[4] definiert.

In der Ship Node befindet sich das Mesh des Raumschiffs als glTF-Datei[5].

Die CollisionMesh Node verfügt über ein vereinfachtes Mesh des Raumschiffs. Mit der Funktion Create Multiple Convex Collision Siblings werden Kollisionsbereiche (Collider[6]) für das Raumschiff erzeugt.

[4] Ein physikalischer Körper in 3D, der sich so bewegt, dass es den Gesetzen der Physik folgt.
[5] Ein Dateiformat für 3D-Szenen und -Modelle.
[6] Dies bezieht sich auf eine Komponente, die feststellt, ob ein physisches Objekt mit einem anderen kollidiert.

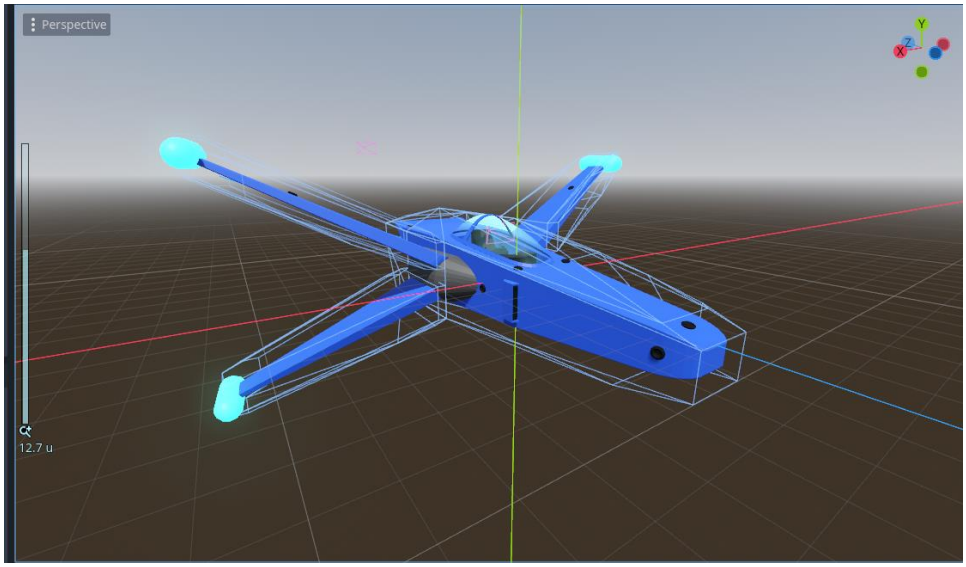

Abbildung 3: Raumschiff in Godot

- **UserControl**: Dieser Node ist für das Kontrollskript des Raumschiffs verantwortlich.
- **Cameras**: Dieser Node implementiert die Kamerafunktionalität.
- **Thrusters**: Hier werden Partikeleffekte implementiert, die mit den Schubdüsen des Raumschiffs in Zusammenhang stehen.
- Der Rigidbody3D FighterGen7-Node enthält das Skript, welches die Bewegung des Raumschiffs steuert.

Für das Mesh des Raumschiffs wurden die exportierten Materialien aus Blender verwendet. Für das Cockpit wurde in Godot ein separates Material erstellt. Es wurde die „Proximity Fade[7]“-Funktion aktiviert, um dem Material das Aussehen von transparentem Glas zu verleihen. Dies ist jedoch nur eine vorübergehende Lösung.

[7] Eine Funktion, die das Material wie transparentes Glas erscheinen lässt.

Ringe modellieren

Die Ringe wurden mithilfe von einen „Icosphere“ und einen „Cylinder“ erstellt.

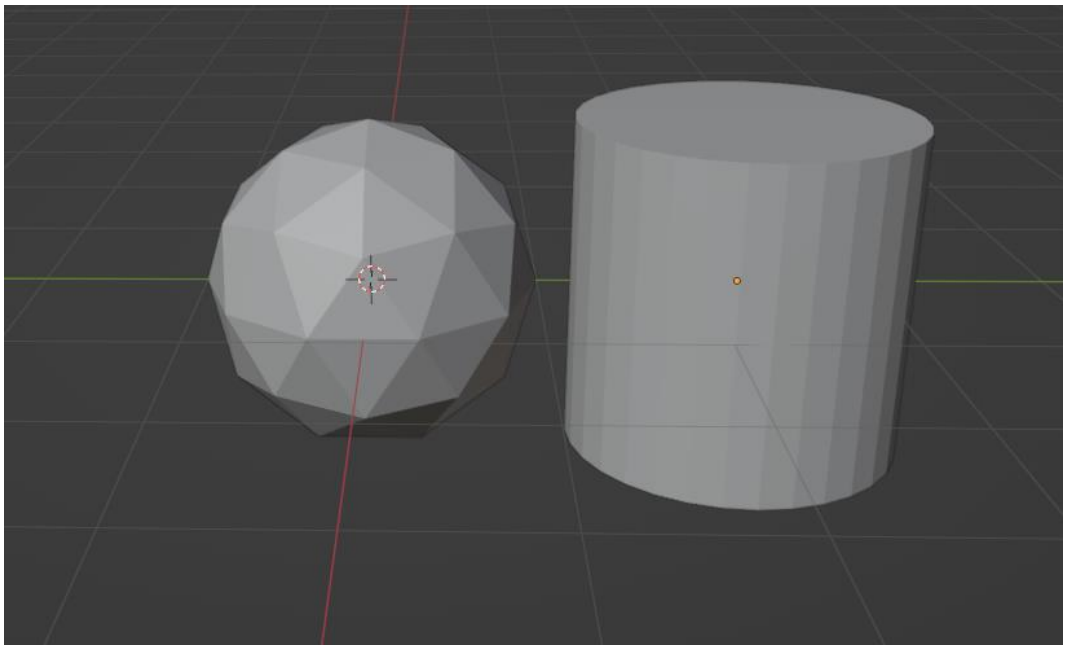

Abbildung 4: Icosphere und Cylinder

Die X-Achse der Icosphere wird verkleinert, sodass es fast schon zu einer Scheibe wird. Danach wird auch der Cylinder noch etwas verkleinert und er wird um 90° gedreht und in das Icosphere Objekt eingeschoben.

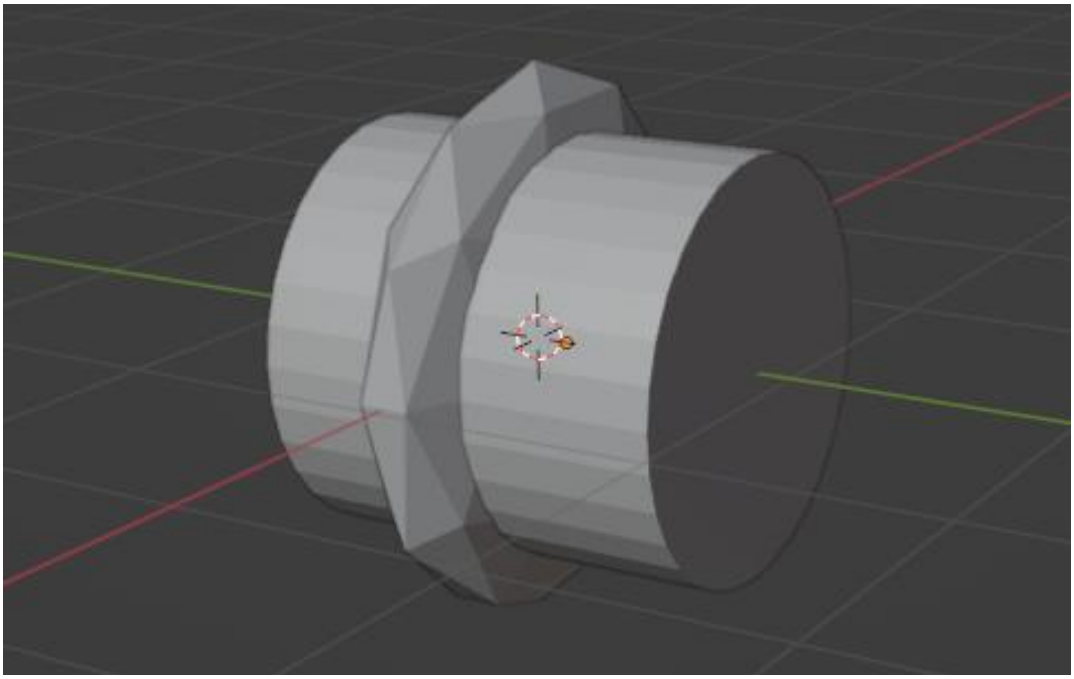

Abbildung 5: Icosphere

Als nächstes wird beim Cylinder ein „Boolean“ Modifier eingefügt. Dieser wird auf „Difference“ gestellt. Bei „Object“ wird anschließend der Cylinder ausgewählt. Wenn man danach den Cylinder ausblendet, ist der Icosphere bereits zu einem Ring geworden.

