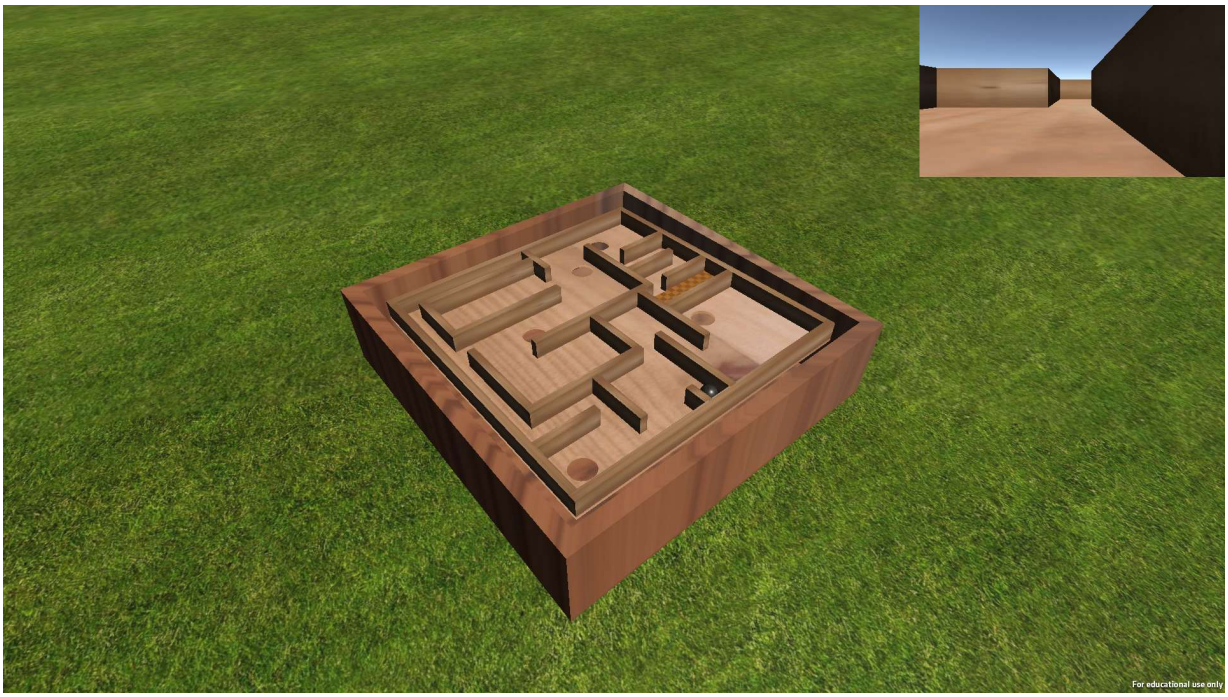
Ball-in-the-Maze



Ziele

Das Ziel dieses Übungsblattes ist ein Ball-in-the-Maze Game und soll zum Verständnis und Vertiefung der folgenden Kompetenzen führen:

- einfaches Importieren von Assets (.fbx)
- einfaches Texturieren
- einfaches Verwenden von Physical Materials
- Verstehen von Time.deltaTime
- Bewegen von Objekten mittels direktem Zugriff auf die Position und Rotation über Code
- Erzeugen von GameObjects und Anpassen in deren Größe
- Collider (isTrigger + normal) sowie den Zugriff auf die jeweilige Funktion
- Erzeugen von Win- und Lose-Conditions über Code + Reset des Games
- public Variablen in Skripten + dessen Auswirkungen in Unity
- Verwenden mehrerer Kameras in einer Szene

Anleitung

Wir werden nun Schritt für Schritt das Ball-in-the-Maze Game nachbauen. Das Spiel kann unter https://antx.at/games/maze/ im Browser mittels WASD gespielt werden.

Zuerst muss ein neues Projekt erstellt werden. Dabei kann der Unity Hub verwendet werden. Der Projekttitel heißt "BallInMaze".

