SZEGEDI TUDOMÁNYEGYETEM
Természettudományi és Informatikai Kar
Képfeldolgozás és Számítógépes Grafika Tanszék

Szakdolgozat

Számítógépes videojáték fejlesztése Unity környezetben, alakzat felismerő algoritmus megvalósításával.

Videogame software development in Unity implementing image and shape recognition algorithm

Puskás Patrik
Programtervező informatikus

Témavezető: Dr. Bodnár Péter, egyetemi adjunktus

Szeged
2025

Feladatkiírás

A feladat egy kétdimenziós térben játszódó játék fejlesztése, mely implementál egy képfelismerő metódust, és egy képernyőre rajzoló algoritmust. A rajzoló algoritmus szerepe a játékon belüli képernyőre való rajzolás biztosítása, melyet értelmez egy alakzatfelismerő algoritmus, és alakzat azonosítása után végrehajt egy akciót, melynek kimenete a felismert alakzattól függ. A program futásidőben a lehető leggyorsabban kell végrehajtsa a ezt a feladatot.

A szoftvernek tehát három fő funkciót kell ellátnia:

- A játéktérre való rajzólás
- A rajzolt alakzatok felismerését
- Felismert alakzatokhoz tartozó funkciók végrehajtását
- Játszható környezet és játékmenet megvalósítását

Alkalmazandó főbb technológiák:

- C# programnyelv
- Unity játékmotor

Tartalmi összefoglaló

A téma megnevezése:

Számítógépes videojáték fejlesztése Unity környezetben, alakzat felismerő algoritmus megvalósításával.

A megadott feladat megfogalmazása:

A szakdolgozat célja egy olyan játékszoftver fejlesztése melyben a felhasználó a képernyőre rajzolva interaktálhat, és az alakzatok felismerésével a programon keresztül befolyást gyakorolhat a játék menetére futásidőben.

A megoldási mód:

Egy kezdeti kudarc után Pygame helyett a program Unity-ben került megtervezésre, majd megvalósításra. A backend C# nyelven van írva, ez alapján működik az irányítás és a játék alapvető logikája. Frontendre ingyenesen felhasználható open source assetek kerültek a programba, melyek textúrái adott objektumoknak, például a játékosnak, platformoknak, háttérnek vagy éppen ellenség objektumnak. A fejlesztés során főleg macOS operációsrendszert fókuszáltam, de a Unity cross-platform megoldásai miatt Windows rendszeren is teszteltem a program futtatását.

Alkalmazott eszközök, módszerek:

- Unity – a játékfejlesztést támogató motor, amiben készült a program
- C# programozási nyelv
- Gitlab és Github - verziókezeléshez

Elért eredmények:

Elkészült egy futtatható játékprogram, mely megvalósítja a képernyőre rajzolást, és a rajzolt alakzat felismerését. A felismert alakzathoz tartozó feladatot pedig végre is hajtja futási időben a program. A szoftver fejlesztés alatt tovább bővíthető új egyéni alakzatokkal, ezeknek feltételei adottak is a kódban.

A program több rendszerre is felépíthető, és működőképes. A fejlesztés alatt Windows és macOS rendszereken lett tesztelve.

Kulcsszavak:

Unity, játékfejlesztés, képfelismerés, alakzatfelismerés, képfeldolgozás játékmenetben, C# programnyelv, objektumorientáltság, cross-platform fejlesztés

Tartalomjegyzék

Bevezetés

A videojáték ipar az utóbbi két évtizedben kicsit sem túlozva felrobbant. Rengeteg hatalmas, kisebb és egyedülálló („indie”) videojátékokra fókuszáló fejlesztőstúdiók jöttek létre, és terjedtek el. Az egyre népszerűbbé váló iparban nagyjából a 2010-es évektől, meg lehetett figyelni egy átalakulást, ami a közönséges szórakoztatásból egyes esetekben már-már művészi alkotásokká kovácsolta a játékfejlesztés egyes produktumait. Az egyre szebb- és valósághűbb grafikai motorok, illetve az egyre növekvő ebbe fektetett összegek, is arra engedik az embert következtetni, hogy már közel sem annyira réteg dolog ez a stílusú szórakozás, mint a videojáték szoftverek hajnalán volt.

Egy videójáték sokkal jobban behúzza a nézőt azzal, hogy részese lesz a történetnek, akár bele is szólhat. Egy moziélmény helyett - ami általában másfél- maximum háromórás játékidő – sokkal több idejük van a játékfejlesztőknek elmesélni egy történetet, egy élethelyzetet vagy egy egész világot. Néhány híres példát említve ilyen volt a „The Last of Us”, vagy a „Fallout” is, amelyekről sikerességük miatt, élőszereplős sorozatot is készítettek. Csak hogy az én személyes kedvencemet is említsem, az „Expedition 33”. Ez a játék (röviden összefoglalva) a gyász feldolgozásáról szól, és olyan témát feszeget, hogy ha módunkban állna az olyan folyamatokat, mint például a halált bolygatni, vajon megtennénk-e. Persze ez a mélység leginkább a történet alapú játékokat érinti, és a piacon azért bőven akadnak a puszta szórakozásra fejlesztett, történet nélküli címek is.

Rengeteg újdonságot hozott magával az ipar technológia szempontból is, és nagyon sok egyedi megközelítést is hoztak. Ilyen volt például a nem Euklideszi geometria és optikai illúziók köré épülő játékok, mint például. a „Superliminal” és a „ViewFinder”. Az open-world szabadságával újított a méltán híres „Grand Theft Auto” sorozat is mely hatalmas szabadon bejárható mondhatni játszóteret adott a játékosoknak.

A megfigyelésem, ami ihlette a témaválasztásomat, hogy a játékiparban a grafikai motorokon kívül játékmenet szempontjából alig fellelhető képfelismerés, a szoftveres alakzatdetektáció. Mit értek ez alatt? Nos a szakdolgozatomban azt tűztem ki célul, hogy készítsek egy olyan játékot olyan játékmenettel, amelyben egérrel történő rajzolás után egy algoritmus felismeri mire hasonlít az alakzat, és végrehajtja a hozzá társított funkciót. Kicsit hasonlóan, mint a klasszikus régi mesében, a varázsceruzában láthattuk. Én ezt az új megközelítést szeretném behozni a játékiparba.

Irodalmi áttekintés

1. Videojátékok technológiai fejlődése

1.1 Mik azok a játék motorok?

A játék motorok alkotják a játékok alapjait. Tulajdonképpen ezek szoftver keretrendszerek, rend szerint grafikus felülettel. Ezekben fejlesztenek a készítők, amelyek tartalmazzák a szükséges speciális könyvtárakat és csomagokat, amik kellenek a játék működéséhez. Ilyenek lehetnek például vetülő fényt szimuláló csomagok, vagy a játék fizikáját adó könyvtárak. Ezek a csomagok határozzák meg, hogy egyes entitások a játéktérben hogyan működnek, pl. egy lufi felszáll, vagy egy doboz a gravitáció hatására leesik. Vannak persze a játéktereknek statikus elemei is, amiket interakciókkal nem lehet befolyásolni, ezeknek is sajátos működésük van, nem lehet őket elpusztítani keresztül menni rajtuk, vagy ha hozzáér a játékos, szimplán megfordul. Ilyenek jellemzően a játéktér vertikális határait szolgáló láthatatlan vagy esetenként texturával rendelkező határelemei, az úgynevezett „sky-boxok”.

1.2 Játékmotorok piaci áttekintése

A piac növekedésével rengeteg motor készült, melyeket több csoportra lehet osztani: nyilvánosan használhatóak, egyéni célra fejlesztettek és akár saját magunknak is készíthetünk akár teljesen a nulláról. Egyes motorok inkább kétdimenziós játékokat támogatnak, mások 2D és 3D játékokat is, de van olyan, ami inkább a 3D és realisztikus textúrára fókuszál. Ha valaki játékfejlesztésbe akar kezdeni, programozási nyelvtől függően sok lehetősége van, az open-source motoroknak köszönhetően. Egyesek teljesen ingyenesek, mások ingyenesek, egy bizonyos eladott példányszám alatt, azonban azt meghaladva részesedési díjat kérhet a motor készítője. Ezen felül pedig természetesen ott vannak az abszolút profi programok speciális igényekre tervezve. Ezek az utóbb említett motorok jellemzően egyedi célra készülnek nagy játékfejlesztők által, akár egy-egy játék, vagy játékszéria kedvéért programozzák le. Nagy részük ezért vállalati titok is, nem férhet hozzá akárki, de ezek közül is akad kiemelkedő, ami nyilvános a nagyközönség számára részesedési díj ellenében. Az sem mindegy, hogy milyen célközönségnek készül egy adott játék. Az egyetlen játéktér ma már nem csak a számítógépek. Számos konzolgyártó, (Microsoft – Xbox, Sony – Playstation, Nintendo) és ezeknek is több verziójú rendszerük van, a különböző generációjú konzoloktól függően. Emellett ott vannak a mobilra fejlesztett játékok is. Ezek mindegyikére egyenként más és más szoftververziót kell kiadni a játékszoftverből, ha azt akarjuk, hogy minél több rendszeren elérhetőek legyenek,

vagyis úgymond „cross-platform” legyen a fejlesztett termékünk. Erre a célra is külön motorok vannak, ugyanis nem mindegyik rendszer támogatja a platformfüggetlen fejlesztést.