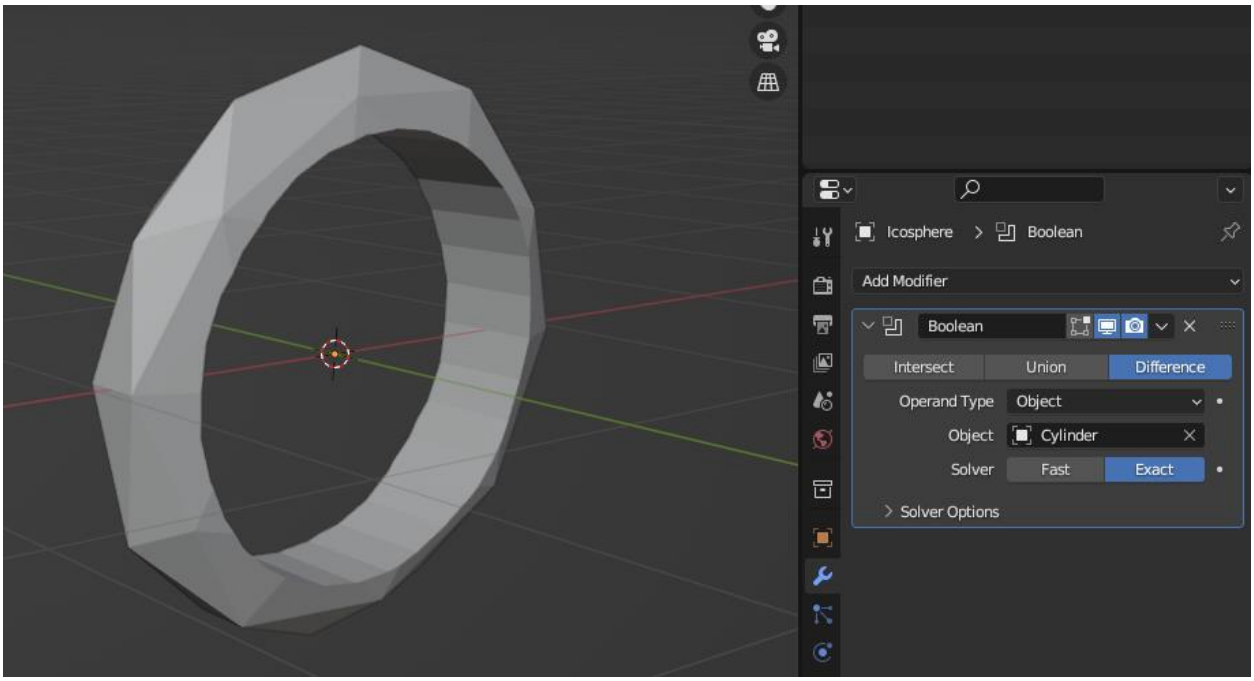

Abbildung 6: Boolean Modifier

Dieser Modifier wird anschließend angewandt. Zum Schluss wird noch beim Ring ein „Material“ angelegt. Unter „Surface“ wird die Base Color auf einen leichten Grauwert gestellt. Danach wird der Wert bei „Metallic“ auf den erwünschten Wert erhöht. Am Ende sieht der endgültige Ring folgend aus:

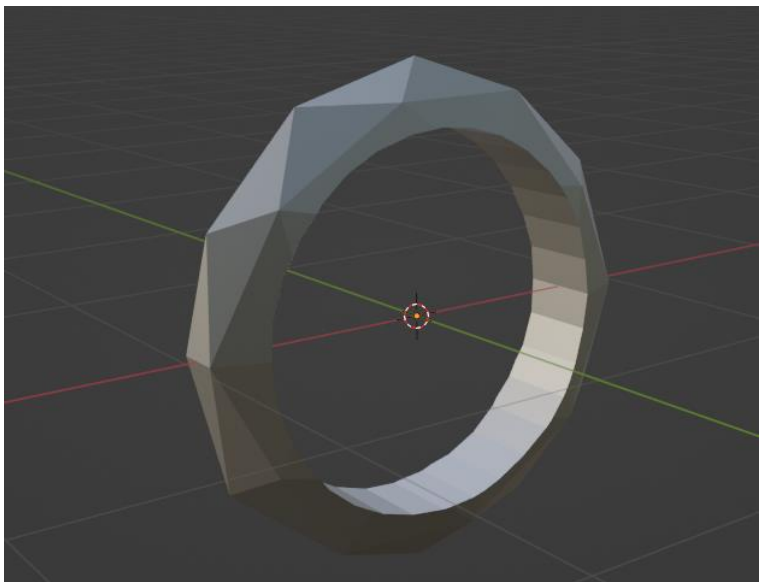

Abbildung 7: Ring

Asteroiden modellieren

In Blender wird zu Beginn durch Drücken von „Shift-A“ ein neues Mesh-Objekt namens „Rock Generator“ hinzugefügt. Anschließend wird in den Einstellungen im Bereich „Presets“ die Voreinstellung von „Default“ auf „Asteroids“ geändert. Das Kontrollkästchen bei „Use a random seed[8]“ wird deaktiviert. Danach wird der entsprechende „User seed“ ausgewählt.

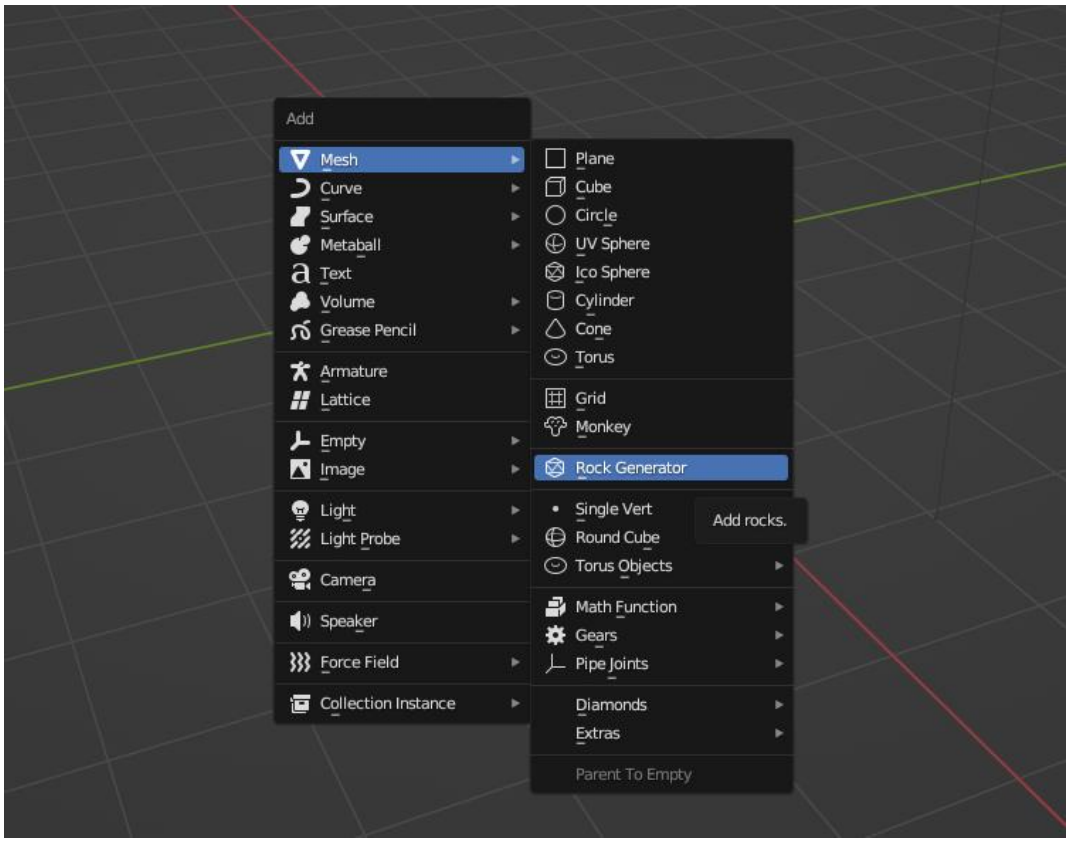

Abbildung 8: Rock Generator einfügen

[8] Ein Eingabewert oder eine Zahlenfolge, die den Zufallsgenerator in einer Softwareanwendung beeinflusst, um konsistente und wiederholbare Ergebnisse zu erzeugen.

Nachdem dieser Schritt abgeschlossen ist, wechselt man in den Shading-Modus. In den „Material-Einstellungen“ wird über „Add“ eine „Image Texture“ hinzugefügt. Anschließend wird das entsprechende Bild ausgewählt und gemäß dem folgenden Bild korrekt verknüpft.

