Szakdolgozat

Miskolci Egyetem

Karaktermozgatás állapotgépekkel a Godot játékmotorban

Készítette:
Halász Máté Sándor
Programtervező informatikus

Témavezető:
Dr. Hornyák Olivér

Miskolc, 2025

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Halász Máté Sándor (T1TNWL) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: állapotgép, videójáték

A szakdolgozat címe: Karaktermozgatás állapotgépekkel a Godot játékmotorban

A feladat részletezése:

A szakdolgozat célja egy játékoskarakter mozgásának és interakcióinak megvalósítása állapotgépek alkalmazásával. A munka középpontjában egy olyan szoftverarchitektúra kialakítása áll, amely a játékos aktuális állapotát állapotgéppel reprezentálja, és ezen keresztül biztosítja a különböző mozdulatok, akciók és interakciók szabályozását.
A megvalósítás során minden mozdulat meghatározott előfeltételekhez lesz kötve, amelyek az állapotgépben egyértelműen modellezhetők. Például az ugrás végrehajtásának feltétele a "Grounded" (földön tartózkodó) állapot, míg a mozgás történhet földön és levegőben egyaránt. Az állapotgépek hierarchikus és egymásra épülő struktúrája lehetővé teszi az állapotok közötti átmenetek (pl. "Grounded" → "Jumping", "Grounded"→ "GroundedMoving", "Jumping" → "InAirMoving") pontos és bővíthető kezelését.
A téma jelentősége abban rejlik, hogy az állapotgépek használatával a játéklogika átláthatóbbá, modulárisabbá és könnyebben karbantarthatóvá válik. Az így létrehozott rendszer rugalmasan bővíthető új mozdulatokkal és állapotokkal, miközben a kód tisztább és strukturáltabb marad.

Témavezető: Dr. Hornyák Olivér (egyetemi docens)

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Halász Máté Sándor**; Neptun-kód: `T1TNWL` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Karaktermozgatás állapotgépekkel a Godot játékmotorban* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....
Hallgató

1.

A szakdolgozat feladat módosítása

szükséges (módosítás külön lapon)

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

A szakdolgozat bírálatra

bocsátható

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Contents

Chapter 1

Bevezetés

A videójátékok az elmúlt években egy dollármilliárdos iparággá nőtték ki magukat. Nem csak a rohamosan fejlődő technológiának, de a szoftveres ágon tett előre lepéseknek is hála. A 2000-es évek elején a videójáték gyártás mindössze a programozók (matematikusok) számára egy hobbi volt, ami az akkori hardverből a lehető legtöbbet való kihozásról szólt. Manapság a játékmotoroknak hála bárki elkezdhet játékokat gyártani, viszont ez nem azt jelenti, hogy sokkal egyszerűbb lenne a videójáték gyártók dolga. A rohamosan fejlődő ipar és a játékfejlesztés elérhetőségének a növekedése az elvárásokat is megnövelte a játékok iránt.

Mivel a számítástechnika a matematikából nőtte ki magát, főleg a kezdetekben, ezért sok gyakorlatot alkalmazunk belőle. Egyike ezeknek a gyakorlatoknak az állapotgépek alkalmazása egyes esetekben. Név szerint, azon esetekben ahol fontos a könnyű bővíthetőség és az, hogy ne kelljen újabb változókat létrehozni annak érdekében, hogy kiküszöböljünk egyes eseteket. A legnépszerűbb példa egy állapotgép használatára az a videójátékokhoz kapcsolódik. A kezdetekben ez nem volt olyan fontos, mivel maguk a játékok egyszerűbbek voltak, lásd: Doom, Wolfenstein, Elite (1984) ahol a legfontosabb a grafika megoldása és helyes kirajzolása volt. Manapság, ahol ún. "Omni-Movement" van a játékokban, ami, ahogyan Charles[1] is megfogalmazta a posztjában, egy olyan mozgásfajta, ahol a játékos teste a fejétől függetlenül mozog, tehát a test és a fej (kamera) mutathat két különböző irányba. Ezen esetben a játékosoknak közelről kell figyelni a mozgásukat és cselekedeteiket, és jól definiált struktúrákat kell létrehozni egyes mozgássorozatoknak. A játékosok képesek különböző mozdulatok végrehajtására, többek között: futás, ugrás, csúszás és vetődés. Ezek mind precíz beviteleket követelnek meg a játékostól és az állapotok pontos nyilvántartását. Ennek a kivitelezéséhez az állapotgépek használata elengedhetetlen, mind a mozgásrendszer kivitelezéséhez,

mind az animációk helyes lejátszásához.

A témámat, viszont nem az Omni-Movement inspirálta, az én projektemhez csak példaképpen kapcsolódik, hogy demonstráljam egy sokkal bonyolultabb koncepción is a témámat és megmutassam, hogy az iparban is használatos. Sokkal inkább a [Shibuya-punk esztétika](Shibuya-punk) királya és a mai napig a Sega egyik legkevésbé elismert kiadása a: Jet Set Radio (2000), és az az által inspirált Team Reptile játék a: Bomb Rush Cyberfunk (2023). Ezekről sokan nem hallottak, viszont, ha azt mondom, hogy "Tony Hawk Pro Skater", akkor az már több embernek fog ismerősen hangzani. A koncepció ugyan az mind a kettőnél: gurulj és csinálj menő trükköket (miközben próbálod magad nem összetörni, természetesen). Ha a felszínt nézzük is már eléggé bonyolultnak néz ki a helyzet, mind mozgás, mind pontszám számítás szempontjából; és elkezd járni az agyunk azon, hogy mégis mennyi állapotot kellet nyomon követniük a fejlesztőknek, hogy ezeket megvalósítsák. Annak érdekében, hogy egy kicsit mélyebbre ássak és megválaszoljam ezt a kérdést úgy döntöttem, hogy magam is nekiállok és létrehozok egy mozgásrendszert, amiben tesztelhetjük, hogy mennyire is érné meg állapotgépeket használni, miért és mennyivel eredményezne szebb, jobban strukturált, átláthatóbb, könnyebben bővíthető kódot, és gyorsabb programot, mint egy hagyományos if-and megoldás.

A szakdolgozat végére szeretnék egy teljesen működőképes és játszható mozgásrendszer-prototípust elkészíteni, amely bemutatja az állapotgépek fontosságát a mozgásrendszerek kialakításában és a játékfejlesztés más terein. Mind ezt a Shibuya-punk esztétikában többnyire magam által elkészített modellekből.

Chapter 2

Véges Állapotú Automaták

Ahogyan az informatika fejlődött és egyre több matematikai koncepciót adaptált, hogy segítse a fejlesztést, úgy távolodott is tőle. Meglepő módon nem kell ismernie az embernek a pontos matematikai definícióit egyes koncepcióknak annak érdekében, hogy használja őket fejlesztés során. Ez leginkább itt fog majd érződni, az automaták terén.