Legismertebb játék motorok és programozási nyelveik:

- **Unity** ..... C#,
- **Godot** .....GDScript, C#, C++,
- **GameMaker** ..... GML,
- **Unreal Engine** .....C++,
- **Source** ..... C, C++,
- **Anvil** ..... C++, C#,
- **Frostbite** .....C++,
- **Pygame** .....Python,

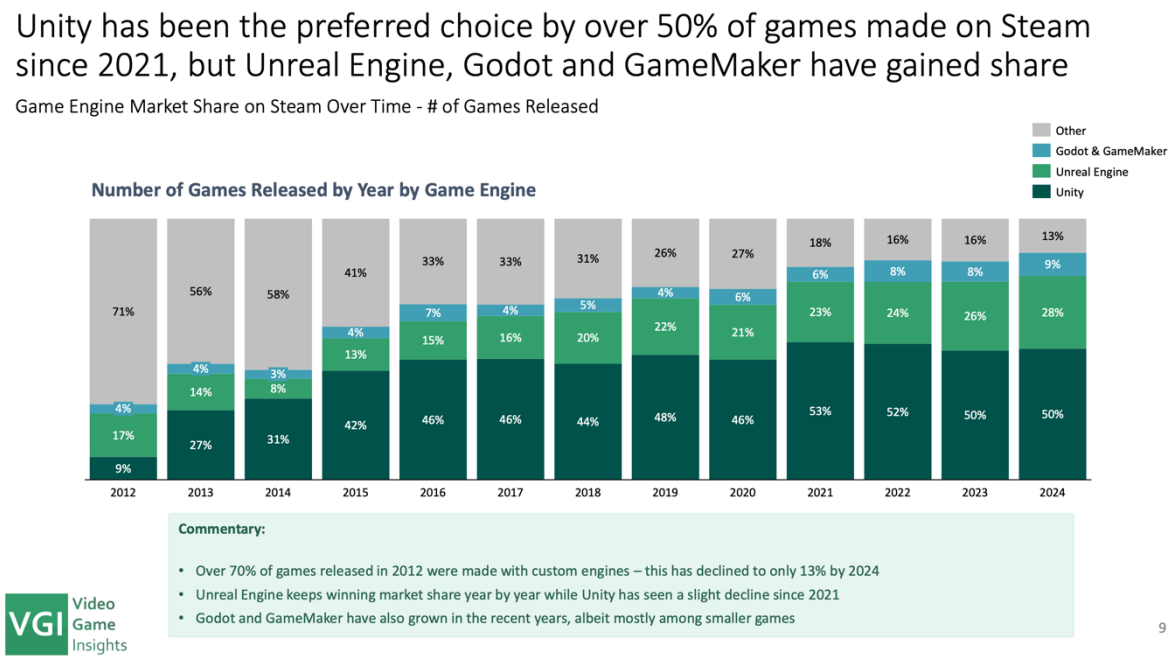
1.3 Piaci megoldások

Az 1.2 fejezetben felsorolt listából is látszik, mennyi lehetőségünk van válogatni a játékmotorok közül, és még nincs is említve az összes.

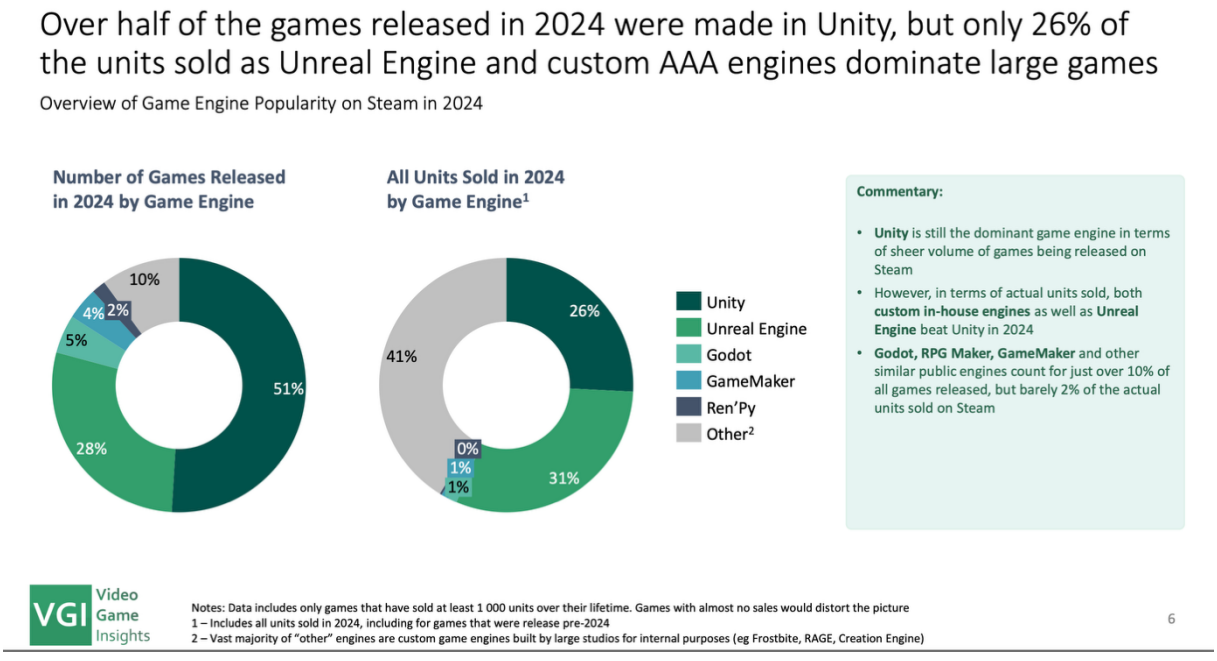
Az utóbbi években a legnépszerűbb motor bizonyosan a **Unity**. A legtöbb indie játékfejlesztő ezt választja könnyen tanulhatósága miatt. Leginkább a 3D játékokban láthatjuk a piacon, de rengeteg 2D-s játékot is készítenek benne, erre is remekül használható a fő funkciója mellett. Rengeteg támogatást kap a motor, közösség által rengeteg csomagot készítettek már ehhez a platformhoz. Különösen előnyös az is, hogy cross-platform fejlesztést tesz lehetővé, így a lehető legkevesebb munkával több platformra is könnyen átvihető a kész játékszoftver. A Unity-ben továbbá kedvező az is, hogy van teljesen ingyenes személyes használatra való csomagja is. Ez a csomag lehetővé teszi az egyszemélyes vagy kis létszámú csapatok számára, hogy éves $200,000 USD bevételig ingyenesen fejleszthessenek és adhassanak ki játékokat ezzel a motorral, viszont. ez még nem tartalmazza cross-platform fejlesztést. E fölött a csomag fölött található a Unity Pro, ami ilyen szintű korlátozásokkal már nem rendelkezik, ára/fő/év: $2,200 USD és használható éves $25 millió USD bevételig. Az utolsó csomag az Enterprise, ami nagyobb cégeknek opció, sokkal nagyobb skálában.

Az alábbi (1.ábra) alapján is látszik, hogy 2021 óta mennyivel megnőtt az érdeklődés a Unity engine iránt. Az is leolvasható az ábráról, hogy a többi játék motor is nagyobb a kereslet évről évre, visszaszorítva az egyéb piaci megoldásokat.

A 2.ábráról az látszik, hogy 2024-ben több mint a játékok fele Unity-ben készült, azonban ennek ellenére is az összes eladott példányszám alapján csupán 26%-ot tesznek ki az évben.



1. ábra: Játék motorok népszerűségének eloszlása 2021-től
https://app.sensortower.com/vgi/assets/reports/The_Big_Game_Engines_Report_of_2025.pdf



2. ábra: 2024-ben készült és eladott játékok eloszlása játékmotorok szerint csoportosítva
https://app.sensortower.com/vgi/assets/reports/The_Big_Game_Engines_Report_of_2025.pdf

2. Technológiai lehetőségek

A fejlesztő nincs egyenes rákényszerítve arra, hogy a fent említett motorokból használjon egyet is. Sőt még arra se, hogy írjon magának egyet. Nézzük csak meg a méltán elhíresült „Minecraft”-ot. Az a játék például egy LWJGL nevű Java könyvtárból készült. További jó példa a „Terraria” nevű népszerű játék is ami pedig a Microsoft XNA Framework-ben készült, ami lényegében csak C# eszközök és könyvtárak egyvelege. Szóval a tanulság ebből, hogy egyáltalán nem kötelező egy játék motorban játékot készíteni, azonban a piac ilyen szintű kitágulása miatt, egyre inkább ez a tendencia.

Egy játékmotor sok előre gyártott komponenst kínál: scene-management, fizika, animáció, asset import, UI, szerverkapcsolat stb. Ha ezeket kihagyjuk vagy saját magunk próbáljuk megvalósítani, akkor több munkánk van, de nagyobb szabadságunk is. „Engine-mentes” fejlesztés alatt tehát azt értem, hogy nem használjuk a megszokott, „mindent tudó” engine-t, hanem vagy könyvtárakra támaszkodunk, vagy teljesen alulról építkezünk.

3. Eddigi képfeldolgozás játékokban

3.1. Grafikai megoldások

Természetesen a videójátékok eddig is használtak képfeldolgozást, de leginkább csak grafikus megjelenítésre, illetve felskálázásra. A jobb teljesítmény elérése érdekében például az NVIDIA 2018-ban meghirdette a DLSS funkciót, ami tulajdonképpen egy felskálázó algoritmus, aminek feladata, hogy szebb képet adjon, mint amit a játék kínál, természetesen a lehető legkevesebb számítási kapacitás igénybevételével. Ennek első iterációja egy CNN-re, azaz konvolúciós neurális hálóra támaszkodott, ami egy olyan típusú neurális háló, ami mélytanulást alkalmazva próbálja megjósolni az elveszett, vagy nem is létezett adatrészeket akár képeken, hangfájlokban, vagy szövegekben.[1] Például egy távolságtól nagyon pixelessé vált emberalakot próbál restaurálni, kirajzolni az arcát, egyéb részleteit.

3.2 Alakzatfelismerés

Az alakzatfelismerés egy eléggé elterjedt, a technológiában már jól körüljárt és behatárolt megoldás, és rengeteget fejlődött az évek alatt. A probléma nagyon egyszerű, adott egy input, ami lehet egy fénykép, képfájl vagy akár élő kamera felvétel is, amin detektálni akarunk formákat, körvonalakat, mozdulatokat. A python nyelv segítségével nagyon sok rendszerbe implementáltak már olyan funkciókat, amiket például autófejlesztésben használnak, mint a

[1] https://en.wikipedia.org/wiki/Deep_Learning_Super_Sampling

LiDAR rendszerek is. Természetesen ez csak egy példa, rengeteg jó felhasználási módja van. A fotós szakmában az utómunkában is segíthet egyes esetekben, például szegmentálni az emberi alakokat a háttértől, vagy egy adott színt kiszűrni. Vannak már megvalósított projektek arra is, hogy kézmozdulatokkal irányítsunk számítógépeket, ezáltal akár okosotthonokat is. Szóval kijelenthetjük, hogy az alakzatfeldolgozás számtalan lehetőséget kínál az informatikában, azonban a videójátékokban még eléggé kiaknázatlan ez a megközelítés.