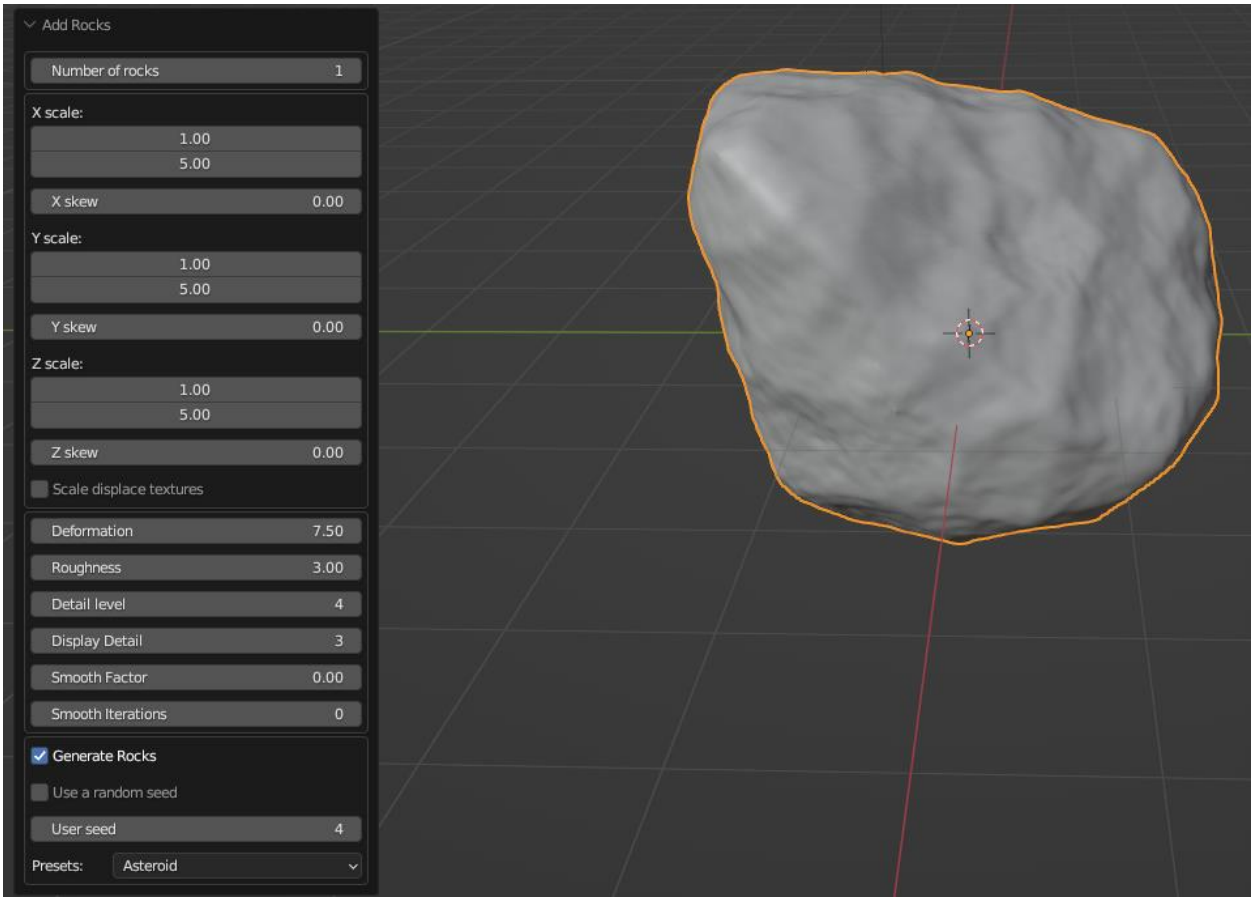

Abbildung 9: Passender User Seed auswählen

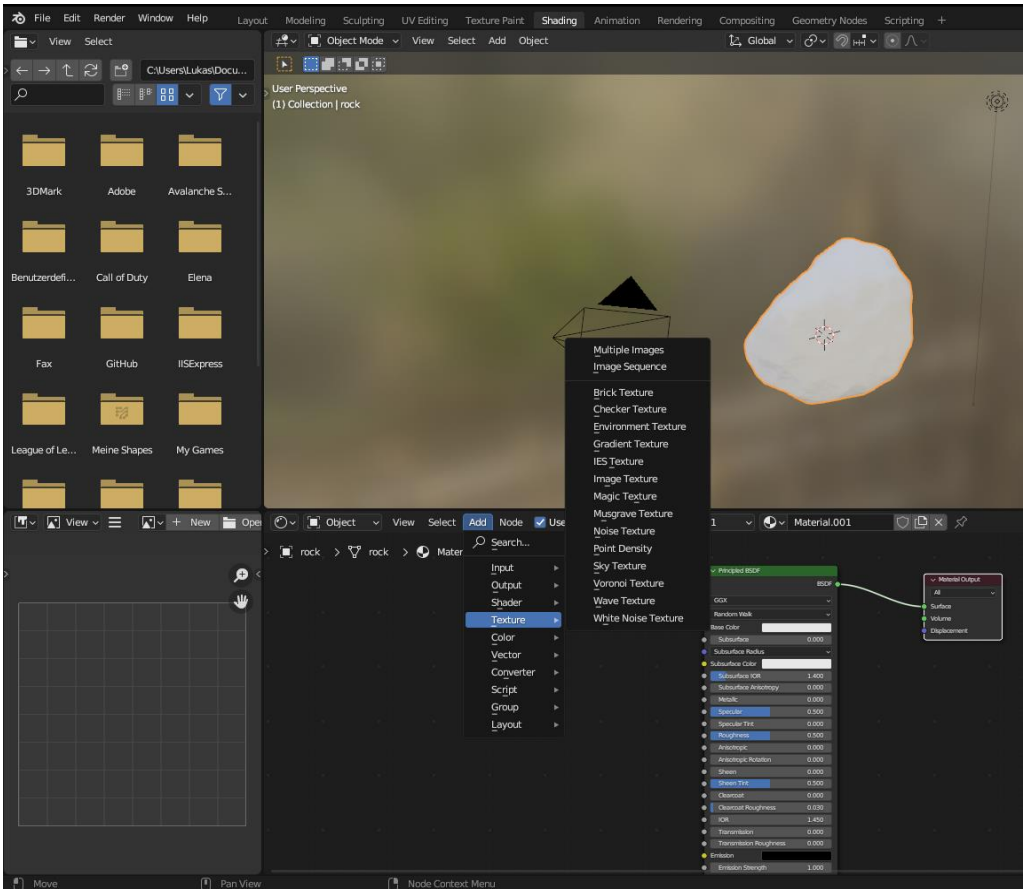

Abbildung 10: Image Texture

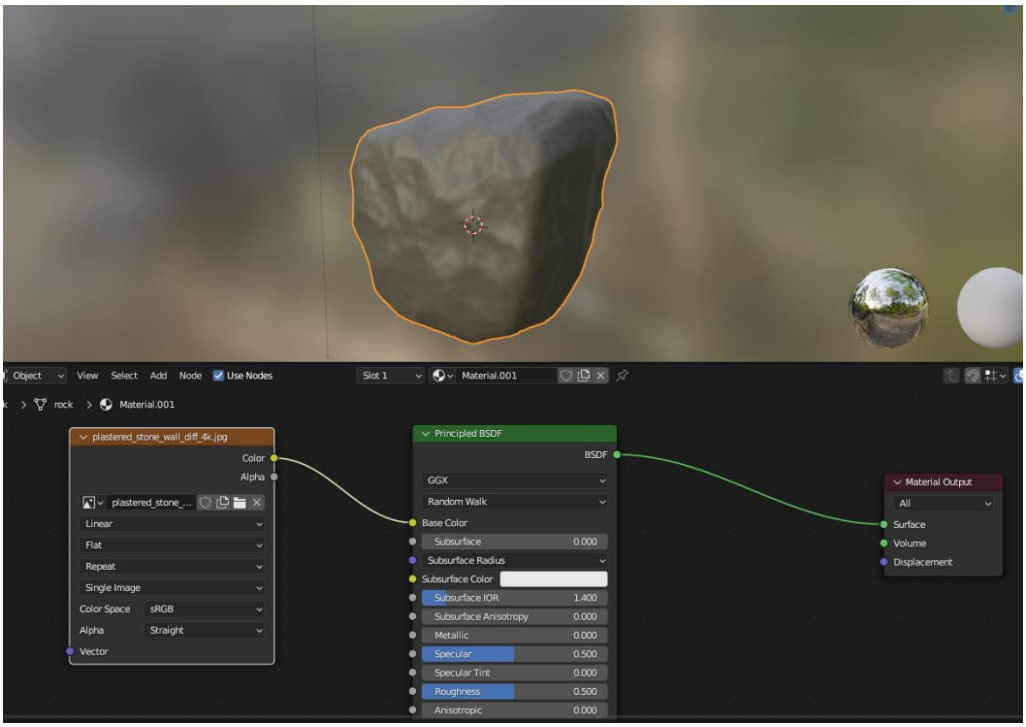

Abbildung 11: Entsprechende Textur auswählen

Asteroiden in Godot

In Godot werden die Asteroiden als StaticBody3D erstellt und bekommen ein Skript, das die Asteroiden zufällig dreht.

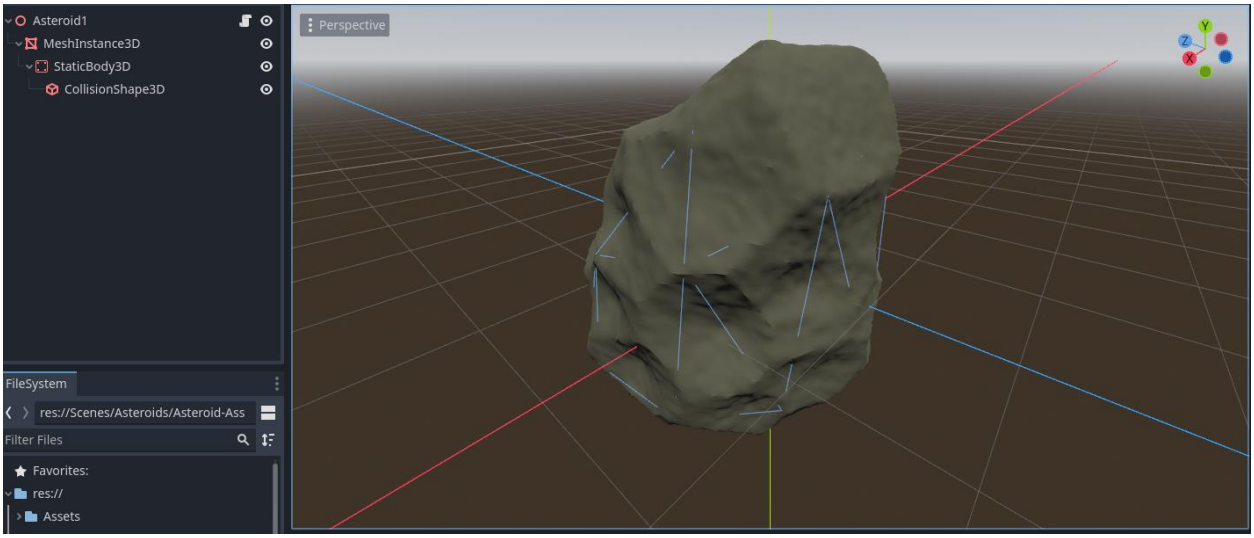

Abbildung 12: Asteroid in Godot

```
extends Node3D

var final_rotation : Vector3

# Called when the node enters the scene tree for the first time.
func _ready():
    var rng = RandomNumberGenerator.new()
    var rotation_direction = Vector3(rng.randf_range(-1, 1), rng.randf_range(-
1, 1), rng.randf_range(-1, 1)).normalized()
    var rotation_speed = rng.randf_range(0.1, 1.5)


    final_rotation = rotation_direction * rotation_speed

# Called every frame. 'delta' is the elapsed time since the previous frame.
func _process(delta):
    rotate(Vector3(1, 0, 0), final_rotation.x * delta)
    rotate(Vector3(0, 1, 0), final_rotation.y * delta)
    rotate(Vector3(0, 0, 1), final_rotation.z * delta)
```

