

Nation Feathers

Omar Sharaki

18.04.17

Inhaltsverzeichnis

1. Motivation	1
2. Spielverlauf.....	1
3. Einsetzung von Unity	2
4. Implementierungsdetails	2
4.1. Das Gelände	2
4.2. Object Pooling	3
4.3. Vorüberlegungen zur Kameraorientierung	4
4.4. Collision Oracle.....	4
4.5. Kamera-Animation	5
4.5.1 Animationszustandsmaschinen und Blend Trees	6
4.5.2. Implementierung der Animation.....	6
4.5.3. Coroutinen	7
4.6. Bewegungsbeschränkung	8
4.6.1 Seitliche Bewegungsbeschränkung	8
4.6.2. Untere und vordere Grenzen	9
4.6.3. Obere Grenze	9
4.7. Nutzereingabe	9
4.7.1. Erste Ideen zur Eingabe	9
4.7.2. Analog-Stick	10
4.7.3. Weitere Eingabe Möglichkeiten	11
4.8. Kollision.....	11
5. Benutzeroberfläche	12
6. Audio	13
7. Ausblick und Fazit	14
Installationshinweise	15
Literaturverzeichnis	16
Asset- und Spielobjektquellen	16

1. Motivation

In der heutigen Welt besitzt fast jeder Mensch, ob alt oder jung, ob technisch begabt oder nicht, ein Smartphone. Diese leistungsfähigen, in einer Hosentasche tragbaren Geräte begleiten ihre Besitzer überall und nehmen die meiste Zeit ihre Aufmerksamkeit, wohl oder übel, in Beschlag. Allein diese Portabilität ist Grund dafür, dass Mobilgeräte eine der größten Interaktiven Medien Plattformen darstellen. Wer also z.B. ein Spiel für das größtmögliche Publikum zur Verfügung stellen will, tut gut daran, Smartphone-Unterstützung für sein Spiel in Betracht zu ziehen. Dies war Anregung für ein Spiel, das Spielern nicht nur viel Spaß bereitet sondern auch in Bezug auf Vogelkunde informativ sein sollte. Diese Kombination aus Lernen und Spaß ist kein leichtes Ziel und braucht viel Innovation. Und, obwohl die ursprüngliche Planungsphase diesen Bildungsaspekt im Visier hatte, ist die Kombination mangels Zeit nicht gelungen. So entstand ein interessantes, herausforderndes Spiel, das zwar nicht so viel Wert wie gewollt auf Informationsverbreitung legt, den Weg aber trotzdem dafür bereitet.

2. Spielverlauf

Nation Feathers ist ein endloses Fliegen Spiel für iOS und Android Geräte. Das Spiel versetzt den Spieler in die Rolle eines Vogels, der eine hügelige, sumpftartige, mit Tintenfischen bewohnte Landschaft durchqueren muss. Ziel des Spiels ist es so viele Tintenfische wie möglich zu fangen ohne das umgebende Gelände oder jegliche Gefahren zu treffen. Der Vogel wird mit der Zeit hungrig und muss ständig Tintenfische sammeln, um seinen Hungerbalken wieder aufzufüllen, der oben links angezeigt ist. Überschreitet der Hunger-Grad die Hälfte des Balkens, blinkt der Bildschirm rot, um den Spieler zu warnen. Läuft der Hungerbalken aus, ist das Spiel zu Ende. Die normalen, greifbaren Tintenfische sitzen ohne Bewegung an der Wasseroberfläche. Im Laufe des Spieles sinkt der Wasserspiegel langsam bis zu einem bestimmten Punkt. Mit dem Wasser sinken auch alle Tintenfische. Das macht die Kollisionsgefahr mit dem Gelände größer, da der Vogel tiefer sinken muss, um die Tintenfische zu fangen und so der Weg nach oben zur Sicherheit länger wird. Mit dem gesunkenen Wasserspiegel tauchen aber weitere Gefahren auf. Springende Tintenfische springen unerwartet aus dem Wasser in die Höhe. Diese Tintenfische kann der Vogel nicht fangen und beenden das Spiel bei Kollision. Andere ungefährliche Tintenfische, die nie ans Licht kamen, sind welche, die eine gewisse Zeit lang unter dem Wasser bleiben, bevor sie wieder auftauchen in einer Art Auf- und Abbewegung. Das macht selbstverständlich die Aufgabe des Vogels schwerer, da er auf den richtigen Zeitpunkt warten muss. Der Spieler steuert den Vogel vorwärts und rückwärts sowie nach rechts und nach links mithilfe eines virtuellen Analog-Sticks. Zwei Knöpfe sind allerdings für Steig- sowie Sinkflüge verantwortlich. Ein Hilfemenü, das sowohl im Hauptmenü als auch während Spielablaufs aufzurufen ist, gibt dem Spieler Informationen zum Spiel. Am Hauptmenü kann auch der

Schwierigkeitsgrad eingestellt werden. Es gibt zwei Schwierigkeitsgrade: Leicht und schwer. Im schweren Modus erhöht sich die Geschwindigkeit des vorbeilaufenden Geländes und der Hungerbalken läuft schneller aus.

3. Einsetzung von Unity

Zur Entwicklung des Spieles wurde die Spiel-Engine Unity eingesetzt mit C# als Programmiersprache. Unity ermöglicht die einmalige Entwicklung und den vielfältigen Einsatz von 3D und 2D Spielen. Anders formuliert kann der Entwickler ein Spiel einmal schreiben und nach verschiedenen Plattformen exportieren. Dabei muss der Entwickler jedoch Tatsachen wie verfügbare Eingabegeräte sowie Rechenleistung der Zielgeräte beachten.

Ein Spiel wird in Unity in sogenannte Scenes (deutsch Szene) zerlegt. Jede Szene stellt eine neue Umgebung wie z.B. ein Spielmenü oder ein Level dar und enthält verschiedenste Spielobjekte, Töne, Animationen usw. Die Logik dieser Objekte kontrollieren C# bzw. Javascript Skripte, die den jeweiligen Objekten angehängt werden. Nation Feathers besteht aus zwei Szenen. Die Erste ist das Hauptmenü und die Zweite das eigentliche Spiel. Im Laufe des Spiels werden beide Szenen in der Regel mehrmals aufgerufen.

4. Implementierungsdetails

Im Folgenden werden die wichtigsten Implementierungsdetails zu ausgewählten Spielaspekten erläutert und genauer betrachtet. Zuerst wird auf die Vorgehensweise zur Erstellung des Geländes und den Einsatz von Object Pooling eingegangen. Spielelemente, die in direkter Verbindung mit der Kamera stehen wie Kamera-Animation, Kamera-Abgrenzung und Coroutinen werden dann erklärt. Außerdem soll auch die Funktionsweise einer Art Frühwarnungssystem zur Positionsorientierung des Vogels nach diesem Abschnitt klarer sein.