Alapvető alakzat felismerési módszerek[2]:

- Centroid – (más néven tömegközéppont) egy alakzat geometriai középpontja, amelyet általában az alakzat pixeleinek (vagy pontjainak) koordinátáinak átlagolásával számítanak ki. A bináris képen például az alakzathoz tartozó pixelek x- és y-koordinátáinak átlaga adja meg a centroid-ot.
- Bounding box - (befoglaló téglalap) egy olyan téglalapot jelent, amely teljes egészében tartalmazza az alakzatot. Gyakorlatilag megkeressük az alakzat legkisebb és legnagyobb x- és y-koordinátáit, és ezek alapján definiáljuk a téglalapot. Ez az eljárás elég egyszerű, de nagyon gyakran használt normalizációs lépés.
- Stroke-matching - (vonal- vagy vonal-sorrend összehasonlítás) egy olyan technika, amely során egy rajzolt vonal (stroke) és egy sablonvonal közti hasonlóságot határozzuk meg. A matching során általában a két vonal pontjait hasonlítjuk össze és kiszámítjuk az átlagos távolságot közöttük, lehetőleg a legjobb rotáció és transzformáció mellett.

3.3 Módszerek

A képfeldolgozásban leggyakrabban két módszerrel dolgoznak, Deep-learning segítségével, illetve a Rule-based alapú felismeréssel.

Deep learning: A deep learning röviden úgy működik, hogy felhúzunk egy modellt, amit rengeteg példával és adattal betanítunk, akár több százezer mintával, és ezt használjuk utána a mintáink felismerésére. Ennek a módszernek is van rengeteg változata, például az actor critic módszer, melyben a modellünk míg tanul, folyamatosan egy „szakértő” visszacsatolást ad a neurális hálónak az alapján, hogy milyen eredményt adott egy-egy betanító mintára. Ez rengeteg mintát igényel, rengeteg időt is felemészt ennek a modellnek a betanítása, így a

[2] https://www.inf.u-szeged.hu/~pkardos/oktatas/kepfeld/DIP_10.pdf

programom elkészítése során végül nem ezt a módszert alkalmaztam. Továbbá a program futásidőben is sokkal nagyobb gépigényt igényelt volna. A módszer alapvető működésén kívül nem célja a dolgozatnak ezt a témát hosszasabban kifejteni, hiszen csak erről az egy témáról lehetne írni egy teljes szakdolgozatot.

Rule-based: szabályrendszer alapú felismerés, aminél egy algoritmus alapján szegmentálod a képet vagy az alakzatokat és a szegmentált képeken számításokat végezve, vagy mintához hasonlítva, valamekkora bizonyossággal megpróbáljuk beazonosítani az objektumot. Ez a fajta megközelítés sokkal kevésbé erőforrás igényes, persze sokkal nagyobb esély is van arra, hogy a modell hibázik, főleg, ha egymáshoz nagyon hasonlítanak a felismerhető alakzatok. Azonban mivel a programomban futásidőben terveztem felismerni az alakzatokat és ezek alapján döntést hozni a játékmenetben, így ezt a módot valósítottam meg.

Ilyen szabályok például:

- **Canny edge detection** – ez egy éldetektáló operátor több szintű algoritmus, ami egy széles tartományát detektálja az adott képnek. 1986-ban fejlesztette John F. Canny [3]
- **Douglas–Peucker algoritmus** - más néven Ramer-Douglas-Peucker algoritmus– ez egy olyan eljárás, amely vonalszakaszokból álló görbét egyszerűsít olyan módon, hogy azt kevesebb pont használatával, de az eredeti inputhoz képest megegyező alakban reprezentálja. Ez volt a legkorábbi algoritmus, melyet kartográfiai generalizálásra fejlesztettek. [4]
- **Template matching**[5] – ez egy olyan OpenCV által is megvalósított módszer, melynek működési elve az, hogy egy neki megadott úgynevezett mintaképet (ez a template) megtaláljon egy nagyobb képben.

A dolgozat elkészítéséhez végül egyfajta rule-based megoldást használtam, a „template matching”-hez hasonlót, megvalósításában kicsit módosítva, oly módon, hogy előre megadott alakzatokat adtam a projekthez, aminek a módja csupán azok egyszer-kétszer történő megrajzolása, amiket az algoritmus utána beilleszt a mintákhoz.

[3] Canny edge detector - Wikipedia
[4] Ramer–Douglas–Peucker algorithm - Wikipedia
[5] OpenCV: Template Matching

Célkitűzés

A játékipar széleskörű elterjedése és a napról napra növekvő fejlesztők száma azt mutatja, hogy igencsak értékes az ebben szerzett tapasztalat.

Az évek során kialakultak alapvető elvárások a gyártók felé, ezek egy része technikai, másik része pedig inkább anyagi. A technikai alapelvárások egyik fontos szempontja, hogy a szoftver megfelelően fusson adott specifikációk alatt, azaz a program futtatása a lehető legkevesebb számítási kapacitást igényelje, a lehető legjobb grafikai megjelenítéssel, és természetesen stabilan fenn is tartva ezt, konzisztens teljesítménnyel. A sietve kiadott vagy összecsapott programoknál esetenként megfigyelhető volt ezen szempont elhanyagolása a játék grafikai megjelenése javára. Ezeket a hibákat a többségben azonban a fogyasztók panaszára megjelenés után jellemzően pár hónapon belül javították a múltban.

Az anyagi szempont állandó vitát képez a kiadók, gyártók, és fogyasztói réteg között. A kiadó jellemzően több pénzt akar, a gyártó szintén, azonban ismerve a piaci reakciókat, próbálhat minimalizálni, a fogyasztó pedig egyértelműen a minimumot szorgalmazza. Jellemzővé vált, hogy a AAA-s játékok $60 USD-ba kerülnek, azonban sajnos ennek is növekvő tendenciája van. Az egyéni kiadók hozhatnak ebbe reformot a jövőben. A nagyobb cégekben egyre többet csalódó vásárlók sokkal hajlamosabbak a kisebb fejlesztők felé pártolni azon játékok minősége és ára miatt is. Jellemzően olcsóbbak az indie fejlesztésű játékok, de a tapasztalatok alapján, mivel nem sürgeti őket egy kiadó, ezért több idő alatt ugyan, de sokkal minőségibb termékeket gyártanak, melyeket a játékosok jobban is értékelnek vásárlással mintsem a nagyobb bizalmukat vesztett cégeket.

Ezek a cégek és egyéni fejlesztők jól belakták már a piacot, azonban a változó és egyéni igények miatt nem mondható telítettnek. Mindig lehet ebben az iparágban újdonságot belevinni, új értéket teremteni. Újat hozni leginkább kétféleképpen lehet jelenleg. Az egyik lehetőség történetben, írói oldalon meglepni a közönséget, az érzelmekre hatva. A másik lehetőség technológiailag újítani, valami olyat belevinni a játékmenetbe, vagy a grafikai részébe a játékoknak, amit eddig más nem, és korszakalkotó lehet, vagy legalább felpezsdíti a megszokott sztenderdeket.

A szakdolgozathoz elkészített program erre az újdonság behozására törekszik, és célja egy olyan kezdetleges videójáték létrehozása, amely formailag újít, eltér a megszokottól és beleviszi a képfeldolgozást, alakzatfelismerést a játékmenetbe, ezzel megmutatva, hogy ennek a módszernek komolyabb, nagyobb skálájú megvalósítása milyen újdonságokat hozhatna a

globális játékiparba. A fejlesztett játék célja az lesz, hogy eddig nem látott, módon alkalmazza a felhasználó eszközeit bevitelként. Feladata lesz:

- az egérrel történő rajzolás megvalósítása,
- az inputként szolgáló rajzot a lehető legpontosabban felismernie
- végrehajtani az adott alakzathoz rendelt akciót,
- arra ösztönözni a játékost, hogy a program lehetőségeit kreatívan és teljes mértékben kihasználja, mérlegelve a különböző alakzatok nehézségeit,
- felkeltenie a játékos figyelmét, hogy érdekelt legyen a gondolatban, miszerint helye van ennek az új megközelítésnek a videójáték iparban

Felhasznált anyagok és eszközök

A dolgozatban számos forrást, információt és módszert használtam fel, főleg a játék elkészítése során. Leginkább a technológiai megvalósításban merítettem ihletet.

- Pygame – Kezdetben ebben készült a játékprogram, azonban ennek nagyon bonyolult felépítése miatt felhagytam ennek használatával, és Unity engine-re váltottam.
- Unity Drawing Recognition projekt[6] – Ebből a nyílt forráskódú projektből merítettem inspirációt rajzolási rendszer fejlesztésére és alakzat felismerésére. A projekt jól dokumentáltságának köszönhetően, remekül be tudtam építeni a programomba ezzel elősegítve a dolgozatom céljának elérését. Ebben a projektben fellelhető a rajzoló és alakzatfelismerő algoritmus is.
- Unity Asset store – a Unity fejlesztő környezet saját asset oldala. Az itt fellelhető open-source, azaz szabadon felhasználható assetek biztosították a programban található grafikákat, karakter mozgásokat képkockánként megrajzolva.
- Unity game engine - ebben a szabad felhasználású játékmotorban készült a játék érdemi része, a frontend, és a pályák. A játék működését és logikáját biztosító kód is ebben fellelhető melynek írása azonban nem ebben készült, saját szövegszerkesztő hiányában.
- C# programnyelv
- Visual Studio Code - A Unity lehetőséget nyújt integrálva használni a Visual Studio-t, ezzel is települ, azonban ez a macOS rendszeren nem elérhető, így Visual Studio Code-ot használtam végül a program logikájának megírásához. Ezt hasonlóképpen nagyon egyszerű volt társítani a Unity-hez.

Ezen eszközök felhasználása, és ezeknek jól dokumentáltsága, vagy éppen széleskörű támogatottsága jelentős szerepet játszott a célként kitűzött platformfüggetlen program lefejlesztésében.

[6] https://github.com/gilbertdyer2/UnityDrawingRecognition

Alkalmazott módszerek

A munkám során igyekeztem a lehető legátláthatóbban dokumentálni a munkámat, emlékeztetésül, hogy hol hagytam abba a munkát, vagy hogy mivel foglalkoztam eddig, konzulensem javaslatára naplót vezettem.