Flugmodell

Das Flugmodell **Oddity** bestimmt, wie das Raumschiff gesteuert wird und wie sich das Raumschiff im Spiel verhält. Viele Spiele dieses Genres verwenden ein „Planes-in-Space“-Flugmodell. Ein Beispiel von einem Spiel mit einem solchen Flugmodell wäre „No Man’s Sky“. „Planes in Space“ bedeutet, dass Raumschiffe wie Flugzeuge fliegen. Das Flugmodell von Oddity ist von realistischen Spielen wie „Star Citizen“ und „Elite: Dangerous“ inspiriert. Man hat 6DOF und wie das Raumschiff fliegt ist realistisch simuliert.

Das Flugmodell ist auf zwei Skripte geteilt: „UserControl.gd“ und „SimpleFlightControlSystem.gd”.

UserControl.gd

Wie der Name sagt, ist dieses Skript für Spielerinput gedacht. Das Skript reagiert auf Tastendrücke und sendet ein Signal mit einem Wert zwischen 0 und 100 zu dem „SimpleFlightControlSystem“ Skript. Dieses ist dann dafür verantwortlich, das Raumschiff zu bewegen.

Mit dieser Unterteilung können wir im „SimpleFlightControlSystem“ Skript die Input Methode komplett ignorieren. Dies macht die Programmierung einfacher.

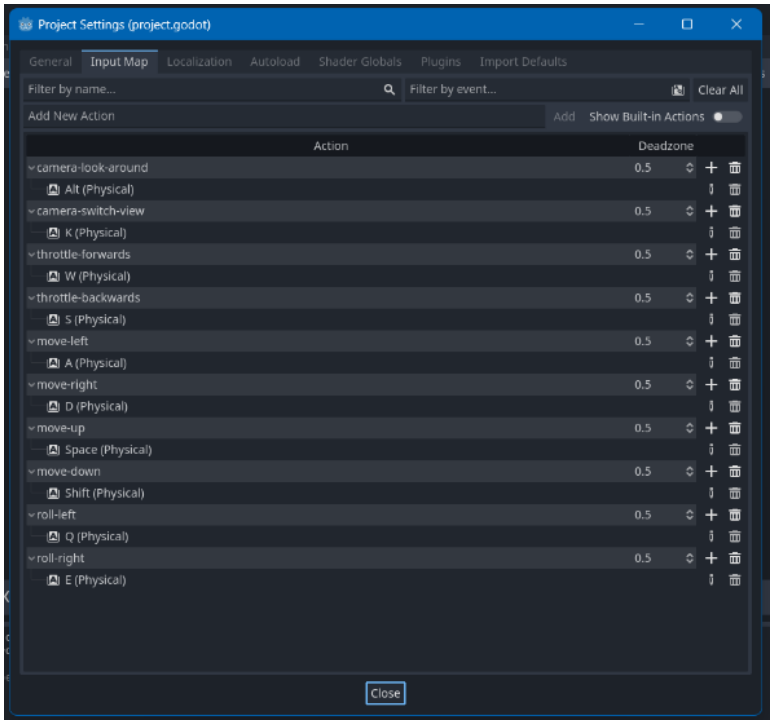

Abbildung 13: Input Map

Momentan wird nur Kontrolle über das Keyboard und der Mouse unterstützt. Der Spieler kann das Raumschiff mit W, A, S, D, Q, E, Shift und Space kontrollieren. Beim Drücken der jeweiligen Tasten wird ein Signal gesendet.

Das „SimpleFlightControlSystem“ Skript erhält beim Tastendruck einen Wert von 100 für die jeweilige Richtung. Bei der Implementierung von analogen Inputs, können wir einen beliebigen Wert von 0 bis 100 senden.

```
if (Input.is_action_pressed("move-down")):
      thrust_down.emit(100)
else:
      no_thrust_down.emit()
```

Die Drossel wird mithilfe der Tasten W und S gesteuert. Im Unterschied zu den anderen Tasten wird hierbei kein fester Wert von entweder 0 oder 100 übertragen, sondern es wird eine analoge Eingabe simuliert. Wenn die Taste W gedrückt wird, steigt der übertragene Wert bis auf 100 an, und wenn die Taste S gedrückt wird, sinkt der Wert bis zu -100. Dies ermöglicht eine stufenlose Steuerung der Drossel und erlaubt eine präzisere Steuerung.

```
func _process(delta):
    if (Input.is_action_pressed("throttle-forwards")):
        throttle = throttle + throttle_sensitivity * delta
        throttle = clamp(throttle, -100.0, 100.0)

    if (Input.is_action_pressed("throttle-backwards")):
        throttle = throttle - throttle_sensitivity * delta
        throttle = clamp(throttle, -100.0, 100.0)

    if (throttle < throttle_deadzone and throttle > -throttle_deadzone):
        if (timer.is_stopped()):
            timer.start()

    if (throttle > 0):
        thrust_forwards.emit(throttle)
    else:
        thrust_backwards.emit(throttle)
```

Mit der Maus werden Pitch und Yaw, gesteuert. Die Maus-Bewegungen werden addiert und in einem Wertebereich zwischen -100 und 100 gebracht.

```
func _input(event):
    if event is InputEventMouseMotion:
        mouse_yaw += lerp(0,1,clamp(event.relative.x *
get_process_delta_time(),-1,1)) * 5
        mouse_pitch += lerp(0,1,clamp(event.relative.y *
get_process_delta_time(),-1,1)) * 5
```

Verbesserungsmöglichkeiten

Momentan sendet das Skript für jeden Tastendruck ein separates Signal. Idealerweise würde man ein Vector, der in die gewünschte Richtung zeigt, senden.

SimpleFlightControlSystem.gd

Dieses Skript erhält die Eingabe Signale vom UserControl Skript und steuert die Bewegung des Raumschiffs.

Allgemein, dieses Skript bekommt die gewünschte Bewegung und passt die Geschwindigkeit und das Drehmoment an, damit das Raumschiff in der gewünschten Richtung fliegt.

Der User sendet für jede Richtung und Rotation eine gewünschte Bewegung. Diese kann entweder als Prozent der maximalen Geschwindigkeit oder der maximalen Beschleunigung in der jeweiligen Richtung gesehen werden. Zukünftig sollte der User dies selbst definieren können, aber momentan wird eine Mischung angewendet. Das SimpleFlightControlSystem steuert die Triebwerke des Raumschiffs, um die gewünschte Geschwindigkeit oder Beschleunigung in der gewünschten Richtung zu erreichen.

Die Flugcharakteristik eines Raumschiffs wird durch die maximale Kraft, die die Triebwerke liefern können, die maximale Drehgeschwindigkeit und die maximalen Geschwindigkeiten definiert.

Abbildung 14: SimpleFlightControlSystem.gd

Das SimpleFlightControlSystem hält sich an diese Parameter.

Beispiele vom Code:

```
if (yaw == false and flight_assist):

    var throttle = calc_torque_throttle(local_angular_velocity.y)

    if (local_angular_velocity.y > 0):
        fire_thrusters_yaw(throttle)
    else:
        fire_thrusters_yaw(-throttle)

if (yaw == true and flight_assist):

    var throttle = calc_torque_throttle(local_angular_velocity.y)

    if (abs(local_angular_velocity.y) >
calc_speed_at_percentage(abs(user_yaw), max_yaw_velocity)):

        if (local_angular_velocity.y > 0):
            fire_thrusters_yaw(throttle)
        else:
            fire_thrusters_yaw(-throttle)
```

```
if (thrust_up == false and thrust_down == false and flight_assist):
    var throttle = calc_throttle(velocity.y)

    if (velocity.y > 0):
        fire_thrusters_down(throttle)
    else:
        fire_thrusters_up(throttle)

if (throttle_based_on_max_speed):
    if (-velocity.z > calc_speed_at_percentage(user_throttle,
max_total_velocity) and velocity.z < 0):
        fire_thrusters_retro(-calc_throttle(abs(velocity.z)))
    elif (velocity.z > calc_speed_at_percentage(-user_throttle,
max_total_velocity) and velocity.z > 0):
        fire_thrusters_forwards(calc_throttle(abs(velocity.z)))
else:
    if (user_throttle >= 0):
        var throttle = calc_throttle(velocity.z)

        if (velocity.z > 0):
            fire_thrusters_forwards(throttle)
```

Es gibt hier einige Schichten von Abstraktion.