Als nächstes erstellen wir die passende Ordnerstruktur, damit unser Projekt auch zusammengeräumt wirkt. Dabei erstellen wir die Ordner Imports, Materials, PhysMaterials, Scenes, Scripts und Textures.

Damit sich keine Spinnweben in den Ordnern bilden, werden wir sie auch gleich passend befüllen. Dazu eignen sich die beiliegenden Dateien. Wir packen die Texturen (.jpg) in den "Textures" Ordner und die zwei zu importierenden Assets in den Imports Ordner (.fbx).

Bevor wir uns mit dem Programmieren beschäftigen, versuchen wir eine Umgebung zu schaffen, auf der wir später unsere Codestücke auch testen können. Dementsprechend ziehen wir unsere zwei Objekte aus dem Imports Ordner in die Szene und platzieren sie so, dass die große Box unter dem Labyrinth ist (z.B. mit Y = -4). Dem Labyrinth geben wir einen Rigidbody, wobei wir die Eigenschaft Is Kinematic auf true setzen (Häckchen setzen!) und dazu geben wir noch einen Mesh Collider. Der darunter befindlichen Box geben wir einen Box Collider, wobei wir die Größe des Box Colliders so anpassen, dass lediglich der Boden vom Collider erfasst wird.

Wenn wir schon dabei sind, sollten wir das Ziel erstellen, also jener Bereich, der als Win Condition dient. Dafür erstellen wir ein neues GameObject und ziehen es auf eine Stelle auf die Platte. Dabei machen wir es recht dünn.

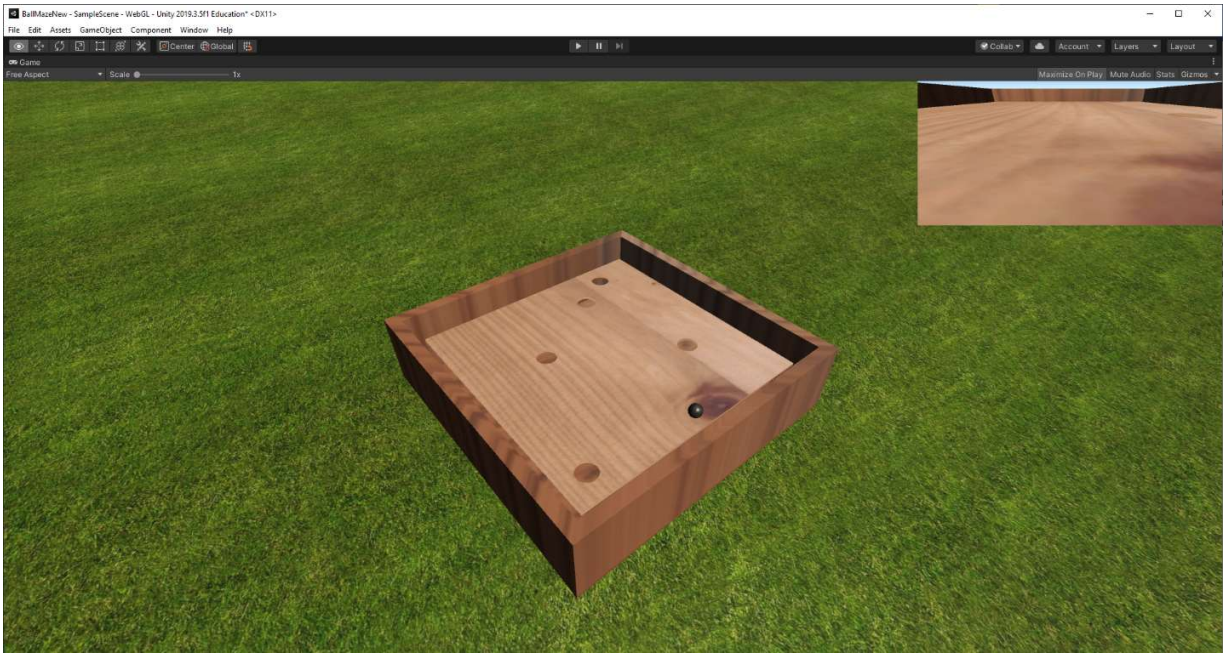
Damit wir die Szene gut überblicken können, geben wir der Kamera eine dementsprechende Position. Und wenn wir schon dabei sind, legen wir uns auch gleich eine zweite Kamera an, die wir am Bild rechts oben platzieren. Diese zweite Kamera soll dann unsere First-Person-View von der Kugel ausgehend sein. Mittels dem Viewport Rect in der Komponente Camera kann die Position der Kameraausgabe am Bildschirm verändert werden, mittels der Tiefe können wir bestimmen, welche Kameraausgabe über welcher liegen soll. Die First-Person-Kamera soll über der Übersichtskamera liegen.