- Clean Code: A program írása során törekedtem az elvre, miszerint átlátható, rendezett és könnyen értelmezhető legyen a forráskód a későbbi bővíthetőségre való tekintettel. Emellett pedig ahol érdemes volt, ott a programban kommentekkel magyarázva tettem egyértelművé a kód működését.
- Verziókezelés: Gitlab és Github; a fejlesztés során felhasznált kétfajta verziókezelő iterációja a Git-nek. Ezek használatával volt biztosítva a kód minden iteráció után a programban történő változások elmentését, és hiba esetén a változtatások visszagörgetését. Továbbá ez lehetővé tette a több munkaállomáson történő fejlesztést is, így nem korlátozódott a munka egyetlen laptopra, vagy asztali számítógépre, hanem mindig mindenhol az aktuális verziót lehetett módosítani az összes eszközömön.
- Agilis fejlesztési elvek: A munka során sprintekbe rendezve dolgoztam, iterációnként kijelöltem egy készítendő funkciót, és azt teljesen körbejártam míg készen nem lett. Az elkészült funkciókat később vissza-vissza ellenőriztem ahogy haladtam előre a többi feladattal is. Ez a fajta fejlesztés azért tette számomra könnyűvé a haladást, mert a Unity-ben könnyű funkciónként haladni, nem sok dolog függ más funkció megvalósításától, ezért rugalmasan tudtam haladni egy-egy objektummal, logikával, vagy pályával.

Megvalósítás

1. Backend

A projektet kezdetben Pygame-ben készítetten az OpenCV csomag miatt, ugyanis azt hittem, abban könnyebb lesz képfeldolgozást írni a programba. Ez még így is lenne, de a pygame keretrendszer nagyon nem kezdőbarát, mivel egyetlen pályarész megvalósításához is több száz sornyi kódot kell írni, minden vizuális segítség nélkül.

Ebből az okból fakadóan a dolgozat és a projekt végül Unity segítségével készült el. A projektet a Unity 6.0 verziójában terveztem meg, és az ezen verzió által biztosított csomagok segítették a munkámat. A projekt nagyját grafikusan össze lehet pakolni, majd ebből az engine fájlokat készít, esetenként nekünk kell manuálisan, létrehozni scripteket. Ezekre egyenként a dolgozat során részletesebben kitérek. A programban ügyeltem az objektumorientáltságra, ezt szorgalmazza maga az engine is. Minden objektum adattagját lehet a kódból módosítani, de az engine-ből csakis a publikus láthatóságúakat. Ez utóbbiakat a játék teszteléséhez akár futásidőben is módosíthatjuk a program leállításáig, ezek a változtatások nem mentődnek el, viszont nagyon hasznosak tudnak lenni a játék működésének finomhangolásában. A Unityben érdemes almappákat létrehozni a különböző feladatoknak szánt objektumokhoz, vagy kódrészekhez, hogy könnyebben megtaláljuk, amit keresünk, és rendszerezve tartsuk a projektünket.

Hasznos csoportosítások, amiket követtem a dolgozat készítése közben:

- Assets – főmappa, ami tartalmazza az egyes részeket
- Animations – az animációkat tartalmazó mappa, melyben az animációk több képkockányi összefűzött úgynevezett sprite-ok.
- Drawing Recognition – az alakzatfelismerő algoritmust tartalmazó mappa
    - Scripts – az alakzatfelismerőnek különféle módszereit, a Character és CharacterLibrary osztályokat tárolja. Ezek felelnek az algoritmus megfelelő működéséért.
- Prefabs – az előre elkészített „template” -ként szolgáló objektumokat tárolja, ezekből példányosítunk a programba, például egy ellenfél objektumot vagy egy „gem” -et. Ezek a prefabek gyakorlatilag sütiformaként tekinthetőek, lehelyezési szabályokkal gyorsan és erőforrástakarékosan lehet több egyedet a játékba helyezni anélkül, hogy a memóriában túl sok példányt tárolnánk előre.

- Scenes – a játék színtereit tároló mappa. A színtereken lelhetőek fel az előre felépített pályák, a grafikus interfészeket is ilyen scene-ekre kell felhelyezni és ezek láthatóságának állításával lehet a kívánt helyre tenni őket.
- Scripts – a játék logiáit tartalmazó szkriptek mappája
- Sprites – itt tárolom a játék grafikájához szükséges „asset”-eket, minden objektumnak aminek kinézete, textúrája van, annak a kinézete itt fellelhető
- Tilemap – ez a mappa tartalmazza a pálya rajzolásához szükséges blokkokra osztott egyenként egy pályakockányi méretű objektumokat, ahonnan rajzolva ecsetszerűen felépíthető egy pálya

Nagyon intuitív a Unity kezelőfelülete és a rengeteg dokumentációnak köszönhetően könnyen ki lehet igazodni a használatán. Széleskörű elterjedése miatt pedig rengeteg forrást lehet találni fórumokon, melyek segíthetnek az esetleges elakadásokban, vagy ötletet adhatnak egy-egy funkció megvalósításának mikéntjében.

1.1 Objektumok

Alapvetően a projekt jól szétválasztható apró részekre, amelyeket objektumok alkotnak. Egy ilyen objektum a játékos, ellenfél, egy adott fal, aminek a játékos nekiütközhet, vagy például a kamera is, amelyik követi a játékost. Az objektumoknak lehet adni textúrát, viselkedési scripeteket, hierarchikusan lehet őket rendezni, örököltetni egyiket a másikból, triggerként beállítani őket, és még sok más lehetőség van bennük. Az objektumoknak rengeteg alapvető fajtája van, és ezek létrehozásakor a projekt mappába fajtájuk alapján kapnak alapértelmezett paramétereket, amiket nekünk kell beállítani. Például egy kamera vár egy játékos paramétert alapértelmezetten, de kódból adhatunk bárminek bármilyen paramétert, amelyet a grafikus felületen társíthatunk egy scripttel, egy másik objektummal, amivel interakcióba lép, vagy éppen egy textúrával.

1.1.1 Játékos

Az irányítható karakter, a billentyűzeten a nyilakkal, illetve w,a,s,d-vel irányítható, egérrel pedig a képernyőre való rajzolást lehet vezérelni.

1.1.2 Ellenfél

Enemy prefab nevezetű mintaként elkészített osztályból példányosíthatunk működő ellenfeleket a pályára, melyek, ha ütköznek a játékossal, akkor sebzést okoznak neki.

Megsemmisülésükkor úgynevezett „loot table”-jük alapján dobhatnak a pályára „gem” -et, élettöltő szív objektumot vagy semmit.

1.1.3 Gyűjthető „gem” -ek

A játéktérre időnként véletlenszerűen lekerülő kövek, ezek jelentik a játékosnak a pontot. Ezek összegyűjtésével, a képernyőn látható haladási sáv elkezd feltöltődni, és amint tele lesz, az „e” betű nyomva tartásával, vagy egy csillag alakzat sikeres lerajzolásával új pályára kerülünk.

1.1.4 Heart item

Egy ellenfél megsemmisítése után bizonyos eséllyel az ellenfél addigi pozíciójára kerülhet egy pélány, mely felvételével a játékos visszatölthet életerőt, amennyiben az nincsen maximális értéken.

1.1.5 Fireball bullet

A játékos által kör alakzat rajzolására kilőhető prefab példánya, mely rendelkezik egy ütközési zónával, egy időbeli élettartammal, ameddig a pályán marad, és egy sebzéssel, amit az ellenfélnek oszt ki ameddig az érintkezik az ütközési zónájuk.

1.2 Pálya

A pálya a játszható játékterületet határolja. Ezen belül mozoghat a játékos. A pályán találhatóak különböző rétegen levő blokkok melyek különböző működést tesznek lehetővé. Ilyenek a *ground* és a *wall* szint.

- A *ground* szinten levő textárval rendelkező pályaelemeken a játékos sétálhat, rajta marad, és innen kezdeményezhet a ugrás akciót is.
- A *wall* szinten levő pályaelemeknek a játékos nekiütközik, nem tud áthaladni rajta, és ezeknek nekiugorva tud a falakról tovább ugrani falról-ugrást kezdeményezve az ugrás gomb megnyomása után.

1.2.1 Pályák betöltése

A pályák betöltéséért több szkript is felel, de a legfontosabb a *GameController.cs,* itt állítjuk be a játék kezdőpozícióját, az első pályát, majd itt is ellenőrizzük, hogy a játékos elért-e már elég pontot a tovább haladáshoz. Itt növeljük a pontszámot, ami szükséges a továbbjutáshoz, és megjelenítjük a haladási sávon a képernyő jobb alsó sarkában található csúszka segítségével. A pályák egy listában vannak eltárolva, melyek sorban követik egymást, és ezen listán végig iterálva tudjuk betölteni mindig az aktuálisan következő pályát. Ha a lista végére érünk, a számozás elölről kezdődik, ezzel biztosítva, hogy a játék végtelenségig játszható legyen. A játékmenet célját miszerint minél több napon keresztül maradjunk életben,

is egy pálya változtatásánál növeljük. Leegyszerűsítve: minél több pályán jutottunk túl, annál ügyesebbek voltunk, annyi napot éltünk túl. Melyet a játék elvesztésekor ki is írunk.

2. Frontend

A frontend feladata a játék képi megjelenítése, ez az ami visszajelzést ad a játék menetéről, interaktívan láthatjuk a játék folyamatát, aktívan részt vehetünk benne úgy, hogy a számolásokat nem a szemünk előtt, hanem a backenden végzi.

A frontendet a Unity engine szinte teljeskörűen elkészíti nekünk, nekünk csak az általunk készített objektumokhoz szükséges grafikákat, textúrákat készítenünk, ha azt akarjuk, hogy megjelenjenek a játékban.

Ezen grafikák lehetnek folytonos vektoros rajzok, vagy pixelenkénti diszkrét értékeket tartalmazó úgy nevezett sprite-ok is. Ezeket el lehet készíteni az engine-ben is, de lehetőségünk van külső programok használatával is elkészíteni ezeket. Amennyiben nem szeretnénk saját grafikákat használni, esetleg nem vagyunk jó grafikusok, vagy csak egy demo összerakására készülünk, ahol a kinézet nem az elsődleges szempont, abban az esetben a Unity lehetőséget biztosít a Unity asset store hozzáférésével mások által már elkészített textúrák használatára.

A dolgozat elkészítése során én is használtam innen package-eket, a legtöbb textúrámat innen illesztettem be, szigorúan nyílt forráskódú és szabad felhasználású grafikákat használva. Ilyen például a karaktert alkotó róka figura, vagy a hátteret és a pálya dizájnt is adó stílusok. A programom jövőbeli továbbfejlesztésénél tervezem ezek lecserélését saját szellemi tulajdonú, általam készített textúrákra.