Die fire_thruster_x Methoden steuern die Triebwerke. Diese sind eine Vereinfachung von der apply_central_force Methode von Godot. Somit muss man sich nicht um den Richtungsvektor und sonstige Variablen kümmern.

```
func fire_thrusters_forwards(throttle):
    apply_local_thrust(-transform.basis.z, max_thrust_main, throttle)


func apply_local_thrust(direction: Vector3, force: float, throttle: float):
    apply_central_force(direction * force * (throttle / 100.0) *
get_process_delta_time() * thrust_multiplier)

func apply_local_torque(direction: Vector3, force: float, throttle: float):
    apply_torque(direction * (throttle / 100.0) * force *
get_process_delta_time() * torque_multiplier)

func calc_local_velocity():

## https://ask.godotengine.org/123178/how-do-i-get-local-linear-velocity ##

    var b = transform.basis
    var v_len = linear_velocity.length()
    var v_nor = linear_velocity.normalized()

    velocity.x = b.x.dot(v_nor) * v_len
    velocity.y = b.y.dot(v_nor) * v_len
    velocity.z = b.z.dot(v_nor) * v_len

func calc_local_angular_velocity():
    local_angular_velocity = transform.basis.inverse() * angular_velocity
```

Lokale Geschwindigkeiten werden auch manuell errechnet, da Godot keine eingebaute Funktion dafür hat.

Kamera

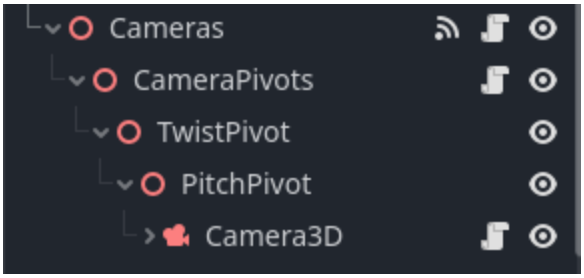

Abbildung 15: Kamera Nodes

Kamera Bewegung

Während der Spieler die „Look Around“ Taste (standardmäßig ALT) gedrückt hält, kann er herumschauen.

```
func _process(delta):
    if Input.is_action_pressed("ui_cancel"):
        Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)

    if Input.is_action_pressed("camera-look-around"):
        twist_pivot.rotate_y(twist_input)
        pitch_pivot.rotate_x(pitch_input)

    twist_input = 0
    pitch_input = 0

func _input(event):
    if event is InputEventMouseMotion:
        if Input.get_mouse_mode() == Input.MOUSE_MODE_CAPTURED:
            twist_input = -event.relative.x * mouse_sensitivity
            pitch_input = -event.relative.y * mouse_sensitivity
```

Die Twist- und Pitch-Pivots werden jeweils mit der entsprechenden Mausbewegung um die Y- und X-Achse gedreht.

Diese Separation der Bewegungen hilft mit der Implementierung der Kamera Effekte.

Kamera Erste / Dritte Person

Die Kamera kann zwischen einer Innen- und Außenansicht wechseln.

Dies geschieht momentan mit einer einfachen Animation, bei der die Kamera Position fix eingestellt ist. Zukünftig sollte die Kamera Position einstellbar sein.

```
func _input(event):
    if Input.is_action_just_pressed("camera-switch-view"):

        if (is_first_person and not animation.is_playing()):

            animation.play("SwitchCameraToThirdPersonView")

            set_default_position.emit(third_person_position)
            is_first_person = false
            is_first_person_signal.emit(is_first_person)

        elif (not is_first_person and not animation.is_playing()):

            animation.play_backwards("SwitchCameraToThirdPersonView")

            is_first_person = true
            set_default_position.emit(first_person_position)
            is_first_person_signal.emit(is_first_person)
```

Kamera Effekte

Um den Spieler ein Gefühl der Bewegung zu geben, haben wir uns folgende drei Effekte überlegt:

- Je schneller man fliegt, desto größer wird das Sichtfeld (FOV)
- Je stärker man beschleunigt, desto mehr Screen-Shake
- Beim Beschleunigen wird die Kamera in die entgegengesetzte Richtung bewegt

Mit nur diesen drei Effekten kann man dem Spieler viel über was das Raumschiff gerade macht, mitteilen. Ohne auf irgendwelche Zahlen zu schauen, kann der Spieler seine Geschwindigkeit, wie viel und in welche Richtung er gerade beschleunigt.

Im CameraShake.gd Skript kann man jeden wichtigen Parameter ändern.

Abbildung 16: CameraShake.gd

Somit kann man zukünftig für neue Raumschiffe eine eigene Persönlichkeit beim Fluggefühl geben, ohne den Code zu ändern.

Code

```
func adjust_camera_position():
    position = position.lerp(acceleration + defaultPosition,
get_process_delta_time() * 2.0)

func adjust_fov():
    # Define the minimum and maximum speeds for mapping
    var min_speed = -400.0
    var max_speed = 400.0

    # Ensure that the speed is within the specified range
    var current_speed = clamp(speed, min_speed, max_speed)

    # Calculate the FOV based on speed using linear mapping
    fov = lerp(current_min_fov, current_max_fov, (current_speed - min_speed) /
(max_speed - min_speed))

func camera_shake():
    var offset = Vector2(randf(), randf()) * acceleration.length() /
current_camera_shake_strength
    h_offset = offset.x
    v_offset = offset.y
```

Das Sichtfeld wird durch eine lineare Interpolation gesetzt.

Der Screen Shake ist nur ein zufälliger Offset, der von der Beschleunigung und eines stärke Parameters abhängig ist.

Die Kamera Position wird der negativen Beschleunigung gleichgesetzt.

```
func LogarithmicTransform(vector):
    var transformedVector = Vector3()

    # Apply the logarithmic transformation to each component of the vector

   transformedVector.x = clamp(log(1 + abs(vector.x)) * sign(vector.x) *
current_acceleration_movement_factor, -current_max_movement, current_max_movement)
   transformedVector.y = clamp(log(1 + abs(vector.y)) * sign(vector.y) *
current_acceleration_movement_factor, -current_max_movement, current_max_movement)
   transformedVector.z = clamp(log(1 + abs(vector.z)) * sign(vector.z) *
current_acceleration_movement_factor, -current_max_movement, current_max_movement)

    return -transformedVector
```

Die kalkulierte Beschleunigung wird zuerst durch eine logarithmische Transformation gestellt, um ihn auf einem viel kleineren Wertebereich einzuschränken.

Gameplay

Das Gameplay ist momentan sehr einfach. Man muss nur durch eine Strecke von Ringen durchfliegen.

