Eötvös Loránd Tudományegyetem

Informatikai Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Logo fejlesztőkörnyezet óvodás gyerekeknek

Dévai Gergely
Tanársegéd

Szecsődi Imre
Programtervező Informatikus BSc

Budapest, 2017

Tartalomjegyzék

1. Bevezetés

Ahogy egyre jobban belemegyünk a 21. századba, egyre több számítógép és okos eszköz vesz körbe minket. Okos telefon, tv, óra és a jelenleg nagy popularitásnak örvendő Internet of Things eszközök. Az informatika tudománya egyre jobban kezd tágulni, egyre több egymástól merőben eltérő eszköz áll rendelkezésre az új és tapasztalt programozóknak. Ezért nagyon fontos, hogy mindig legyen elegendő szakember, aki tudja programozni ezeket.

Ahogy haladunk a korral, a fiatalok egyre korábban találkoznak a számítógéppel, és veszik igénybe azt, hogy megalkossák az első programjukat. Mikor még én voltam általános iskolás, informatikát csak a hetedik és nyolcadik évfolyamos diákoknak tanítottak. Ezek az órák inkább csak egy kis ízelítők voltak, hogy mire képesek a számítógépek. Majd idővel egyre lejjebb került a határ, mikor kezdik el valóban használni a számítógépet a diákok.

Ezek az órák nagyon fontosak, mivel itt tapasztalják meg először, hogy milyen élmény programozni, alkotni egy programot. A fejlesztés örömét nem mindenki osztozza, de ez érthető, nem mindenkiből lesz informatikus. A szakdolgozatom ezekhez az emberekhez szól. Rá szeretnék mutatni arra, hogy egy kis kódolást minden embernek érdemes tudnia, mivel ez egy egész más gondolkodásmódot eredményez. A problémamegoldó-képességének növelése egy fontos dolog, egy kis programozás ismeret segít megérteni és értelmezni egy összetett problémát. Ezután már fel tudjuk bontani a feladatot kisebb feladatokra. Több kisebb problémára egyszerűbb megoldást találni, mint a teljes egészet egyben megoldani.

A programozást nem lehet elég korán elkezdeni, ezért már sok nyelv és fejlesztőkörnyezet jött létre, hogy segítse az ifjú fejlesztő, alkotó embereket. Magyarországon az egyik legelterjedtebb program a Comenius Logo. Nemzetközi szinten még ismeretes és széles körben használt program az MIT által fejlesztett Scratch.

Ez a két program között a cél ugyanaz, megközelítés különbözik, hogy szerettessük meg a programozást a fiatal gyerekekkel. A Comenius Logo, ezután már csak Logo-nak fogom nevezni, egy egyszerű rajzoló program egyszerű utasításkészlettel. A felhasználó parancsok segítségével rajzol egy lapra, beírja a parancsot és az rögtön látható eredményt produkál a rajzlapon. A Scratch egy hasonló elvet követ, ám itt egy vásznon különböző szereplőket mozgatunk, és azok lépnek interakcióba egymással. Ám míg a Logo gépelt parancsokra épül, ezt a Scratch kiváltja egy grafikus programozási felülettel. Itt egér segítségével tudjuk a különböző színkódolt parancsokat úgy rendezni, hogy azok megmozgassák a szereplőket. Ezek a parancsok összetettsége már kicsit magasabb, ezért nem lehet a legfiatalabb korosztálynak még ajánlani.

Látván ezt a két programot, jött az ötlet, miért ne lehetne kombinálni a két program előnyét, legyen egyszerű, mint a Logo, ám legyen vizuális, mint a Scratch. Ezzel egy egész új korosztálynak nyitni a programozással, az ovisoknak.

2. Felhasználói dokumentáció

2.1. Bemutatás

A program egy egyszerűsített fejlesztőkörnyezet, ami az óvodás gyerekeknek lett kialakítva, hogy megismerkedjenek a Logo nyelvvel. Fontosabb nyelvi elemek a Comenius Logo programban is megtalálhatóak. Továbbá a Python Turtle csomagjában található parancsok közül is van pár elkészítve. A program fő célja, hogy a gyerekek megbarátkozzanak a programozással. Ennek érdekében csak a legegyszerűbb, ám fontos alapelveket mutatom be.

A Logo rész csak a rajzolásért felel. A másik alapköve a programnak a vizuális programozás. Tanítóprogramok között akadnak nagyon jó példák erre. Az egyik széleskörűen elterjedt program a Scratch. A Scratch egyszerű alaputasítás készletet használ, kódját pedig vizuális blokkokból lehet összerakni. Ezek az utasítások szövegesen vannak leírva a felhasználónak. Hasonló parancsok kategóriánként vannak csoportosítva. Különböző kategóriák különböző színűek, így ránézésre is látható a program működésének alapötlete.

A megvalósított programom is ezt az alapelvet követi, a parancsok blokkokból összeállíthatóak. Számomra a cél az volt, hogy óvodás gyerekeknek is élvezhető és játékos legyen a program. Mivel még nem tudnak olvasni ilyen idősen a gyerekek, ezért a szöveges parancs blokkok nem voltak meglelőek a számomra kitűzött cél eléréséhez. Ezért lettek a különböző függvények egy-egy képpel jelölve. A másik kihívást a programmal való interakció jelentette. Ekkor jön képbe a Logo nyelv, ahol egy rajzolótábla áll rendelkezésünkre, és ezzel tud visszajelezni a program a felhasználó felé.

2.2. Telepítési útmutató

2.2.1. Rendszerkövetelmény

	Minimum	Ajánlott
Rendszer	Windows 7, 8, 10, Linux, MacOS (OSX)	
Processzor	1 x 900 MHz	1 x 3.50 GHz
Tárhely	30MB	
Felbontás	1360 x 768	1920 x 1080
Bemenet	Egér vagy touchpad, billentyűzet	

2.2.2. Általános futtatási információk

A Python egy scriptnyelv azért ajánlott a . Ezzel bizto-sítjuk az összes függőséget a futáshoz, továbbá így lehetőségünk van a legtöbb rendszeren futtatni a programot. A pontos leírása ennek a fejlesztői dokumentációban található meg.

Másik lehetőség, hogy az előre lefordított programot használjuk. Ennek előnye az, hogy egyszerűen tudjuk futtatni. Ellenben itt biztosítani kell a program futásához szükséges csomagokat.

- Windows rendszer alatt nincs külön dolgunk, mivel ez az egyetlen rendszer, ami tartalmazza az összes szükséges fájlt.
- Linux Mint 17.3 alatt fordítottam a kódomat, és az alatt teszteltem is. Ott minden rendben volt. Ám mivel Linuxok között hatalmas különbség lehet, ezért ha hiba lép fel ajánlott rákeresni a hibaüzenetre interneten. Feltételezhető hibák lehetnek, hogy nincs SDL telepítve, és SDL-hez tartozó különböző könyvtárak. Másik fontos követelmény, ami hiányozhat, az a képfeldolgozó modul egyik szükséges követelménye a „Libpng”.
- MacOS alatt sok dologra szükség van. A felhasznált irodalomban megtalálható a következő helyen egy link, ahol a szükséges könyvtárakat sorolják fel[1] Lefordított programkód működése nem garantált. Ajánlott forráskódból futtatni. Részletek a fejlesztési útmutatóban, és a projekt GitHub[2] és Bitbucket[3] oldalán.
- Futtatás Windows és Linux rendszereken dupla kattintással működik probléma nélkül.

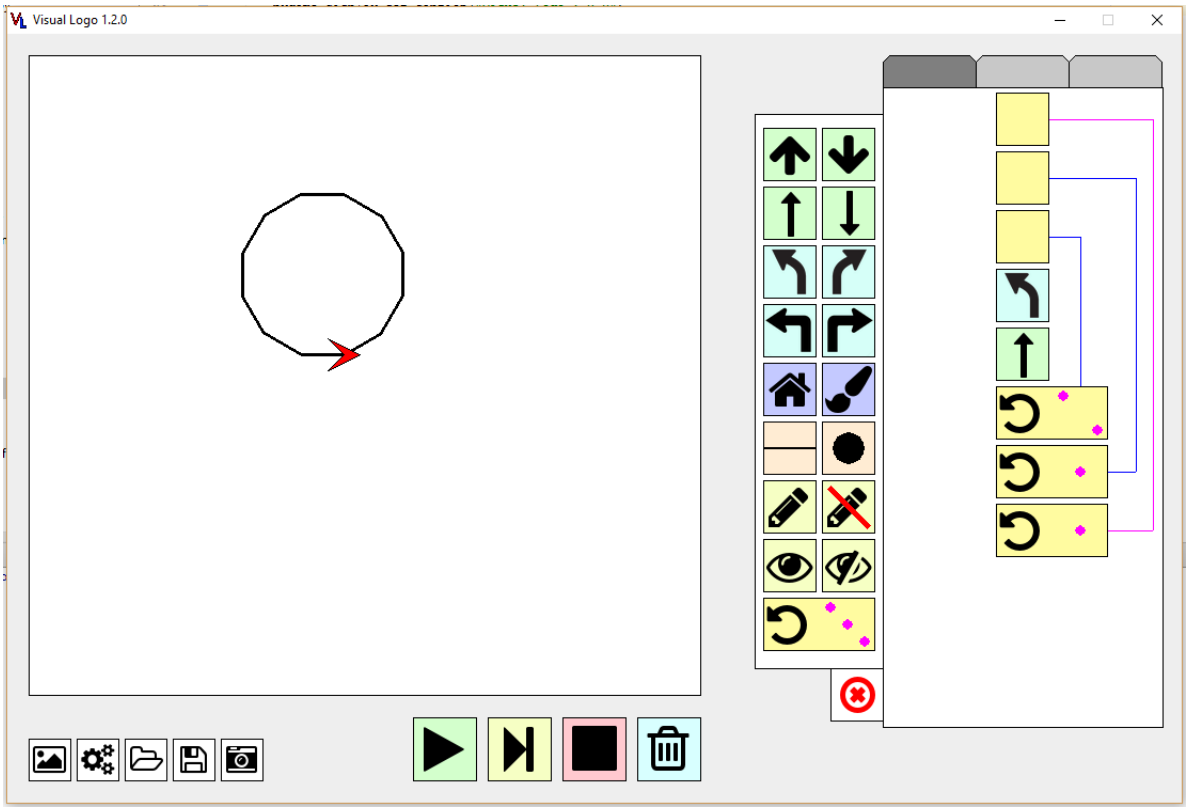
Mivel a program Python nyelven íródott, és minden modul, amit használtam a standard PyPI csomagok közül van, ezért támogatott a legtöbb modern rendszer. Mindemellett nem csak a X86 és X64 alapú rendszerek támogatottak, hanem az ARM alapú Raspberry Pi mini pc is képes futtatni. Külön van jelezve a letöltések között melyik lefordított futtatható állományt kell letölteni. Ajánlott a Raspberry Pi 2-es vagy a 3-mas verzió használata. Raspberry Pi-n futtatott rendszer ajánlott, hogy Raspbian vagy Ubuntu Mate legyen.

