19.1.2020

Gamephysik
Abgabe GD 1017

Alexander Doege
GD 1017, MDH BERLIN

Inhalt

.....0
1 Kollision 2
1.1 Vorwort 2
1.2 Sphere-Collision 2
1.3 Capsule-Collision 2
1.4 Box-Collision 2
2 Physik 3
2.1 PhysicsComponent 3
2.2 PhysicsModifier 3
2.3 GravityPhysics 3
2.4 FirstPersonController 4
3 Sonstiges 4
3.1 Sound 4
3.2 Shoot 5
3.3 Importierte Assetpacks 5

1 Kollision

1.1 Vorwort

Bei der Erstellung der Kollision des Produktes, habe ich lediglich die von Unity bereitgestellten Collider genutzt, die Logik hinter der Kollision habe ich dabei selbst eingefügt. Dies habe ich für 3 verschiedene Collider-Typen getan (Sphere-, Capsule- und Boxcollider). Um die Kollision für den jeweiligen nächsten Frame berechnen zu können, läuft jegliche Bewegung von Objekten über dieselbe Komponente (PhysicsObject). Bewegungen werden hierbei von Skripten, welche von „PhysicsComponent“ erben, an die „PhysicsObject“ Komponente weitergegeben, welche nach dem Ausrechnen aller Bewegungseinflüsse den eigentlichen Laufweg des Objektes innerhalb des Frames bezüglich der Kollision abfragt. Hierbei liest die Komponente die Größen des Unity-Colliders des Objektes aus und sucht auf der nächsten Position nach Kollisionen. Sollte es Kollisionen feststellen, drückt das Skript das Objekt aus der Kollision so weit wie nötig heraus, sodass es sich direkt an der Kollision, aber nicht in der Kollision befinden kann. Im falle eines Laufweges, welcher einer längeren Distanz als des Radius des Objektes entspricht, wird dies mehrfach angewandt, um keine Kollision auf dem Laufweg zu verpassen. Sollte eine hierbei gefunden werden, wird die Position nach dem setzen aus der Kollision berechnet und diese wird dann letztendlich auch gesetzt. Im Falle von mehreren Kollisionen im selben Schritt, werden bis zu 3 berücksichtigt. Diese Zahl ist im Inspektor der Komponenten einstellbar, sollte aber nicht zu hoch sein, sollte sich ein Objekt in einem anderen verfangen, würde eine Zahl im dreistelligen Bereich die Framerate des Projektes gefährden. Alle Kollisionen werden in der Klasse „PhysicsObject“ berechnet.

1.2 Sphere-Collision

Diese Kollision war die leichteste der 3 Kollisionstypen, da eine Kugel sehr einfach zu berechnen ist. Ich habe lediglich die Kugel entlang ihres Laufweges entsprechend ihres Radius gesetzt und nach Kollisionen innerhalb ihres Radius gesucht. Sollte eine Kollision gefunden werden, Berechne ich den Punkt des anderen Objektes, welcher dem Mittelpunkt meiner Kugel am nächsten ist, berechne von dem Mittelpunkt der Kugel den Vektor zum Punkt auf meinem anderen Objekt und ziehe letztendlich diesen von einem Vektor ab, welcher dieselbe Richtung, aber die Länge gleich dem Radius der Kugel hat. Dadurch erhalte ich den exakten weg, den meine Kugel gehen muss, damit sie gerade nicht mehr innerhalb des anderen Objektes ist.

1.3 Capsule-Collision

Bei diesem Kollisionstypen berechne ich zuerst wieder den Punkt, welcher sich am nächsten zur Mitte meines Colliders befindet. Danach berechne ich eine Linie, welche vertikal genau durch die Mitte meines Colliders verläuft und zu jedem Punkt auf der Oberfläche meines Colliders genau den Abstand des Radius des Colliders hat. Auf dieser Linie suche ich dann den nächst gelegenen Punkt zu dem Punkt auf meinem anderen Collider, um mich von diesem aus genau wie bei der Kugel aus dem anderen Collider wieder heraus zu bewegen.

1.4 Box-Collision

Dieser Kollisionstyp war mit Abstand der am schwersten zu berechnende, da ich hier mit keinem Radius und dem finden eines Mittelpunktes oder einer mittleren Linie arbeiten konnte. Daher habe ich wieder damit angefangen, den Punkt pro Collider zu finden, welcher dem Mittelpunkt meiner Box am nächsten kommt. Von diesem aus berechne ich die Richtung, in welcher meine Kollision stattfindet und finde heraus, welche Seite meiner Box ausgehend vom Mittelpunkt meiner Box am frühesten von dieser Richtung getroffen wird. Damit weiß ich, in welche Richtung ich das Objekt bewegen muss, damit es mit möglichst wenig Bewegung aus der aktuellen Kollision herauskommt. Danach berechne ich den Punkt, welcher auf der richtigen Seite meiner Box dem Kollisionspunkt auf dem anderen Objekt am nächsten ist und berechne dann ausgehend von diesen 2 Punkten den

Vektor, um welchen sich das Objekt bewegen muss. Bei der Berechnung des Punktes auf der richtigen Seite der Box, finde ich zuerst heraus, ob die Seite an dem Punkt mit den kleinsten Werten (in World Space) oder dem mit den größten Werten angrenzt. Daraufhin kann ich von dem Kollisionspunkt den X, Y oder Z Wert auf den des kleinsten bzw. größten Punktes setzen, je nach Richtung, in die das Objekt bewegt werden soll. Die anderen Werte setze ich dann dem Kollisionspunkt auf dem anderen Objekt gleich, so erhalte ich den Punkt am nächsten zum Kollisionspunkt auf der Außenseite meines eigenen Colliders.

