Notizen Kapitel 6: Dynamic Motion: Implementing Movement and Spawning

Implementing movement

Unser Spieler (Robot) soll sich bewegen können!

- Erstelle Player Figur → **Polyart Mesh** Prefab in die Scene ziehen → Name zu **Player** ändern → **Add Component / New Script** namens **PlayerMovement** hinzufügen.
- Neues Script im **Scripts Ordner** ablegen, dann Script öffnen.
- Zu Bewegung wird nur die **Update**-Funktion benötigt → **Start()** löschen.
- 1. Versuch die Bewegung hinzuzufügen:
  `transform.Translate(0,0,1);`
  → nach **Save** und **Run** bewegt sich der mesh viel zu schnell (längs z-Achse)!
  → Zur Mitverfolgung ev. Kameraposition ändern!
  Begründung: Der Computer generiert sehr viele (z.B. 50) frames per second (fps). Das wird momentan nicht kontrolliert und dann sind 50 m/s eben viel zu viel.
- 2. Versuch: Kontrolliere die Bewegung mittels **speed-Variable**:
  `public float speed;` → `transform.Translate(0,0,speed);`
  → nun können wir den speed-Wert im Inspector deklarieren, z.B. 0.05 → Bewegung ist moderater, aber immer noch zu schnell und v.a. invariabel → korrigieren wir bald.

Zur Gegenüberstellung: **Bewegung des Flying Enemy's mittels visual graph:**

- Flying Enemy anwählen → **Add Component / Visual Scripting / Script Machine**.
- Im Inspector und Script Machine **New** anwählen → **FlyingEnemyMovement** unter Scripts anlegen, dann öffnen (Doppelklick).
- **On Start**-event löschen.
- Im Inspector unter Variables neue float-Variable **speed** erzeugen → Wert z.B. 0.03 → die Variable in den Graph hinein ziehen (an den grauen Balken links).
- An **On Update** einen **Transform**-node anhängen: **Transform: Translate (X, Y, Z)**.
- → Ausgang **Get Variable-node** mit **Z-Komponente des Translate-nodes** verbinden.
- Run zeigt, wie sich der Player und der Flying Enemy bewegen. Noch ruckelt es und zwar genau dann, wenn der non-flying enemy das Terrain berührt und umkippt… darum kümmern wir uns momentan nicht.

Input verwenden

Unser Robot Mesh soll sich nach Tastatureingabe bewegen. Dazu programmieren wir:

```
5   public class PlayerMovement : MonoBehaviour
6   {
7       public float speed;
8
9       // Update is called once per frame
10      void Update()
11      {
12          if (Input.GetKey(KeyCode.W))
13          {
14              transform.Translate(0, 0, speed);
15          }
16      }
17  }
18
```

Erläuterungen zu diesen Programmschritten:

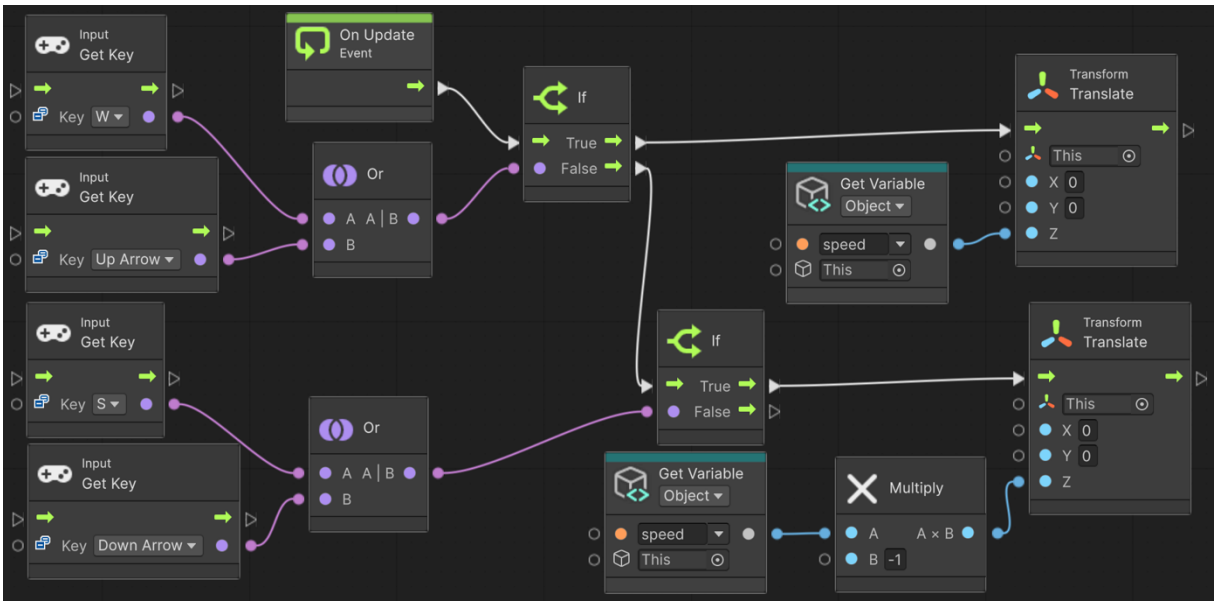
- Wir verwenden nach wie vor die **speed**-Variable.
- In der Update-Funktion wird jeweils überprüft, ob die Taste W gerade gedrückt ist. Wir sehen, wie das gemacht wird: `Input.GetKey(KeyCode.W)`. Diese ganze Struktur wird in der **UnityEngine**-Library definiert. Wir brauchen uns nicht darum zu kümmern, sondern wenden diese vordefinierten Befehle einfach an. Wie praktisch!
- Falls die W-Taste gedrückt ist, wird eine **Verschiebung in z-Richtung** vorgenommen.

Natürlich möchten wir auch Bewegungen in die anderen (horizontalen) Richtungen ausführen können. Zudem sollen diese Bewegungen auch unter Verwendung der Pfeiltasten erfolgen können. Wir erweitern daher unsere Programmierung wie folgt:

```
10          void Update()
11          {
12              if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow)) { transform.Translate(0, 0, speed); }
13              if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow)) { transform.Translate(0, 0, -speed); }
14              if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow)) { transform.Translate(-speed, 0, 0); }
15              if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow)) { transform.Translate(speed, 0, 0); }
16          }
17      }
18
```

- Die Doppelstriche || stehen jeweils für ein logisches ODER. Wenn die W-Taste oder die Taste mit dem Pfeil nach oben gedrückt wird, erfolgt eine Bewegung nach vorne.
- Beachte die Minuszeichen für die Bewegungen nach links oder nach hinten.

Wir bemerken hier, wie mühsam eine entsprechende Eingabe mittel visual graph wäre:



Und dies wäre lediglich die Eingabe für die vorwärts-rückwärts-Komponente!
Wir erkennen: Sobald es komplizierter wird, ist schriftliche Programmierung angesagt!
(Profis arbeiten praktisch ausschliesslich schriftlich.)