A program első indításakor a futáshoz szükséges fájlokat generál ki, ezért megtörtén-het, hogy kisebb várakozás után fog csak elindulni. Ezért fontos, hogy olyan helyen legyen a program, ahol van jogosultága létre hozni fájlokat a felhasználónak, aki futtatni fogja majd a programot. Ellenőrizzük le, hogy megtalálható-e a program mellett a Resources mappa. Ha nem akkor másoljuk ki a forráskódból. Továbbá ha szükséges, akkor hozzuk létre a Screenshots és UserData mappát. Ha a program nem indul el és vannak fájlok a UserData mappában. Töröljük tartalmát és indítsuk el a programot ismét.

A programból kiléphetünk kétféle módon, a billentyűzeten használjuk az Esc gombot, vagy az X-et az ablak sarkában.

2.3. Használati útmutató

2.3.1. Fő ablak



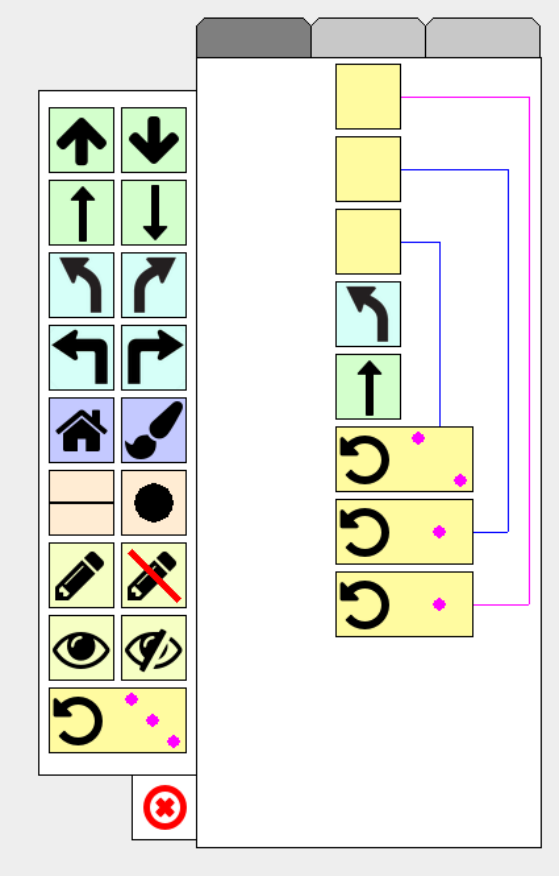
1. ábra, A program fő ablaka

Amit a felső (1.) ábrán látunk az a program fő ablaka. Négy fő részre bontható fel.

1) A jobb oldalon található rajzlapra, ami az általunk összerakott program végeredményét mutatja. A piros nyíl a tollunk, ami segítségével rajzolunk.
2) Alatta bal felől látunk öt gombot, ezek azok a funkciók, ami nem befolyásolják a készített program futását, hanem a fő programot segítik. Balról jobbra rendre:
   a) Háttérszín csere. Erre kattintva meg tudjuk változtatni a panelek között kitöltő színt
   b) Beállításokat mutató gomb. Alaphelyzetben csak az első kettő gomb látszik, mivel a maradék három gomb funkciója memória írás és olvasással kapcsolatos műveleteket használ. Ezért a menüt csak jobb egér gombbal lehet megnyitni vagy az alternatív billentyű kombinációval. Bővebb információ a 2.3.4 részben található.
   c) Programkód betöltése. A már előzőleg eltárolt program visszatöltéséért felelős ablakot hozza elő ez a gomb. Bővebb információ a 2.3.3 részben található.
   d) Programkód mentése. Ezzel menthetjük el a jelenleg aktív programkódot, amit készítettünk. A gomb felhozza a program mentéséért felelős ablakot. Bővebb információ a 2.3.3 részben található.
   e) Képernyőkép mentésre. A jelenlegi rajzot lementi egy fájlba a program mellett lévő Screenshots mappába.

3) Mellette jobb oldalt találhatóak a futáshoz kapcsolódó gombok. Balról jobbra rendre:
   a) Zöld futtatás gomb. Az általunk készített kódot tudjuk futtatni. Bal egér gombbal lassan, jobb egérgombbal vagy alternatív billentyű kombinációval gyorsan. Bővebb információ a 2.3.4 részben található.
   b) Sárga léptető gomb. Ez szolgál arra, hogy lépésről lépésre futtassuk a programot. Jól jön, ha meg akarjuk keresni egy hiba forrását, vagy figyelni a program működését. A gomb nem fog működni rögtön az összeállított kódon. Ez azért van, mert mikor a futtatásgombot nyomjuk meg háttérfolyamatoknak kell lefutni, mielőtt egyáltalán a kód futhat. Ha szeretnénk ezzel kezdeni a program futását, akkor kezdjük el futtatni, majd rögtön nyomjuk meg ezt a gombot, és már tudjuk is tovább léptetni ezzel. Léptető helyzetből bármikor vissza tudunk lépni futó állapotba a futtató gombra kattintva.
   c) Piros megállító gomb. Megállítja a program futását, újraindítás szükséges, nem lehet innen léptetni. Nem törlődik az eredmény.
   d) Kék kuka gomb. Míg a megállító gomb nem törli az általunk kirajzolt ábrát, addig ez igen és alaphelyzetre állítja a rajzlapot.
4) Sidepanel, forráskód mező. Itt tudunk létrehozni kódot, és azt szerkeszteni. Mivel a panel terjedelmes azért a következő részben (2.3.2.) fogom taglalni.

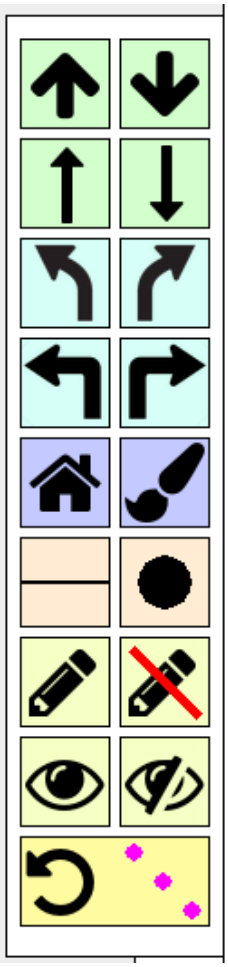
2.3.2. Sidepanel, forráskód mező



2. ábra, Sidepanel, forráskód mező

A mellettünk lévő (2.) ábrán látható eszköztár a forráskód kezelő. Itt készíthetjük el a kódunkat. Az ablak négy főbb részre osztható fel.

1) A felhasználható parancsokat jobb oldalon a kis részben találhatjuk.
2) Jobbról mellette a forráskód mező maga. Itt található az általunk készített kód. Az itt lévő blokkok automatikusan rendezve vannak, elég csak a megfelelő régióba tenni a parancsot, és azt a helyére rakja a program.
3) A két eszköztár között egy piros X áll rendelkezésünkre ha szeretnénk törölni a jelenleg összeállított kódunkat.
4) A fülek a jelenlegi forráskód felett. Ezek a fülek külön-külön tárolnak el egy aktív kódot. Egymástól függetlenül működnek, ezért ha több felhasználó használja a programot, akkor mindegyik tud tartani gyors elérésen egy kódot. Az aktív fület a sötétebb festés jelzi.



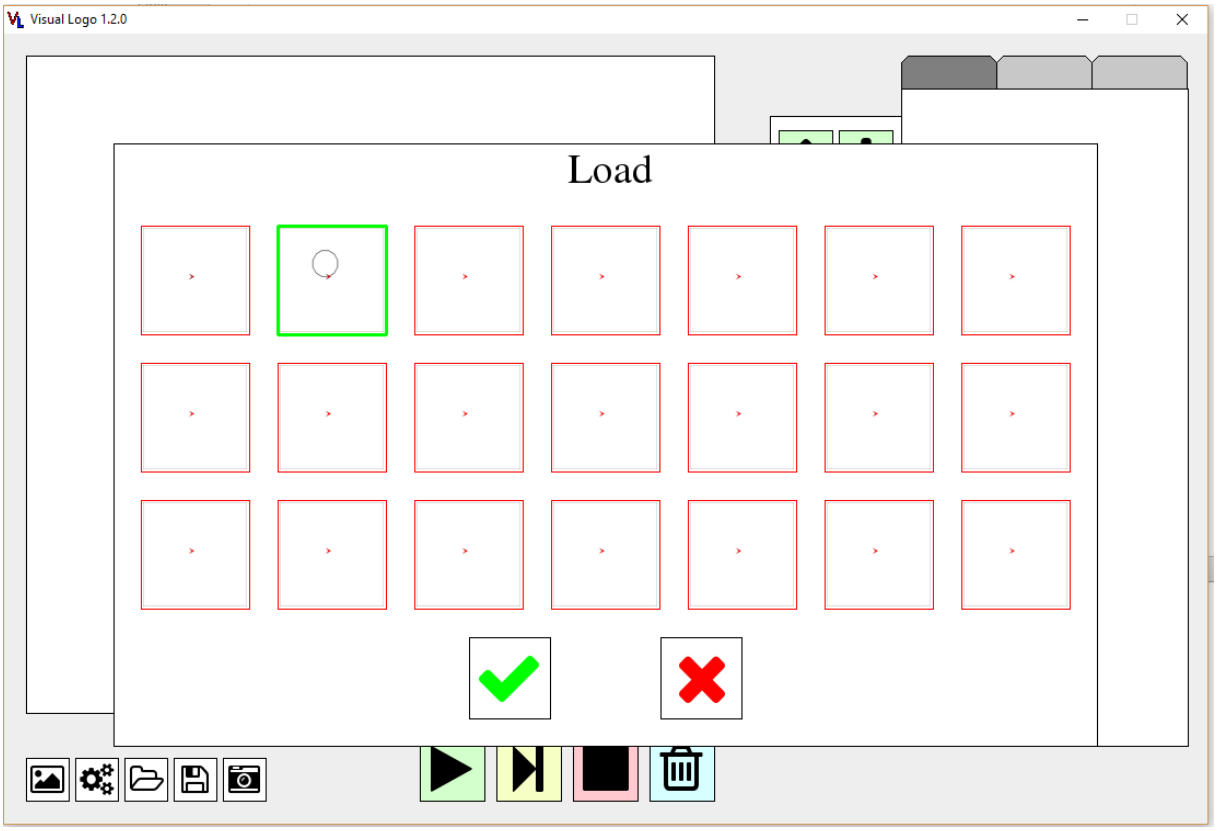
3. ábra, Parancsok

Jobb oldalt látható (3.) ábra a felhasználható parancsok összessége. Fentről lefelé a funkciók leírása:

1. sorban találhatóak az előre és hátra mozgás parancsok, ezek egy egységet mennek a mutatott irányba.
2. sorban találhatóak ezek párjai, ezek kettő egységet mennek a megfelelő irányba
3. sorban vannak a kis fordulások, ezek csak 30° fokban fordulnak el
4. sorban vannak a fent lévők párjai, ezek 90° fordulnak a mutatott irányba
5. sorban van két különböző funkciót betöltő jel. A ház a nyilat, akárhol is van, visszaviszi a kezdőpontba, ha lent van a toll, akkor húz maga után vonalat is. Mellette található az ecset, feladata, hogy akárhol is áll a teknős az alatta lévő területet kifesti a jelenlegi színre. Ha egy vonalon áll épp, akkor a vonalat fogja átszínezni, ha pedig egy zárt területen van, akkor azt fogja kifesteni a következő különböző szín határáig
6. sorban a festéssel kapcsolat parancsokat látjuk. A baloldalon lévő vékony vonal a tollvastagság. Ezt tudjuk állítani jobb egér gombbal, vagy az alternatív billentyű vezérléssel. Jobb oldalon meg a jelenlegi szín van. Ezt is lehet változtatni, mint a vonal vastagságot. További információ a 2.3.4 részen található.
7. sorban található tollak jelzik, hogy húzzon-e vonalat a nyíl vagy ne.
8. sorban vannak az opciók arra, hogy szeretnénk-e látni nyilat futás közben vagy nem.
8. sor egy parancs, ez pedig a ciklus. Ez egy két részből álló parancs. Itt csak egy látható, de mikor bekerül a rajzmezőre a párja is létrejön, ami bezárja majd, és a ciklus határát jelzi. A jobb oldalon látható pontok jelzik, hányszor fut le majd. Ez változtatható a jobb egér gombbal, vagy az alternatív billentyű vezérléssel. További információ a 2.3.4 részen található.

Mint a (2.) ábrán látható, a kód részen található több parancs. Az egyik érdekesebb a ciklus. Mivel az két darabból áll mindig tudnunk kell, hogy melyik a hozzátartozó pár. Ezért vannak egy színes vonallal összekötve az egymással kapcsolatban lévő blokkok. Vegyük észre, hogy ha több ciklus van egymásba ágyazva, maximum kettő mélységig, akkor különböző vonalat húz neki a program, hogy egyértelműen látszódjon melyik hova tartozik. Ám vigyázzunk, mert a program csak azokat a ciklusokat érzékeli belsőnek, aminek nyitó és csukó tagja egyértelműen a másik ciklusban van. Szóval, ha a külső előbb záródik, mint a belső, akkor a vonalak egymást fedni fogják, mivel nincs benne egyértelműen az egyik a másikban. A másik meg hely hiányában fordul elő, ha több mint 2 mélyen vannak ciklusok egymásba ágyazva, akkor az őket összekötő vonalak egymást fogják felülírni, és a legutoljára hozzáadott ciklus színe lesz kirajzolva a fedő vonalakon.

2.3.3. Betöltő és mentő ablak



4. ábra, Betöltő és mentő ablak

Ez az ablak felel a betöltésért és forráskód mentéséért. Míg ez az ablak aktív addig a háttérben lévő elemekre nem lehet kattintani. Az ablakon látható a huszonegy darab mentési hely, a módtól függően Load vagy a Save felirat tükrözve, hogy épp mit szeretnénk csinálni. Mikor megnyitjuk ezt az ablakot, akkor nincs kijelölve egy mentési mező sem. Rá kell kattintani egyre, és akkor megjelenik a zöld keret körülötte. Miután kijelöltük a zöld pipára kattintva végezhetjük el a mentést vagy betöltést. Ha nem jelöltünk ki semmit, nem fog történni semmi, mivel nem volt egyértelmű az utasítás. Támogatott még a duplaklikk a megadott mezőkön és akkor automatikusan megtörténik a feladat végrehajtása. Mint látható mindegyik mentési hely tartalmaz egy kis képet. Ezek a képek megmutatják, hogy mit fog kirajzolni a kód, amit már előzőleg elmentettünk. Betöltés esetén a jelenleg aktív fülön lévő kód törlődni fog, és a betöltött kód lesz a helyére téve. Mentés esetén, be fog záródni az ablak, és a program le fog futni gyorsan. Mivel a programban csak számláló ciklust találunk, ezért mindig meg fog állni. Mikor a program megáll, egy képernyőkép készül a jelenlegi állapotról és az kerül mentésre a memóriába a jelenlegi parancslistával együtt. Mentés közbeni futást nem lehet megállítani.

2.3.4. Programmal való interakció

A program alapbeállításai futtatás indításakor: a rajzlap üres, a forráskód mező üres. Az első fül van kijelölve, a memóriakezelő beállítások el vannak rejtve, szürke a kitöltő szín. A rajzlapon lévő nyíl jobb felé mutat, tollvastagság a legvékonyabb, a rajzolási szín fekete, és a toll lent van, azaz ha mozogni kezd, rögtön húzza a vonalat maga után.

Mint már az előző részekben jelezve volt, több féle módon lehet vezérelni a programot. Elsődleges módja csak az egér használata. Legtöbb helyen elég csak a bal egér gombot használni. Gombokra kattintani, váltani füleket. A program kódot drag 'n drop módon kell összeállítani. Megfogjuk a parancsokat és behúzzuk a forráskód mezőbe. Mint már említettem itt automatikusan fog helyet találni magának, és rendeződnek a parancsok. Ha el szeretnénk távolítani parancsot, akkor megfogjuk és kihúzzuk a forráskód mezőből a parancsot, figyeljünk arra, hogy az egér mutatója is legyen kívül a mezőn. Ha ciklust távolítunk el, akkor a hozzá tartozó pár is törlődni fog. Ha a ciklust a számlálós résznél fogjuk, meg akkor tudjuk ki és be rakni a kódba, ám ha a párját fogjuk meg, és azt húzzuk ki a mezőből automatikus törlésre kerül a ciklus is meg a fogott párja is.

Ha valami speciális parancs van, mint színcsere a tollon, vagy épp a vastagság beállításakor, akkor a jobb egér gombot használjuk. Ebbe a csoportba tartozik még a ciklus futásszámának állítása, és a lejátszás gomb gyors módjának elindítása is, valamint a beállítások előhozása, ami rejti mentés funkciókat. Ha ezt használjuk, a kis részen ahol van az összes parancs akkor, ha következőnek akarunk egy megváltozott parancsot berakni, az megtartja a beállításokat. Másik lehetőségünk, hogy a már jelenleg forráskód mezőben szereplő parancsokat változtatjuk meg így, akkor az eredeti parancs nem fog megváltozni, hanem csak az a másolata, amit megváltoztattunk. Ez által nem kell újra berakni egy ilyen változtatható parancsot, hanem elég helyben változtatni az igényeinknek megfelelően.

Mikor már elég sok kódot raktunk be a forráskód mezőbe, nem fog kiférni az ablak magasságára ezért lefelé fog növekedni a mező. Megnövekedett mezőt az egér görgő segítségével tudjuk mozgatni. Ahogy távolítjuk el a blokkokat úgy fog csökkenni a nagysága a megnövekedett mezőnek, még vissza nem áll az alaphelyzetre. Ha hozzá akarunk adni kódot a jelenlegihez, ám már a megnövekedett mezővel dolgozunk akkor, ha odahúzzuk az egeret a paranccsal a felső vagy az alsó részhez, akkor a program automatikusan görgetni fog a kívánt irányba, ezzel megkönnyebbítve a nagy kód készítését.

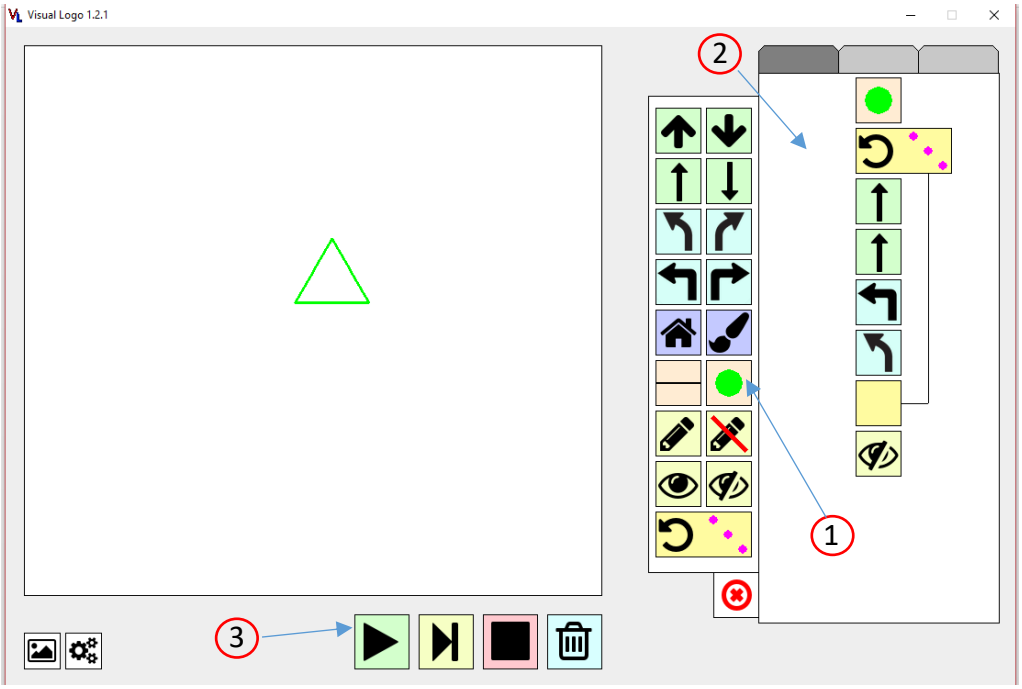
Sidepanel esetén még van pár kiegészítő segéd kombináció. Ez főleg akkor jön jól, ha touchapdot használunk. (Ez előfordulhat több esetben is, ha épp nincs kéznél egy egér, vagy csak több mint ezerhétszáz dollárt költöttünk egy laptopra, amire csak USB Type-C csatlakozókat raktak. | Nem ragaszkodok ehhez a mondathoz.) Ezért billentyűzet segítségét is használhatjuk, és ki tudjuk váltani a jobb egér kattintást az „S" gomb lenyomásával. Így a második bekezdésben felsoroltakat végre tudjuk hajtani az „S" gomb tartásával és bal egér gomb kattintással a megfelelő elemeken. A megnövekedett forráskód mezőt tudjuk mozgatni a nyilak segítségével.