2.0.1 Matematikai szempont

A véges állapotú automata (Finite State Machine, avagy FSM) egy matematikai model, amely képes ábrázolni a dinamikus viselkedését egy rendszernek véges számú állapotot alkalmazva, valamint ezen állapotok közötti átmeneteket, és az átmenetekhez szükséges cselekvéseket. Bármely adott pillanatban egy rendszer egy bizonyos állapotot vesz fel és egy átmenet egy válasz egy bizonyos cselekvésre, vagy történésre.

A véges állapotú automaták kategorizálhatóak **determinisztikus** és **non-determinisztikus** automatákra a viselkedésük szempontjából:

- **A determinisztikus véges automaták (DFSM)** esetében minden állapotnak van egy egyedi átmenete minden lehetséges bemenetre (input). A következő állapotot csakis a jelenlegi állapot és a kapott input dönti el, amely egy kiszámítható és egyértelmű viselkedést eredményez.
- **A non-determinisztikus automaták (NDFSM)** esetében több átmenet létezhet egy inputra és a jelenlegi állapotra. A rendszer következő állapota nem egyedien lesz eldöntve, ami rugalmasságot eredményez, de egyben kiszámíthatatlanságot és komplexitást is bevezet.

A mi esetünkben csak determinisztikus automatákkal fogunk foglalkozni, de érdemes volt megemlíteni a non-determinisztikus automatákat, mivel matem-

atikai és tervezésbeli jelentőséggel bírnak.
Egy determinisztikus véges automata hivatalos definíciója így szól:
$M = (Q, \Sigma, \delta, q_0, F)$
Ahol:

1 Q, egy véges halmaz, az állapotok halmaza

2 Σ, egy véges halmaz, az ABC

3 δ, $Q \times \Sigma \to Q$ az állapotátmenet függvény

4 $q_0 \in Q$ a kezdeti állapot

5 $F \subset Q$ a végállapotok halmaza

Egy determinisztikus automata felismer (elfogad) egy jelsorozatot, ha $w = x_1, x_2 \ldots x_n$ Σ-beli jelek sorozata. M felismeri w-t, ha létezik olyan $r_0, r_1, \ldots, r_n$ Q-beli állapotok sorozata, hogy:

- $r_0 = q_0$
- $\delta(r_i, x_{i+1} = r_{i+1})$, ahol $i = 1, \ldots n-1$
- $r_n \in F$

Definíció: M DFSM felismeri az A nyelvet, ha

$A = \{w \mid M\ felismeri\ w - t\}$
Jelölés: A = L(M), avagy "A, az M automata nyelve"

Nézzük meg ezeket a részeket egy kicsit részletesebben is.

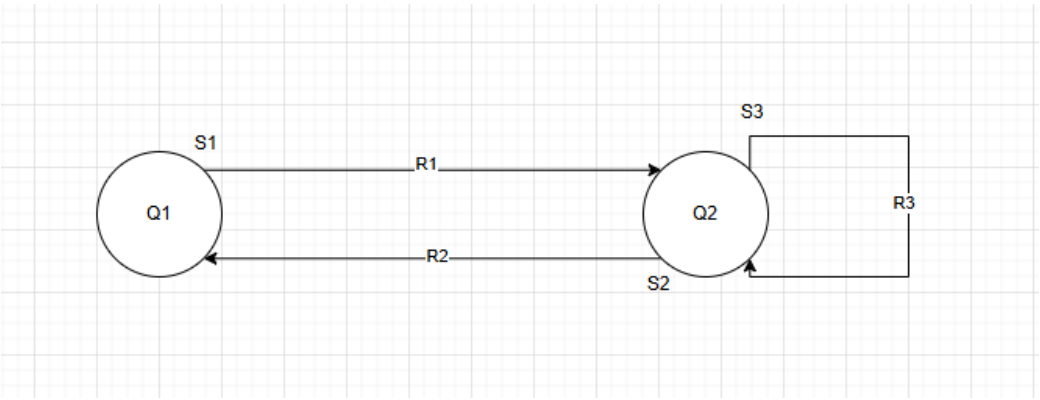

Figure 2.1: Példa állapotgép

Egy egyszerű példa egy ajtó nyitására, nyitva tartására és csukására.

Állapotok:

- Az állapotok segítségével tudjuk megmondani, hogy milyen feltételek és módok mellett operál a rendszerünk a jelenlegi pillanatban. Minden állapot egy egyedi viselkedést ír le, amit a rendszer produkál. Fontos, hogy az állapotok száma *egy véges halmaz.* - $Q_1, \ldots, Q_n$ -nel jelöljük az állapotokat, amikor vizualizáljuk az automatánk

Átmenetek:

- Az átmenetek írják le a rendszer válaszait egyes bemenetekre, vagy eseményekre, amelyek átlökik az automatát az egyik állapotból a másikba.
- Csak bizonyos ingerek (stimuli, a példában S_1, S_2, S_3) eredményezhetnek átmenetet a rendszeren belül, olyanok amelyek benne vannak az automata **nyelvében.**
- Az átmeneteket mindig irányított nyilakkal jelöljük az ábrákon.

Ingerek:

- Az ingerek olyan események, vagy bemenetek, amelyek átmenetet eredményeznek az automatán belül. Lehetnek pl. felhasználói parancsok, szenzor érzékelések, hálózati üzenetek, stb.

Válaszok:

- Egy automata válasza (vagy kimenete) mindig egyedi az állapotot nézve. Ezek lehetnek: egy üzenet kiírása/elküldése, egy egész algoritmus lefuttatása, vagy csak egy változó felülírása.
- A példában a válaszokat (responses) az R_1, R_2, R_3 ábrázolja, az S_1, S_2, S_3 ingerekre.

2.0.2 Programozói szempont

Programozás szempontjából az automaták egy nagyon hasznos része az informatikának, mivel végtelenül egyszerűvé teszi egyes rendszerek nyomonkövetését, megtervezését és automatizálását. Egy rendszeren belül, vegyünk most egy játékot példának, akármilyen objektumnak lehet állapota és, ahogyan ezt már említettem, az állapottól függően fognak változni az objektumok viselkedése is, például:

- A pálya időjárása (eshet, süthet a nap, fújhat a szél),
- a karakter fizikai állapota (futhat, ugorhat, csúszhat, támadhat),
- a fegyver állapota, amivel támadni akarunk (készenlétben, töltődik újra, elhasználva)

Vegyük észre, hogy az állapotok kihathatnak egymásra is, példaképpen: A karakterrel éppen csúszás közbe vagyunk és a csúszda végén ott van egy ellenfél, akit meg akarunk támadni, ezért elővesszük a fegyverünk, de ki van fogyva és ez átrak minket az "újratöltés" animációba, ami miatt csak belecsapódunk az ellenfélbe, a támadás helyett. Azzal, hogy mindig tisztában vagyunk vele, hogy milyen állapotban vannak a világunk egyes részei, megkönnyebbítjük a magunk dolgát a hibakeresésben és a játékosok dolgát azzal, hogy nem zavarjuk őket össze.
Amikor egy rendszernél állapotgépeket használunk, akkor általában kéz-a-kézben jár a "kompozíció", és az "öröklődés" használata mint Objektum Orientált Programozásban használt tervezési minta. Ahhoz, hogy jobban megértsük hogyan is nézne ez ki kódban kezdés képpen készítettem egy mintaprogramot, amely mintaprogram egy egyszerű jelzőlámpa működését mutatja be.
Elsőként is létrehoztam egy vázat az állapotok számára, minden állapot egy külön Node lesz a Godot-n belül. Ez egy teljesen absztrakt osztály, amit valójában nem fogunk futtatni mint script, ezt csak örökölni fogják az egyes állapotaink.

```
using Godot;
using Godot.Collections;

public partial class State : Node
{
	public StateMachine fsm;

	public virtual void Enter() {}
	public virtual void Exit() {}

	public virtual void Update(double delta) {}
	public virtual void PhysicsUpdate(double delta) {}

	public virtual void HandleInput(InputEvent @event) {}
```

```
}
```