Sollte eine Box versuchen, sich in den Spieler herein zu bewegen, führt sie diese Bewegung nicht aus, sondern geht nur so nah wie möglich an den Spieler heran. Programmiertechnisch hätte ich zwar auch sagen können, dann wenn der Spieler innerhalb eines Colliders gleichzeitig mit einem Collider auf der entgegengesetzten Richtung kollidiert, er einfach stirbt, aber ich fand es für den arcade-artigen Plattformer passender, dass die Dinge im Environment den Spieler nicht so einfach töten können.

Da es in diesem Spiel um präzise Sprünge gehen würde, habe ich es so programmiert, dass die Boxen den Spieler nicht irgendwo hindrücken können. Würden sie den Spieler drücken können, wäre die Bewegung des Spielers nicht mehr so präzise, das kann in diesem Genre zu Frustration führen.

2 Physik

2.1 PhysicsComponent

Alle Komponenten, welche eine Bewegung des Objektes herbeirufen wollen, erben von der Klasse „PhysicsComponent“. Diese registriert sich in ihrer Awake() Funktion im „PhysicsObject“ Skript des Objektes. Ein von „PhysicsComponent“ erbendes Skript muss zudem die Funktion ApplyPhysics() definieren. Die Funktion übergibt dem Skript die aktuelle Position des Objektes nach dem Berechnen anderer Komponenten mit einer niedrigeren Reihenfolge als dieses Skript. Bei dieser Funktion muss die Position nach berechnen der eigenen Bewegungseinflüsse zurückgegeben werden, damit andere „PhysicsComponent“ Skripte ebenfalls ihre Berechnungen durchführen können. Sobald die Berechnungen aller Komponenten des Objektes abgeschlossen sind, wird der Laufweg des Objektes auf Kollisionen überprüft und falls nötig entsprechend geändert, sodass es sich durch keinen Collider bewegen kann, welcher kein Trigger ist. Um ein „PhysicsComponent“ Skript vor oder nach einem anderen auf die aktuelle Position zugreifen und diese verändern zu lassen, muss lediglich die Reihenfolge des Skriptes im Inspektor geändert werden. Als Beispiel hierfür dient beispielsweise die „FPController“ Klasse, oder die Klasse „GravityPhysics“.

2.2 PhysicsModifier

„PhysicsModifier“ können als Komponente einem Objekt hinzugefügt werden. Sie richten sich an ein „PhysicsComponent“ Skript auf diesem und verändern es entsprechend der eigenen Definition. Als Beispiel Skript habe ich hierfür „SpeedModifier“ erstellt, welches sich selbst in den Parent des „FPControllers“ registriert und dort die Geschwindigkeit des Spielers entsprechend der eigenen Stärke festlegt. Diese kommt aus seiner Parentklasse „PhysicsModifier“. Die Logik zur Reduzierung oder Erhöhung der Geschwindigkeit des Spielers ist in der „ControllerBase“ Klasse in der Funktion OnModifiersChanged() beschrieben.

2.3 GravityPhysics

Dieses Skript orientiert sich an realen Werten zum Aufbau von Geschwindigkeit basierend auf Schwerkraft. Es hat jedoch eine maximale Geschwindigkeit, welche sie nicht übersteigen kann. Dies ist nötig, da Sinn des Spiels war, einen 1st-Person Plattformer mit Shooter-Elementen zu gestalten, dieser würde sich mit einer zu schnellen Fallgeschwindigkeit nicht mehr gut steuern lassen können und entsprechend würde dies für Frustration sorgen.

Die Ausrichtung der Schwerkraft kann in diesem Skript geändert werden, sodass Objekte in einer Szene entsprechend gegen die Decke „fallen“ können, bzw. gegen Wände. Die Änderung der Richtung wird ebenfalls an die „PhysicsObject“ Komponente des Objektes weitergegeben, damit diese entsprechend feststellen kann, ob sie das Objekt auf dem Boden befindet, in welchem Falle unsere Schwerkraft nicht mehr auf das Objekt angewendet wird. Ebenfalls wird hierbei die aktuelle Fallzeit des Objektes zurückgesetzt.

Es ist zudem möglich, die Schwerkraft für eine Zeit oder bis zur Reaktivierung zu Deaktivieren. Dies ist über die Funktion DisableGravity() möglich, welche optional einen Parameter annimmt, welcher bestimmt, für wie lange die Schwerkraft deaktiviert werden soll. Wird kein Parameter angegeben, wird die Schwerkraft bis zur Reaktivierung deaktiviert.