Elemek hozzáadása a forráskód mezőhöz történhet a fent leírt módon, hogy belehúzzuk a parancsot a mezőbe, ám van egy másik mód. Ha azt szeretnénk, hogy egy parancs kerüljön automatikusan a program végére, akkor a megfelelő parancson nyomjuk meg az egér görgő gombját, vagy tartsuk lenyomva a „A” gombot és jobb egér gombbal ugyanaz elérhető. Ez működik a parancsokat tartó kis eszköztáron, és még a forráskódon is. Pl. Ha van egy már előre beállított ciklusunk, amit le akarunk másolni, akkor elég a forráskódban rákattintani, és lemásolja azt. Ha törölni szeretnénk egy megadott elemet, akkor kihúzhatjuk a blokkot a mezőből, másik lehetőség az, ha lenyomva tartjuk a „D” gombot, akkor jobb egér gomb segítségével is megtehetjük ezt.

2.4. Felhasználói tesztesetek

2.4.1. Zöld háromszög rajzolása

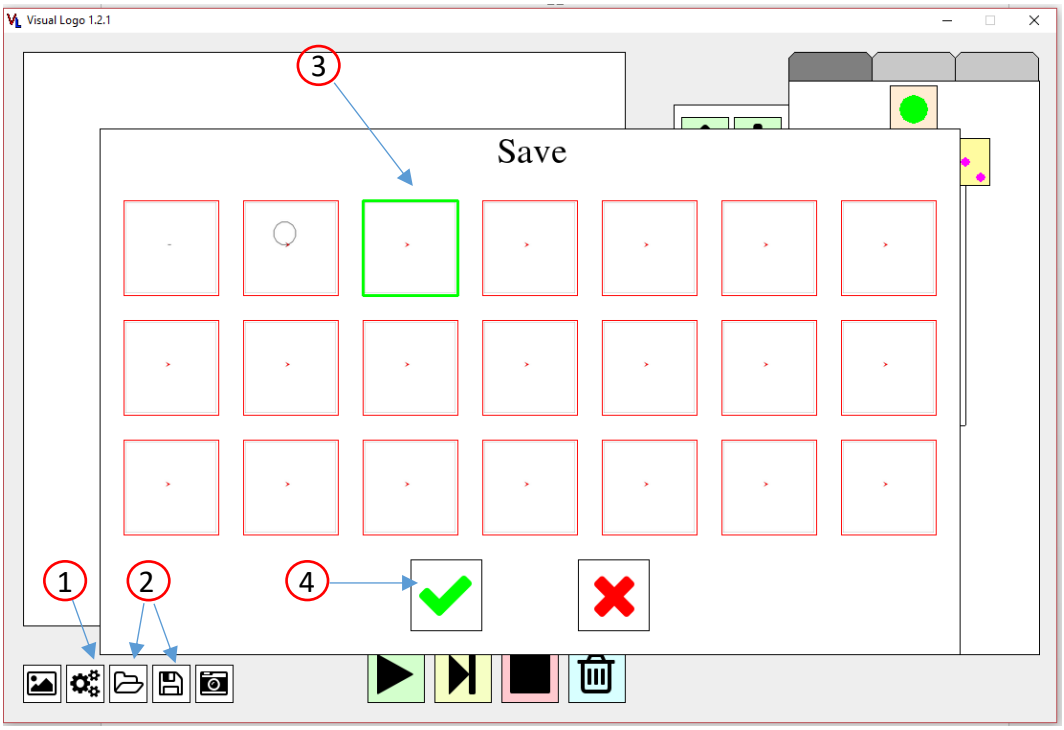
1. Hozzunk létre a forráskódot, amit majd kirajzolja az alakzatot. Ahhoz, hogy zöldre állítsuk a rajzolást, kattintsunk jobb egérgombbal a megfelelő parancsra míg nem kapjuk meg a megfelelő színt.
2. A parancsok rendre a következők: színcsere zöldre, ciklus hármas számlálóval, előre, előre, nagy fordulás balra, kis fordulás balra, ciklus bezáró elem, rajzoló mutató elrejtése.
3. Mikor elkészültünk ezzel, nyomjuk meg a zöld futtatás gombot, ezután már látható is az eredmény.



5. ábra, Zöld háromszög elkészítése

2.4.2. Rajz elmentése / betöltése

1. Kattintsunk jobb egérgombbal a beállítások menüre, hogy megjelenjenek a rejtett gombok.
2. A nyitott mappa a betöltő, a floppy a mentés gomb. Nyomjuk meg amire szükségünk van.
3. Válasszuk ki a mentés vagy betöltés helyét, és kattintsunk rá majd a zöld pipára.
4. Várjuk meg, hogy lefusson a program, és készüljön el a mentés. Ha betöltjük a programot, akkor a jelenlegi forráskód kukázódik, és oda fog kerülni az új.



6. ábra, Mentés folyamata

2.4.3. Adatok kézi megnyitása

Ha szeretnénk egy rajzunkat megmutatni másnak is, akkor két lehetőségünk van:

- Használjuk a kamera gombot a rejtett menüben. Ez a kép a Screenshots mappában lesz megtalálható, a készítés időpontja néven.
- Ha már el van mentve, akkor nézzük meg, hogy melyik mezőre mentettük le, és sorfolytonosan számoljuk ki melyik az, figyeljünk arra, hogy 0-tól kezdődik a számozás. Nyissuk meg a UserData mappát a program mellett, és csomagoljuk ki a megfelelő tömörített fájt, és ott fogjuk megtalálni a képet.

3. Fejlesztői dokumentáció

3.1. Tervezés

3.1.1. Inspiráció más hasonló programokból

Ha megnézzük a jelenleg nagy népszerűségnek örvendő programnyelveket, észrevehetjük, hogy nincs köztük vizuális programozás. De miért van ez? Erre a kérdésre kerestem a választ munkám írása során. Első feladatom az volt, hogy keressek működő és közismert nyelveket, amikben van lehetőségünk vizuálisan bármilyen jellegű programot készíteni. Ha végzünk egy keresést Bing segítségével, akkor a Scratch lesz az első, amit ajánl nekünk. Ezt a programot az MIT fejleszti jelenleg. A program nagyon egyszerű és hatékony. Gyerekek számára van fejlesztve, hogy egyszerűen és játékos módon tudjanak szereplőket mozgatni egy vásznon. A parancsok színkódolva vannak, és különböző formájuk van. Ezt még kíséri a tény, hogy van fordítás sok nyelvre, köztük magyarra is. Így minden kisiskolás gyerek a saját nyelvén tanulhat meg programozni. Mikor összerakjuk kódunkat a kis ablakban, már futtathatjuk is, és látjuk mi lesz a végeredménye alkotásunknak. Ezért sokan szeretik, és nagy előszeretettel tudom én is ajánlani mindenkinek, akit érdekel.

A program annyira sikeres, hogy az MIT-nak van még egy projektje, ami pont ezt a blokkosított, egyszerű és letisztult környezetet használja, ez pedig az MIT App Inventor. A program tökéletes mása a Scratch-nek. A különbség abban rejlik, hogy ezzel a programmal nem szereplőket mozgatunk, hanem akár egy teljes értékű Android alkalmazást készíthetünk. A program futtatása is nagyon egyszerű, egy QR-kód segítségével Android-os telefonunk felismeri és futtatja. Sok ilyen applikáció található a GooglePlay Store-ban, ami ezzel a programmal készült. Egy kisebb magyar startup cég is ezt a technológiát használta egészségügyi alkalmazásuk elkészítéséhez.

Következő cég, amelyik foglalkozik vizuális programozással az Epic Games. Ők a fejlesztői a híres Unreal Engine-nek. Nem áll tőlem messze a játék programozás, ezért ismerem az általuk készített programot is. Az Unreal Engine-t kétféleképpen lehet programozni, C++ nyelven vagy vizuálisan. UE3 alatt ezt a technológiát Kismet-nek hívták, UE4 alatt pedig Blueprints néven találjuk meg. Ők más megközelítésből hasznosítják a vizuális programozás adta előnyöket. Míg a Scratch program, merev blokkokat, és letisztult struktúrát készít, addig az UE4 egy gráfos megoldást választott. Minden parancs egy gráf csúcs. Különböző csúcstípusok más-más célt szolgálnak. Vannak olyan csúcsok, ahonnan indul a vezérlés, valamelyik feladatot végez az adatokkal és vannak lekérdezési csúcsok. A legtöbb csúcsba kell húzni a vezérlési folyamatot mutató élt, míg lekérdezési csúcsba nem kell bekötni. Mikor egy vezérlő csúcs lefut, akkor lefuttatja a rá csatlakozó lekérdező csúcsokat is, pl. matematikai számítások.

A fent felsorolt programok mind alkotásra szolgálnak, kapunk egy eszköztárat, hogy játsszunk vele, és a kreativitásunkra bízzák, mit hozunk ki belőlük. Ám akadnak játékok is, amik vizuális programozás segítségével akarják, hogy megoldjunk több fejtörőt. Kisebb korosztály számára egy ilyen játék a Lightbot alkalmazás. Egy robotot kell irányítanunk, és fel kell kapcsolnunk lámpákat a pályán. Öt darab parancsunk van, és pályáktól függően tudunk használni procedúrákat. A feladatok változatosak, és a felhasználó-felület is gyerekbarát. Ajánlani tudom jelenlegi programozóknak is.

Végül a személyes kedvencem a Human Resource Machine játék. Játékosan mutatja be egyre nehezedő feladványokon, hogy kell használni különböző programozási technikákat. Blokkok segítségével lehet összerakni a kódot. Az utasítások egyszerűek, vegyünk egy elemet a bemenetről, tegyük el a memóriába, adjunk hozzá, vegyünk a tárolt mező értékéből, tegyük a blokkot a kimenetre. Amiben különbözik a Ligthbot-tól, hogy míg ott rekurzió volt az alapja a programnak, addig itt Assembly nyelvhez hasonló ugrásokat tudunk használni. Ezt is csak ajánlani tudom, mivel gyakorlott programozóknak is tud elég kihívást mutatni.

Ezek után már volt egy elképzelésem, hogy fogom a vizuális részét megcsinálni a programnak, ám a másik része, hogy legyen interpretálva a nyelv, ez volt még egy kihívás. A nyelvnek, amit meg kellett alkotni, Turing teljes nyelvnek kellett lennie. Ez annyit jelent, hogy a nyelv képes legyen megoldani olyan feladatot, amit egy Turing-gép is meg tud oldani, ha adunk neki elég időt. Két lehetséges út volt járható a kód ismétlés megoldására, a ciklus vagy a rekurzió használata. Ha megnézzük a fent felsorolt programokat, akkor láthatjuk, hogy rekurziót a Lightbot használ, mint elsődleges vezérlési egység. Mikor felmerült a véges sok rekurzió ábrázolása a program tervezésekor, akkor látszott már, hogy az csak sok munka és tervezés után tudna működni, ezért inkább a ciklus mellett döntöttem. Könnyebb ábrázolás, egyszerűbb kód és feldolgozás. További segédanyag, amit felhasználtam a tervezéshez, a Fordítóprogram tárgyból tanult módszerek, és ott elkészített (számomra ABAP nyelv) fordító volt a kiindulási pont. Ez mellett volt még egy Pythonban írt Brainfuck értelmezőm is.