4.1. Das Gelände

Das Gelände im Spiel besteht aus 5x10 sich bewegenden Plane-Objekte (deutsch Ebene od. Fläche). Sobald eine Flächenreihe die Sicht der Kamera verlässt werden alle Flächen in der Reihe zerstört und eine neue Flächenreihe am Ende des Feldes erstellt. Der Nebel (siehe Abbildung 4.3.) sorgt dafür, dass neue Flächenreihen beim Initialisieren verbergt sind, sodass sie nicht plötzlich aus dem Nichts erscheinen. Jede Fläche ist 10x10 Unity-Einheiten groß und hat somit 121 Eckpunkte. Um die fließigen, nahtlosen Steigungen zu erstellen wurde die Perlin-Noise Funktion eingesetzt. Perlin-Noise liefert pseudozufällige Werte, die die Form einer Welle annehmen, deren Werte langsam größer und kleiner werden, wobei die selben eingegebenen Werte immer dieselben Ergebnisse liefern und ähnliche Eingabewerte zu ähnlichen Ausgabewerten führen (Unity. Mathf.PerlinNoise). Zum Erstellen des Geländes im Spiel

berechnet die Funktion für jeden Eckpunkt einen neuen y-Wert, indem die x- und z-Koordinaten des Eckpunktes als Parameter übergeben werden. Hier stellt die y-Koordinate die Tiefe dar. Zwei nahe beieinanderliegenden Eckpunkte bekommen also ähnliche y-Werte. Und da keine zwei Eckpunkte die gleichen x- und z-Werte haben, werden auch keine genau denselben y-Wert haben. Dies bewirkt, dass die erstellten Hügel natürlich und kontinuierlich aussehen.

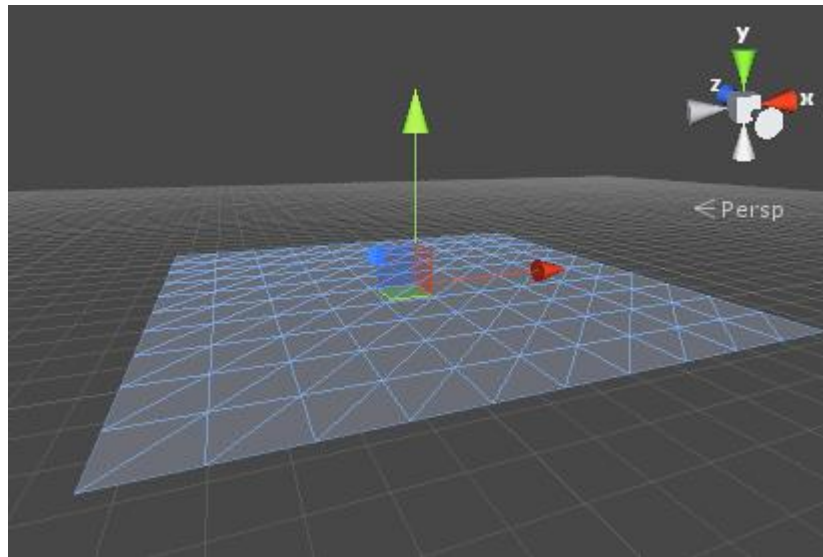


Abbildung 4.1.: Ein 10x10 Plane. (aus: Unity. Primitive and Placeholder Objects)

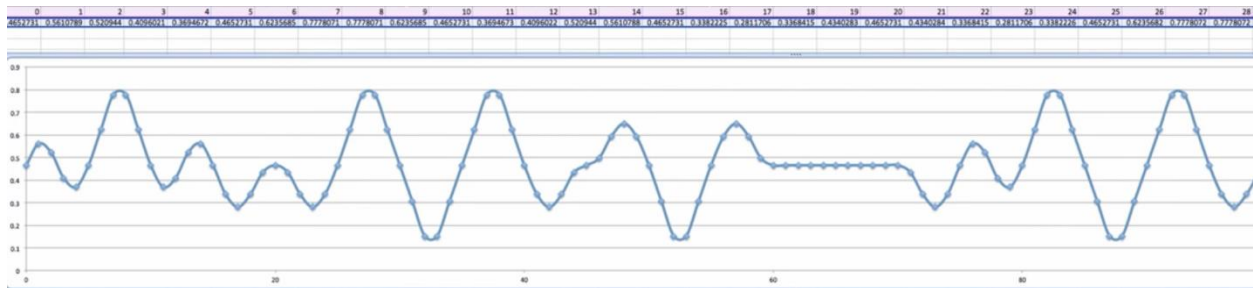


Abbildung 4.2.: Beispiel einer Perlin-Noise Welle (aus: Holistic3d. Making Minecraft with Unity 5 Part 1: Landscapes and Digging)

4.2. Object Pooling

Zu jeder Fläche gehören weiterhin Kindobjekte, die zum Erstellungszeitpunkt der Fläche aus einem sogenannten Objektpool geholt und zum Kind der jeweiligen Fläche gemacht werden. Im Programm gibt es zwei Objektpools, nämlich den Treepool, wo Bäume gespeichert sind und den Foodpool, wo Tintenfische gespeichert sind. Ein Objektpool kann in Verbindung mit Objekten nützlich sein, die immer wieder im Spiel erzeugt und zerstört werden sollen. Diese ständige Initialisierung und Zerstörung von Objekten ist zu Rechenaufwendig. Der Objektpool umgeht

diese Problematik, indem er alle Objekte im Pool nur einmal am Anfang erzeugt und im Speicher bis zum Programmende behält. Die Poolobjekte werden dann je nach Verfügbarkeit und Bedarf in der Szene verteilt. Da die Flächen sich ständig bewegen ist es wichtig, dass auch alle Bäume und Tintenfische mit den Flächen in Bewegung sind. Als Kindobjekte einer Fläche haben Bäume und Tintenfische immer die selbe relative Position zur Fläche. Es gibt drei Arten Bäume im *treePool*. Welche Art ausgegeben wird, entscheidet das Programm zufällig. Aus den zwei Arten Tintenfischen wird bei hohem Wasserspiegel nur die greifbare Art ausgegeben. Sinkt der Wasserspiegel aber zum festgelegten Punkt so hat jedes Flächenobjekt eine dreizehprozentige Chance einen springenden Tintenfisch statt eines Normalen zu bekommen.