A felhasznált grafikákhoz tartozó asset csomagok:

- A róka karakterhez: https://assetstore.unity.com/packages/2d/characters/sunny-land-103349
- Pálya részek: https://assetstore.unity.com/packages/2d/environments/pixel-fantasy-caves-152375
- Pálya részek: https://assetstore.unity.com/packages/2d/environments/platformer-set-150023

3. Alakzatfelismerő algoritmus

3.1 Működési elve[7]

A program futásidőben eltárolja a rajzolt alakzatokat egy „Bitmap” osztály példányába. Ez a példány átkonvertálja a rajzolt pontokat négy különböző háló stílusra. Ezek a „GridMap”, „CircleMap”, „HorizontalMap” és „VerticalMap”. Mindegyik Bitmap másképpen vizsgálja az inputot és futásidőben az összes Bitmap a saját logikája alapján összehasonlítja a mintát az inputtal, és ad rá egy egyezési pontszámot. Mindegyik map stílusnak megadható egy súlyozás, arra a célra, hogy pontosan beszabályozhassuk, hogy az adott Bitmap mennyire legyen releváns az eredménypontszám kiszámításában. (SetWeights() függvény a kódban).

Miután mind a négy Bitmap fajta kiértékelte a rajzolt alakzatot, ezeknek az összegét a súlyuk alapján átlagolva összeadjuk és az együttes pontszám jelzi majd, hogy összességében melyik minta alakzattal egyezik leginkább az input. Az eredmény az lesz, amelyiknél a legkisebb a hibapontszám, tehát a legmagasabb az egyezési pontszám.

Egyenként mind a négy fajta Bitmapnak megvan a maga hibája, ami alapján egyik nem alkalmas az egyenes alakzatok összehasonlítására, másik pedig pont a kerekeket téveszti össze, azonban a négy együttes alkalmazásával elkerülhető, vagy legalábbis minimalizálható ez a fajta átlagos hiba érték.

Fontos megjegyezni, hogy ez a módszer nem veszi számításba a pontok lerajzolásának sorrendjét, nem tartja számon azt, hogy melyik pontot rajzoltuk hamarabb a másiknál, és melyiket melyik irányból húztuk. Ezáltal nem téveszt össze két kört csak amiatt mert nem ugyanabból az irányból húztuk, mint a mintát. Emiatt a program és pontozás eredménye teljesen független a felhasználó rajzolási stílusától.

3.1.1 GridMap

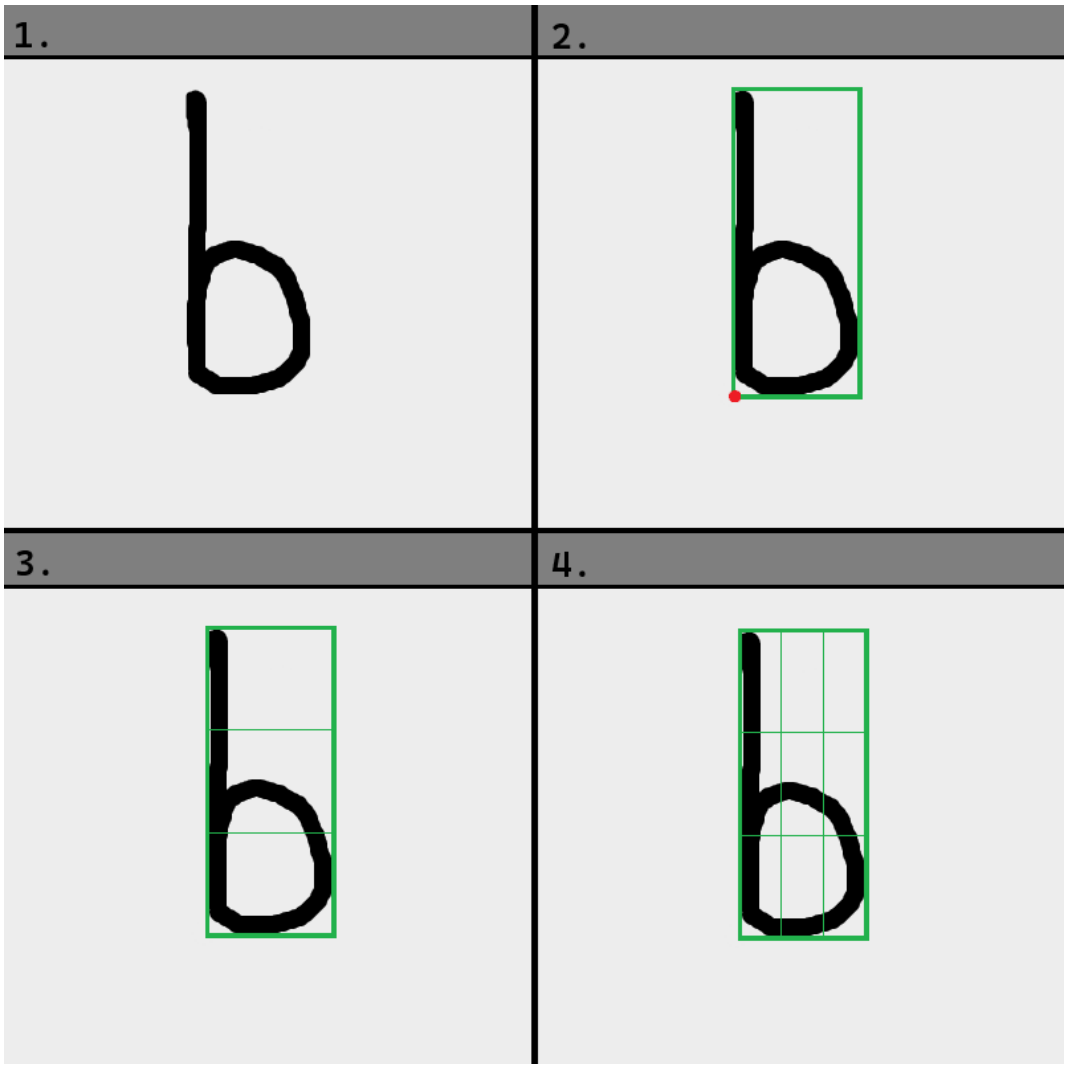
A modell jól működik a torzult alakzatokkal, amik nyújtottakra, vagy nyomottabbra sikerültek, de hajlamos hibázni akkor amikor egy karakter a szárhosszán vagy elhelyezkedésén kívül ugyanolyan, mint egy másik. Ilyen hiba példa a ’p’ és ’b’ betű keverése.

[7] A teljes 3.1 fejezet forrása a „UnityDrawingRecognition” programomba beépített projekt „How it Works” részből származik, kiegészítve a saját gondolataimmal és értelmezésemmel.

A GridMap a pontok eloszlását reprezentálja egy a karaktert befogó téglalapban. Ez a módszer hasonlít a Bounding boksz alapvető alakzat felismerési módszerhez, miszerint az egész alakzat köré egy azt befoglaló téglalapot húzunk fel.

A módszer stratégiája:

1. Először is megszerzi a rajzolt alakzat lerajzolt pontjait.
2. Megtalálja a befogó pontjait az alakzatnak, és köréhúzza a befoglaló téglalapot és kijelöli a bal alsó sarkát az új középpontnak (origo-nak) a két irány szempontjából, ami egyfajta x és y tengellyé válik ezzel.
3. Felosztja a függőleges tengelyt ’n’ darab azonos méretű szeletre.
4. Felosztja a vízszintes tengelyt ’n’ darab azonos méretű szeletre.
   *(A két lépésben az egyik ’n’ ugyanannyit jelent mint a másikban, ezáltal egy n*n-es grid-et kapva)*



3. ábra, (GridMap működése)
forrás: GitHub - gilbertdyer2/UnityDrawingRecognition

Ezután meghatározza, hogy az így felállított kis mezők, egyenként mekkora százalékban tartalmazzák a teljes rajzot. Ezt egy kétdimenziós tömbben tároljuk. Ennek adatai az erre a ’b’ betűre tekintve rendre:

0.167	0.0	0.0
0.225	0.069	0.069
0.225	0.069	0.179

1. táblázat

Emiatt is összekeverhető például a ’p’ és ’b’ betű, hiszen hasonló százalékok jönnek ki, a bal oldali 3 függőleges kis mező miatt, amiben jól látható, hogy a rajznak legnagyobb százaléka található. Ahhoz, hogy a tömbben kapott értékeket használjuk, vesszük a négyzetes átlag hibát a két karakter GridMap értékei között, amely kettő a tárolt minta alakzat és az input.

A kiszámítás úgy történik, hogy vesszük a két karakter GridMap értékeit, és egyenként hozzájuk adjuk az egy-egy érték közötti különbség négyzetét az összesített hibaponthoz. (Ebben a hibaösszegben határozzuk meg mennyiben tér el egymástól a kettő alakzat) Ezután, ha végzett az összes mező közti különbségével, ezután elosztjuk a teljes hibaösszeget a mezők számával, ami ugye n*n, jelen esetben 9. Ezzel megkapunk egy normalizált értéket a két GridMap alapján, amely azt jelzi, hogy minél kisebb az érték, annál inkább egyezik a két alakzat.