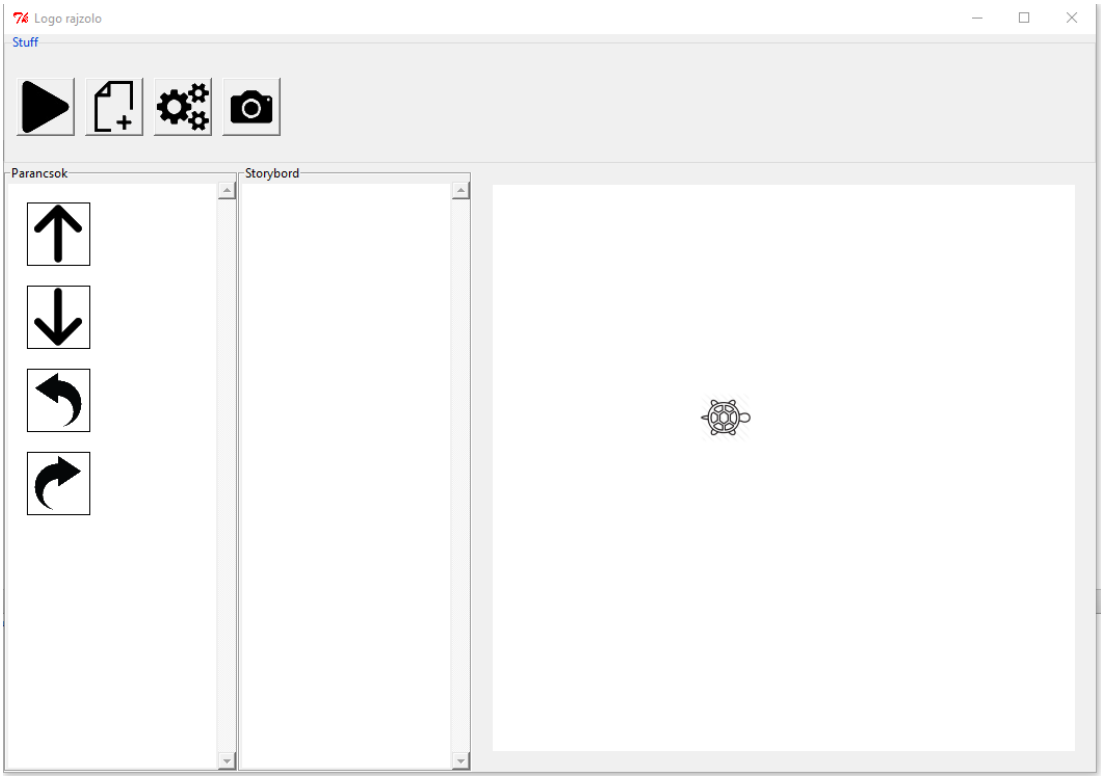
3.1.2 Első prototípusok

A programot Python nyelven írtam meg, mivel script nyelv, és gyorsabb benne fejleszteni programot. Mivel gyengén típusos, ezért kevésbé kellett azért aggódnom, hogy típushiba történik, és az, hogy Listában nem feltételen egy típus elemeit tudom elhelyezni, ez jelentősen felgyorsította a fejlesztést. Mivel ez így lehetséges, a program futási időben számol ki mindent, mikor hivatkozik egy változóra a Python értelmező, nem tudja a szimbólumtáblát használni, hogy mi volt a típusa az objektumnak, ezért helyben kell kiértékelnie azt. A folyamat nagyon időigényes feladat, így a program futási ideje lesz rosszabb. Ez mellett Pythonban szemantikai hibát sokkal könnyebb ejteni, mint szintaktikust.

Első legfontosabb dolgom a téma elfogadása előtt az volt, hogy felmérjem, egyáltalán lehetséges-e amit meg akarok csinálni, ezért több prototípust készítettem. Első volt, hogy a Pygame lehetőségeit teszteljem, hogy mit lehet vele csinálni, és mit nem. A Pygame a közismert SDL (Simple DirectMedia Layer) könyvtárat használja. Ez egy elég régi és széles körben elterjedt technológia, ezért minden modern operációs rendszer valamilyen formában támogatja azt. A CPython is megtalálható a legtöbb modern rendszeren alapból telepítve, Windowson manuális telepítés szükséges, ezért elegendő egyszer megírni a kódot, ami használja az OS modult, ez aztán platform független képes működni.

A következő prototípus feladata volt, hogy egy már meglévő vizuális fejlesztőkörnyezethez hasonló rajzoló felületet hozzak létre. Első gondolatom volt, hogy az UE4-hez hasonló környezetet készítések. Az egyetlen bökkenő ezzel csak az volt, hogy az ovis gyerekek nem tudják mik azok a gráfok és azok hogyan működnek, így inkább a Scratch és a Human Resource Machine megoldásait elemeztem.

3.2. Teszt Verzió 1.0



7. ábra, Első teszt verzió

Az első teszt verziót (5. ábra) a Python Tkinter ablakkezelő moduljával készítettem el. Ez már egy működő verziót eredményezett. Felül találhatóak a vezérlő gombok. Alatta láthatók a rendelkezésre álló parancsok, mellette a forráskód mező, és végül egy beillesztett Pygame ablak. Itt a kódot úgy kellett berakni a forráskód mezőbe, hogy rákattintottunk, és automatikusan berakta az ablakba. A ciklust is hasonlóan hozzáadta. Ha ciklusba akartunk rakni parancsokat, akkor rá kellett kattintani a ciklusra a forráskód mezőn és utána oda rakta be a parancsokat a program.

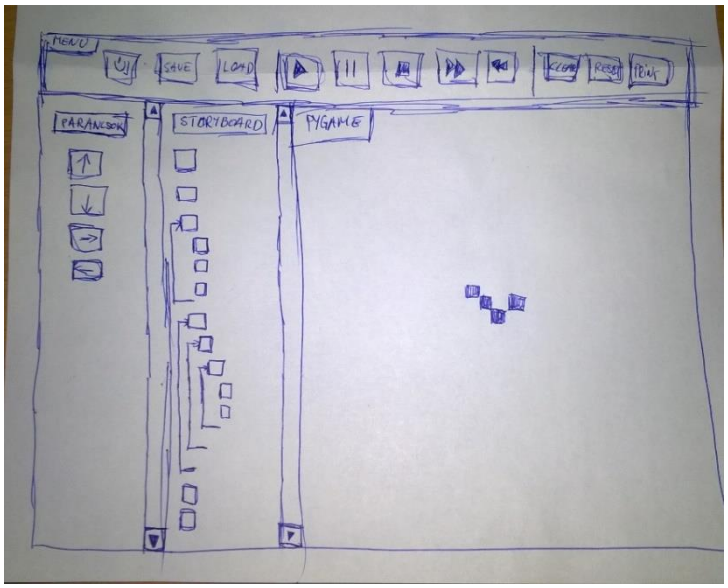
Ez a megoldás működőképes volt, de mikor komolyabb rajzolási dolgokat kellett végrehajtani a parancsok mezőben vagy a forráskód mezőben, akkor a Tkinter korlátjai miatt nem volt elég jó. Korlátozások kikerülése nem volt a célja a programomnak, ezért elvetettem az ötletet, hogy Tkinter segítségével legyen a program megvalósítva. Néztem emellett más alternatív ablakkezelő könyvtárakat, mint a WXPython, és ez sem volt jobb. Főleg, hogy a WX-nek nemrég volt verzió váltása, és a régi Linuxokon még nem működik az új, de az új Linuxok meg már felszámolták a régi WX támogatást. Ezért megint csak vakvágányra jutottam. Itt jött a döntés, hogy nem fogok használni semmilyen segédkönyvtárat az ablak kezelésre, hanem az egészet megírom tisztán Pygame segítségével.

3.3. Leírás információk

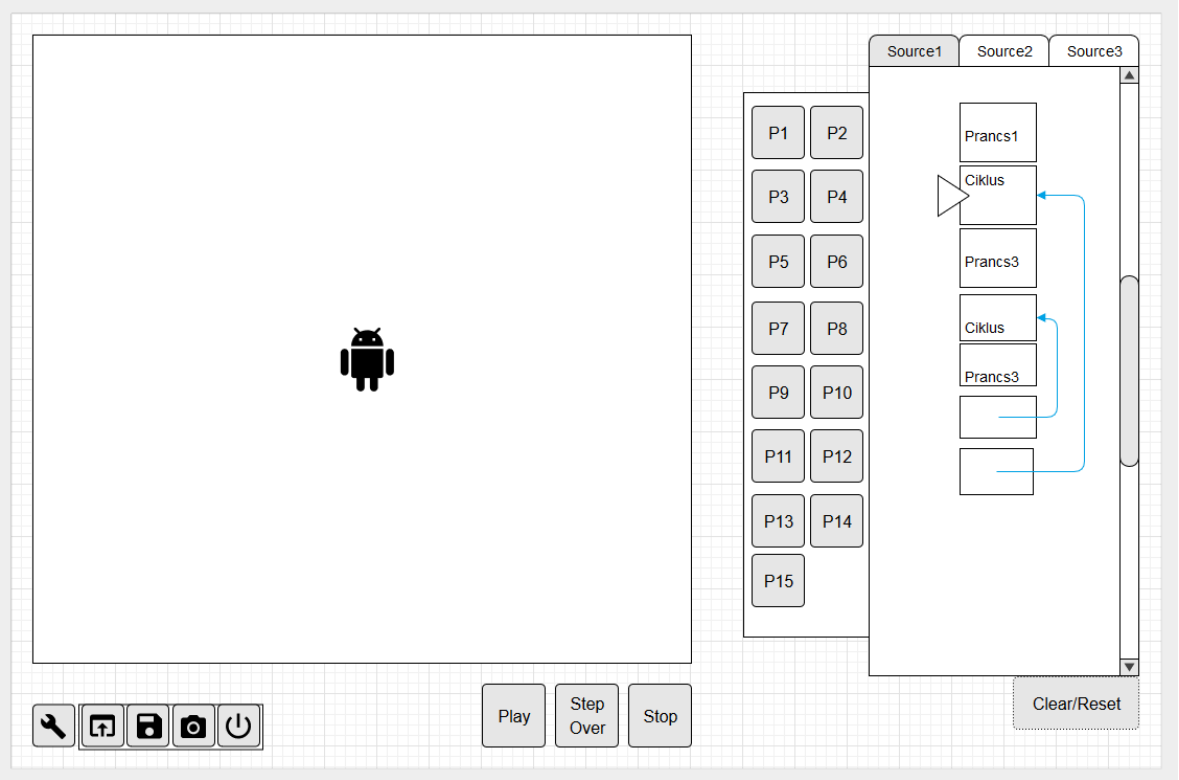
Az előző részben már taglaltam, miért vetettem el a beépített ablakkezelő használatát, ezért nekem kellett az alapoktól építeni egy újat. A dokumentáció során nem fogom lépésről lépésre elemezni az általam készített keretrendszer minden apró építőkövét, hanem inkább az elméletet, ami mentén fel lehet építeni ezt, azt fogom elemezni és bemutatni. Az adathordozón, illetve a projekt GitHub és Bitbucket oldalán megtalálható a pontos UML diagramja a programnak.