Lehet, hogy megfordul a gondolat: "De, akkor miért nem egy absztrakt osztály?" Azért mert, akkor nem tudnánk felvenni az állapotokat egy Dictionary-be, erre később vissza is térek. Egy állapotnak 5 metódusa van: Enter, Exit, Update, PhysicsUpdate, HandleInput. Ezek mind a megfelelő pillanatban lesznek meghívva, lehet, hogy nem mindegyik állapot enged majd inputot feldolgozni, vagy lehet, hogy nem mindegyikre fog kihatni a fizika, viszont ez a metódus-ötös az alapja a legtöbb állapotnak. Az "Enter" metódusban az állapot ellenőrizni fog egy pár feltételt és fel fogja építeni az állapotban végrehajtható cselekvésekhez szükséges környezetet. Az "Exit" egyértelműen ennek az ellenkezőjét fogja csinálni. Az "Update" és "PhysicsUpdate" metódusok mind az eltelt (delta) idő szerint fognak végrehajtani eseményeket, pl.: a levegőben töltött idő mérése, esési sebesség, féktáv, stb. Végül a HandleInput akármilyen bemenetre fog reagálni és azokat lekezelni. Ha például a karakter éppen a levegőben van és elkezd mozogni, akkor azt elkapja a HandleInput metódus és megoldja, hogy ne csak egyhelyben tudjon a karakter ugrani, de különböző irányokba is. A metódusokon kívül van még egy változónk, ami az "fsm".

```
using Godot;
using System.Collections.Generic;

public partial class StateMachine : Node
{
        [Export] public NodePath initialState;

        private System.Collections.Generic.Dictionary<string, State>
            _states;
        private State _currentState;

        public override void _Ready()
        {
                _states = new Dictionary<string, State>();
                foreach(Node node in GetChildren())
                {
                        if(node is State s)
                        {
                                _states[node.Name] = s;
                                s.fsm = this;
                                s._Ready();
```

```
                        s.Exit();
                }
        }
        if(initialState != null)
        {
                _currentState = GetNode<State>(initialState);
                _currentState.Enter();
        }
        else
                GD.Print("Initial State not set");
}

public override void _Process(double delta)
{
        _currentState.Update((float)delta);
}

public override void _PhysicsProcess(double delta)
{
        _currentState.PhysicsUpdate(delta);
}

public override void _UnhandledInput(InputEvent @event)
{
        _currentState.HandleInput(@event);
}

public void TransitionTo(string key)
{
        if(!_states.ContainsKey(key) || _currentState == _states[
            key])
        {
                return;
        }

        GD.Print("Transitioning from ", _currentState.Name.
            ToString(), " To ", _states[key].Name.ToString());

        _currentState.Exit();
        _currentState = _states[key];
        _currentState.Enter();
```

```
        }
}
```

Ez itt a "StateMachine" osztályunk. Ő fel lesz használva mint futtatható script, ő lesz nekünk az állapotgépünk gyökere, minden más Node, ami állapot, hozzá fog tartozni. Itt sok minden történik, de leegyszerűsítve az "initialState" változónk, egy "NodePath", avagy Node-hoz vezető út. Az [Export], ami elé van írva egyszerűen annyit jelent, hogy az editor-ból is meg tudjuk változtatni annak értékét.

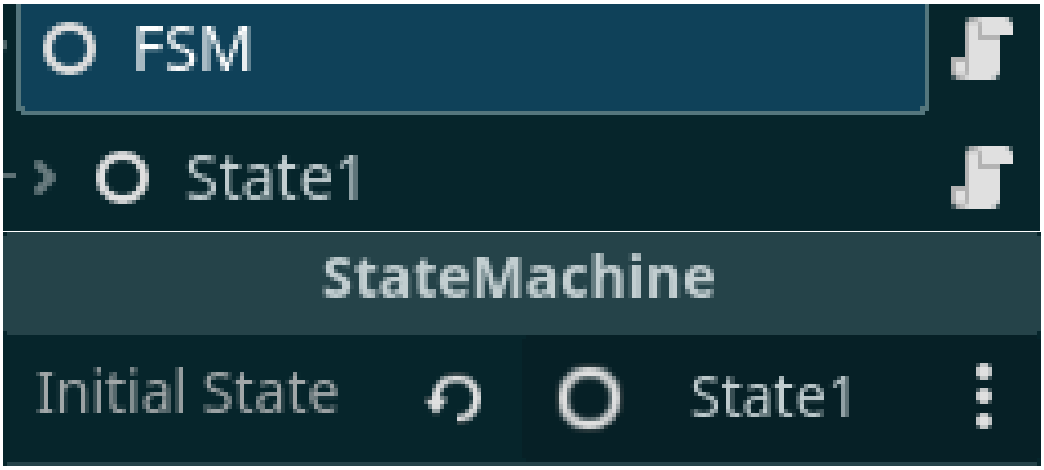

Figure 2.2: Az alap állapot beállítása

Az állapotgépünknek minden képpen kell egy kiindulópont, egy első állapot és ezt mi választjuk ki itt. Továbbá egy Dictionary-ként tárolom a helyes állapotokat és lementem egy "_currentState" változóba a jelenlegi állapotot, hogy mindig számon tudjam tartani. A _Ready() metódusban, ahogyan látjuk végigiterálunk minden "Node" típusú változón. Itt térek vissza az előbbi kijelentésemre, muszáj volt megtartani a "State" osztályt Node-ként, mivel a "GetChildren()" beépített metódus Node típusú változókat ad vissza. A _Process(), _PhysicsProcess() és _UnhandledInput() metódusok mind meghívják a jelenlegi állapot saját metódusát ezekre az alkalmakra. Végül a TransitionTo(string key) metódus felelős az állapotok közötti váltásért. A "kulcs" ebben az esetben nem más mint egy string, ami magának a Node-nak is a neve. Ez a jobb olvashatóság és struktúra érdekében volt így megoldva.