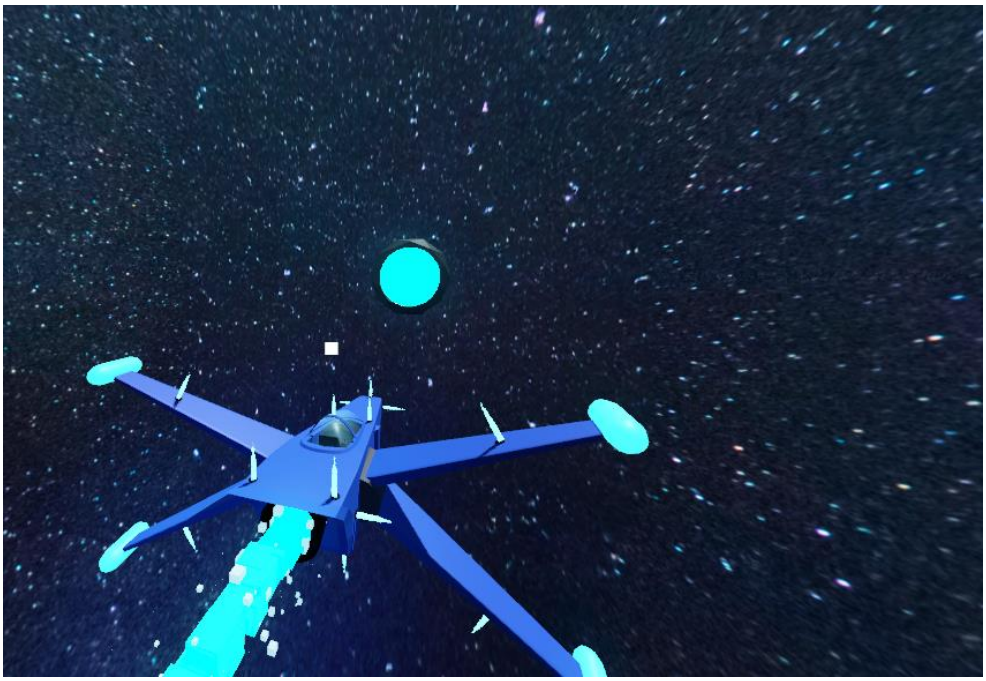

Abbildung 17:Gameplay

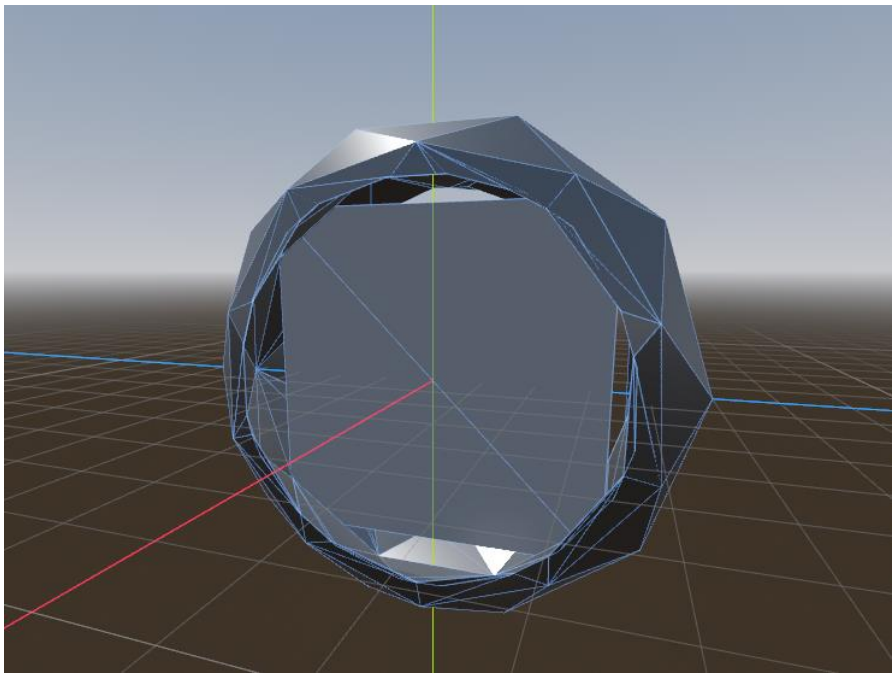

Abbildung 18: Ring in Godot

Die Ringe haben je einen Trigger in der Mitte, der beim Durchfliegen ein Signal senden.

```
func _on_area_3d_body_entered(body):
    course.ship_flew_through_ring()
```

Die Ringe sind alle in der Course Szene. Diese Szene beschreibt eine Strecke.

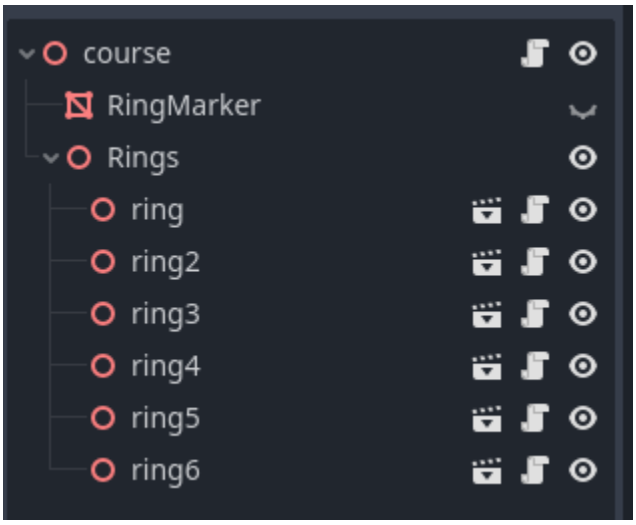

Abbildung 19: Ring Scene

Im course.gd Skript werden alle Ringe im „Rings“ Node zu einem Array hinzugefügt.

```
# Called when the node enters the scene tree for the first time.
func _ready():
    for _i in $Rings.get_children():
        course_rings.append(_i)
```

Beim Erhalten des Durchfliege Signals setzt dieses Skript einen Marker auf dem nächsten Ring in der Strecke.

```
func ship_flew_through_ring():
    if (rings_flown_through == 0):
        RingMarker.show()
    if (rings_flown_through == (course_rings.size() - 1)):
        rings_flown_through = 0
        RingMarker.hide()
    return 0;
rings_flown_through += 1
RingMarker.global_position = course_rings[rings_flown_through].global_position
```

Markierte Ringe werden mit diesem Effekt angezeigt.

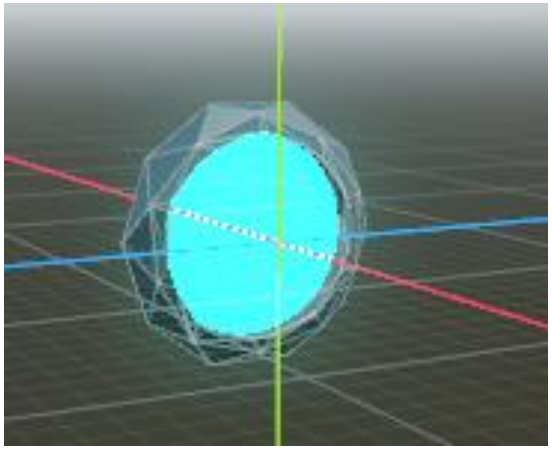

Abbildung 20:Ring Effekt Godot

Je näher man ist, desto durchsichtiger werden sie.

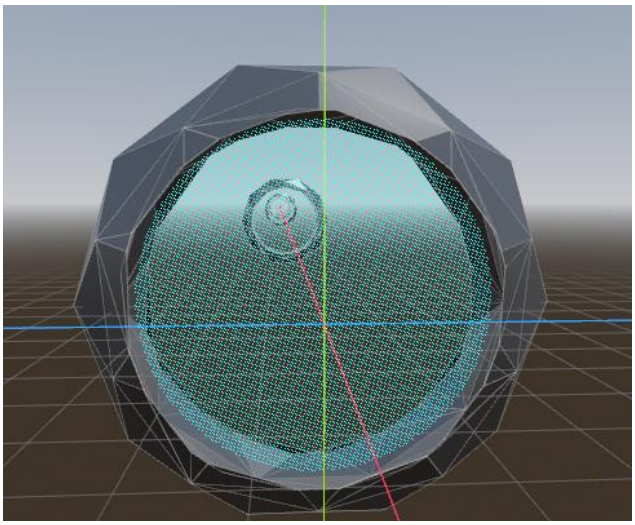

Abbildung 21: Annäherung an Ring

Grafik

Skybox

Eine Skybox ist eine 360°-Projektion eines Bildes um den Spieler. Bei der Erstellung der Skybox für das Projekt wurden folgende Schritte durchgeführt:

1. **Bildquelle**:

Ein Bild des Nachthimmels wurde von Unsplash ausgewählt und heruntergeladen.

Das Bild wurde aufgrund seiner dunklen Farbgebung gewählt, was die Erkennbarkeit von Objekten wie Ringen und Kometen im Spiel erleichtert.

https://unsplash.com/photos/photography-of-night-sky-L8126OwlroY

2. **Bildbearbeitung in Photoshop**:
   - Das heruntergeladene Bild wurde in Photoshop 2024 geöffnet.
   - Unter **Filter > Verzerren > Polar-Koordinaten**[9] wurde das Bild in eine 360-Grad-Ansicht konvertiert.
   - Eventuelle Unregelmäßigkeiten im Zentrum des Bildes wurden korrigiert.
   - Das generative Füllwerkzeug wurde verwendet, um Fehlstellen automatisch zu füllen.
   - Das Patch-Tool wurde genutzt, um zusätzliche Anpassungen vorzunehmen und das Bild zu perfektionieren.
3. **Bit-Tiefenänderung und Export**:
   - Im Menü wurde die Option ausgewählt, um die Bit-Tiefe des Bildes auf 32 Bit zu ändern.
   - Anschließend wurde das bearbeitete Bild als komprimierte EXR-Datei[10] exportiert.
4. **Einbindung in Godot:**

- Die exportierte EXR-Datei wurde in das Godot-Projekt importiert.
- Innerhalb von Godot wurde das Bild als Skybox-Material verwendet und entsprechend konfiguriert.

[9] Ein mathematisches Konzept und ein Filter in Photoshop, das verwendet wird, um ein Bild in eine 360-Grad-Ansicht zu konvertieren.
[10]Ein Dateiformat für Bilder, das eine hohe Farbtiefe und eine breite Palette von Helligkeitswerten unterstützt, ideal für Grafikanwendungen und VFX-Arbeiten.

Zusätzlich wird noch ein Stern hinzufügt.