3.4. Mocup

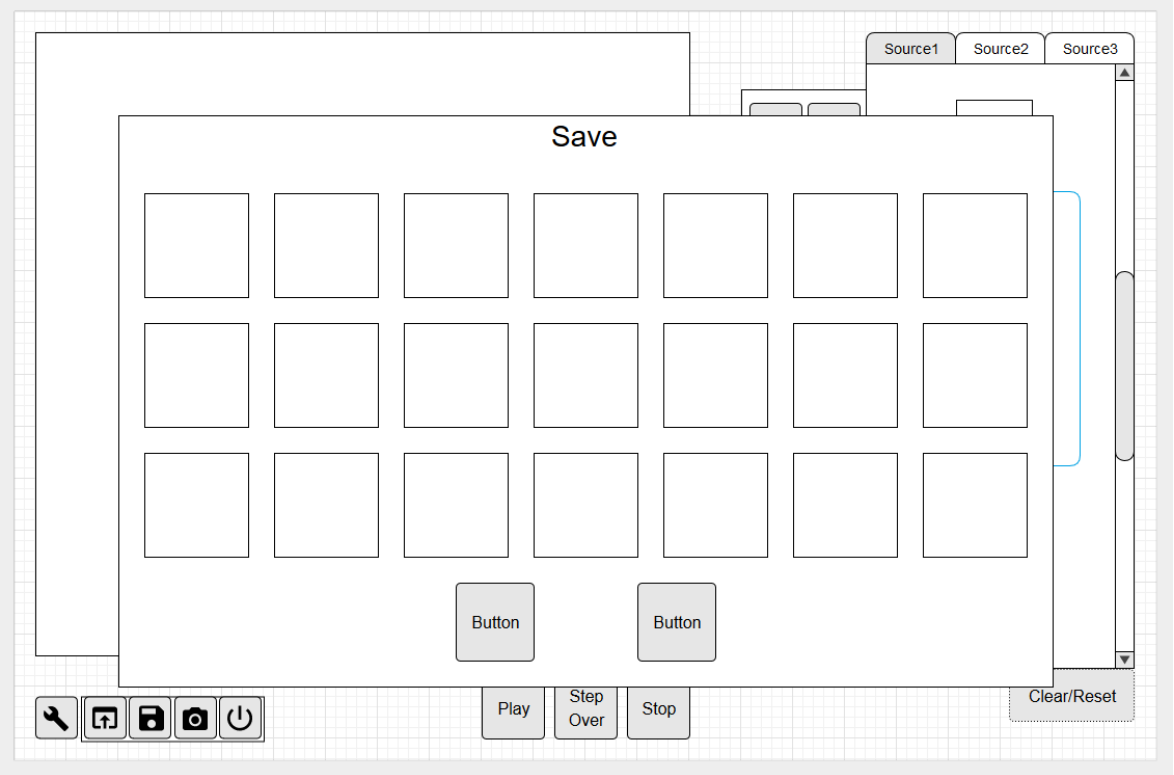
Mivel grafikus programról van szó, ezért az első lépés lerajzolni vázlatosan, mit is fogunk építeni. Ha modernebb módszert használnánk, akkor ott a Mockup. Az általam használt program erre a célra a Moqups volt. Ez egy ingyenesen használható online program. Sok ikont tartalmaz, ezért egy egész jó vázlatos rajzot lehet készíteni. Az általam tervezett programnak itt láthatók a vázlatrajzai.



8. ábra, Első teszt változat vázlatos rajza papíron



9. ábra, Release verzió első vázlatrajza, Moqups



10. ábra, A program mentő és betöltő ablakának a vázlata, Moqups

A programom felbontása nem változtatható, a főbb elemek helyzete abszolút koordinátákkal vannak megadva. Ennél léteznek kifinomultabb megoldások is, könnyebb fejlesztés érdekében a rajzon látott képet egy az egy arányban készítettem el. A másik képen látható a mentés ablak vázlatrajza. Itt a távolságok egy abszolút pozícióból vannak kiszámítva.

3.5. Felépítés alapok

A keretrendszer építésének első és legfontosabb lépése egy olyan ősosztály létrehozása volt, aminek a feladata, hogy olyan függvényeket biztosítson, amik segítségével ki tudjuk majd rajzolni a felhasználói felületet. Ez lett az AbstractDrawable osztály. Feladata, hogy a kirajzolandó objektumoknak tárolja a koordinátáit, nagyságát. Továbbá még tartalmaz függvény definíciókat arra, hogy meg tudja mondani egy osztály esetén, lett-e rá kattintva. Emellett még tartalmazza a másik legfontosabb függvény definíciót, hogy kell kirajzolni egy adott objektumot. Ezt a két függvényt minden ebből származtatott osztálynak felül kell írnia. Ebből származik a maradék osztály túlnyomó része. Ide tartoznak azok az osztályok, amik a Drawable mappában vannak.

A program még tartalmaz két absztrakt osztályt, az egyik a Command, a másik a Polygon. A Command felel a programozásra felhasznált parancsokat leíró mezőkért. Mivel ezek megjelenítése teljes mértékben megegyezik, ezért ha létrehozunk egy új parancsot, elég csak megadni a kinézetre vonatkozó paramétereket, és a többit az osztály feladata kiszámolni. A Polygon osztály egy módosított kirajzolási technikát használ, mivel nem elég megadni a sokszög egy pontját, szükség van a többire is. Ezért egy ponthalmazt tárolunk el, amit a Pygame átalakítások nélkül rögtön tud használni és kirajzolni.

3.6. Kirajzolható osztályok

Rectangle A különböző osztályoknak más-más feladatuk van. A legtöbbet használt közülük a Rectangle. Feladata, hogy téglalapot tudjunk kirajzolni a rajzlapra. Meg vannak adva a bal felső sarok koordinátái, emellett a szélessége és a magassága. Tartozik még hozzá egy paraméter, amely megmondja, hogy átlátszó legyen-e. Ez a beállítás azt jelenti, hogy kirajzoláskor elég-e kirajzolni a körvonalát a négyzetnek, vagy kell-e még egy másodlagos téglalapot rajzolni, aminek a pontjai megegyeznek, ám ez mindig egy kitöltött téglalap, ami a hátterét adja a körvonalnak.

Button Az osztály feladata, hogy egy gomb működését valósítsa meg. Ezért tartalmaz egy OnClick() függvény. Ez egy törzs nélküli függvény, amit akkor kell lefuttatni, ha a kattintáskor igazat kaptunk vissza az IsInside() függvénytől. Mivel nincs törzse a függvények, mikor létrehozzuk, akkor kell a *bind* műveletet használni, ami paraméterként várja egy függvény definícióját. Pythonban egy függvény felüldefiniálása probléma nélkül megtörténhet, ezért használható ez a megvalósítás.

Sprite Mivel nem csak négyzeteket szeretnénk kirajzolni, hanem különböző képeket, ezért van szükségünk a Sprite osztályra. Feladata, hogy olvasson be egy képet, és rajzolja ki a megfelelő koordinátákra. Mikor ezt az osztály használjuk, a konstruktorban megadott paraméterekre lesz a beolvasott kép is méretezve. Ajánlott jobb minőségű képek használata, hogy kicsinyíteni kelljen és ne nagyítani.

TextIcon A program a képek mellett támogatja mindenféle írásjel kirajzolását is. A Font Awesome egy szabad felhasználású font készlet, amit nagy előszeretettel használnak a web-fejlesztők. Érdekessége az, hogy nem betűket tartalmaz, hanem különböző jeleket. Ezek az írásjelek azért fontosak a programban, mert a képek minősége romlik, ha kicsinyítjük azokat, de mivel a Font vektorgrafikusan van ábrázolva, ezért minden méretben ugyanolyan éles lesz. Ezek a jelek képzik a programozási parancsok képeinek a nagy részét.

DrawingIcon Ez az osztály felel azért a kis nyilacskáért, ami a programban fog rajzolni. Elég a középpontját megadni és matematikailag lesz kiszámolva. A mozgásért és a fordulásért is ez felel. Ha valamelyik parancsot szeretnénk végezni rajta, akkor feladatunk, hogy a pontok mindegyikére végezzük el a megfelelő mátrixszorzási műveletet. Az így kiszámított pontokkal írjuk felül a jelenlegi pontokat. Vigyázzunk, mert a kerekítési hiba problémákat okozhat. Pythonban az int() függvény minden lebegőpontos számot lefelé kerekít, ezért inkább ajánlatos használni round() függvényt kerekítésre.

RunPointer A program futása közben mindig látható, hogy épp melyik műveletet végzi el. Ennek a mutatására van használva a RunPointer osztály. Elég megadni neki a jelenlegi parancsnak az indexét és függvény kiszámítja, hogy a képernyő melyik koordinátáira kell kirajzolni ezt a mutatót.

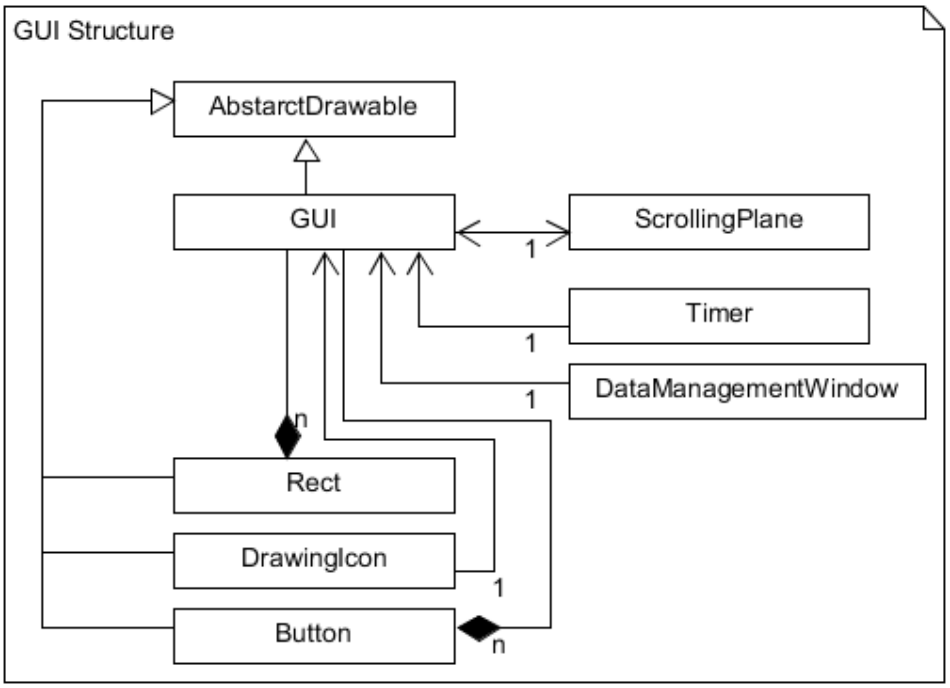
Tab A program tervezése során felmerült a kérdés, mi van akkor, ha több személy szeretné használni? Erre az eshetőségre lett létrehozva a Tab osztály. A különböző füleket a forráskód mező felé rajzoljuk ki. Mivel változó, hogy épp mennyi felhasználó szeretné igénybe venni, ezért megvalósításkor paraméterrel lehet beállítani, hogy mennyi rajzolódjon ki ebből az osztályból.