Példaképpen:

Figure 2.3: A befejezett állapotok fája

Az állapotgépünk áll négy állapotból. Mindegyik állapothoz hozzá van rendelve egy fényforrás és egy időzítő.

```
public partial class State3 : State
{
	public override void Enter()
	{
		GetNode<Node3D>("Light").Visible = true;
		GetNode<Timer>("Timer").Start();
	}

	public override void HandleInput(InputEvent @event)
	{
		if(@event is InputEventKey eventKey)
		{
			if(eventKey.Pressed)
			{
				if(eventKey.Keycode == Key.Key1)
				{
					EmitSignal(SignalName.Transition
					, "State1");
				}
				if(eventKey.Keycode == Key.Key2)
				{
					EmitSignal(SignalName.Transition
					, "State2");
				}
				if(eventKey.Keycode == Key.Key3)
				{
					EmitSignal(SignalName.Transition
					, "State3");
				}
```

```
                        if(eventKey.Keycode == Key.Ctrl)
                        {
                                EmitSignal(SignalName.Transition
                                    , "State4");
                        }
                }
        }
    }

    public override void Exit()
    {
        GetNode<Node3D>("Light").Visible = false;
        GetNode<Timer>("Timer").Stop();
    }

    private void _OnTimerTimeout()
    {
        EmitSignal(SignalName.Transition, "State1");
    }
}
```

A "State3"-nak elnevezett állapotban vagyunk jelenleg, ami ebben az esetben a "Piros" állapot a jelzőlámpánkon. Amikor erre az állapotra váltunk, akkor a State3-hoz csatolt fényforrást láthatóvá kell tennünk, valamint el kell indítanunk az időzítőnket, hogy nehogy benne ragadjunk. Amikor az időzítő lejár átmegyünk a "State1"-ba, ami a mi estünkben a "Zöld". *Amikor* átlépünk a következő állapotba fontos, hogy az előző állapotból sikeresen kilépjünk és megszüntessünk minden olyan dolgot, ami bezavarna más állapotok működésébe. Ezért az állapotgépünk meg fogja hívni a State3-nak az "Exit()" metódusát, amelyben is elrejtjük a fényforrását és megállítjuk az időzítőjét, arra az esetre, ha nem állna meg magától.
Ezen felül van egy egyszerű switch statement, ami felhasználói bemenetre reagál ezzel lehetővé téve az állapotok közötti egyszerű ugrálást.

Chapter 3

Koncepció

Az évek alatt sok olyan játékkal játszottam, amelyek mélyen kidolgozott mozgásrendszerekre vannak építve, ilyen műfajok például a: Movement-Shooter, bizonyos Sport játékok többnyire azok, amik gördeszkákra, görkorcsolyákra és a hasonlókra épülnek. Fontosnak tartottam itt megemlíteni a Movement-Shooter műfajt, mivel az ebbe tartozó játékok szerettették meg igazán velem a szofisztikált mozgásrendszereket és belőlük tanultam sokat. Ezen játékok tökéletesen összehangolják a finom mozdulatokat az aréna-lövöldék gyors döntéshozatalával és precíz célzás igényével. Erre már az eredeti Quake játék többjátékos módját is fel tudnám hozni, de manapság inkább az "Ultrakill" nevű játékot szokták felhozni, mint ékes példája. A projekt maga viszont egy játszható prototípus, amely egyben egy hordozható rendszer is, így nincs műfajhoz és programhoz kötve, hanem mint egy Lego darab, kivehető és átrakható egy másik programba.

3.0.1 Játékmenet

Játékról, mint úgy nem lehet beszélni, viszont hordozható rendszerről már inkább. A mozgásrendszert úgy terveztem meg, hogy akármilyen olyan környezetben használható legyen, amelyben a játékos karakteren, vagy esetleg nem-játékos karakteren (NPC-n) valamilyen gurulásra vagy csúszásra alkalmas eszköz van (pl.: gördeszka, görkorcsolya, síléc, jégkorcsolya). Ehhez a Tony Hawk's Proskater játéksorozatot és Bomb Rush Cyberfunk / Jet Set Radio játékmenetét és mozgásrendszerét vettem alapul. Azzal az eltéréssel, hogy a prototípusban, megnyerési pont nincs, mivel ez túlmutatott a prototípusnak megszabott feladaton.

3.0.2 Használt technológiák

A használt technológiákat két részre osztanám fel: Szoftver és Hardver. Most egyenlőre, csak egy rövidebb felsorolást tennék a technológiákról és a későbbiekben, amikor a megfelelő részekhez érek, bővebben beszámolok róluk.
Szoftver szempontjából:

- Blender-t használtam a 3Ds modellek elkészítéséhez.
- Krita-t használtam a Modellek textúráinak megrajzolásához és SLK_img2pixel-t használtam a színek állításához.
- A hangok megvágásához Audacity-t és LMMS-t használtam.
- A diagramok megszerkesztéséhez Draw.io-t használtam.
- A verziókövetéshez Git-et használtam.

Hardver szempontjából a szoftvert két különböző rendszeren teszteltem és fejlesztettem:

- Asztali Számítógép:
  - Windows 10
  - Processzor: AMD Ryzen 5 7600X 6-Core
  - AMD Radeon RX 6600
  - Memória (RAM): 32GB
- LENOVO IdeaPad Slim 3 15AMN8:
  - Kubuntu 25.10
  - Processzor: 8 x AMD Ryzen 5 7520U with Radeon Graphics
  - Videókártya: AMD Radeon 610M
  - Memória (RAM): 16GB

Megjegyezném, hogy ez nem jelenti azt, hogy csak ezen gépigényeket elérő rendszereken működne a rendszer, egyszerűen csak ezeken tudtam kipróbálni. Valamint a digitális koncepció rajzok elkészítéséhez és a textúrák megalkotásához az: XP-Pen Deco 02-őt használtam.

3.0.3 Fejlesztési Környezet Választása

A fejlesztési környezet választásakor több szempontot is figyelembe kellett vennem:

- Elérhetőség: Mivel a fejlesztést két különböző rendszeren, két különböző operációs rendszer alatt végeztem fontos volt számomra, hogy a fejlesztési környezet elérhető legyen mind a két operációs rendszeren.
- Programozási Nyelv Támogatása: A Godot-n belül van számos programozási nyelvhez támogatás, ezek a:
  - GDScript a Godot saját script nyelve,
  - C# a Godot Mono támogatásával,
  - Valamint C és C++ számos kiegészítő támogatásával.

Fontos volt, hogy ezek közül az egyikhez legalább támogatást nyújtson a fejlesztési környezet.

- Kényelem: A fejlesztési környezet által nyújtott kiegészítési, valamint emlékeztető segédletek fontosak voltak, mivel a rendszer sok hasonló kódrészt tartalmaz (pl. Dictionaries, Leképzések, stb.).
- **Integráció a motorral:** Ez egy bónusz pont a listán, de sokat segít, ha a fejlesztési környezet integrálja a játékmotort az egyszerű dokumentáció olvasás és metódus / függvény hívás érdekében.

3.0.4 Játékmotor Választása

A játékmotor megválasztása során fontos volt számomra az elérhetőség, kompatibilitás és a hordozhatóság.