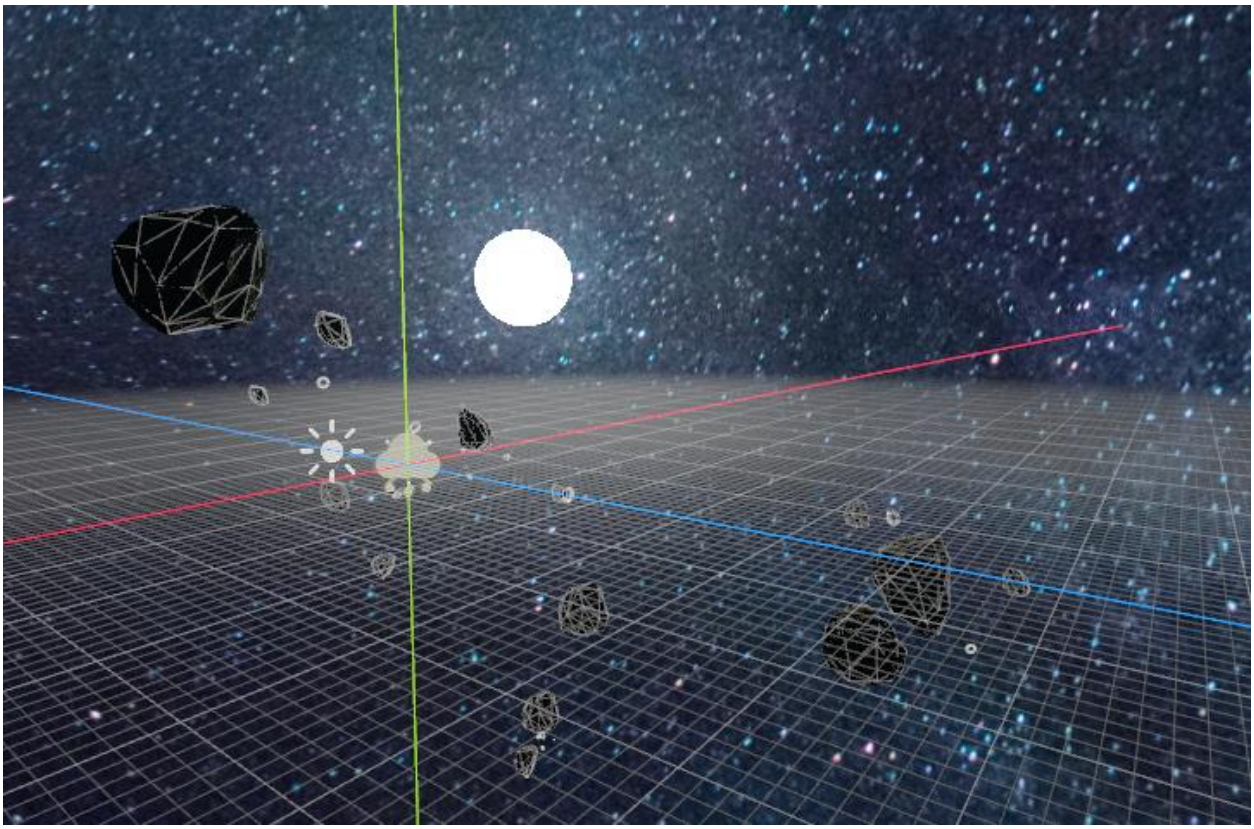

Abbildung 22: Stern in Godot

Dieser Stern ist nur ein sehr dünner Zylinder, mit einem leuchtenden Material.

Particles

Das Haupttriebwerk verfügt über zwei unterschiedliche Partikeleffekte: *InnerGlow* und *OuterGlow*. Jeder Thruster ist mit einem eigenen Partikelsystem ausgestattet, um Abgas- und Antriebseffekte visuell zu simulieren.

Partikelsystem in Godot

In Godot werden GPUParticles3D zum Generieren von Partikel verwendet. Diese können an Objekte oder in der Welt platziert werden.

Dokumentation 3D Particles: Godot Dokumentation 3D Particles

Partikelsysteme des Haupttriebwerks

1. **InnerGlow**

Der InnerGlow-Partikeleffekt visualisiert die zentrale Energiequelle des Haupttriebwerks und ist darauf ausgelegt, die intensive Energie des Antriebssystems darzustellen.

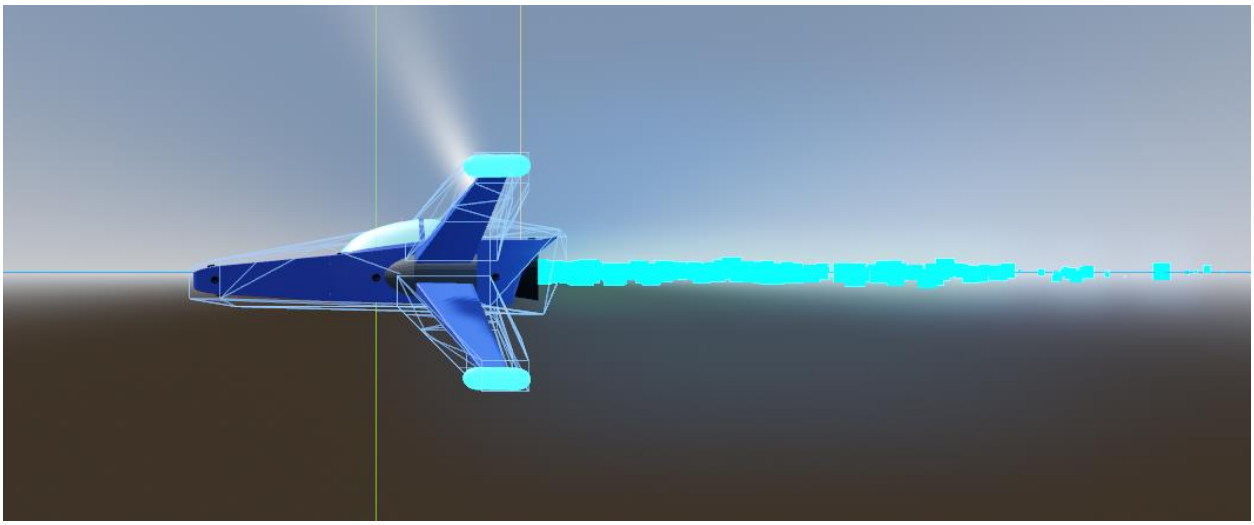

Abbildung 23: Main Thruster

Spezifikationen:

Emissionsrate: Sehr hoch eingestellt, um die erhebliche Energiemenge zu repräsentieren, die im Zentrum des Triebwerks erzeugt wird.

Lebensdauer der Partikel: Bewusst kurzgehalten, um die Fluktuation und die dynamische Natur der Kernenergie nachzuahmen.

Bewegungsverhalten: Die Partikel bewegen sich schnell und mit variierenden Geschwindigkeiten, um Turbulenzen und die ungezähmte Natur der Triebwerkskernenergie widerzuspiegeln.

Der InnerGlow-Effekt trägt entscheidend zur Visualisierung der rohen Kraft und des energetischen Pulses des Haupttriebwerks bei und unterstützt eine glaubwürdige Darstellung der Antriebsenergie.

2. OuterGlow

Der OuterGlow-Partikeleffekt repräsentiert den Abgasstrom, der vom Haupttriebwerk ausgeht. Der Effekt zielt darauf ab, das visuelle Erscheinungsbild des Abgases zu erzeugen, das sich hinter dem Triebwerk ausbreitet.

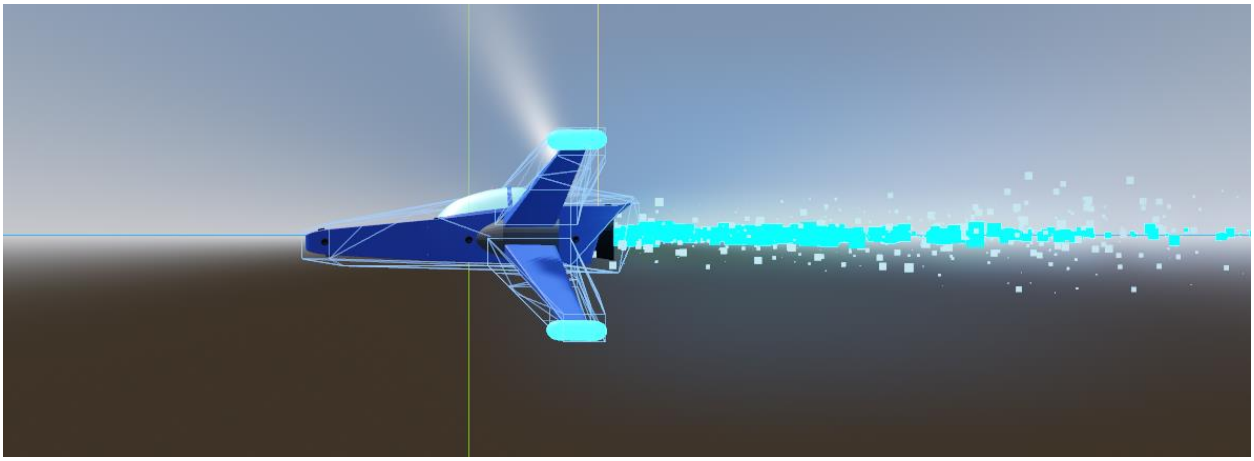

Abbildung 24: Main Thruster mit Smoke

Spezifikationen:

- **Emitter:** Nutzt einen Partikelemitter, der eine kontinuierliche Strömung von Partikeln erzeugt, um das Abgas darzustellen.
- **Emissionsrate:** Hoch genug, um eine konstante Abgasfahne zu simulieren, mit der Möglichkeit, die Rate für verschiedene Triebwerkszustände zu verändern.
- **Lebensdauer der Partikel:** Kurz aber mit hoher Geschwindigkeit, um den Eindruck von schnell expandierendem und sich auflösendem Abgas zu vermitteln.
- **Skalierung:** Die Partikel skalieren im Laufe ihrer Lebensdauer auf, um die Ausbreitung des Abgases mit der Entfernung vom Triebwerk zu imitieren.

Diese Konfiguration des OuterGlow-Partikeleffekts unterstützt die Visualisierung der Antriebskraft und Dynamik des Haupttriebwerks und trägt zu einer immersiven Simulationserfahrung bei.

Partikelsysteme der Düsen

Jede Düse ist mit einem spezifischen Partikelsystem ausgestattet, das darauf abzielt, die Triebwerksfunktion visuell darzustellen und die Reaktion auf unterschiedliche Betriebszustände zu simulieren.