4.3. Vorüberlegungen zur Kameraorientierung

Bei einem 3D Spiel ist es äußerst wichtig die richtigen Kameraeinstellungen zu finden. Im Laufe der Entwicklung kamen viele Kameraorientierungen in Betracht. Ursprünglich war Nation Feathers als Top Down 3D Spiel gedacht in Anlehnung ein bisschen an Chicken Invaders jedoch mit Bewegung im 3D-Raum nicht nur im 2D. Die Sicht von oben machte aber die mit viel Mühe erstellten Hügel und deren genauen Größen unerkennbar. Das bedeutete Sink- sowie Steigflüge kaum möglich waren ohne mit hoher Wahrscheinlichkeit mit dem Gelände zu kollidieren. Dazu wurde ursprünglich der Collision Oracle (deutsch Kollisionsorakel) konzipiert, um den Spieler einen Hinweis auf bevorstehende Hindernisse zu geben.

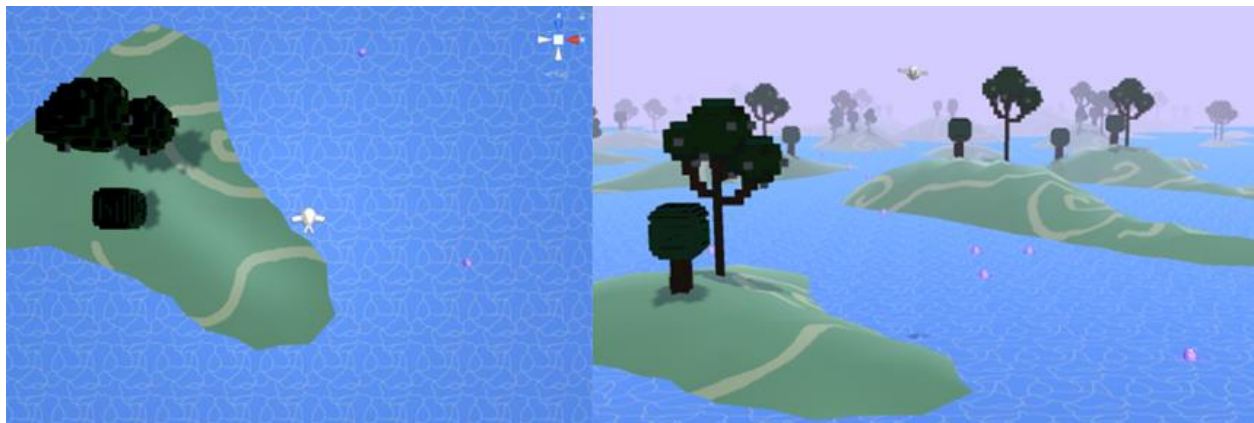


Abbildung 4.3.: Vergleich der Kameraorientierungen. Bilder zur selben Zeit aufgenommen. Links: Die ursprüngliche Perspektive (Vogelperspektive). Rechts: Die aktuelle Perspektive.

4.4. Collision Oracle

Das Kollisionsorakel dient als eine Art Frühwarnungssystem. Im Grunde ist es nicht mehr als ein einfacher, lang gezogener, unsichtbarer Würfel, der die selbe Position wie der Vogel hat aber eine gewisse Länge vor ihm streckt und etwas nach unten gedreht ist, sodass er auf den Boden

deutet. Es ist ein Kindobjekt des Vogels und bewegt sich somit immer mit ihm. Das Kollisionsorakel ist als Trigger angelegt. Das heißt bei einer Kollision mit einem anderen Objekt entsteht keine wirkliche physische Kollision sondern wird die Kollision erkannt und als Event (deutsch Ereignis) registriert, das dann ausgewertet und entsprechend bearbeitet werden kann. Wenn eine Kollision des Kollisionsorakel registriert wird, ändert sich die Farbe des Vogels auf rot. So erhält der Spieler eine Warnung, dass er sich auf Kollisionkurs befindet. Diese Spielweise war aber weder sehr angenehm noch praktikabel.

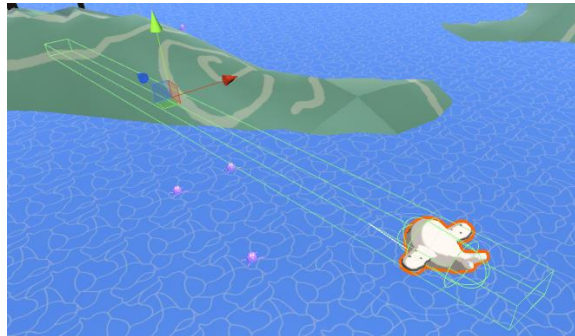


Abbildung 4.4.: Das Kollisionsorakel.

4.5. Kamera-Animation

Obwohl das Kollisionsorakel immer noch Anwendung im Spiel findet musste die Kameraorientierung noch umgearbeitet werden. Darauffolgend kamen verschiedene Kamerawinkeln, die jeweils eine neue Spielweise anboten bis hin zur aktuellen Version. Sinkflüge stellen das Hauptproblem dar. Wenn der Vogel sinkt, dann muss die Kamera entweder ihm Folgen, die Winkel ändern oder beides, damit erstens der Vogel immer im Sichtfeld der Kamera ist und zweitens, damit der Spieler bevorstehende Hürden wahrnehmen kann. Dass die Kamera dem Vogel beim Sinken den ganzen Weg nach unten folgt zeigte sich als ungünstig, da für manche Spieler so die Bewegung nach unten unerkennbar war, als würde sich der Vogel gar nicht bewegen. Die gleichzeitige Bewegung der Kamera und der Vogel hat auch dazu geführt, dass das Bild beim Sinkflug etwas zitterte, was keiner gern sieht. Dieses Phänomen lässt sich lösen, indem die Kamera sich nicht zu viel bewegt. In anderen Worten eine Verringerung der Distanz. Statt also, dass die Bewegung der Vogel direkt mit der Kamerabewegung verbunden ist, beginnt die Kamera mit ihrer Bewegung erst dann, wenn der Vogel einen bestimmten Punkt auf dem Bildschirm beim Sinken erreicht hat. Diese Auswirkung kann mit einer Drehung der Kamera nach oben kombiniert werden, um den Vogel weitestgehend direkt von hinten betrachten zu können. So ist die Tiefe für den Spieler immer leicht erkennbar und so bleibt der Vogel beim Sinken immer im Bild. Beim Steigflug hingegen müssen aber diese Änderungen zur Kameraorientierung wieder rückgängig gemacht werden und zwar analog mit dem selben Tempo wie beim Sinkflug. Außerdem, bricht der Spieler den

Sink- bzw. Steigflug ab, so muss die Kamera ihre aktuelle Transformation, d.h. Position und Rotation, beibehalten.

4.5.1 Animationszustandsmaschinen und Blend Trees

Zwei sehr einfachen Animationen mit einem Blend Tree in der Animationszustandsmaschine (englisch Animation State Machine) können diesem Zweck gut dienen.