Weiters platzieren wir eine Kugel in der Szene, die dann vom Spieler oder der Spielerin gesteuert werden kann. Der Kugel geben wir auch einen Rigidbody. Abschließend platzieren wir noch eine Plane (GameObject -> 3D Object -> Plane) unter der Box mit dem Labyrinth in der Szene und machen sie so groß, dass der gesamte Kamerabereich sie erfasst. Dort werden wir dann unseren Grasuntergrund rauftexturieren.

Jetzt haben wir einen Haufen Texturen aber noch keine Materials. Legen wir also für jede Textur ein Material im Ordner Materials an und ziehen die jeweilige Textur auf das entsprechende Albedo. Danach geben wir der Box die dunkle Holztextur, und dem Labyrinth die helle. Der Sphere geben wir das metallene Aussehen, der Plane unten die Grastextur (Tiling X=5, Y=5). Abschließend texturieren wir noch das Ziel mit der entsprechenden Textur. Wenn wir gerade beim Ziel sind, setzen wir auch noch beim Box Collider am Ziel Is Trigger auf true (Häckchen setzen!).

Ein Physic Material erzeugen und legen wir auch noch im entsprechenden Ordner an und ziehen es auf das Labyrinth. Die Parameter können wir nach Belieben anpassen.

So in etwa sollte die Szene jetzt aussehen:



Nun schreiben wir den Code für die 5 Skripte nach den unten angegebenen Spezifikationen, wobei folgendes zu beachten ist:

Wir beginnen mit dem MazeControl Skript und hängen es an die Platte mit den Löchern. Wir programmieren erst weiter, wenn das MazeControl Skript funktioniert (wir können auf beiden Achsen die Plattform nie zu weit kippen lassen). Das MazeControl Skript hat zwei öffentliche (public) Eigenschaften. Die Kugel muss auf „Player“ im Inspektor gezogen werden, mit dem Movement Speed können wir uns spielen, um eine passende Geschwindigkeit zu finden (15 z.B.).

Danach machen wir das Won Skript. Das GameOver Skript ist recht ähnlich. Das Won Skript hängen wir an das Ziel GameObject, das GameOver Skript an die Box. Das Won und das GameOver Skript haben eine öffentliche (public) Variable, nämlich das Maze selbst. Hier ziehen wir das GameObject hinein.

Abschließend schreiben wir das CameraFollow Skript und hängen es an die Kamera in der Kugel, um danach das TurnCameraToDirection Skript zu schreiben und ebenfalls an die Kamera hängen. Auch hier haben beide Skripts öffentliche Eigenschaften, wobei wir bei beiden die Kugel als GameObject reinziehen.

Nachdem wir programmiert haben, wird es Zeit, das Labyrinth selbst zu erstellen. Dabei erstellen wir mittels GameObject -> 3D Object -> Cube immer neue Würfel und ändern ihren Scale. Sobald wir der Meinung sind, dass das Labyrinth unseren Anforderungen entspricht, wird es Zeit für einen ersten Playtest. Wir dürfen dabei aber nicht vergessen, die ganzen Teile des Labyrinths unserem GameObject unterzuordnen, das für das Kippen verantwortlich ist, ansonsten werden sich die Teile des Labyrinths nicht mitbewegen :-).