«Delta Time» verstehen

- Momentan hängt die **FPS-Rate** («frames per second») von der Computerleistung, aber auch davon ab, was denn im jeweiligen frame zu rechnen ist. So entstehen meistens sehr schnelle Bewegungen, die dann aber eben plötzlich durch kurzzeitige Stillstände unterbrochen sind, wenn auf einmal mehr Rechenpower benötigt wird. Das ist aus vielen, sehr offensichtlichen Gründen schlecht für das Gameplay und würde so auch dazu führen, dass sich ein Game auf verschiedenen Rechnern unterschiedlich gut spielen lassen würde – völlig inakzeptabel!
- Der Wert **«Delta Time»** – in der Implementierung: `Time.deltaTime` – liefert uns nun eine sehr praktische Angabe, nämlich wie viel Zeit seit Beginn des frames vergangen ist.
- Indem wir alle aus den Inputs hervorgehenden Werte mit deltaTime multiplizieren, skalieren wir die Bewegungen entsprechend der Zeit, mit der der einzelne Computer die Operationen ausführen kann. Damit werden die Bewegungen auf allen Maschinen gleich schnell aussehen:

```
11      void Update()
12      {
13          if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow)) { transform.Translate(0, 0, speed * Time.deltaTime); }
14          if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow)) { transform.Translate(0, 0, -speed * Time.deltaTime); }
15          if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow)) { transform.Translate(-speed * Time.deltaTime, 0, 0); }
16          if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow)) { transform.Translate(speed * Time.deltaTime, 0, 0); }
17
18          float mouseX = Input.GetAxis("Mouse X");
19          transform.Rotate(0, mouseX * rotationSpeed * Time.deltaTime, 0);
20      }
```

Erzeugung eines «bullet»-Prefabs

Unser Demonstrationsobjekt für das **spawning** (= engl. *etw. ablegen/hervorbringen/erzeugen* oder *laichen*) sollen Kugeln (= bullets) für die Schüsse des Player-Robots sein. Dazu benötigen wir einen **Bullet-Prefab**. Bei jedem Schuss soll davon eine neue Version ins Spiel kommen.

- Erzeuge eine Sphere via **GameObject / 3D Object / Sphere** und nenne sie **Bullet**.
- Erzeuge ein neues **Material** in der **project view**, nenne es ebenfalls **Bullet** und lege es im Ordner **Materials** ab.
- Wähle das Bullet-Material an und gib ihm unter **Base Map** und **Emission Map** die Farbe **Rot** – unter Emission muss der Check gesetzt sein!
  (N.B.: In RGB hat Rot die Koordinaten (255, 0, 0) – das hilft.)
- Wende das Bullet-Material auf die Bullet an (einfach Material auf Bullet ziehen).
- Schrumpfe die Kugel: **Scale** auf 0.2 in allen drei Raumrichtungen.
- Versieh die Bullet mit einem **Script** namens **ForwardMovement** mit einer konstanten Bewegung (in Abhängigkeit einer Variable **speed**) in z-Richtung. Lege das neue Script im Ordner **Scripts** ab.
- Ziehe die Bullet aus der **Hierarchy view** in den Ordner **Prefabs**, um den Prefab zu erzeugen.
- Entferne schliesslich noch das ursprüngliche Bullet-GameObject.
- Wir verfügen nun über einen Bullet-Prefab, von dem wir beliebig viele Kopien herstellen und in die Scene einfügen können.

Instanzierung einer Bullet-Ausgabe

Der Player (resp. sein Robot) soll schiessen können. D.h., wenn wir beim Spielen auf die linke Maustaste klicken, soll ein Schuss abgegeben werden resp. eine Kugel ins Spiel kommen. Dazu muss der Player (!) ein zusätzliches Script erhalten:

- Erzeuge im Inspector zum **Player**-GameObjects ein neues **Script** namens **PlayerShooting**. (Verschiebe auch dieses Script in den Script-Ordner.)
- Sorge dafür, dass das Script den folgenden Code enthält:

```
5       public class PlayerShooting : MonoBehaviour
6       {
7           public GameObject prefab;
8
9           // Update is called once per frame
10          void Update()
11          {
12              if (Input.GetKeyDown(KeyCode.Mouse0))
13              {
14                  Instantiate(prefab);
15              }
16          }
17      }
```