Animationszustandsmaschinen steuern den Fluss der Animation eines Objekts. Wenn ein Objekt z.B. zwischen mehreren Animationen nach bestimmten Ereignissen wechseln soll kann in einer ASM bestimmt werden was, wann und in welchem Umfang passiert.

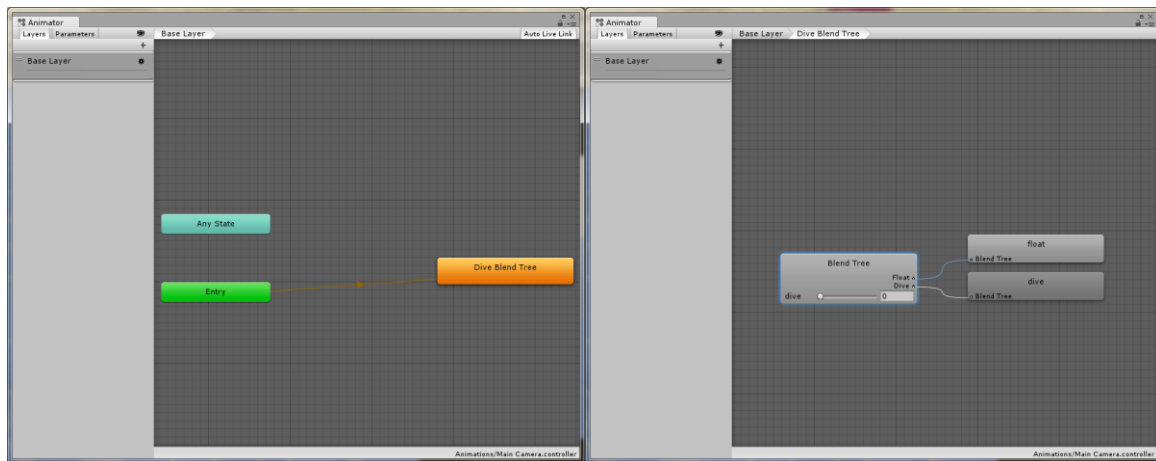


Abbildung 4.5.: Links: Animation State Machine. Rechts: Inhalt des Dive Blend Trees.

Blend Trees sind eine spezielle Art von Zustand einer ASM. Hier können mehrere Animationen kombiniert bzw. gemischt werden. Inwiefern jede Animation Einfluss hat bestimmt ein numerischer Blendparameter (englisch blending parameter) (Unity: Blend Trees).

4.5.2. Implementierung der Animation

Im Programm enthält der Blend Tree zwei Animationen, die jeweils zu einem bestimmten Transformation der Kamera führen. Animationen bestehen normalerweise aus mehreren Keyframes, die die Werte bestimmter Eigenschaften (englisch properties) wie Position oder Rotation der Animationsobjekt zu verschiedenen Zeitpunkten festlegen. Die zwei Animationen der Blend Tree jedoch enthalten jeweils nur einen Keyframe. Der Keyframe der *float* Animation, die für den Steigflug zuständig ist enthält die höchsten erwünschten Rotations- sowie Positionswerte der Kamera, während umgekehrt der Keyframe der *dive* Animation, die für den Sinkflug zuständig ist, die niedrigsten erwünschten Rotations- sowie Positionswerte der Kamera enthält.

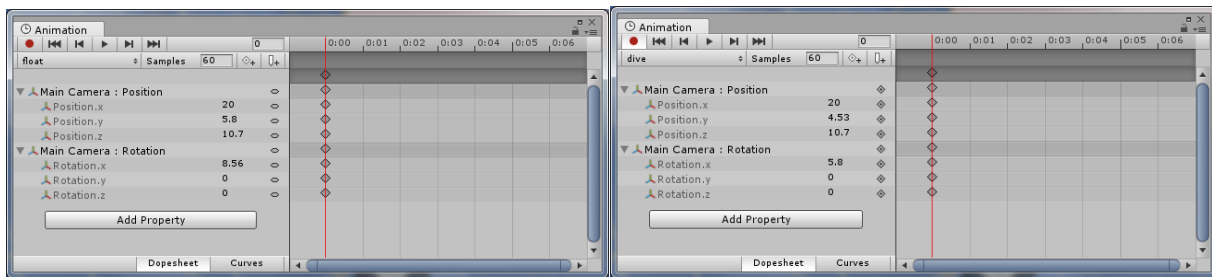


Abbildung 4.6.: Links: Die *float*-Animation. Rechts: Die *dive*-Animation. In beiden Bildern zeigt die rote, senkrechte Linie auf den jeweils einzigen Keyframe.

4.5.3. Coroutinen

Wie zwischen den beiden Animationen gewechselt wird steuert eine Coroutine. Im normalen Programmablauf wird jeder Codeabschnitt bzw. Methodenaufruf in einem einzigen Frame ausgeführt. Wenn also z.B. ein bestimmter Wert sequenziell in einer Schleife geändert wird und diese sequenzielle Änderung für den Nutzer sichtbar sein soll, dann würde einen normalen Methodenaufruf nicht funktionieren, da der Wert zwar sequenziell, aber innerhalb nur eines Frames geändert wird. Das hat zur Folge, dass die Änderung im Wert unmittelbar wirkt. Coroutinen lösen dieses Problem, indem sie ihre Ausführung pausieren und Unity die Kontrolle zurückgeben. Die Ausführung geht dann im nächsten Frame an der selben Stelle weiter (Unity Coroutines). Die im Programm implementierte Coroutine nimmt drei Parameter entgegen. Erstens der Zielwert des Blendparameters *dive*; dies ist entweder 0 oder 1. Die Sink-Animation ist zu Ende, wenn der Wert von *dive* 1 ist. Umgekehrt ist die Steig-Animation zu Ende, wenn *dive* gleich 0 ist. Die Coroutine hat hauptsächlich die Aufgabe *dive* dem Zielwert sequenziell näher zu bringen und dabei die Animation zu steuern. Der zweite Parameter ist eine Fließkommazahl, die eine bestimmte dauer darstellt, in der *dive* geändert wird. Der Dritte ist die Kurve der jeweiligen Animation. Diese Kurve bestimmt wie die Animation die von ihr beeinflussten Properties mit der Zeit ändert. In Jeder Iteration berechnet die Coroutine einen neuen Wert, der an *dive* übergeben wird und so auch die Animation vorantreibt. Dieser Wert merkt sich auch das Programm, damit falls der Spieler einen Steig- bzw. Sinkflug abbricht, kann die Coroutine von der selben Stelle, d.h. dem selben *dive*-Wert, weitermachen, wenn der Spieler wieder sinkt bzw. steigt. Bei einem Richtungswechsel, sprich Sinkflug gefolgt von Steigflug oder umgekehrt, wird die Coroutine gestoppt und dann mit den für die neue Richtung relevanten Parametern wieder aufgerufen. Zum Beispiel, führt der Spieler gerade einen Sinkflug durch, so wird die Coroutine erstmal mit dem Zielwert 1 und die entsprechende Animationskurve ausgeführt. Ändert der Spieler dann seine Meinung und einen Steigflug beginnt, so wird die Coroutine gestoppt und mit dem Zielwert 0 und die entsprechende Animationskurve aufgerufen. Dabei fängt die Coroutine von dem im letzten Coroutinenaufruf erreichten *dive*-Wert an.