MazeControl.cs

Aufgabe		
Steuert das Labyrinth.		
hängt an …		
dem Labyrinth.		
Variablen		
Sichtbarkeit	*Typ*	*Variablenname*
public	GameObject	player
public	float	movementSpeed
private	float	maxAngle
private	Vector3	playerStartPos

Funktionen	
Funktionskopf	*Beschreibung*
`void Start()`	Hier speichern wir uns zu Beginn die Position der Kugel, damit wir sie, wenn wir den Spielzustand wiederherstellen, am gleichen Ausgangspunkt startet. Daher speichern wir uns die position der Variable `player` in `playerStartPos`.
`void Update()`	Hier steuern wir die Platte, auf der sich die Kugel dann befindet. Mittels den Tasten W, A, S, D (oder anderen) soll die Platte auf zwei Achsen rotiert werden, sofern der jeweilige Winkel `maxAngle` nicht übersteigt. Dabei soll `transform.Rotate` verwendet werden, dem ein Vector3 übergeben wird. Dieser Vector3 stellt die Menge dar, um die das Objekt rotiert werden soll. Die Menge gibt dabei `movementSpeed` vor. Um eine frameunabhängige Steuerung gewährleisten zu können, muss `movementSpeed` mit `Time.deltaTime` multipliziert werden. Beim `transform.Rotate` sollte als zweiter Parameter bei einer Achse `Space.Self` und bei der anderen Achse `Space.World` übergeben werden.
`public void resetMaze()`	Stellt den ursprünglichen Zustand des Spiels wieder her. Das Maze wird auf die Einheitsrotation zurückgesetzt (`Quaternion.identity`). Das „`player`“ GameObjekt wird auf seine Startposition zurückgesetzt (`playerStartPos`) und der Geschwindigkeitsvektor der Kugel am `Rigidbody` auf 0 (`Vector3.zero`) gesetzt.
`private float convertAngle(float angle)`	Wandelt den Winkel (0° – 360°) in (-180° - 180°) um.

Won.cs

Aufgabe		
Stellt die Win Condition des Spiels da. Wenn der Spieler oder die Spielerin die Kugel ins Ziel bringt, hat er oder sie gewonnen.		
hängt an …		
Objekt in der Szene, welches das Ziel darstellt.		
Variablen		
Sichtbarkeit	*Typ*	*Variablenname*
public	MazeControl	maze
Funktionen		
Funktionskopf	*Beschreibung*	
`void OnTriggerEnter(Collider other)`	Die Funktion wird aufgerufen, sobald ein GameObject mit einem Collider, das Objekt, an dem dieses Skript hängt, berührt. Sollte das andere Objekt (`other`) der Spieler sein (ist der Tag vom Objekt `other` „Player"?), geben wir ein „Gewonnen!" aus (z.B. mittels `Debug.Log`) und rufen die Funktion `resetMaze()` von `maze` auf, sodass der ursprüngliche Zustand für eine neue Runde hergestellt wird.	

GameOver.cs

Aufgabe		
Stellt die Lose Condition des Spiels da. Wenn der Spieler oder die Spielerin die Kugel ins ein Loch steuert, hat er oder sie verloren.		
hängt an …		
der Box in der Szene. Wenn die Kugel die Box berührt, hat der Spieler oder die Spieler verloren.		
Variablen		
Sichtbarkeit	*Typ*	*Variablenname*
public	MazeControl	maze
Funktionen		
Funktionskopf	*Beschreibung*	
`void OnCollisionEnter(Collision collision)`	Die Funktion wird aufgerufen, sobald ein GameObject mit einem Collider, das Objekt, an dem dieses Skript hängt, berührt. Sollte das andere Objekt (`other`) der Spieler sein (ist der Tag vom Objekt `other` „Player"?), geben wir ein „Game Over!" aus (z.B. mittels `Debug.Log`) und rufen die Funktion `resetMaze()` von `maze` auf, sodass der ursprüngliche Zustand für eine neue Runde hergestellt wird.	