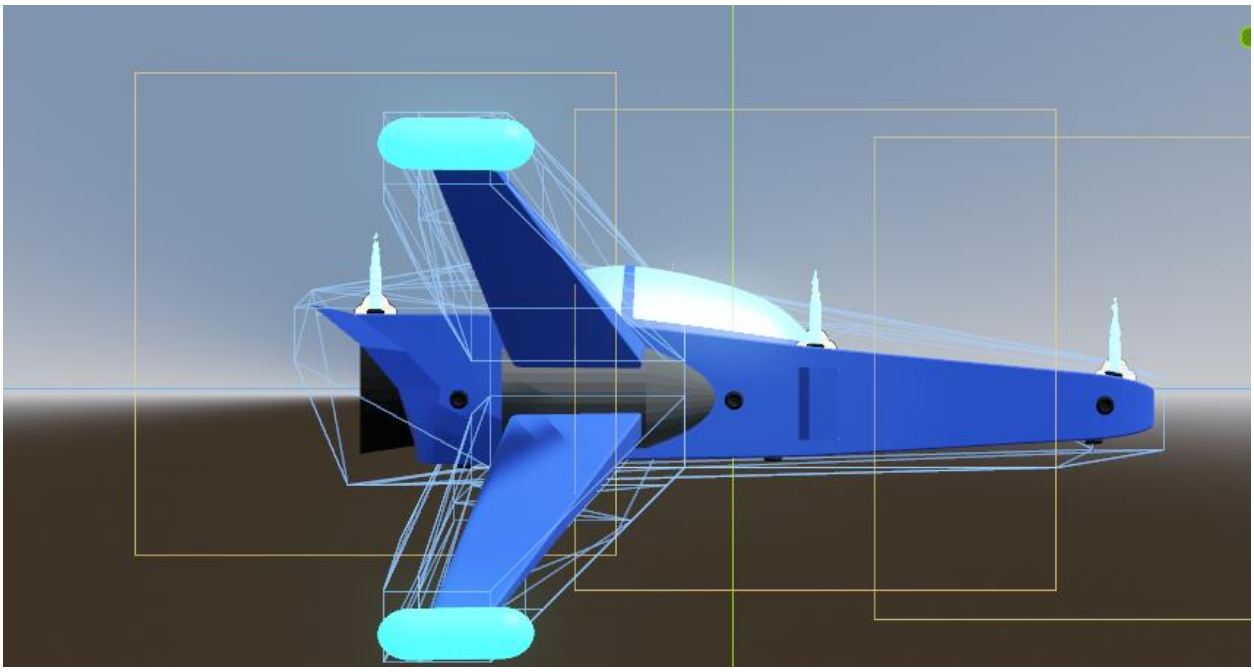

Abbildung 25: Düsen oben

Spezifikationen:

- **Partikelnamen:** Benannt nach ihrer genauen Positionierung an der Raumfähre, um eine eindeutige Identifizierung zu ermöglichen (z.B. wing_top_right für die Partikel der Düse oben rechts am Flügel).
- **Partikelbewegung:** Die Partikel sollten von der Düse aus in einer gerichteten Strömung emittiert werden, die den Ausstoß der Antriebsgase nachbildet.
- **Emissionsverhalten:** Variabel, um die Leistungsänderungen der Düsen zu reflektieren. Eine höhere Emissionsrate spiegelt eine gesteigerte Düsenaktivität wider, während eine niedrigere Rate einen reduzierten Schub signalisiert.

-

Partikelsystem rückwärts Düsen

- **Konfiguration**: Jede Rückwärtsdüse verfügt über ein vertikal orientiertes Düsenpartikelsystem.
- **Visualisierung**: Das Erscheinungsbild ist ähnlich dem des Hauptantriebs, jedoch in einer verkleinerten Form.
- **Zweck**: Diese Konfiguration dient dazu, Manövrierfähigkeit beim Rückwärtsflug zu ermöglichen und zu visualisieren.

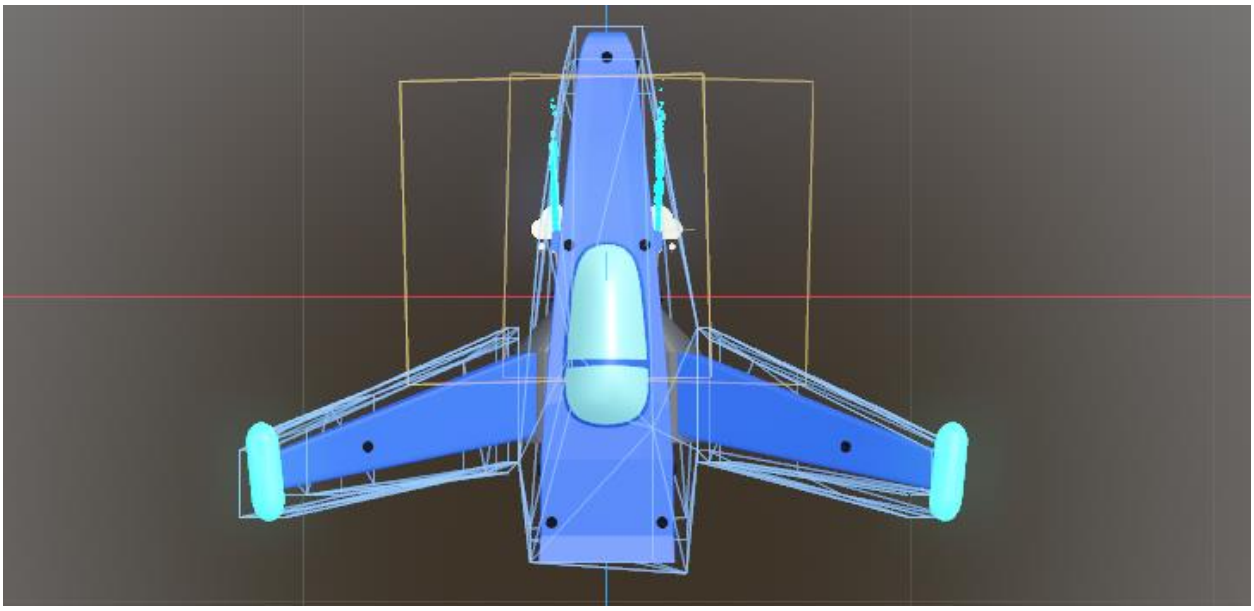

Abbildung 26´: Retro Thrusters

UI

Das User Interface wird dem Spieler drei Komponente Anzeigen:

1. **Geschwindigkeit:** Textanzeige der aktuellen Geschwindigkeit.
2. **Schub (Throttle):** Progress bar zeigt Schubkraft in %.
3. **Steuerung (Pitch/Yaw):** Kreisförmige Anzeige für Richtungsänderungen.

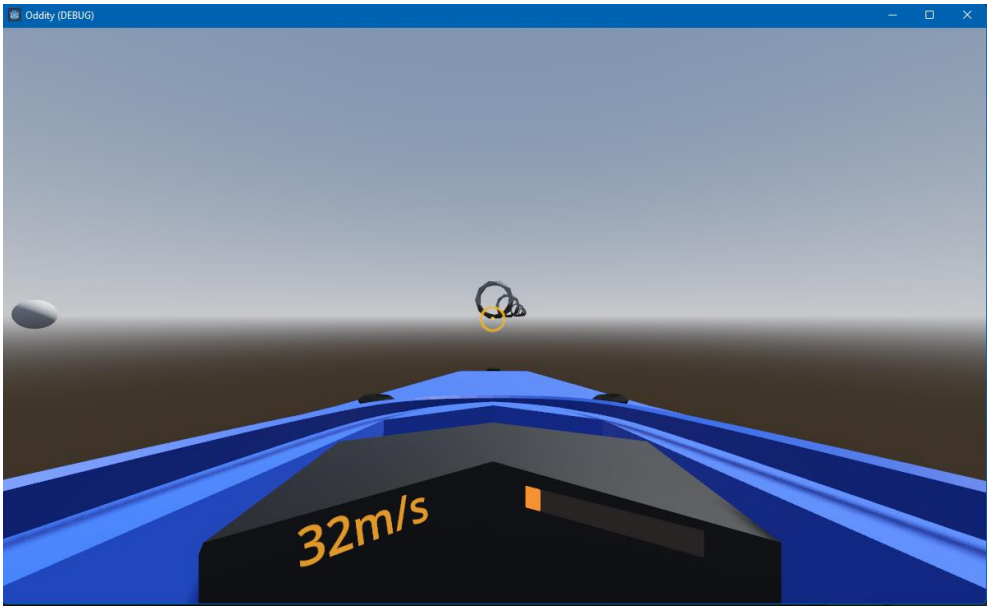

Abbildung 27: UI Oddity

Die Geschwindigkeit wird als einfacher Text dargestellt. Die Drosselposition wird in einer Fortschrittsanzeige angezeigt. Die Pitch- und Yaw-Bewegungen der Maus werden als bewegender Kreis visualisiert.

Zusammenfassung

Das Projekt „**Oddity**“ ist ein einfaches Raumschiffspiel, entwickelt für den Infotag an der IT-HTL Ybbs. Die Nutzung der Godot Game Engine war eine neue Herausforderung für uns, da diese noch nicht Teil unseres Unterrichts war.

Beim Design des Raumschiffs orientierten wir uns an bekannten Sci-Fi-Modellen, die wir in Blender modellierten. Obwohl Blender schon im Unterricht behandelt wurde, brachte die Integration in Godot einige Herausforderungen mit sich, vor allem bei der Steuerung. Wir hatten anfangs Schwierigkeiten, eine präzise und reaktionsfähige Mausbewegung zu implementieren, konnten diese jedoch erfolgreich meistern.

Abgesehen von einigen kleineren Herausforderungen, die schnell gelöst wurden, verlief das Projekt weitestgehend problemlos. Insgesamt haben wir das Projektziel erreicht und wertvolle Erfahrungen im Umgang mit Godot gesammelt.

Bibliography

Unity. (2023, 9 30). *Unity plan pricing and packaging updates*. Retrieved from https://blog.unity.com/news/plan-pricing-and-packaging-updates