RenderItems Ez az osztály egy csomagként funkcionál. Benne található osztályok a Line, és a FloodFill. Ezek nem direkt leszármazottjai az AbsztractDrawable osztálynak, mivel ezek egyszer jönnek létre, és az után már nem változnak. Feladatuk:

- **Line**, kirajzoláskor egy egyenes vonalat húz a jelenlegi helyről a cél irányban a megadott helyig. Színe és vastagsága állítható a felhasználó által.
- **FloodFill**, kirajzoláskor kitölti az alatta levő teret arra a színre, amire jelenleg állítva van a toll. Az algoritmus, amit használ, mohó, ezért ha nem jelenik meg rögtön az eredmény, akkor várnunk kell. Alacsony teljesítményű gépek esetén akár több másodpercet is igénybe vehet. Ám ha egyszer már kiszámolódott, attól fogva már a program újra gyorsan dolgozik. Ha a frissen kirajzolt alakzaton állunk épp, és akkor hívjuk meg a függvényt, akkor az rögtön vissza fog térni számítás nélkül, mivel nem különbözik a szín attól a színtől, amin éppen áll.

3.7. Gyűjtő, tároló rajz objektumok

3.7.1 GUI

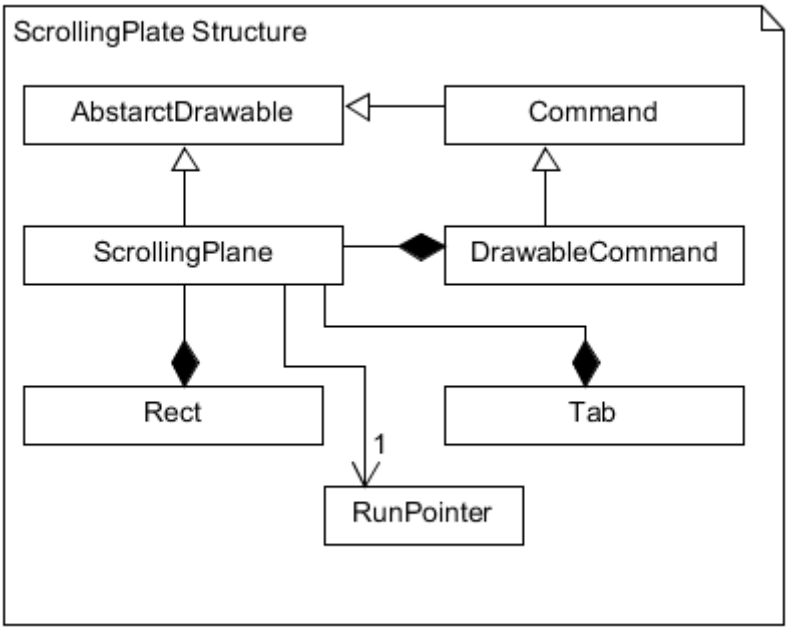


11. ábra, GUI osztály felépítése, nagy vonalakban

Mikor használunk egy meglévő ablakkezelő rendszert, sokszor használunk különböző csoportosító adatszerkezeteket. Ezek segítségével több nagyobb darabra tudjuk felosztani a programunkat. A darabokat meg már könnyebb karbantartani, mintha egy osztályba lenne minden összerakva. Ezért létre kellett hozni ilyen osztályt. A legnagyobb csoportosító egysége programomnak a GUI. Feladata, hogy az összes képernyőre kirajzolandó dolog itt történjen meg. Ezért ez az osztály az összes kirajzolható osztályt használja: Gombokat, a kirajzoló felületet és a forráskód mezőt. Mivel ez még mindig egy nagyon nagy tároló, ezért szükségünk van kisebb tárolókra. A második nagyobb egység a Sidepanel.

A GUI osztályban történik meg több elem eseménykezelése. Első sorban a gombok, mindegyiknek van egy külön függvény felvéve, és azok hozzá vannak csatolva a gombokhoz. Továbbá ez az osztály felel a felületen rajzoló nyíl helyének meghatározásáért, és az összes kirajzolt vonalért a program futása során. Helyet kapott itt még egy időzítő is. További részletek a 3.9.2 részben.

3.7.2 Sidepanel

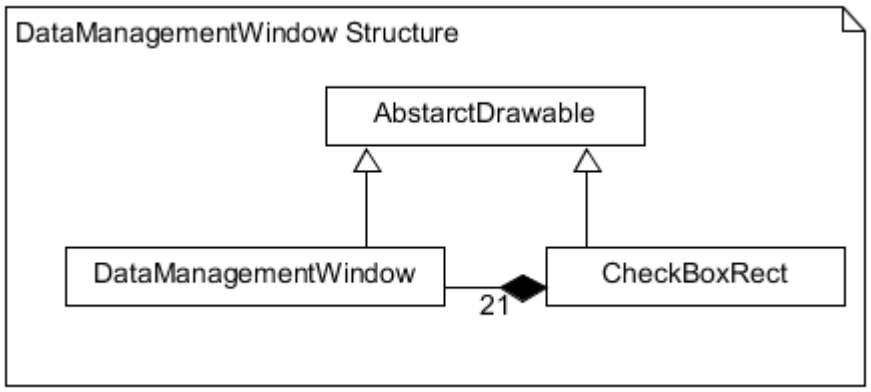


12. ábra, ScrollingPlane felépítése nagyvonalakban

A Sidepanel (forráskód kezelő) feladata, hogy tartalmazza a parancsokat kódoláshoz és az általunk készített kódot. Ha szemügyre vesszük a felépítését, még több kisebb elemre bonthatjuk fel ezt. A program futása közben baloldalon látható rácsszerkezet egy kisebb gyűjtő egység, feladata, hogy az összes felhasználható parancs látható legyen. Mellette látható rögtön a jelenleg elkészített kódunk. Ezt egy listában tároljuk el, hogy aztán tudjuk használni futtatáskor. Fent láthatók a kis fülek. Ezek arra szolgálnak, hogy melyik listát szeretnénk használni. A megoldásom egy forráskód listát tároló lista. Különböző fülek rendelkeznek egy azonosítóval, ami segítségével a program is tudja azonosítani, melyik lista legyen megjelenítve. Ez az osztály felel még azért, hogy a program futását mutató nyíl, a RunPointer, ki legyen rajzolva.

Ez az osztály felel az itt történő összes esemény lekezeléséért. Ide tartozik a fül váltás, az általunk kívánt programkód elkészítése, és annak az elemzése. Mivel minden parancsot egy külön osztály reprezentál, ezért a parancsok értelmezése úgy történik, hogy meghatározzuk melyik osztály a soron következő elem. Részletes leírás a 3.10. részben található a folyamatról.

3.7.3 Data Management Window



13. ábra, Data Management Window vázlatos felépítése

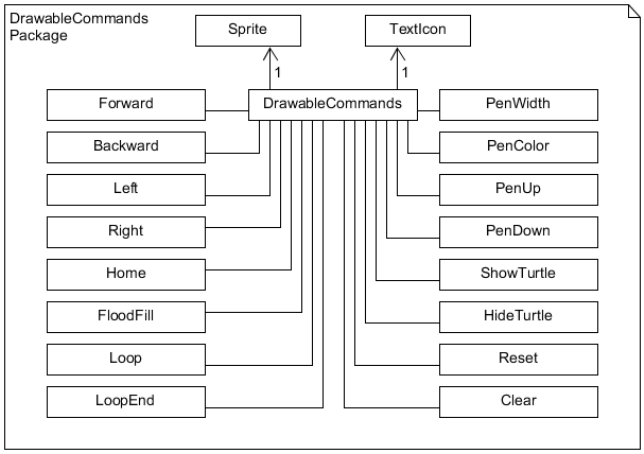
A Data Management Window egy érdekes osztály. Egy különálló ablak, amit a GUI fölé rajzol ki a program. Feladata, hogy kezelje a felhasználói fájlokat. Kétféle módban lehet megnyitni, mentés vagy betöltés. Mindkét esetben ugyanaz az ablak nyílik meg, csak a fenti szöveg változik, attól függően milyen módban van. A felhasználó számára itt 21 darab mentési hely van, amit szabadon használhat fel. Az alsó két gomb meg elvégzi az adott műveletet. Míg ez az ablak nyitva van, addig a mögötte levő ablakon található gombokra nem lehet kattintani.

Az osztály egy speciális, erre a célra készített **CheckBoxRect** osztály elemeit jeleníti meg. Ez szintén az AbstractDrawable osztályból származik, ezért megvan minden tulajdonsága a kirajzoláshoz. Ami különbözik egy sima gombtól itt, az az, hogy ez tartalmaz egy Sprite objektumot is, ami lehetővé teszi, hogy a felhasználó lássa, mi található a különböző mentési helyeken. Pár kiegészítő mezőt is tartalmaz, mivel úgy kell, hogy működjön, mint egy kijelölő mező, ezért ha rákattintunk az egyikre, egy zöld keretet kap, ami jelzi, hogy ki van jelölve. Egyszerre csak egy mező lehet kijelölve az egyértelműség érdekében.

Miután megválasztottuk, melyik mentési helyet szeretnénk használni, a zöld pipára kattintva betöltődik vagy elmentődik az adott munka. Ha nincs kijelölve egy elem se, és úgy kattintunk a zöld pipára, azt a program úgy kezeli, mintha a piros x-re kattintunk volna, és nem fog művelet végrehajtódni. Ez mellett még támogatott a dupla kattintás is, ami megkönnyebbíti a kezelést, ha elég gyorsan, 60 képernyő frissítésen belül kattintunk kétszer az egyik mezőre, akkor az automatikusan elvégzi a hozzárendelt műveletet.

Egy érdekessége a programnak, ez az egyetlen osztály, ahol a Python Imaging Library szükséges. A modul fejlesztése során észrevettem, hogy a Pygame nem tudja jól kezelni a képeket, ugyanis ha módosítani kell a képet magát, akkor nem alkalmaz anti-aliasing algoritmust. Ezért kellett a külön könyvtár, ami ezt támogatja, de mivel Pygame kirajzolásához egy képre van szükség, a PIL bytestream objektummá alakítja a módosított képet, amit már a Pygame tud kezelni, mintha csak fájlból olvasott volna.