- Was macht dieser Code genau?
    - Die neue Klasse **PlayerShooting** enthält die Variable **prefab**. Dies ist nun nicht mehr irgendeine Zahl (z.B. float, integer), sondern es handelt sich um ein im Prinzip beliebiges GameObject, dem im Code der Name **prefab** gegeben wird.
    - Wird die linke Maustaste gedrückt, so wird dadurch in der Update-Funktion eine Ausgabe (= Instanz) des prefabs ins Leben gerufen. Dieser Vorgang nennt sich **Instanzierung**.
    - **Beachte:** Hier wird nun nicht mehr der Abfragebefehl **GetKey**, sondern stattdessen **GetKeyDown** verwendet. Unterschied: GetKeyDown liefert die Information, ob seit dem letzten frame die neu Taste gedrückt wurde, während GetKey schaut, ob die Taste jetzt gerade gedrückt ist. (Wo liegt wohl der Unterschied? Wir können es bald ausprobieren…)
    - **Zum genaueren Verständnis:** Sobald der Player im Spiel ist – und das ist er von Anfang an – läuft für ihn das Script PlayerShooting. Dieses Script wird aktiv, wenn der Mausklick erfolgt. Dann wird eine Ausgabe des prefabs erzeugt.
- **Aber woher weiss denn nun unser Script, welches GameObject mit dem prefab gemeint ist?**
- Die Antwort ist eigentlich ganz einfach: Selbstverständlich müssen wir das dem Script noch sagen, und zwar genau so, wie wir früher z.B. bei der Variable speed einen Wert eingeben mussten. Das passiert aber nicht im Script, sondern im Inspector zum Player-GameObject → **speichere das Script** und kehre zu Unity zurück. Dort erscheint nun im Inspector bei unserem neuen Script die Variable **Prefab**, die noch nicht mit einem konkreten Wert belegt ist («None»). Ziehe den Prefab **bullet** aus dem Prefabs-Ordner in der Project View in dieses Variablen-Fenster. (Alternativ kannst du auch auf den kleinen Ring mit Punkt rechts daneben klicken und dort den Prefab bullet aussuchen.)
- Nun können durch Mausklicks Kugeln abgefeuert werden → speichere und lass das Spiel laufen und probiere das Schiessen aus!

- **Aber wo sind denn unsere Kugeln?!** Vermutlich siehst du sie nicht – weil wir keinen guten Erzeugungsort deklariert haben! Das werden wir gleich tun. Die Kugeln werden aber ganz bestimmt erzeugt, wie wir in der **Hierarchy view** mitverfolgen können.
- **Die Instanz der Kugel passend platzieren – Variante 1:** Beim Instanzieren können Position und Orientierung direkt gesetzt werden. Dazu werden die entsprechenden Informationen des Objekts abgerufen, zu dem das Script gehört. Der **Instantiate**-Befehl weiss, was gemeint ist:

```
12          if (Input.GetKeyDown(KeyCode.Mouse0))
13          {
14              Instantiate(prefab, transform.position, transform.rotation);
15          }
```

- **Die Instanz der Kugel passend platzieren – Variante 2:** Wir verwenden den vorherigen Instantiate-Befehl, um ein neues GameObject namens **clone** zu erzeugen und setzen hinterher dessen Ort und Orientierung:

```
12          if (Input.GetKeyDown(KeyCode.Mouse0))
13          {
14              GameObject clone = Instantiate(prefab);
15              clone.transform.position = transform.position;
16              clone.transform.rotation = transform.rotation;
17          }
```