- Elérhetőség: Mivel a szoftvert két különböző operációs rendszeren fejlesztettem ezért az elérhetőségi szempont elengedhetetlen volt. A játékmotorok mindig szóba jön a Unity és az Unreal Engine, viszont ezeknél az opcióknál a licensz kérdése is szóba jön. Nem egy utolsó szempont az se, hogy ezzel a rendszerrel én majd a jövőben tovább dolgozzak akármilyen fennakadás vagy hátráltatás nélkül.
- Verziókezelés támogatása: Akármilyen rendszer fejlesztése során fontos a megbízható verziókezelés. Háromdimenziós terekben
- Tapasztalat: Végül, de nem utoljára fontos volt a meglévő tapasztalat a választott motorral, hogy a lehető legjobbat tudjam kihozni a választott projektből reális időn belül.

Így mindezeket a szempontokat összevetve a választásaim végül a: Godot motorra, C# programozási nyelvre és a Visual Studio Code fejlesztőkörnyezetre esett.

3.0.5 Vizuális Világ Megtervezése

A vizuális világ, amint már említettem a Shibuya punk esztétikára alapszik. [3.1] Agresszív vonalak, elmosott, neon színek erőteljes használata és dinamikus kompozíciók definiálják. Műfajalkotó játéknak számít a Jet Set Radio (amire mostantól **JSR**-ként fogok hivatkozni).

Figure 3.1: Demonstráció az agresszív vonalakra

A JSR 2000-ben jött ki a DreamCast konzolra. És korszakalkotó volt a játék vizuális világa, [3.2] játékmenete és mozgásrendszere. Százezreket fogott meg és évekig utána nem volt semmi olyan játék, ami el tudta volna csípni, hogy mi is tette a játékot annyira ikonikussá. A játékmenetében is megtartja a dinamikusságot és az agresszív vonalakat, ami a játékosban azt az érzést kelti, hogy gyorsabb és pontosabb mint valójában.

Figure 3.2: Rövid gif a JSR játékmenetéről

Ahhoz, hogy pontosan replikálni tudjam ezeket a tulajdonságokat mélyen bele kellett magam ásnom a játék világába, de mivel egy eléggé régi játékról van szó, nem könnyen hozzáférhető. Szerencsére erre a problémára megoldást találtam egy 2023-mas ún. "Szerelmi Vallomásban" (Love Letter), a Bomb Rush Cyberfunk-ban [3.3] (amire mostatntól **BRCF**-ként fogok hivatkozni).

Figure 3.3: Reklámkép a Bomb Rush Cyberfunk-ról

A játék teljes mértékben a JSR-t veszi alapul és megpróbálja replikálni a játékot annyira, amennyire csak tudja [3.4], miközben újít és iterál a meglévő koncepciókon. A játékmenet kifinomultabb, mint a JSR-nak, de képes megtartani azt az érzést, amit annak idején az elődje keltet. A Team Reptile leírása alapján: "Dion Koster elmevilágában, ahol saját-stílusú bandák személyi jetpack-ekkel vannak felruházva, a graffiti művészet új magaslatokba szárnyal. Vágj bele te is a világba és táncolj, fess, trükköz, verd vissza a korrupt rendőrséget és foglald el a város minden zegét-zugát egy alternatív jövőben, amit életre kelt Hideki Naganuma zenéje"[2].

Figure 3.4: Játékbeli kép a Bomb Rush Cyberfunk-ról

Látszatra a két játék ugyan azt a stílust tartja, azzal a különbséggel, hogy míg a JSR egy földhöz-ragadtabb játékmenetet és világot ad át, mint ha egy, a 20-as éveiben lévő lázadó punk tini lennél, addig a BRCF egy sokkal sci-fi-sebb világba dob minket, ahol mindenkinek lehet egy jetpack a hátán, ami eszméletlen sebességekre gyorsít fel minket és rásegít a trükkjeinkre A vizuális világ megtervezésében tehát ezeket a szempontokat vettem alapul és ezek szerint alakítottam a világot és a karaktert.

3.0.6 Mozgásrendszer megtervezése

Most, hogy a vizuális világ megbeszélésre került, ideje áttérni arra, ami igazán fontos: a mozgásrendszer.
Ahhoz, hogy egy minél folyékonyabb mozgásrendszert tudjak alkotni sok tervezés és előre gondolás kellet. A véges automaták egyik előnye az, hogy nagyon könnyen bővíthetőek, így ha el is rontottam volna valamit, vagy esetleg kellett volna még egy állapot ahhoz, hogy az animáció vagy a mozgás jól jöjjön ki, akkor azt könnyen megtehettem volna. Ennek a tulajdonságnak köszönhetően volt egy kis mozgásterem a kísérletezésre, ami újoncként nagy előny.

Állapotok

Át kellet gondolnom, hogy hogyan is szeretném, hogy kinézzen egyes mozdulat és, hogy egyes, a játékos által megadott bemenetre, milyen mozdulat lehet végrehajtható, bizonyos időkben. Egyszóval, a mozgás-láncokat meg kellett alkotnom. Viszont, nem csak a játékos kezdeményezhet állapotváltozásokat, hanem a program maga is. Előfordulhatnak események, amelyek befolyásolják a játékos mozgását és egyben az állapotát is. Ahhoz, hogy megkülönböztessem a program eseményeit a játékos bementeitől elneveztem azokat az eseményeket, amik a játékos irányítása fölött vannak "E"-nek, mint "Event" és a játékos bemeneteit "I"-nek, mint "Input". Most már tárgyalásra kerülhet az állapot diagram.
Minden a belépéssel vagy leidézéssel (spawn) kezdődik. A felhasználó itt kapja meg az irányítást a karaktere fölött, ezt a [3.5] kép demonstrálja.

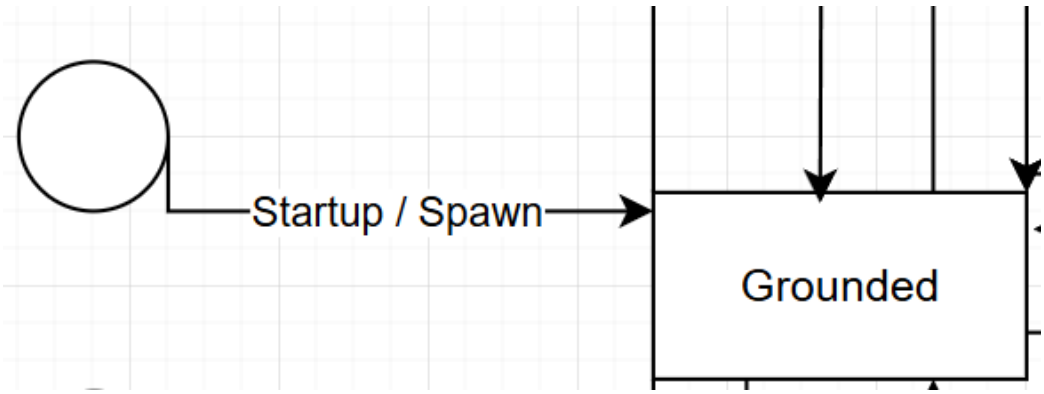

Figure 3.5: A program elindítása / a karakter újraéledése utáni állapot

A karakter minding egy szilárd, állható talaj fölött éled le, vagy éled újra, ezzel garantálva az állapotok közötti folytonosságot. Az alap állapot minden esetben az ún. "Idle", avagy nyugalmi állapot lesz, ahol a karakter egy állható, szilárd talajon van és játékos nem ad meg bemenetet, ennek a vizualizálása a [3.6] képen látható.