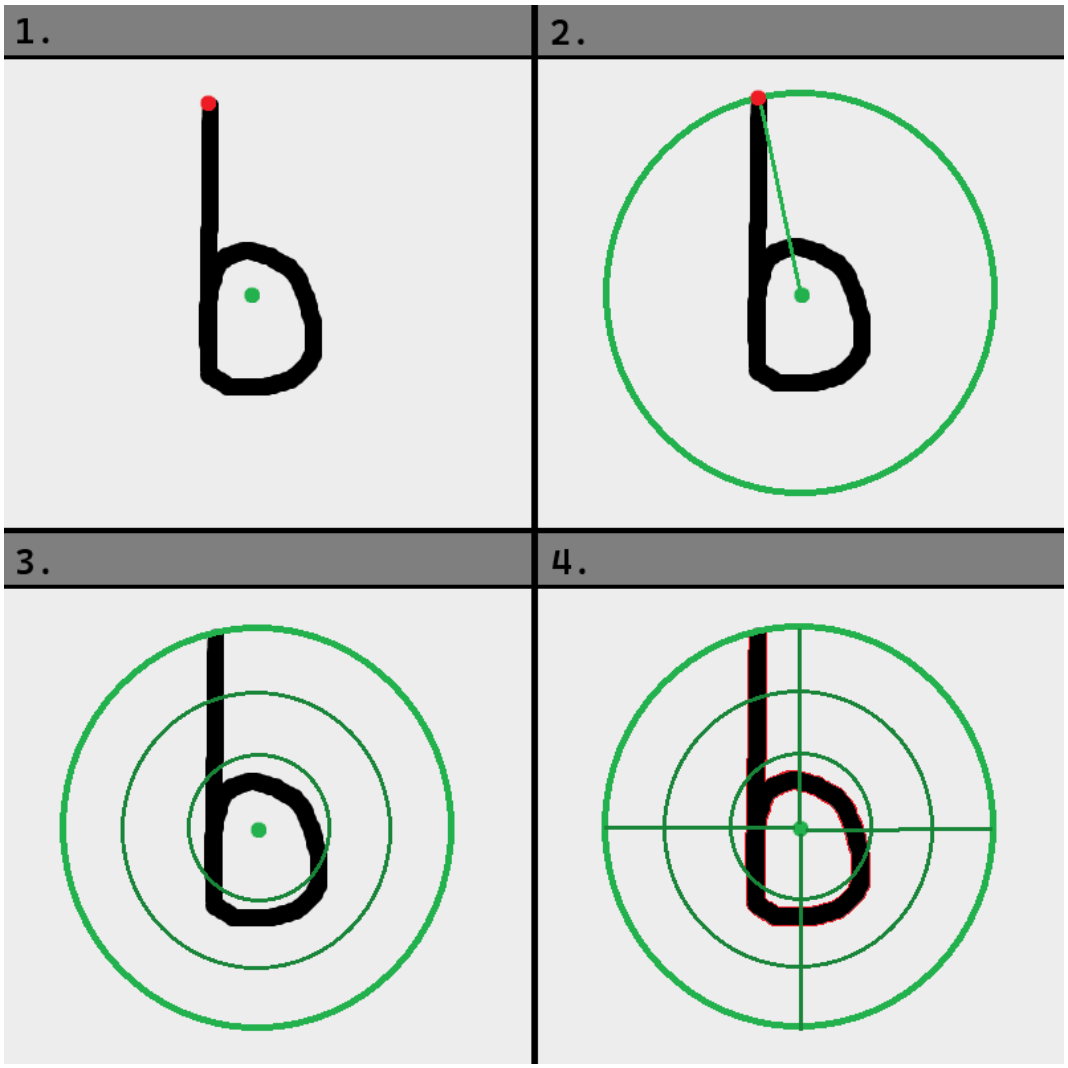
3.1.2 CircleMap

Ez a modell jól működik, nagyjából az összes inputra és alakzatra, kevésbé érzékeny azokra, mint a GridMap, viszont ami a GridMap előnye volt, hogy jól kezeli az eltorzulásokat, az sajnos ennek a megoldásnak a nehézsége, ebben a módszernek az a hátránya, hogy ha kicsit nyújtottabb vagy nyomottabb az inputunk a mintához képest, akkor hajlamosabb hibázni.

A CircleMap egy kétdimenziós rácsszerkezetként működik, hasonlóan, mint a GridMap, azonban nem befogó téglalap, hanem egy kör alakú rácsban határozza meg a rajz pontjait.

A módszer stratégiája:

1. Kiszámolja a mediánját az összes pontnak, amit rajzoltunk az inputban, és a mediánból húzott legtávolabbi pontot is meghatározza.
2. Rajzol egy kört az alakzat köré, egy befoglaló kört, aminek a középpontja a medián pont, a sugara a körnek pedig a mediánból a legtávolabbi pontig húzott távolság.
3. Felosztja a kört ’n’ darabszámú gyűrűre, amelyek egyenlő szélesek egymáshoz képest.
4. Felosztja az így kialakult körgyűrűket 4-be, egy vertikális és egy horizontális vonallal.



4. ábra, (CircleMap működése)
forrás: GitHub - gilbertdyer2/UnityDrawingRecognition

A GridMap megoldásához hasonlóan, az így kapott negyedkörökben kiszámolja a program, hogy az adott alakzatnak mekkora százaléka helyezkedik el benne. Minden negyedkörnek így 'n' darab kisebb gyűrűje van, jelen esetben 3 mindegyikben. Ezekben a kis szeletekben számol a program, és ezeket az adatokat rendre így értékeli ki ennél a példánál:

- Első körgyűrű (legbelső; 2.táblázat), az adatok az ábrás elrendezéshez igazodnak

bal felső: 0.153	*j.f.:* 0.102
b.a.: 0.093	*jobb alsó:* 0.025

2. táblázat

- Második körgyűrű (3.táblázat)

0.119	0.000

0.280	0.110

3. táblázat

- Harmadik körgyűrű (4.táblázat)

0.119	0.000
0.000	0.000

4. táblázat

Az egyezési pontok kiszámításához ugyanazt a négyzetes átlag hiba számítást alkalmazzuk, mint a GridMap-nél, csak cellák helyett a körcikkelyekkel.

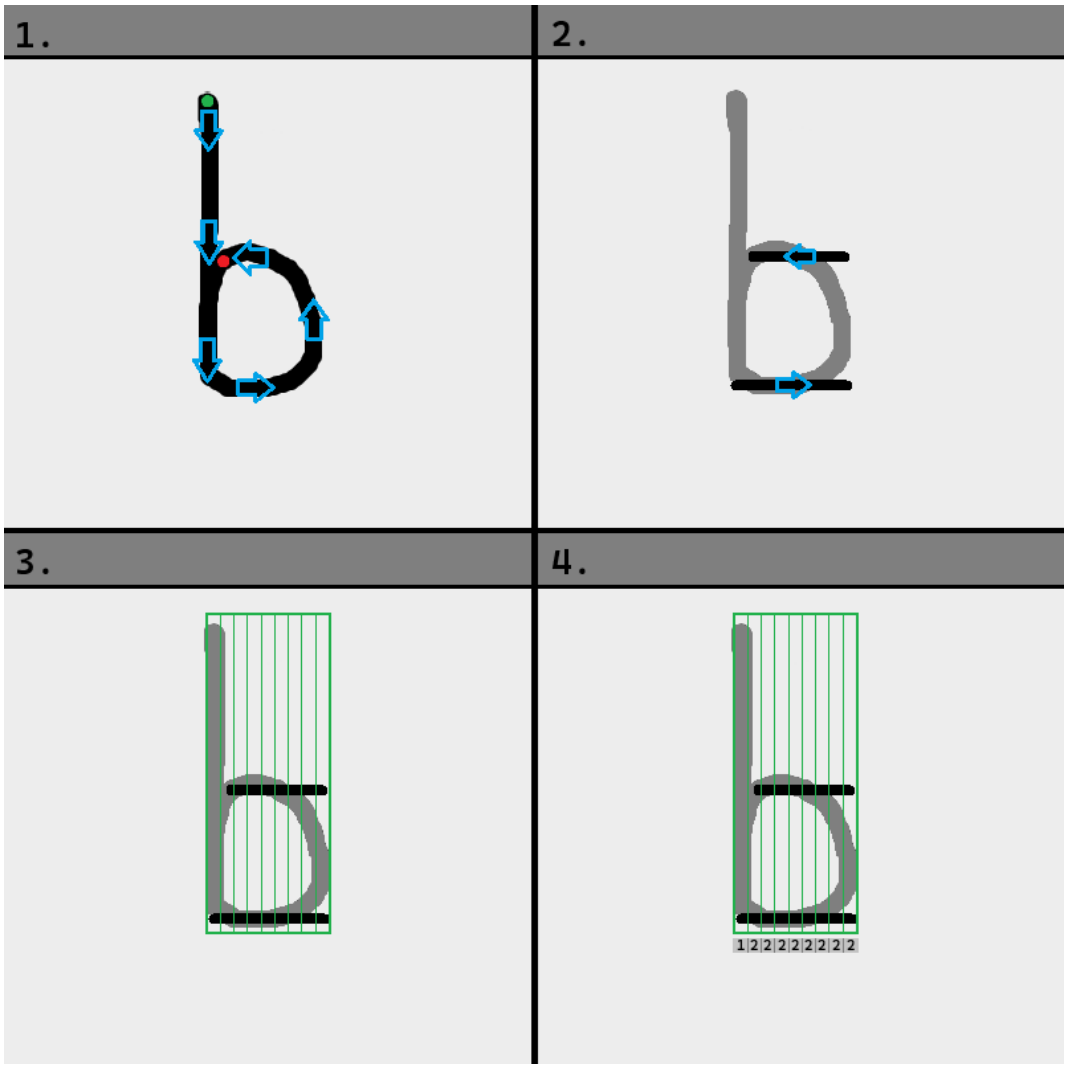
3.1.3 HorizontalMap és VerticalMap

Ennek a két bitmapnak egybefoglalható a működése, hiszen leginkább csak abban térnek el, hogy az egyik vízszintesen, a másik pedig függőlegesen szeleteli a kapott inputot. Mindkettőnél fontos megjegyezni, hogy ezek működés közben az előzőkkel ellentétben figyelembe veszik a rajzolás sorrendjét, de csupán vonalegyszerűsítés szempontjából, az egyezés pontszámába nem számít bele.

Mindkét Bitmap egy egydimenziós reprezentálását vizsgálja a kapott inputnak. A VerticalMap a pontok előfordulását függőleges tengelyen vizsgálja, a HorizontalMap pedig a vízszintes tengelyen teszi ugyanezt.

HorizontalMap stratégiája:

1. Megszerzi az input rajzolt pontjait, és ezeknek rajzolás béli sorrendjét.
2. A kezdőponttól indít egy vonalat, és rögtön befejezi amint az x tengely béli iránya változik.
3. Felosztja a rajzot az x tengely mentén egyenlő méretű oszlop szeletekre.
4. Kigyűjti az így kapott értékeket, amely számok azt jelölik, hogy hány alkalommal fordul elő rajzolt pont a függőleges szegmensben. Ezeket a számokat egy tömbben tárolja.