4.6. Bewegungsbeschränkung

Da dies kein Open-World-Spiel ist, darf sich auch der Spieler nicht komplett nach freiem Willen bewegen. Sonst gerät der Spieler ganz schnell ins Nichts, weil die Spielobjekte nur in einem bestimmten Bereich und zu bestimmten Zeiten generiert werden. In diesem Abschnitt geht es darum, wo die Bewegungsfreiheit aufhört und wie das technisch realisiert ist.

4.6.1 Seitliche Bewegungsbeschränkung

Die seitliche Bewegung des Vogels beschränken zwei unsichtbare Flächen, die Kindobjekte der Kamera sind und somit immer die selbe Bewegung haben. Der Grund warum keine einfachen Zahlenwerte statt Flächen benutzt werden hat viel mit der Kamerasicht zu tun. Da die Kameraprojektion keine orthografische Projektion ist sondern die Perspective zeigt, strecken die Sichtlinien nicht gerade von der Kamera aus in form einer Rechteck sondern sie schwenken nach außen aus wie eine Art abgeschnittene Pyramide. Diese Pyramide nennt sich Viewing Frustum und alles, was innerhalb dieses Frustums liegt ist für die Kamera sichtbar (Wikipedia. Frustum Culling).

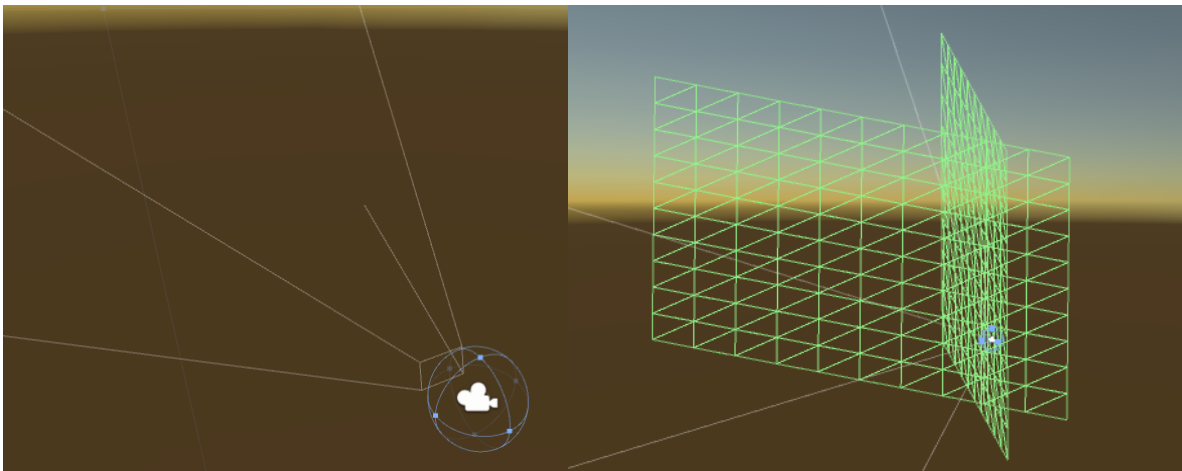


Abbildung 4.7.: Links: Frustum der Kamera. Rechts: Die Boundary Planes.

Das heißt würde der Vogel nah vor der Kamera fliegen, so hat er weniger seitliche Bewegungsfreiheit, bevor er außerhalb Kamerasichtfeld landet, als wenn er weiter vorne stehen würde. Die Flächen sind so positioniert und gedreht, dass sie die seitlichen Sichttränder entlang verlaufen. Die Flächen dürfen außerdem nur mit dem Vogel interagieren. Das heißt, während die Kamera bei der *dive*-Animation sinkt, müssen alle anderen Spielobjekte für die Flächen immateriell sein und keine Kollisionen herbeirufen. Jedem Spielobjekt kann ein Layer (deutsch Schicht) zugeordnet werden. Mithilfe der Layer Collision Matrix kann eingestellt werden, welche Layers und damit auch welche Spielobjekte bei einer Kollision interagieren

dürfen. Die selbe Funktionsweise nutzt das Kollisionsorakel, um für den Vogel ungefährliche Spielobjekte wie normale Tintenfische und Wasser zu ignorieren.

4.6.2. Untere und vordere Grenzen

Nach vorne darf sich der Vogel nur bis zu einem bestimmten Punkt bewegen. Ab diesem Punkt bleibt der Vogel stehen aber hält der Spieler den Analog-Stick immer noch gedrückt so wird die Geschwindigkeit des Geländes größer, was den Eindruck einer erhöhten Geschwindigkeit des Vogels erweckt. Von unten ist der Vogel durch das Wasser und Gelände eingeschränkt. Trifft er das Wasser, so kann er nicht weiter sinken. Trifft er das Gelände, dann ist das Spiel zu Ende. Nach oben ist aber die Bewegung nicht eingeschränkt. Dies hat damit zu tun, dass manchmal seitliche Manöver nicht ausreichen, um Hürden auszuweichen. In solchen Fällen kann der Spieler die Hürden nur überfliegen.

4.6.3. Obere Grenze

Die Kamera folgt dem Vogel jedoch nicht, wenn er beim Steigflug außerhalb des Sichtfeldes fliegt. Eine interessante Idee aber, die nie ans Licht kam, ist eine Änderung der Kamerarotation beim Erreichen einer bestimmten Höhenlage außerhalb des Sichtfeldes, wo Gelände und Bäume keine Gefahr mehr sind und so die Tiefenerkennung nicht mehr von großer Bedeutung ist. Die Kamera wäre so gedreht, dass sie direkt von oben herunterschaut, um eine Art Vogelperspektive zu verschaffen (siehe Abbildung 4.3.).

4.7. Nutzereingabe

Das Spiel soll für alle Nutzer ohne Bedarf an zusätzlichen Geräten oder Zubehör spielbar sein. Aus diesem Grund soll hauptsächlich als Eingabegerät der Touchscreen benutzt werden. Die folgenden Abschnitte gehen auf die verschiedenen implementierten und nicht implementierten Methoden zur Eingabe.