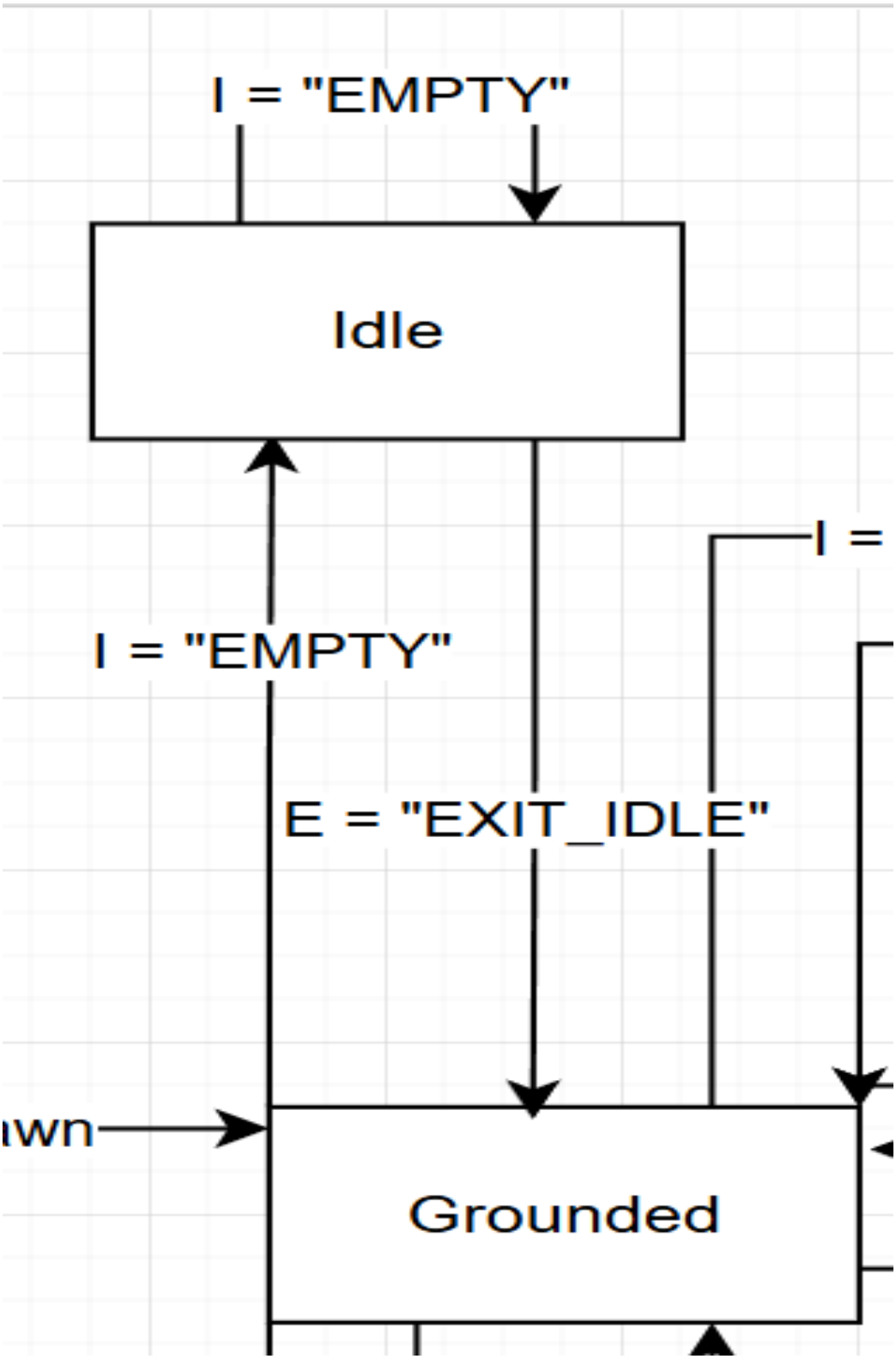

Figure 3.6: A karakter egyhelyben állása

Ebben az állapotban egy nyugalmi animáció játszódik le, ami életszerűbbé teszi a karaktert. A szilárd talajon való állásból (Grounded) elindulhat a játékos valamely mozgás gomb lenyomásával az egyik irányba ezzel átlökve az automatát a mozgásban levés állapotába (Moving), ahogyan azt a [3.7] kép is mutatja.

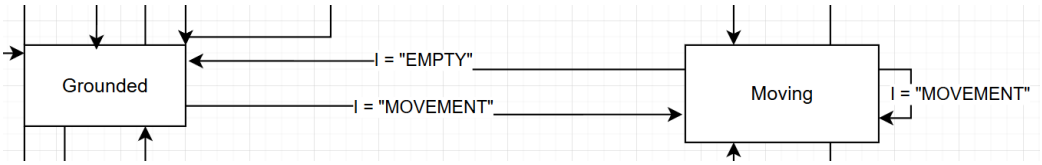

Figure 3.7: A karakter állható talajon való mozgása

Ebből az állapotból a játékos visszatérhet a Grounded állapotba ha abba hagyja a mozgást. Viszont, ha a játékos mozgás közben lenyomja az ugrás gombot (ami ebben az esetben a "SPACE"), akkor átlöki az automatát a levegőben lévő állapotba (InAir), ami az alábbi [3.8] képen látható.

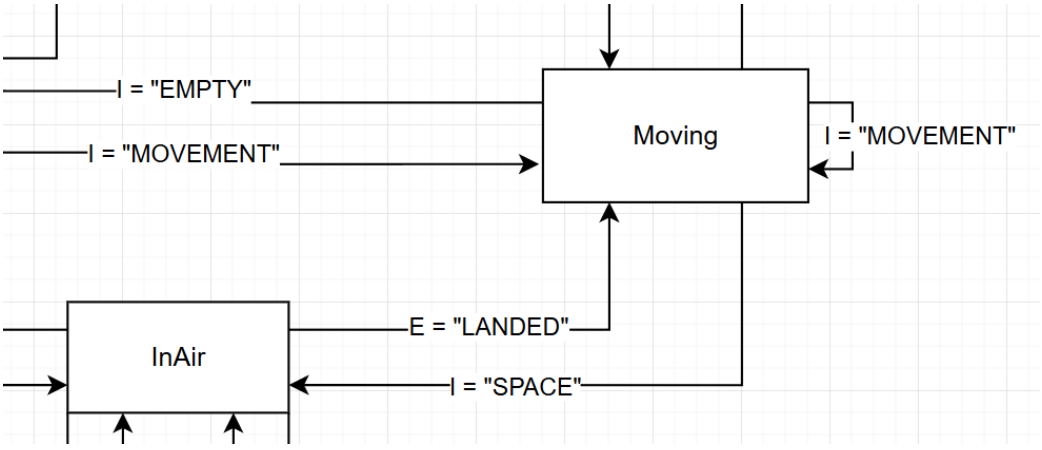

Figure 3.8: A karakter mozgásból való ugrása

Az ugrás időhöz van kötve, amit a gravitációs vonzóerőből és a sebességből számol ki a program. Amint ez az idő lejár és a karakter földet ér a program kiad egy "földet ért" (LANDED) eseményt, amiből, abban az esetben ha a játékos még mindig mozog, akkor visszatér a Moving állapotba. Ammenyiben, viszont a játékos már nem ad meg bemenetet a mozgásra, akkor az InAir állapotból visszatérünk a Grounded állapotra, amit a [3.9] kép demonstrál.