CameraFollow.cs

Aufgabe		
Bewegt eine Kamera (oder ein anderes GameObject) in Abhängigkeit eines anderen Objekts. Das heißt, bewegt sich das eine Objekt um 1 auf der x-Achse nach vorn, so wird dieses Verhalten durch das Skript auf das Objekt, an dem das Skript hängt, übertragen. Zusätzlich kann ein Translationsvektor übergeben werden (wenn der Translationsvektor 0,0,0 ist, ist die Kamera an derselben Stelle wie das zu verfolgende Objekts).		
hängt an …		
der Kugelkamera in der Szene.		
Variablen		
Sichtbarkeit	*Typ*	*Variablenname*
public	Vector3	translationVector
public	Transform	transformToFollow

Funktionen	
Funktionskopf	*Beschreibung*
`void Update()`	Die Position des Objekts (`transform.position`), an dessen dieses Skript hängt, entspricht immer der Position des übergebenen Objekts (`transformToFollow.position`) plus eines Translationsvektors (`translationVector`).

TurnCameraToDirection.cs

Aufgabe		
Dreht die Kamera (oder ein Objekt) in Abhängigkeit eines Richtungsvektors eines anderen Objekts.		
hängt an …		
der Kugelkamera in der Szene.		
Variablen		
Sichtbarkeit	*Typ*	*Variablenname*
public	GameObject	gameObjectToGetDirectionFrom
private	Rigidbody	rb

Funktionen	
Funktionskopf	*Beschreibung*
`void Start()`	Hier speichern wir uns in `rb` den Rigidbody vom anderen Objekt (`gameObjectToGetDirectionFrom`), damit wir später einfacher darauf zugreifen können. Dafür verwenden wir die `GetComponent<Rigidbody>()` Funktion!
`void Update()`	Um die Kamera in die Richtung des anderen Objekts schauen lassen zu können, müssen wir in jedem Update die Orientierung (Rotation) anpassen. Wir setzen hier daher die Rotation auf den Richtungsvektor durch Verwenden von der Funktion `Quaternion.LookRotation` und übergeben dabei normalisiert die Eigenschaft `velocity` vom Rigidbody (`rb`).

Punkteliste

Aufgabe	Punkte
Die Plattform lässt sich mittels WASD steuern.	2
Das Labyrinth besteht zumindest aus 8 Wänden.	2
Das Spiel setzt den Ball und die Rotation der Spielfläche beim Gewinnen oder Verlieren zurück.	2
Das Spiel kann, sobald die Kugel die Zielfläche berührt, gewonnen werden.	2
Das Spiel kann, sobald die Kugel die Box unterhalb des Labyrinths berührt, verloren werden.	2
Die Kugelkamera verfolgt mittels Skript die Kugel.	2
Die Kugelkamera dreht sich in Abhängigkeit des Richtungsvektors der Kugel.	2
Das Spiel hat eine kleine zusätzliche Veränderung (z.B. eine Rampe, bewegende Wand).	1
Das Spiel hat weitere, kleine Veränderungen, die über die Anforderungen des Übungsblatt hinausgehen. Pro Veränderung 1-2 Punkte (je nach Schwierigkeit), max. 5 Punkte.	5 (Bonus)

P.S.: Ihr könnt gerne die zu importierenden Objekte bzw. Texturen selbst modellieren bzw. erstellen und müsst nicht die Objekte aus der Angabe nehmen. Es soll am Schluss halt noch als Ball-in-the-Maze Game erkennbar sein :-).

Abgabe

Bitte das Spiel für Windows builden, zippen und ggf. ein Textfile mit Besonderheiten dazulegen (z.B., wenn andere Tasten als WASD verwendet wurden, wenn etwas nicht funktioniert, oder zusätzliche Veränderungen dokumentiert werden sollen).

Das Zip-File sollte bitte folgendes Format haben:

3DGAMEDEV_UB1_<Nachname>.zip

z.B.: 3DGAMEDEV_UB1_Güldenpfennig.zip