4.7.1. Erste Ideen zur Eingabe

Der erste Ansatz war, dass der Vogel dorthin fliegt, wo der Finger gerade angetippt hat. Dies war leichter gesagt als getan, da alle Berührungen auf dem Bildschirm im 2D-Raum stattfinden, während die Spielwelt 3D ist. Das heißt, die Fingerposition ist unter Berücksichtigung der Kameratransformation aus dem 2D-Raum in die 3D-Welt zu mappieren. Mit einer Wischgeste hingegen lässt sich die Eingabe viel leichter in Bewegung übersetzen. Die zurückgelegte Distanz des Fingers liefert die Funktion *Touch.deltaPosition* als 2D Vektor, der dann als Kraft zum Vogel hinzugefügt wird oder als neue Geschwindigkeit eingesetzt wird. Obwohl funktionsfähig, war dieser Einsatz nicht genug ergonomisch und manchmal schwer zu handhaben. Und da kam der Analog-Stick ins Spiel.

4.7.2. Analog-Stick

Im Unity Asset Store können Entwickler voll funktionsfähige Analog-Sticks herunterladen und mit wenig Umarbeitung für die meisten ihrer Zwecke nutzen. Für Gelegenheitsspieler war dieser Stick akzeptabel. Für Spieler mit mehr Erfahrung, die mehr an traditionelle Joysticks oder sogar höherwertige, virtuelle Analog-Sticks gewöhnt sind, war er nicht flexibel genug. Normalerweise muss der Finger nicht unbedingt bis zur Standardposition (d.h. den mittleren Teil eines Analog-Sticks mit den Koordinaten $(0, 0)$), um den Stick zu bewegen. Der Spieler soll einfach irgendwo am Rande (siehe Abbildung 4.8.) antippen können und der Stick bewegt sich dorthin wo sich sein Finger befindet. Das war mit diesem Stick aus den Standard Assets von Haus aus nicht möglich. Es war also entweder eine Umarbeitung fremden Codes nötig oder eine ganz neue Implementierung eines flexibleren Sticks, was am Ende sich durchgesetzt hat. Im Grunde besteht dieser Stick aus zwei Bildern, nämlich einem Hintergrund und einem beweglichen Teil, der nur als optische Hilfe dient.

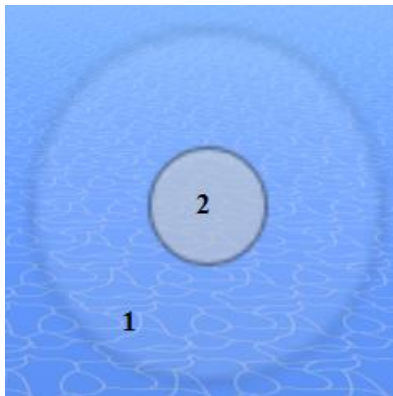


Abbildung 4.8.: 1. Hintergrundbild. 2. Stickbild

Die Funktion *RectTransformUtility.ScreenPointToLocalPointInRectangle* gibt *Wahr* zurück, wenn der rechteckige Bereich rund um den Hintergrund angetippt wird und speichert die angetippte Position in einer Variable. Diese Position ist dann durch die Größe des Rechtecks zu teilen, um den Koordinaten einen Wert zwischen -1 und 0 zu geben. Optimal wäre es aber, wenn 0 die Mitte darstellt und nicht rechts. Die Position wird also in einen neuen Vektor umgewandelt, dessen Koordinaten Werte zwischen -1 und 1 haben. Der normalisierte Wert dieses Vektors wandelt das Verhalten des Rechtecks in das eines Kreises. Das heißt, tippt der Spieler auf der unteren rechten Ecke des Rechtecks, dann sind die x-y-Koordinaten nicht -1 und 1 sondern normalisierte Werte dieser Koordinaten als hätte der Spieler den unteren linken Rand des runden Hintergrunds angetippt. Das Joystick-Bild bewegt sich dann zur entsprechenden Position.

4.7.3. Weitere Eingabe Möglichkeiten

Der Nachteil eines Analog-Sticks auf einem Handy überhaupt ist die eingeschränkte Sicht. Je mehr ein Spieler den Bildschirm berühren muss, umso höher die Wahrscheinlichkeit, dass ein wichtiges Spielobjekt durch den Finger bedeckt wird, was zu einem negativen Spielerlebnis führen kann. Der nächste Schritt wäre den Accelerometer (deutsch Beschleunigungssensor) des Handys als Eingabegerät für die Bewegung auf der x-z-Ebene zu benutzen. So wäre die Betätigung des Touchscreens auf die Steig- bzw. Sinktasten und das Spielmenü beschränkt.

4.8. Kollision

Für den Vogel sind nur greifbare Tintenfische immateriell. Alles andere soll den Vogel anhalten bzw. seine Transformation beeinflussen. Dazu müssen alle Objekte, die mit dem Vogel oder mit sich selbst interagieren sollen, Collider-Komponenten haben. Diese können entweder primitive Collider-Komponenten (siehe Abbildung 4.9.) sein wie Kugeln, Würfel und Kapseln oder Kombinationen dieser Komponenten in Form von zusammengesetzten (englisch compound) Colliders (Unity. Colliders). Wenn Collider-Komponenten als Trigger eingestellt sind, können Kollisionen abgefangen und bearbeitet werden. Kollidiert der Vogel mit einem Tintenfisch, so wird der Score inkrementiert und der Tintenfisch unsichtbar gemacht, was ihn als Poolobjekt wieder zur Verfügung stellt. Die Flächen des Geländes und die Bäume darauf schicken andererseits bei Kollision ein Signal an die GameController-Klasse, die für die Steuerung des allgemeinen Spielablaufs zuständig ist. Dieses Signal hält das Spiel an und ruft ein Spielmenü herbei, das den erreichten Scorewert anzeigt. Von hier aus kann der Spieler entweder erneut spielen oder zum Hauptmenü zurückkehren. Der Spieler kann auch jeder Zeit das Spiel pausieren. In diesem Menü kann er die Hilfeseite öffnen, die Lautstärke ändern, das Spiel fortfahren oder zum Hauptmenü zurückkehren.