Über die Funktion FlipGravity() Kann die Schwerkraft für das Objekt umgedreht werden. Hierbei wird ebenfalls ein Material auf das Objekt eingesetzt, welches im Inspektor festgelegt werden kann. Entsprechend wird auch die „PhysicsObject“ Komponente des Objektes neu ausgerichtet, sodass diese herausfinden kann, ob wir unseren Boden durch die Schwerkraft erreicht haben.

2.4 FirstPersonController

Wie andere „PhysicsComponent“ Komponenten berechnet er über ApplyPhysics() die neue Position je nach Input des Spielers. Zudem sorgt der Controller dafür, dass sich der Spieler mit der Maus umsehen und zielen kann. Er beinhaltet auch die Logik für Sprünge sowie einen Counter, welcher den Spieler 2x springen lässt, sobald er den Boden berührt werden diese wieder zurückgesetzt.

Ich habe mich hierbei bewusst dazu entschieden, dem Spieler nicht einem Sprung zu berauben, sollte er von der Plattform herunterlaufen, da das Spiel als kleines Projekt sehr auf casualplayer ausgerichtet wäre und diese für leichte Fehler nicht zu sehr bestraft werden sollten. Auch ist es aktuell nicht sehr einfach, zu sehen, wo man sich auf einer Plattform gerade befindet, wenn man an der Decke nach Blöcken zum abschießen sucht. Um das zu kompensieren habe ich den 2ten Sprung eingebaut.

Über die Funktionen DisableControlls() und EnableControlls() kann dem Spieler die Steuerung seines Charakters genommen werden. Jedoch gilt dies aktuell nicht auch für das Schießen, dieses muss zusätzlich deaktiviert werden.

Die Rotation des Spielers über die Maus funktioniert sehr einfach, wobei ich lediglich die Kamera um die X Achse, aber das Gesamte Spielerobjekt um die Y Achse bewege. Das hat den einfachen Grund, dass das rotieren des Colliders um die X Achse zu vielen Problem führen würde, da man sich beispielsweise in eine Wand reinrotieren könnte, oder alternativ an einer Wand nicht nach oben sehen könnte.

3 Sonstiges

3.1 Sound

Alle Sounds des Spiels werden über einen SoundManager abgespielt. Dieser kreiert einfach „AudioSource“ Komponenten, welche wieder zerstört werden, sobald der abzuspielende Sound beendet ist.

Bei dem Erstellen des Sound Systems habe ich darauf geachtet, es möglichst einfach zu machen, neue Sounds dem Projekt hinzuzufügen. Hierbei muss man diese lediglich im überschaubaren Skript „AudioAssigner“ definieren und danach über „SoundManager.Instance.PlayNewSound([definierter enum für den abzuspielenden Sound])“ in einem beliebigen Skript ausführen. Als Beispiel hierfür kann man sich an der HandleShot() Funktion des Skripts „Shoot“ orientieren.

Da die Hintergrundmusik in verschiedenen Leveln variieren soll, man aber gut darüber nachdenken kann, dem SoundManager ein DontDestroyOnLoad() zu geben, um Szenenübergänge einfacher zu gestalten, habe ich diese über ein zusätzliches Skript („BackgroundMusic“) definiert.

3.2 Shoot

Das Skript „Shoot“ funktioniert völlig unabhängig von allen anderen Skripten des Projektes. Es definiert 2 im Inspektor änderbare Keys, welche die Gravitation eines Objektes entweder Einfrieren, oder sie um genau 180° umdrehen. Die beiden Schüsse haben einen gemeinsame Abklingzeit, welche ebenfalls im Inspektor einstellbar ist. Zudem spielen sie eine Animation in dem AnimationsController der Pistole ab, welcher beim Erhalt des Assetpacks nicht mal eine Schnittstelle dafür bereitstellte. Das Skript aktiviert auch einen geshaderten Blitz, welcher vom Endpunkt der Waffe bis zum Punkt, auf welchen man beim Schuss gezielt hat, geht. Dieser wird nach kurzer Zeit wieder deaktiviert.

3.3 Importierte Assetpacks

Alle von mir importierten Assets habe ich in den Ordner „Imports“ im Unity Projekt geschoben. Dazu gehören:

1) Ein Crosshair-Pack, von welchem ich das Crosshair in der Mitte des Screens sowie seinen Variationen in 3 anderen Farben entnommen habe (die Variationen habe ich letztendlich nicht benutzt).
2) Die im Spiel gezeigte Waffe sowie ihre Animation wurden nicht von mir erstellt. Der genutzte Animationcontroller wurde mir vom Assetpack ohne Schnittstellen, aber mit fertig eingefügten Animationen bereitgestellt.
3) Der Blitz-Effekt beim Schießen stammt nicht von mir sondern ist aus dem Pack „LightningBolt“.
4) Der von mir genutzte Sound stammt einmal aus dem Assetpack „Ultra SF Game Audio Weapons Pack v 1“, sowie von der Internetseite „epidemicsound.com“, auf welcher ich berechtigt bin, Musik für private Projekte zu verwenden.