  Beachte, wie sich **transform.position** und **transform.rotation** auf verschiedene Objekte beziehen und beim Abrufen oder Setzen eines Wertes verwendet werden.
- Beide Varianten funktionieren. Die zweite ist zu bevorzugen, wenn wir später mit der Kugel, also mit dem **clone**, noch weitere Dinge anstellen wollen.
- **Problem:** Die Kugeln erscheinen nicht beim Gewehr, sondern am Boden. Dort befindet sich der **Drehpunkt (pivot)** des Players, der per default verwendet wird.
- **Idee:** Wir erzeugen ein neues GameObject **ShootPoint**, das dem Player-Objekt untergeordnet ist, und verwenden dieses zur Festlegung der Abschusspunktes:
  - In **Hierarchy view** ein **EmptyObject** hinzufügen, diesem den Namen **ShootPoint** geben und dem **Player** unterordnen.
  - Platziere in der **Scene view** den **ShootPoint** so, wie es dir passt.
  - Nun muss natürlich dem Player noch beigebracht werden, diesen ShootPoint auch wirklich zu verwenden. Ergänzungen im **PlayerShooting-Script**:

```
5       public class PlayerShooting : MonoBehaviour
6       {
7           public GameObject prefab;
8           public GameObject shootPoint;
9
10          // Update is called once per frame
11          void Update()
12          {
13              if (Input.GetKeyDown(KeyCode.Mouse0))
14              {
15                  GameObject clone = Instantiate(prefab);
16                  clone.transform.position = shootPoint.transform.position;
17                  clone.transform.rotation = shootPoint.transform.rotation;
18              }
19          }
20      }
```

  - Die neue Variable shootPoint muss im Player-Inspector jetzt noch mit dem passenden Wert befüllt werden → **ShootPoint**!
- Nun haben wir gelernt, wie wir mit Scripts Objekte und Vorgänge steuern. So funktioniert Unity!

Cursor-Probleme beheben

Wenn wir momentan das Game laufen lassen und dann den Spieler mit der Maus drehen, so kann es sein, dass der Cursor den Spielbereich verlässt. Klicken wir dann drauf, z.B. weil wir schiessen möchten, so verlassen wir die Game view und wählen irgendetwas anderes an, ohne das effektiv zu wollen. Mit wenigen Ergänzungen können wir diese Problematik vermeiden:

- Wir öffnen das Script **PlayerMovement** und ergänzen es um eine Start-Funktion:

```
10          void Start()
11          {
12              Cursor.visible = false;
13              Cursor.lockState = CursorLockMode.Locked;
14          }
```

- Sobald wir die erste Aktion mit dem Player ausführen, verschwindet der Cursor und es gibt kein Problem mehr mit ungewollten Klicks auf andere Objekte.
- Nun müssen wir ESC klicken, um den Cursor wieder zu sehen.

Timing-Aktionen

Oftmals passieren Auswirkungen einer Aktion erst mit Verspätung. Z.B. möchten wir die abgeschossenen Kugeln nach einer gewissen Zeit wieder zerstören, damit das Memory auf Dauer nicht von Kugeln überflutet ist. Oder wir möchten kontrollieren, mit welcher Rate resp. zu welchen Zeitpunkten die Gegner erzeugt werden.

In unserer Demo sollen die Gegner «in Wellen» erzeugt werden. Alle 20 Sekunden sollen während 5 Sekunden pro Sekunde je zwei Gegner erzeugt werden. Für die Gegner wollen wir denselben Robot Mesh wie für unseren Player verwenden.

- Wir **löschen** alle bisherigen GameObjects namens **Enemy**. Den Enemy Prefab können wir z.B. in Simple Enemy umbenennen. Den Flying Enemy können wir so belassen.
- Ziehe den heruntergeladenen **Mesh Prefab (Polyart_Mesh)** von der **Project view** in die **Scene view**. Gib dem Robot den Namen **Enemy**.
- Füge dem Robot das Script **ForwardMovement** hinzu und setze den **speed** auf 10.
- Ziehe den Enemy in den Prefabs Ordner, sodass eine **Prefab-Variante** des Polyart_Meshs erzeugt wird. Sie soll **Enemy** heissen.
- **Lösche** das jetzt noch vorhandene **Enemy Game Object**.
- Erzeuge den Ort des spawnings → erzeuge ein **Empty Game Object** → platziere es in einer Ecke der Plattform und nenne es **Wave1a**.
- Gib ein Script namens **WaveSpawner** hinzu → **Add Component** etc. → Script im **Scripts Ordner** versorgen.
- Im Script des WaveSpawners werden vier Variablen benötigt:

```
5       public class WaveSpawner : MonoBehaviour
6       {
7           public GameObject prefab;
8           public float startTime;
9           public float endTime;
10          public float spawnInterval;
11      }
```