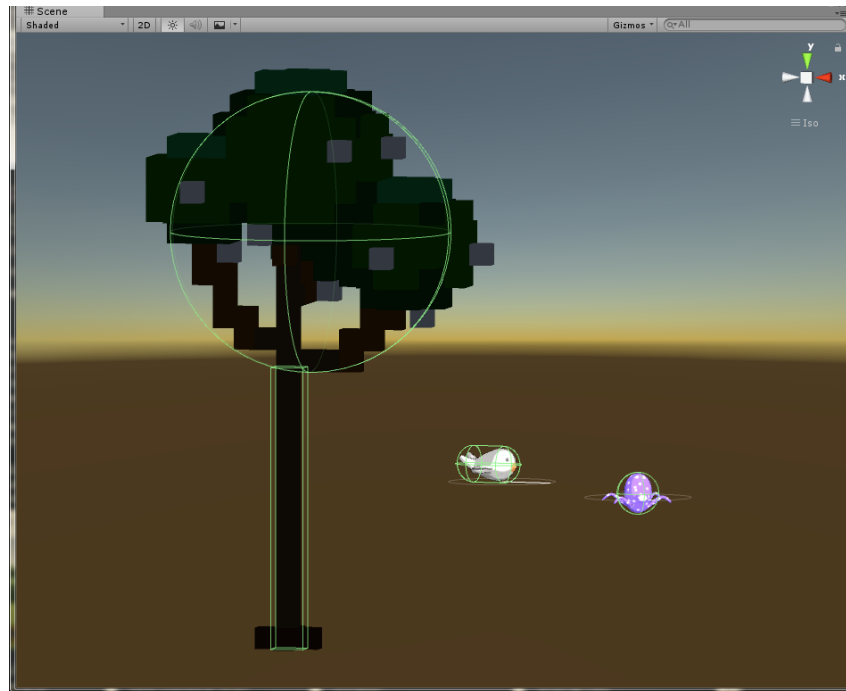


Abbildung 4.9.: Links: Zusammengesetzter Collider eines Baumes bestehend aus einem BoxCollider und einem Sphere (deutsch Kugel) Collider. Mitte: CapsuleCollider-Komponente des Vogels. Rechts: SphereCollider-Komponente eines Tintenfisches.

5. Benutzeroberfläche

Das Hauptmenü selbst ist eine separate Szene, die beim Verlassen eines aktiven Spiels geladen wird. Alle Objekte aus dem laufenden Spiel werden beim Szenenwechsel zerstört.

Normalerweise ist das kein Problem, da das Spiel sowieso zu Ende geht und es keinen Sinn macht den Speicher unnötig zu belasten. Was der Spieler aber mit hoher Wahrscheinlichkeit gerne beibehalten würde sind personalisierte Einstellungen wie in diesem Fall die Lautstärke und den ausgewählten Schwierigkeitsgrad. Unity bietet die Lösung in Form einer Klasse an. *PlayerPrefs* erlaubt es dem Spieler Einstellungen unabhängig von Szenen lokal auf dem Gerät zu speichern. Jedes Mal also, wenn der Spieler die Lautstärke ändert, speichert *PlayerPrefs* den neuen Wert. Wenn eine Szene geladen wird, holt sich der Audioschieberegler den Wert aus *PlayerPrefs* und passt die Lautstärke entsprechend an. Beim Verlassen eines Spiels und Laden des Hauptmenüs holt sich das Spiel den gespeicherten Schwierigkeitsgrad aus *PlayerPrefs* und ändert das Aussehen der relevanten Tasten, indem die Taste des aktuell eingestellten Schwierigkeitsgrads deaktiviert wird und die des anderen Schwierigkeitsgrads aktiviert wird. Der Hungerbalken ist grundsätzlich ein nicht-interaktiver Schieberegler, der immer den aktuellen Hunger-Grad anzeigt. Sammelt der Spieler Tintenfische, wird er entsprechend nachgeladen. Mithilfe einer Coroutine blinkt der Bildschirm beim Erreichen eines gefährlichen Hunger-Grads. Die Coroutine ändert den Alpha-Wert eines roten Bilds, das vor dem Bildschirm

platziert ist, von null bis zu einem gewissen Wert hin und her, um diese Auswirkung zu realisieren. Das Hilfe-Menü bietet nähere Informationen zum Spiel. Es besteht aus mehreren Bildern, die in einem Array gespeichert sind, und die Tasten **Next** (deutsch weiter), **Prev** (deutsch vorherige) und **Back** (deutsch zurück). Die Tasten **Next** und **Prev** steuert ein Skript, das den Bilder-Array sequenziell durchläuft und die entsprechenden Bilder anzeigt.

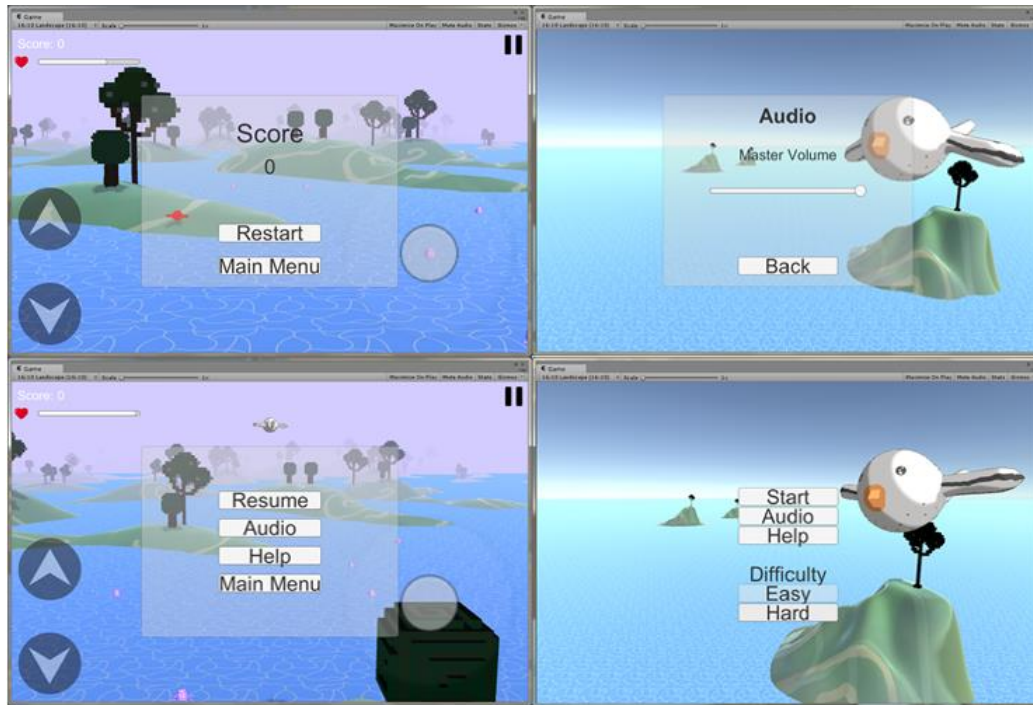


Abbildung 5.1.: Oben links: Gameover-Menü. Oben rechts: Audio-Menü. Unten links: Spiel pausiert. Unten rechts: Hauptmenü.