5. ábra, (HorizontalMap működése)
forrás: GitHub - gilbertdyer2/UnityDrawingRecognition

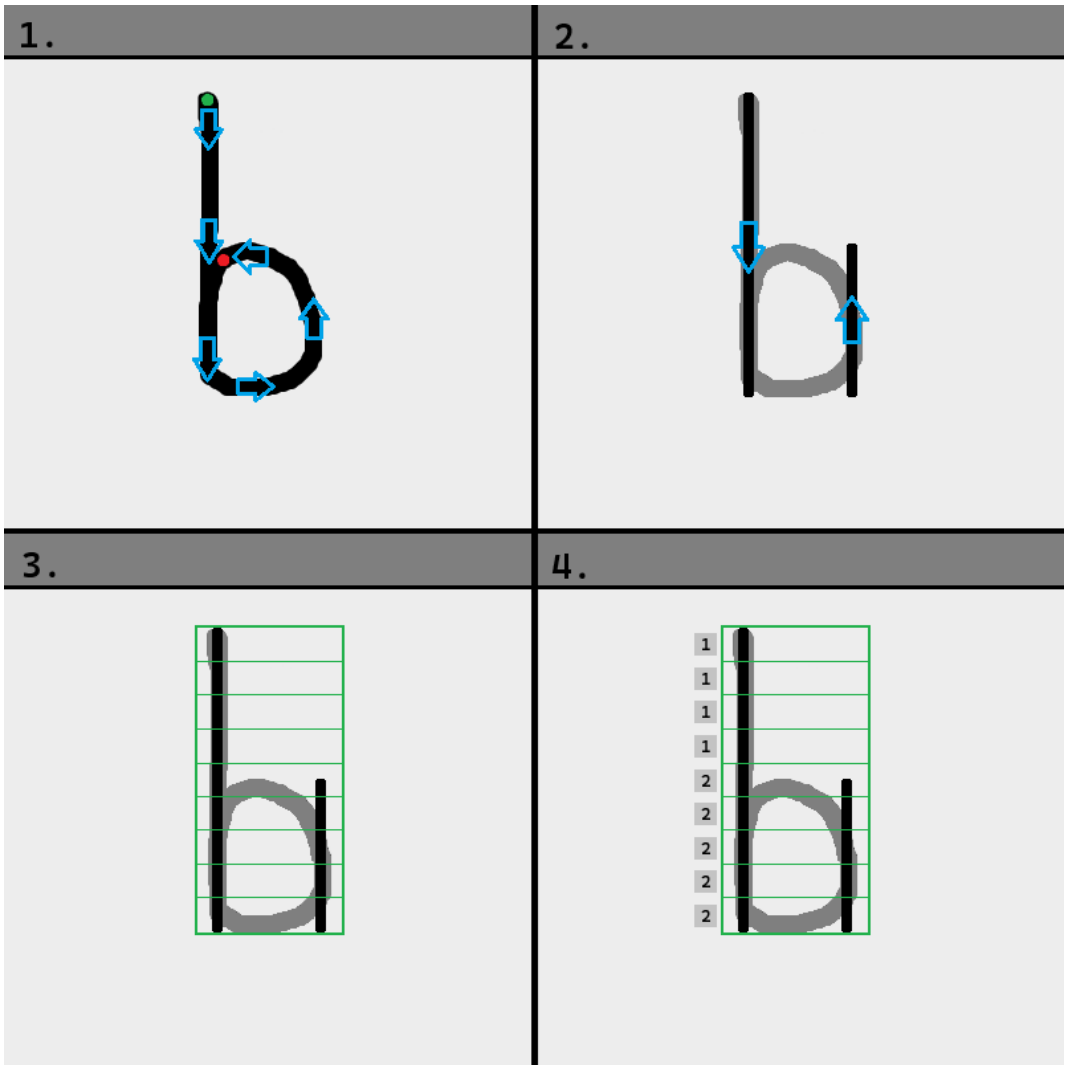
Ebben a példában a tömb elemei sorrendben: {1,2,2,2,2,2,2,2,2}

Az összehasonlítási pontszám kiszámításához összehasonlítjuk a minta és az input HorizontalMap-ból kapott tömbök egyes indexeiből az értékeket. Amennyiben az értékek egy-egy indexnél nem egyeznek, a hibapontot a következőképp számoljuk ki: ***'1/összes vágás száma'***.

Ahhoz, hogy pontos értéket kapjunk akkor is, ha az alakzat nem középpontosan szimmetrikus, további összehasonlításokat végzünk a két tömb elemén azután, hogy az egyik tömb indexeit eltoljuk balra vagy jobbra 'n'-szer, és minden ilyen iterációban végzünk egy összehasonlítást. Ezek után a minimum kapott hibapontot vesszük egyezési alapul, ez lesz a végső pontszámunk.

A VerticalMap stratégiája megegyezik a HorizontalMap-éval, csupán szeleteléskor nem függőleges oszlopokra osztja a képet, hanem vízszintes sávokra:

1. Megszerzi az input rajzolt pontjait, és ezeknek rajzolási sorrendjét.
2. A kezdőponttól indít egy vonalat, és rögtön befejezi amint az y tengely béli iránya változik.
3. Felosztja a rajzot az y tengely mentén egyenlő méretű sor szeletekre.
4. Kigyűjti az így kapott értékeket, amely számok azt jelölik, hogy hány alkalommal fordul elő rajzolt pont a vízszintes szegmensekben. Ezeket a számokat egy tömbben tárolja.



6. ábra, (VerticalMap működése)
forrás: GitHub - gilbertdyer2/UnityDrawingRecognition

Ebben a példában a tömb elemei sorrendben: {1,1,1,1,2,2,2,2,2}.

Az egyezési pontszámot ugyanúgy számolja, mint a HorizontalMap.

4. Rajzoló algoritmus

4.1 Működése

Az algoritmus feladata az, hogy a vászonként szolgáló teljes játéktérre interaktívan rajzolhassunk. A felhasználó a bal egérgomb lenyomva tartásával, és párhuzamosan az egér mozgatásával, bármit rajzolhat a teljes képernyőre. A program nem szabja meg, hogy milyen formát rajzolhat, bármilyen inputot készíthet a felhasználó, olyat is akár, ami nincsen benne a minta adatbázisban. A bal egérgomb felengedésekor regisztráljuk a képre rajzolt pontokat, és mindenképpen az egyik alakzattal azonosítja a program, azzal, amelyikre az egyezési pontszám alapján legjobban illeszkedik. Mivel az szándékosan nincsen lekezelve, hogy mit tegyen a program, hogyha egyik minta alakzatra sem hasonlít az input, ezért véletlenszerű inputokkal is kihatással lehet lenni a játékra.

A működési elv miatt, mivel bal egérgomb lenyomva tartásával húzhatjuk a vonalat, és felengedéskor regisztrálja az alakzatot, csak egy vonallal felrajzolható minta alakzatok vehetők fel a template-ek közé.

A képre húzott vonalak *Vector2* (tehát x, és y koordinátával rendelkező) vektorok, ezért ezeket a bal egérgomb felengedését követően egy *List<Vector2>* listában tároljuk. A Bitmap osztály konstruktorában ezeket a pontokat adjuk át, a játszható terület koordinátáit tartalmazó lista mellett. Ezután példányosítjuk ezen adatok alapján az inputot egy Bitmap példányként. A Bitmap példány konvertálja a kapott adatokat a négy fajta Map-re, illetve kód alapján csupán háromra, GridMap-re, CircleMap-re és FlatMap-re. Ez a FlatMap elnevezés viszont csupán a VerticalMap és HorizontalMap együttese, a kevesebb számítási kapacitás eléréséért, és azért, hogy ezt a kettő egymástól alig különböző map-et egy helyen kezeljük.

4.2 Detektáció és akció

Hogy jelzi a játék ha felismert egy alakzatot? Jelenleg két féle módon tudhatjuk, hogy milyen alakzatot detektált a programunk.

1. **Végrehajtja az alakzatot társított akciót**: Amennyiben az inputot egy olyan alakzatként azonosította mely szerepel a minták között és van hozzá társítva funkció, a bal egérgomb felengedésekor a program végre is hajtja az ahhoz az alakzathoz rendelt folyamatot.
2. **Debug konzolra kiír**: Ha a játékot nem a build-elt .exe fájlban futtatjuk, hanem a Unity programban projektként, akkor a képernyő alján található debug konzolra kiírja felismert alakzat nevét.

4.3 Eredmény kezelése

Mi alapján állítjuk be, hogy miket ismerhessen fel a program? A dolgozat jelenlegi állapotában a DrawingRecognition script-jében lehet egyelőre csak kóddal beállítani adathalmazt. Ezekből az előre beállított mintákból lehet egyszerre több is betöltve a programba, és egy környezeti változótól függővé lehet tenni is azt, hogy melyik legyen aktív. Ha például ez egyik könyvtárban a kör és négyzet van, a másikban pedig csillag és kereszt, akkor ezeket a program futásának valamelyik hatására, például pályaváltásnál cserélhetjük, és onanntól az új alakzatok lesznek felismerhetők.

Ilyen könyvtárakat inicializálhatunk, és tetszőleges számú karakter mintát is tehetünk beléjük. Ezeket el lehet látni névvel is, hogy könnyebben azonosíthatóak legyenek. Miután inicializáltuk a karakter könyvtárakat, be kell állítanunk a Bitmap-ek súlyát, amit a setWeights() függvényben kell megtennünk 4db double érték megadásával (0.0-1.0 tartományban) melyek rendre a *circleMapWeight*, *gridMapWeight*, *horizontalMapWeight*, *verticalMapWeight.* Ezen értékek beállítása azt befolyásolja, hogy melyik Bitmap egyezési pontszámát mennyire vesszük figyelembe az input és a könyvtár elemei közti összehasonlítás során. Ezután végül be kell állítanunk a *precision* értékét is, mely a hibahatár növelését eredményezi. Ennek alacsonyra állítása kevésbé pontos, viszont gyorsan rajzolt karakterek azonosítására hasznos, míg a magasabb érték nagyon pontos egyezést fog elvárni az inputtó. Ez nem jelenti azt, hogy minden inputra pontosan azt a mintát fogja azonosítani a program amire tényleg hasonlít, ugyanúgy a leginkább hasonlót fogja választani, viszont, ha a felhasználó csúnyán rajzol egy kört, az a magas *precision* érték miatt azonosítható lehet téglalappal, ha nem tökéletes kör alakú az input.

A karakter könyvtár beállítása után, már működőképes is a felismerő algoritmus. Azonban a felismert alakzatok ezen a ponton még nincsenek akciókhoz kötve. Ahhoz, hogy egy lerajzolt alakzat működésében befolyásolja a játékmenetet a dolgozatban létrehoztam a játékosnak egy lövés akciót. Bármilyen akciót ki lehet találni és megvalósítani, ezeknek pedig bármilyen bemeneti követelményt be lehet állítani a végrehajtáshoz. Egy gomb lenyomást, egér kattintást stb. Jelen esetben egy bizonyos alakzat felismeréséhez is lehet társítani bármilyen folyamatot, pályabetöltést, ugrást, lövést, akármit.