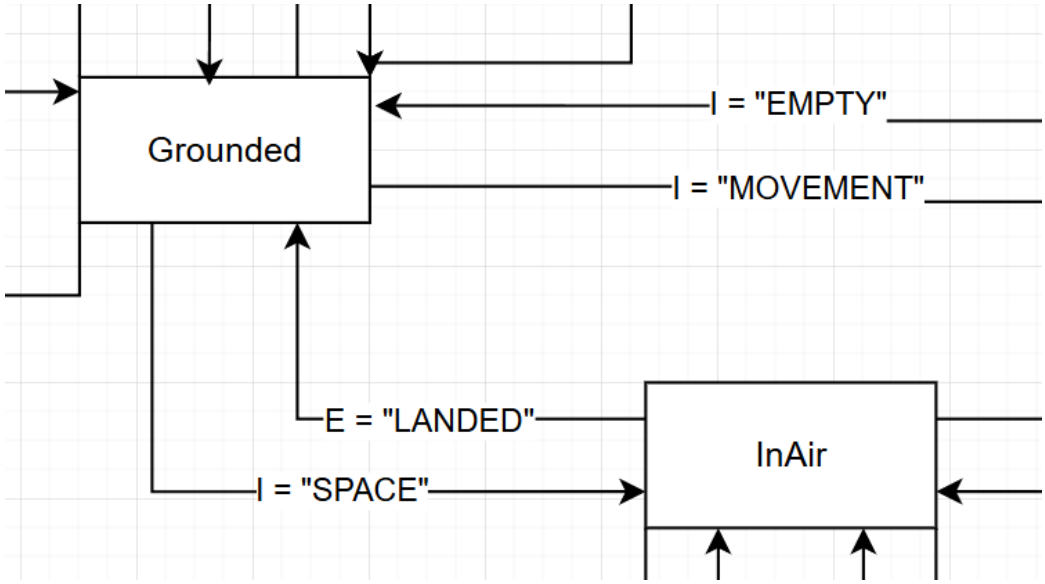

Figure 3.9: A karakter állható talajról való ugrása

Vegyük észre, hogy a Grounded állapotból is el lehet jutni az InAir állapotba, abban az esetben, ha a játékos nem ad meg mozgásra bemenetet, viszont ugrásra igen. Ha a játékosnak viszont kell még egy kis idő a levegőben, akkor mégegyszer megnyomva az ugrás gombot, akkor átlöki az automatát a dupla ugrás (DoubleJump) állapotba, ahogy azt a [3.10] képen is lehet látni. Ebben az állapotban megtartja a momentumát a játékos csak a felfele mutató vektorhoz adódik hozzá megint az érték.

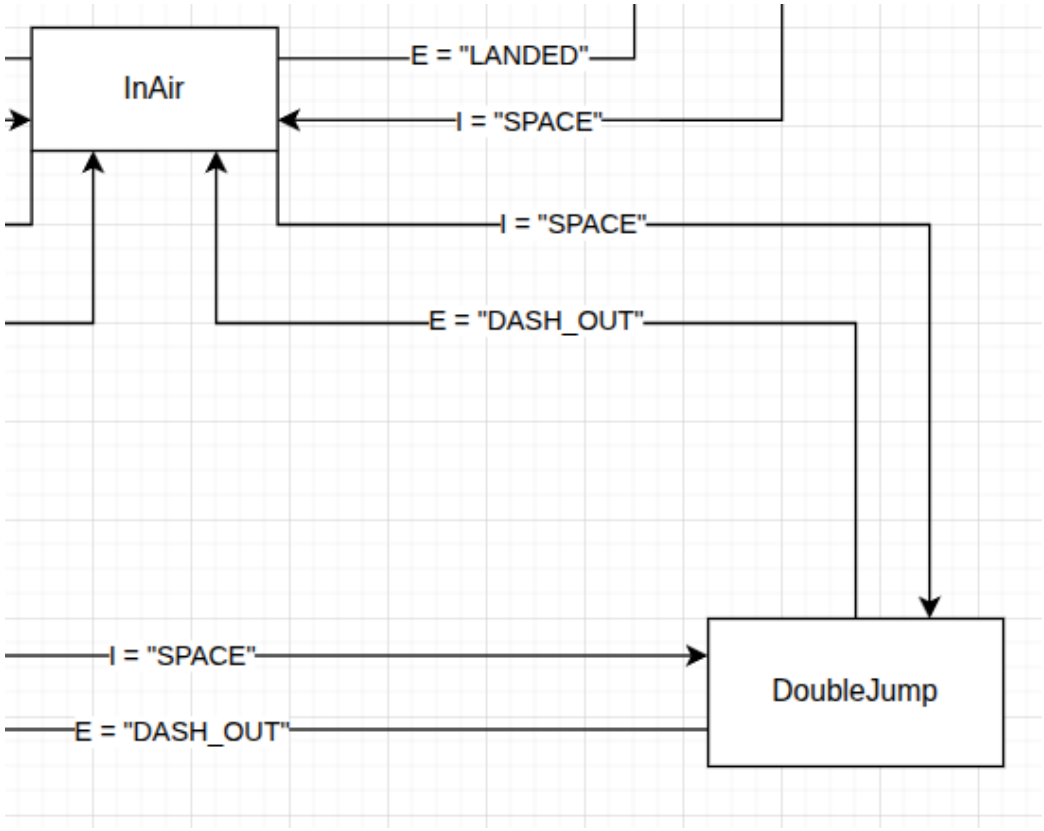

Figure 3.10: A karakter levegőből való ugrása

Vegyük észre, hogy a játékosnak most már nincs irányítása a karakter levegőben maradása fölött, mivel kaptunk egy eseményt, ami megmondta nekünk, hogy nincs több ugrásunk ("DASH_OUT"). Abban az esetben ha a játékos a dupla ugrás után tovább szeretné mozgatni a karaktert a levegőben valamilyen okból kifolyólag, akkor a mozgás billentyűket lenyomva át tudja lökni az automatát a levegőben lévő mozgás (InAirMovement) állapotba, ahogyan azt a [3.11] kép is mutatja. Amíg a játékos lenyomva tartja a mozgás gombokat, addig ebben az állapotban marad.