- Nun muss das spawning geplant werden. Mit der `InvokeRepeating`-Funktion muss das nur einmal gemacht werden, nämlich in der **Start**-Funktion naheliegt.

```
12          // Start is called before the first frame update
13          void Start()
14          {
15              InvokeRepeating("Spawn", startTime, spawnInterval);
16          }
```

Wie muss die **InvokeRepeating**-Funktion aufgerufen werden? Der Name der Funktion, die wiederholt werden soll, muss **in Anführungszeichen** als erstes Argument angegeben werden. Dahinter folgen Startzeitpunkt (startTime) und die Zeit bis zur nächsten Wiederholung (spawnIntervall).

- Innerhalb der Spawn-Funktion muss nun das eigentliche spawning stattfinden:

```
19          void Spawn() {
20              Instantiate(prefab, transform.position, transform.rotation);
21          }
```

Übernommen werden Ort und Ausrichtung unseres spawning-Objekts Wave1a.

- Game ausführen: **Script speichern!** → im **Wave1a** Objekt Variablenwerte eingeben (Prefab: Enemy, startTime: 2, endTime: 7, spawnRate: 0.5).
- Kurz nach dem Start der Ausführung werden während 2 Enemies pro Sekunde erzeugt, die dann auch gerade lossausen. Allerdings: Das Spawning hat kein Ende!
- Selbstverständlich muss auch dieses Ende im Script deklariert werden:

```
12          // Start is called before the first frame update
13          void Start()
14          {
15              InvokeRepeating("Spawn", startTime, spawnInterval);
16              Invoke("CancelInvoke", endTime);
17          }
```

Für die Beendigung des spawnings wird keine neue Funktion benötigt, vielmehr bewirkt die **Invoke**-Funktion mit Argument **CancelInvoke** das Ende anderer Invoke-Funktionen zum Zeitpunkt **endTime**.

Halten wir nochmals fest, wie wir Funktionen timen: Wir packen die eigentlichen Funktionen (im Bsp. oben Spawn()) in andere Funktionen hinein, die das timing übernehmen (InvokeRepeating() und Invoke()). Solche «Verpackungsfunktionen» nennt man **wrapper**.

Objekte entfernen

Wir kümmern uns nun noch um das Entfernen der Kugeln…

- Füge dem **Bullet-Prefab** ein Script namens Autodestroy hinzu → **Add Component** → **New Script** → Name «Autodestroy eingeben» → anlegen und danach Script von Assets in den **Scripts-Ordner** verschieben → Script öffnen und Code eingeben:

```
5       public class Autodestroy : MonoBehaviour
6       {
7           public float delay;
8
9           void Start()
10          {
11              Destroy(gameObject, delay);
12          }
13      }
```

Beachte erstens, dass dieses Script keine Definition der Variable gameObject benötigt. Das Script «weiss», dass sich dieser Ausdruck auf das gameObject bezieht, dem es angegliedert wird. Zweitens können wir die «Zerstörung» des gameObjects mittels einer **delay**-Variable verzögern.

- Speichere das Script und gib in Unity einen Wert für delay ein, z.B. 5 (sec) → beim Ausführen werden nun bullets erzeugt, die sich nach 5 Sekunden selber zerstören.

Das neue Input System installieren

Zwar reichen unsere bisherigen Input-Befehle der UnityEngine im Prinzip bestens. Aber dabei gibt es doch Einschränkungen, vor allem wenn es um die Verwendung neuerer Eingabe-Hardware und -Mappings geht. Daher werden wir ein neues Eingabesystem verwenden. Dieses muss wie folgt installiert werden:

- Öffne den Package Manager (Menü Windows → Package Manager) und suche dann unter Unity Registry nach dem Input Manager. Die aktuelle Version ist 1.7.0.
- Klicke auf Install.
- Im nachfolgenden Dialog ist auf Yes zu klicken, um das neue Input System zu aktivieren.