A dolgozatban alapvetően a következő alakzatokat teszteltem: kör, téglalap, háromszög, csillag, és kereszt. Mindegyik alakzatot felismeri, és a legtöbb esetben jól azonosítja futásidőben a program. Amennyiben csak kattintunk egyet, és nem rajzolunk semmit, úgy egy

üres találatot ad nekünk vissza. A tesztelt alakzatokhoz tartozó egy-egy példa, amit futás közben teszteltem és felismerte a program az adott alakzatként:


7. ábra


8. ábra


9. ábra


10. ábra

7-10. ábra: *rajz példák sorrendben: háromszög, kereszt, kör, csillag.*
A vonal vörösből feketébe való sötétedése indikálja a vonalak húzásának sorrendjét: minél sötétebb, annál régebben lett húzva

Az alábbi alakzathoz valósítottam meg társított funkciót:

- **Kör**: Ha a rajzolt input egy körrel egyezik meg, akkor az egér irányába példányosítunk egy tűzgolyó objektumot, amely a játékostól indul, az egérmutató rajz befejezésekori pozíciójához érkezik, és ott lebegve egy rövid ideig sebzést oszt ki az ellenfél objektumoknak útja és pozíciója során is.
- **Csillag**: Ha a rajzolt input egy csillaggal egyezik meg, akkor a játék ellenőrzi, hogy betelt-e már a haladást jelző sáv, amit „gem”-ek felvételével lehet feltölteni. Amennyiben fel van töltve, úgy a játék betölti a következő pályát, azonban, ha nincs feltöltve, és úgy próbálunk egy csillagot rajzolni, a debug konzolra kiírja a program, hogy „You do not have enough point to progress!” azaz „Nincsen elég pontod még a tovább haladáshoz!”

4.4 Alapvető felismerési hibák

A program természetesen nem tökéletes, akadhatnak benne összetűzések még, főleg, ha két karakter nagyon hasonló. A pontosságot nagyon nehéz úgy beállítani, hogy határozottan ismerjen fel minden alakzatot, viszont ne legyen extra érzékeny a tökéletlenségekre.

A körnél és a téglalapnál a leginkább megfigyelhető ez a hiba. Ha magasra állítjuk a pontosságot, akkor a tökéletes kört nagyon jól felismeri körnek, azonban, ha a felhasználó, nem hibátlan kört rajzol, kerül bele egy kicsi szögletesség, akkor máris téglalapként fogja azonosítani a program. Ha a pontosságot lejjebb vinnénk, akkor könnyebben körnek ismer fel nem szabályos köröket is, azonban a téglalapnak szánt alakzatokat is ide kategorizálhatja, ha az

nem teljesen szögletes. Így a pontossági értéket ennél a modellnél 3-as értéken hagytam, ezzel minimalizálva az ezzel kapcsolatosan előforduló pontatlanságokat, azonban határozottan fellelhető ez a futás során. Természetesen erre megoldás lehet jelentősen eltérő alakzatokat használni, de amint kettő közt lehet átfedés, ez a hiba újra előjöhet.

4.5 Minták bővíthetősége

A tárolt mintáink mellett van lehetőségünk ezeknek listáját bővíteni. Azon túl, hogy vannak a Drawing recognition scriptbe beégetett egyéb alakzat lehetőségünk, amiket bármikor belerakhatunk az értelmezési könyvtárakba, új felvitelére is lehetőség van, azonban egyelőre csak a kódba való beégetéssel, ehhez nincsen a felhasználó számára lehetőség. A program bővítésével azonban ez megoldható lenne.

Ahhoz tehát, hogy fejlesztői oldalról bővítsük az ismert alakzatokat, szükség van a futás során egy gyorsgombra, amely lefuttatja a 'PrintCurCharacter()' metódust. Ennek a metódusnak a feladata az, hogy az imént rajzolt inputnak kiírja a *List<Vector2>* és *List<Vector3>*-as típusú listákban eltárolt rajzpontjait. Ezeket kimásolva és az alakzatfelismerő szkriptünkben a használt Initializer-be beillesztve már működik is. Pontosíthatjuk a várható eredményeinket, ha egy alakzatról nem csupán egyetlen példát rajzolunk és illesztünk be, hanem több mintát is mutatunk a programnak, és ezek vektorjait mindet ugyanahhoz a mintához társítjuk.

```
private void Initialize_Szakdolgozat(CharacterLibrary charLib)
{
    charLib.ClearLibrary();
    List<Vector2> square = new List<Vector2> { new Vector2(448f, 435f), new Vector2(448f, 433f), new Vector2(448f,
    List<Vector3> squareW = new List<Vector3> { new Vector3(-0.5925926f, 3.055556f, 0f), new Vector3(-0.5925926f,
    charLib.AddCharacter(new Character(new Bitmap(square, squareW, 5), "square"));
```

11. ábra
példa egy egyedi minta leírására

A 7.ábrán látható kód hosszasan folytatódik, végig a látható *new Vector(...)* feliratokkal, ugyanis rengeteg vektor található egy ilyen minta karakterben. Ezeknek a kiírásán nem kell gondolkozni, a két listát tartalmazó sorokat az ábrán látható formátumban kiadja a *„PrintCurCharacter()”* függvény az input rajzolása után. Ezeket csak be kell illeszteni a „*char.Lib.CleaerLibrary()”* és „*char.Lib.AddCharacter()”* sorok közé.

Könnyen belátható, hogy ezt a folyamatot a program jövőbeli bővítése során lehetne automatizálni, felhasználói felületekkel is, vagy gombnyomásra is felvihetőek lehetnek karakterek. Ha például lenne egy „új karakter felvitele gombunk, ami kikapcsolja a játék vezérlését, megnyit egy grafikusan könyvnek kinéző új oldalt, megkéri a játékost, hogy legalább háromszor rajzolja meg ugyanazon karaktert, nevezze el az új alakzatot, majd az oda

rajzolt karakterre meghívja a *PrintCurCharacter()* metódust, és az azáltal rajzolt értéket automatikusan beilleszti a kívánt *Initializer*-be, akkor már működőképes is lehet ez az elképzelés. A „könyv” ablak bezárásával pedig folytatható lenne a játékprogram, ezzel pedig futási időben megvalósíthatóvá válik a felhasználó számára is. Ez a funkció elkészíthető a program további bővítése során, azonban nem került bele a dolgozat elkészítése alatt.

Összefoglalás

A szakdolgozatom elkészítése során fontos tapasztalatokra tettem szert a képfeldolgozás és Unity engine-ben való játékfejlesztés során is. A munka során több olyan dolgot is tanultam, amely alkalmazásával ma már hatékonyabban készíteném el ezt a projektet. A szakdolgozatom befejezésével a jövőben folytatni fogom a programot, és célom, hogy egyszer egy piacra bocsájtható indie játék legyen belőle, hogy a dolgozat végső célját is teljesíthessem ezzel, miszerint hatással szeretnék lenni a játékiparra, egy új megközelítés bevezetésével.

A dolgozat elkészítésében nagy szerepet játszott a nyílt forráskódú gilbertdyer2/UnityDrawingRecognition projekt beépítése is, melyet eleinte értelmeznem kellett, majd meg kellett tanulnom használni, melyben hatalmas segítség volt a projekt jól dokumentáltsága. A feladatom a játékprogram saját ízlésem alapján való elkészítésén kívül a projekt átfogó megismerése volt, és annak hatékonyan és hasznosan a programomba való integrálása volt.

A kész dolgozatban elértem, hogy elkészült egy több platformon is futtatható (macOs és Windows) demo játékprogram, mely megmutatja a játékfejlesztés lehetőségeit az alakzatdetektáció felhasználásával. A játékban a felhasználó irányíthatja a karakterét és az egérrel interaktálva alakzatokat rajzolhat, melyek működéséhez funkciókat rendeltem. A játékost kezdetleges ellenfelek kergetik, melyek elől el kell meneküljön és „gem” -eket kell gyűjtögessen, hogy tovább haladhasson a pályákon. Az elkészült játékprogram bővíthető a jövőben új alakzatokkal, okosabb ellenfél intelligenciával, új pályákkal, grafikus felületekkel, és funkciókkal. És természetesen a jövőben saját grafikus dizájnokkal.

Összességében a szakdolgozat nem csak egy feladat volt számomra, de elindított az indie játékfejlesztés útján is, és a képfeldolgozást is jobban megismertem. A program fejlesztése során én is sokat fejlődtem, rendszerszinten kezdtem gondolkozni, erősödött az objektumorientált gondolkodásom is, megtanultam, hogy hogyan lehet játékbéli új funkciókat programokkal megvalósítani, és véleményem szerint egy egyedi formabontó új lehetőséget vihetek tovább a jövőbe ezzel a dolgozattal.

Irodalomjegyzék

1. Szakmai definíciók

cross-platform

Több rendszeren működő program, nincs limitálva a program működése egy operációs rendszerre, vagy platformra.

indie

Egyedülálló vagy független, az „indepentent” szóból ered, az iparban az olyan feljesztőkre használják, akik mögött nem áll kiadó, vagy esetleg nagyobb stúdió. Rendszerint kis létszámmal rendelkező csapat jellemzi.

asset

Játékot alkotó építő elem, lehet ojbektum, sablon prefab, a kamera ami követi a játkost, vagy egy objektum textúráját adó sprite

sky-box

Rendszerint a játék felső határát jelölő láthatalan ütközési pont, mely biztosítja, hogy a játékos ne léphessen nem kidolgozott területre.

scene

A játék egy jelenetét adó vászon. Erre dolgozhatunk a Unity engine-ben, és drag and drop módon alkothatjuk a játékteret.

UI

User interface, általában grafikus felületet jelent, de az már GUI jelölést szokott jelenteni. UI elemnek számít az életerősáv, a haladási sáv és az a vonal is, amit a dolgozatomban rajzolás során húz a kurzor.

sprite

Egy adott játébeli objektum grafikáját megadó kép, ábrázolás animáció esetén egy mozgás több sprite egybefűzése alkot meg képkockánként.

prefab

Prefabricated szóból ered, ami előre elkészítettet jelent. Olyan objektumok a fájlrendszerben melyek teljeskörűen leírják egy objetkum tulajdonságai, működését és textúráját is. Egyszer manuálisan elkészítve, utána mintaként használhatjuk játékmenetet befolyásoló objektumok pélányosításához.

Köszönetnyilvánítás

Ezúton szeretném kifejezni hálámat és köszönetemet konzulensemnek: Dr. Bognár Péternek, aki támogató hozzáállásával segítette a munkámat, és azt, hogy a dolgozatomat elkészíthessem.

Nyilatkozat

Alulírott, Puskás Patrik, programtervező informatikus szakos hallgató, kijelentem, hogy a szakdolgozatban ismertetettek saját munkám eredményei, és minden felhasznált, nem saját munkából származó eredmény esetén hivatkozással jelöltem annak forrását.

Dátum (Szeged, év, hó, nap) _____

aláírás