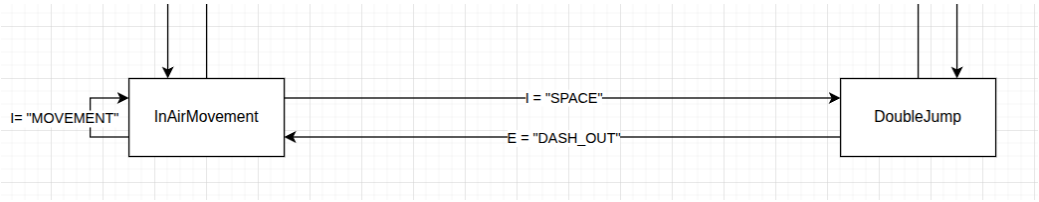

Figure 3.11: A karakter levegőben való mozgása egy dupla ugrás után

Amennyiben a játékos abba hagyja a mozgást, és még nem futott ki a levegőben tölthető időből az automata vissza lesz lökve a levegőbeli állapotba (InAir). Viszont, ha ismét el kezd mozogni, akkor vissza lesz lökve a levegőben lévő mozgás állapotába, ezzel befejezve a kört, ahogy az az alábbi [3.12] ábrán is látható.

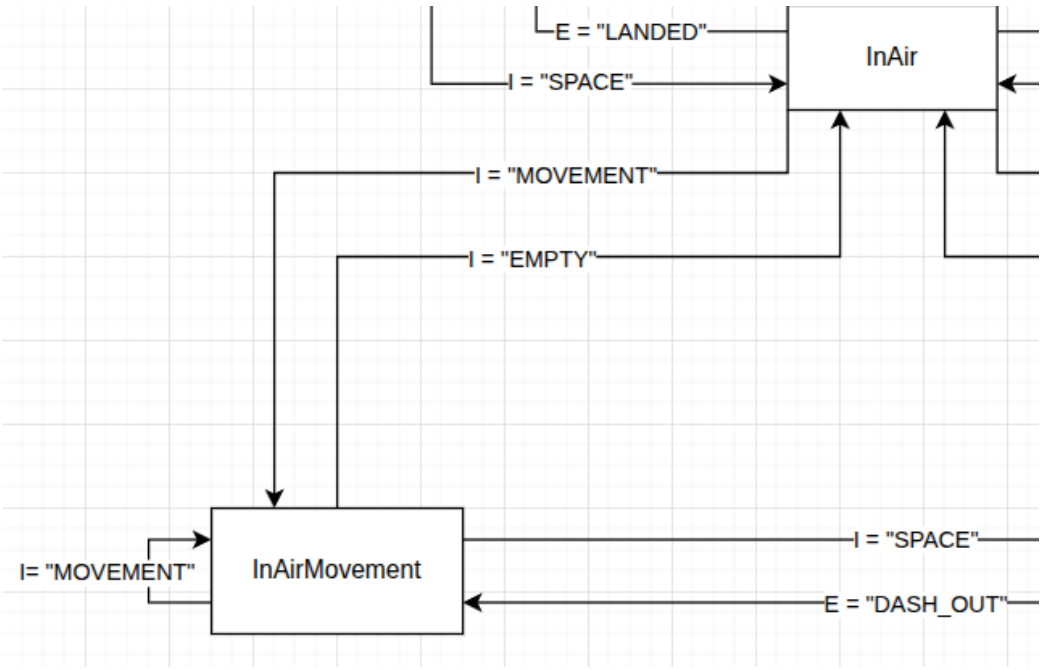

Figure 3.12: A karakter levegőben való mozgása és esésbe való visszatérése

Érdemes megjegyezni, hogy az eddig felsorolt állapotok össze vannak kötve egymással. Ezeket mozgás hármasoknak hívom. A legtöbb mozgásfajtát le lehet bontani kisebb csoportokra amiknek van hozzáférése 1 vagy 2 olyan állapothoz, amely átviszi őket egy másik csoportba. Példa képpen a levegőben történő mozgáscsoport hármasa a [3.13] képen látható.

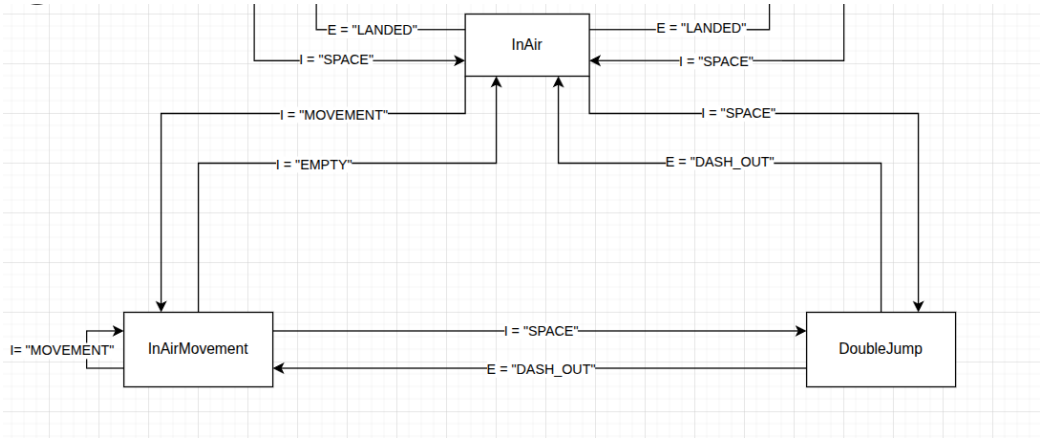

Figure 3.13: A levegőben történő mozgáscsoport

Valamint a földön történő mozgáscsoport hármasa a [3.14] képen.

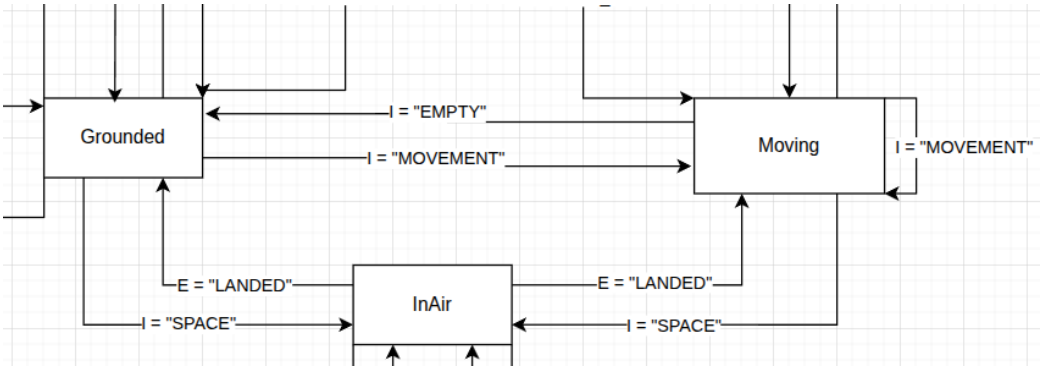

Figure 3.14: Az állható talajon történő mozgáscsoport

Ezen mozgáscsoportok közötti átjárást az **ugrás** cselekvés biztosítja, ami további mozgáslehetőségeket tár fel.

Bibliography

[1] Charles Burgar. How omnimovement works in black ops 6. 2024.

[2] Team Reptile. Bomb rush cyberfunk. 2020.