Input Mappings erzeugen

Einer der grossen Vorteile des neuen Input Systems ist das Input Mapping, mit dem der physische Input sauber vom «spiellogischen» Input getrennt (resp. eigentlich eher übersichtlich mit diesem verbunden) werden kann. Im Code sollte gefragt werden, ob der Schuss-Knopf gedrückt wurde, anstelle der Abfrage des Mausklicks. Wir möchten also direkt mit Begriffen der Game-Logik programmieren und uns nicht fragen, wie der Input physisch eigentlich erfolgt.

- Wähle das **Player** Object an → **Add Component** → **Player Input** → **Create Actions**.
- Gib den neuen Inputactions den Namen **SuperShooter**.
- Falls nicht schon offen, lassen sich die Input Actions durch Doppelklick auf SuperShooter unter Player Input öffnen.
- Wir sehen, dass wir für die Steuerung des Players nun standardmässig drei Bereiche – Move, Look und Fire – offeriert bekommen. (Öffnen der entsprechenden Menüs zeigt, mit welchen Eingabeaktionen diese Spielaktionen verbunden sind.) Das brauchen wir jetzt aber gar nicht mehr genau zu wissen…
- Vielmehr können wir nun unsere Scripts entsprechend anpassen und sie durch die Deklaration «using UnityEngine.InputSystem» auf das neue Input System zugreifen lassen. Unsere Scripts müssen dazu wie folgt modifiziert werden:

```
3      using UnityEngine;
4      using UnityEngine.InputSystem;
5
6      public class PlayerShooting : MonoBehaviour
7      {
8          public GameObject prefab;
9          public GameObject shootPoint;
10
11         // Update is called once per frame
12         public void OnFire(InputValue value)
13         {
14             if (value.isPressed)
15             {
16                 GameObject clone = Instantiate(prefab);
17                 clone.transform.position = shootPoint.transform.position;
18                 clone.transform.rotation = shootPoint.transform.rotation;
19             }
20
21         }
22     }
```

```
3       using UnityEngine;
4       using UnityEngine.InputSystem;
5
6       public class PlayerMovement : MonoBehaviour
7       {
8           public float speed;
9           public float rotationSpeed;
10          private Vector2 movementValue;
11          private float lookValue;
12
13          private void Awake()
14          {
15              Cursor.visible = false;
16              Cursor.lockState = CursorLockMode.Locked;
17          }
18
19          public void OnMove(InputValue value)
20          {
21              movementValue = value.Get<Vector2>() * speed;
22          }
23
24          public void OnLook(InputValue value)
25          {
26              lookValue = value.Get<Vector2>().x * rotationSpeed;
27          }
28
29          // Update is called once per frame
30          void Update()
31          {
32              transform.Translate(
33                  movementValue.x * Time.deltaTime,
34                  0,
35                  movementValue.y * Time.deltaTime
36              );
37
38              transform.Rotate(0, lookValue * Time.deltaTime, 0);
39          }
40      }
```

Hierin sehen wir, wie nun immer dann, wenn eine Bewegung eingegeben wird oder geschossen wird (onMove, onFire) Aktionen ausgeführt werden. Diese können von Eingabewerten abhängen, die vom Input System geliefert werden.
Der Wert zu onMove ist einer 2er-Vektor mit einem Wert in x- und in z-Richtung. Der Wert zu onLook ist ebenfalls ein 2er-Vektor mit einem Rotationswert um die x- und um die y-Achse.
Aufgrund dieser Werte wird schliesslich in der Update-Funktion bewegt und rotiert. Erst dort wird die Zeitkorrektur (Time.deltaTime) ausgeführt.