6. Audio

In Unity sind Audioquellen relevanten Spielobjekten angehängt und entweder bei Initialisierung des Objekts oder bei Eintritt bestimmter Ereignisse abspielbar. Die Lautstärken individueller Audioquellen sind zwar einstellbar aber der Spieler hat hier Einfluss nur auf die Master-Lautstärke, die die Lautstärke aller Audioquellen gleich beeinflusst. Die Master-Lautstärke ist im Hintergrund eine Komponente namens **Audio Listener**, die in jeder Szene nur einmal vorkommt und als eine Art Mikro funktioniert, das die Töne der verschiedenen Audioquellen als Eingabe einnimmt und sie über die Lautsprecher des Zielgerätes ausgibt. Es werden nur die Töne abgespielt, deren Spielobjekte in einer bestimmten Entfernung vom Objekt mit der Audio-Listener-Komponente sind. Aus diesem Grund wird der **Audio Listener** normalerweise der Kamera oder dem Spielerobjekt angehängt.

7. Ausblick und Fazit

Im Spiel gibt es noch viel Spielraum sowohl für ästhetische als auch funktionelle Erweiterungen und Verbesserungen. Viele Ideen sind aus zeitlichen Gründen nicht implementiert worden. Im Folgenden wird auf einige Ideen kurz eingegangen.


An erster Stelle vielleicht steht die Speicherung von Scorewerten. Es ist in der Regel bei Spielen wie Nation Feathers üblich, dass Spieler die Möglichkeit haben erreichte Scorewerte zu speichern und auf Abruf anzeigen zu lassen. Eine Datenbank wäre für diese Aufgabe am besten geeignet. Eine einfache .csv Datei würde aber für die Zwecke und den Umfang dieses Spiels auch reichen. Wie oben schon erwähnt würden weitere Arten Tintenfische und Hürden das Spielerlebnis anspruchsvoller machen. Solchen Herausforderungen hätte sich der Spieler mit einem erweiterten Arsenal in Form von anderen spielbaren Vogelarten gestellt. Diese Vogelarten hätten jeweils verschiedene Eigenschaften die sowohl Stärken als auch Schwächen aufweisen. In Verbindung mit einer sich mit der Zeit ändernden Umgebung, die über einen sinkenden Wasserspiegel hinaus geht, wären manche Vögel für bestimmte Situationen besser geeignet als Andere. Diese Vögel, die an echten Vögeln angelehnt wären, würden auch unterschiedlich klingen und aussehen. Plakate bzw. Infotafeln würden den Spielern Hinweise über die Eigenschaften jedes Vogels geben wie z.B. Ernährung und Lebensraum, was in Form von unterschiedlichen Zielobjekten und Umgebungen dargestellt werden könnte. Diese Vögel wären dann entweder als greifbare Bonus-Objekte im Spielverlauf oder als Errungenschaften freischaltbar. Es fehlen auch wichtige Bestandteile wie einige Animationen und Schatten, die nicht nur einen besseren ästhetischen Eindruck geben sondern auch als optische Hilfe zur Orientierung und Position dienen. Letzteres ist leider nur in der kommerziellen Version von Unity verfügbar, während ersteres in späteren Iterationen des Spiels miteinbezogen werden kann. Eine attraktivere Benutzeroberfläche hat nichts mit den Spielmechaniken zu tun, darf aber nicht komplett vernachlässigt werden und soll, wenn zeitlich machbar in Erscheinung treten.

Abschließend trotz alledem ist Nation Feathers immerhin ein Spiel, das schöne Landschaften anbietet, sich leicht weiterentwickeln lässt und vor allem Spaß macht. Obwohl das Spiel für manche am Anfang etwas gewöhnungsbedürftig ist, ist es nach ein paar Spielrunden erlernt und die Kontrollen nicht mehr so fremd.

Installationshinweise

Installationshinweise für Android und iOS sind hier erläutert.

Android

Unter **Einstellungen** > **Sicherheit** > **Unbekannte Herkunft** aktivieren. Die zur Verfügung gestellte .apk Datei auf das Zielgerät bringen. Zielgerät einfach mittels eines USB-Kabels mit dem Rechner verbinden und die Datei irgendwo auf das Gerät kopieren. Zum auffinden der Datei nach der Kopierung empfiehlt es sich meistens eine Dateimanager-App aus dem Play Store herunterzuladen. Eine Möglichkeit wäre File Manager 

Nach der Installation, die .apk Datei mithilfe des Dateimanagers lokalisieren und zum Installieren antippen.

iOS

Die Installation auf einem iOS Gerät ist etwas anspruchsvoller als die Installation auf Android Geräten. Um die Installation durchführen zu können wird vor allem ein Mac Rechner und natürlich ein iOS Gerät, wo das Spiel läuft, gebraucht. Eine ausführliche Anleitung befindet sich hier: <https://unity3d.com/learn/tutorials/topics/mobile-touch/building-your-unity-game-ios-device-testing>

Literaturverzeichnis

- Unity. (2013, April 22). Primitive and Placeholder Objects [Dokumentation].
aus <https://docs.unity3d.com/Manual/PrimitiveObjects.html>
- Unity. (2017, März 29). Mathf.PerlinNoise [Dokumentation].
aus <https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html>
- Holistic3d. (2016, Januar 11). Making Minecraft with Unity 5 Part 1 [Video]. (9:32)
aus <https://www.youtube.com/watch?v=4hVAzRahcdY&t=622s>
- Unity. (2017, März 29). Blend Trees [Dokumentation].
aus <https://docs.unity3d.com/Manual/class-BlendTree.html>
- Unity. (2017, März 29). Coroutines [Dokumentation].
aus <https://docs.unity3d.com/Manual/Coroutines.html>
- Frustum Culling. (2016, Dezember 8). In *Wikipedia*.
aus https://de.wikipedia.org/wiki/Frustum_Culling
- Unity. (2017, März 29). Colliders [Dokumentation].
aus <https://docs.unity3d.com/Manual/CollidersOverview.html>

Asset- und Spielobjektquellen

- Sail Character Pack:** *Wasser mit Animation, Vogel mit Animation, Gras-Textur, Tintenfisch-Model und -Textur*
aus <https://www.assetstore.unity3d.com/en/#!/content/18880>
- Low Poly Boxy-Stylized Trees:** *Bäumen-Modelle und -Textur*
aus <https://www.assetstore.unity3d.com/en/#!/content/67258>
- Wind Sound Effect:** *Wind-Töne*
aus https://www.youtube.com/watch?v=FYCOW0Np_IA
- Pop Sound Effect:** *Ton beim Tintenfische-Fangen*
aus <https://www.youtube.com/watch?v=xrWnEf74aBA>
- Seagull Sound Effect:** *Vogelruf am Spielanfang und Spielende*
aus <https://www.youtube.com/watch?v=xolWjKkf244>
- Fire Hose for Unity:** *Ton beim Berühren der Wasseroberfläche*
aus <https://www.youtube.com/watch?v=QLhmQ8GFAP4>

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Ithaca, NY, 18.04.17

Ort, Datum

Shazali

Unterschrift