3.8. DrawableCommands alosztályok



14. ábra, DrabableCommands csomag osztályai

A Python nyelv sajátosságából adódóan egy forrásfájlon belül lehet bármennyi osztályt létrehozni, majd egyesével hivatkozni azokra. Ez a technika hasonlít a csomagokhoz. A nyelvnek ezt a tulajdonságát kihasználva egy fájlba van csoportosítva az összes parancs osztály megvalósítása. Ezek a parancsok követik a Logo főbb parancsait. Mivel különböző parancsok nem csak működésben, hanem kinézetben is különböznek, ezért használtam ezt a megvalósítást.

Az előző fejezetben taglaltuk, melyik mit csinál. Az osztályok nem sokban térnek el egymástól. Mindegyik rendelkezik a saját konstruktorával, a többi kód meg megegyezik. Ám akad pár kivétel ez alól, aminek a kirajzolása nem egyértelmű, mint a Loop, LoopEnd, PenWidth és PenColor. Ezek az osztályok felülírják a kirajzoló metódusukat. A fent felsorolt parancsok közös tulajdonsága, hogy változik a jelük attól függően, hogy milyen állapotot reprezentálnak. A PenColor jelzi a jelenleg aktív színt. A PenWidth relatívan mutatja milyen vastagságú lesz a húzott vonal. A Loop egy különleges darab több szempontból is. Első sorban hogy két helyet foglal el, másik tény, mikor példányosítjuk, akkor a LoopEnd parancsból is készít egy másolatot. Továbbá mutatja hány alkalommal fog lefutni.

A Loop és LoopEnd parancsok érdekesek még programozói szempontból is. A programozásban kétféle ciklus létezik, az egyik az elöltesztelő, a másik a hátultesztelő. A különbség a kettő között, hogy a hátultesztelős eggyel többször fog lefutni. Ezt a működést úgy valósítottam meg, hogy amikor a programot elindítjuk futtatni, akkor egy ellenőrzés megnézi, hogy a vége előbb van-e mint az eleje, és ez alapján fogjuk eldönteni, hogy melyik típusú. További részletek a 3.10. részben.

3.9. System csomag

3.9.1. Konstansok

Ez is egy csoportosító osztály, mint a DrawableCommands. Három osztályt tartalmaz. Azért vannak külön ezek, mivel nem direkt kirajzolhatók, ám szorosan kötődnek a program működéséhez.

Első a színek osztálya. Itt található az összes felhasznált szín a program futása során. A színeket RGBA formátumban kell megadni. Tapasztalatom szerint a Pygame nem tud mit kezdeni az így megadott Alpha channel értékével. Korábbi verziókban elég volt az RGB kódot megadni, de a FloodFill implementálása során az összehasonlítandó érték RGBA kódolásban volt, ezért kellett kiegészíteni ezt az osztályt is arra.

Második osztály a képek elérési útvonaláért felelős konstansok osztálya. Az itt szereplő képek csak a relatív útvonalat tárolják a fájl eléréséhez. A Sprite osztály felel az abszolút útvonal előállításáért. Itt minden kép az OS modul segítségével van megadva. Ez azért szükséges, mert különböző rendszeren más felé dől a perjel (\\ , /). Az OS osztály mindig a megfelelőt fogja belehelyettesíteni, ezzel könnyebbé téve a program hordozhatóságát.

Végül a Font Awesome osztály konstansai. Az a legterjedelmesebb, mivel innen van a legtöbb elem használva. Mint már előzőleg említve volt, a Font Awesome egy képet tartalmazó írásjel készlet. Ám mivel ezek az írásjelek nem szokványosak, ezért külön van egy külső fájlban hozzárakva a projekthez. Emellett nem megjeleníthetők alapból, ezért külön-külön meg kell címezni minden karakterkódot, hogy melyik alakzatra mutatnak.

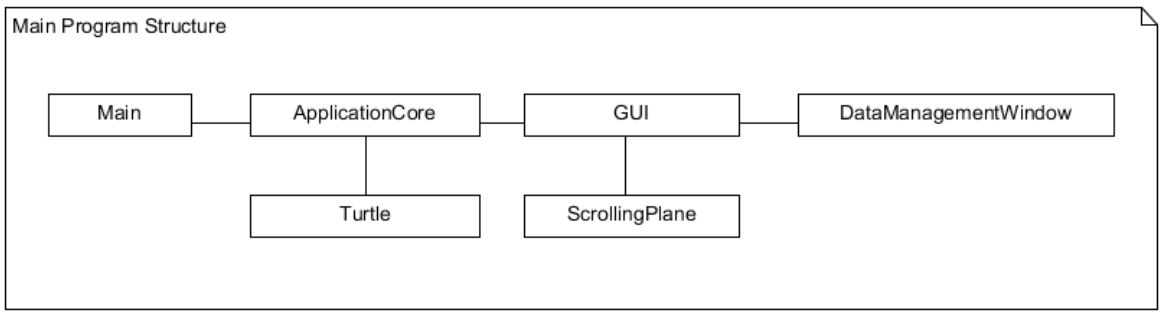
3.9.2. Időzítő

Következő osztály az Időzítő. Már esett szó feljebb arról, hogy nem hívhatjuk meg egy grafikus program futása során a sleep() funkciót, mivel az az egész program futását gátolni fogja. Ezért az időzítést úgy szokták csinálni, hogy a program futásának kezdete óta eltelt időhöz mérik a jelenlegi időt. Ennek több implementációja létezik. Az egyszerűbb és kevésbé elegáns megoldás, amit az én programom is használ, az idő telését a képernyőre kirajzolt képernyő frissítések száma alapján méri le. A módszer működőképes, de gyengébb hardveren jelentősen lassítja a program működését. Régi játékoknál tapasztalható ez az effektus, mikor a játék gyorsaságát numerikusan lehet megadni, és az a processzor feldolgozó képességéhez kalibrálja a kirajzolást. Ekkor lehet tapasztalni, hogy a játékbeli idő sokkal gyorsabb, mint egy másodperc, és minden extrém gyorsan mozog, néha még az egér is elszabadul, és irányíthatatlan az egész. Ennek van már egy jobb implementálása is, amit egy későbbi szekcióban fogok részletezni. Az időzítő a programban a GUI osztályban kapott helyet, mivel ott történik a kirajzolás. Mikor az ApplicationCore (3.10. részben több infó) rajzoló parancsot küld, akkor az időzítőt is frissíti.

3.9.3. Kiegészítő funkciók

A legnagyobb osztály a Kiegészítő funkciók osztálya. Ez lényegében a GUI kiegészítése, de a könnyebb átláthatóság érdekében, külön lett ide szedve. Itt vannak elkészítve a fájlkezeléssel kapcsolatos függvények: Beolvasó és kiíró függvény, daraboló és értelmező. Mivel a program tömörített fájl formátumban tárolja az adatokat, ezért annak a speciális kezelése is itt található meg. Egy érdekes dolog található ezen függvények között. A Pygame kép átméretező függvénye roncsolja a kép minőségét, ezért a Python Imaging Library segítségével vannak itt a képek átméretezve, a mentéseket kezelő ablakhoz. Részletesen a 3.7.3 fejezetben leírva.

3.10. Működési alapelv



15. ábra, Fő program struktúra

Előző részben már fel lettek vázolva a fő alkotó elemei a programnak. A 14. ábra mutatja, hogy a nagyobb elemek hogyan kapcsolódnak egymáshoz. A Main ponton belép a program működni (ez így értelmes?), ami rögtön létrehoz egy ApplicationCore osztályt. Ennek az a feladata, hogy létrehozza a felhasználó adatait, ha előzőleg nem rendelkezett ilyennel, ez mellett létrejön egy GUI és Turtle osztály is. A program egy Model-View-Controller sémát követ. A Turtle osztály a modellje a programnak. Egyetlen feladata tárolni az adatokat, hol van a rajzoló nyíl a parancsok futása közben. A program View része a GUI egy része, és szorosan hozzá kapcsolódik a ScrollingPlane osztály. Itt történik az összes kirajzolás. A Controller meg a GUI másik része és az ApplicationCore maga. A GUI Controller és View részéhez tartozik még a DataManagementWindow osztály. Ez egy felugró ablak feladatát látja el. Mikor megnyitjuk, akkor lehet választani, mit szeretnénk csinálni a megadott memória területtel.

A program futásának első fázisa, hogy ellenőrizze, megvan-e az összes felhasználói fájl, ha nincs, akkor kigenerálja és elindul a program. Ez után várakozik a program felhasználói utasításokra. Mikor kattintunk az egérrel, leellenőrizzük az összes kirajzolt objektumot, melyik felett található az adott időben az egér. Mikor megvan a keresett objektum, egy kis eseménykezelőben fogja a program eldönteni mit csinál az eseménnyel. Ha a ScrollingPlane mező volt a kiválasztott, akkor belépünk az ott található eseménykezelőbe.

A ScrollingPlane több eseménykezelőt tartalmaz. Első a kattintások helyes feldolgozásáért felel. Mivel ez egy gyűjtő osztály, itt is át kell menni az összes kijelölt elemen, és megkeresni mire volt kattintva. Ha megtaláltuk, felírjuk egy változóba, ami bekerül egy másik eseménykezelőbe. Itt van feldolgozva az összes forráskóddal kapcsolatos esemény. Mikor rákattintunk egy parancsra, hogy be szeretnénk rakni a forráskód mezőbe, akkor lemásolódik az osztály. Ekkor egy érdekes hiba keletkezik, mivel a pygame.font.Font nem másolható le, ezért mielőtt a másolás megtörténik, törölni kell a font referenciáját, és mikor megtörtént a másolás, újra létrehozni azt. Képekkel hasonló a helyzet, újra be kell tölteni azt is másolás után.

Mikor már van kész kódunk és indítanánk a futtatást, az első dolga a programnak, hogy optimalizálja a kódot. Attól függetlenül, hogy ez egy valós idejű értelmező, gyors futásidőre szükség van, ezért első körben átmegy a program az összes parancson lineárisan lefelé, és ha talál Loop vagy LoopEnd osztályt, akkor azok tartalmaznak fordítás információt. Felírja a program, hogy az összekapcsolódó blokkok hol találhatók a forráskódban, továbbá azt is, hogy elől- vagy hátultesztelő ciklust használunk. Mikor elkészült a fordító adatok felírása, a program megkezdi a futást. Ha megváltozik a kódunk sorrendje, akkor újra kell a fordító függvénynek optimalizálnia a Ciklus parancsokat.

Miután megtörtént az optimalizálás, a program elkezdi feldolgozni a parancsokat egyesével. Megint átmegy rajtuk, ám itt mikor meghatározza a típusát, ha van paramétere a parancsnak, lekérdezi, és a GUI osztályon keresztül kommunikálni kezd a modellel. A modell lekészíti a kirajzolandó dolgok listáját, és továbbküldi kirajzolásra az adatokat. Mivel a FloodFill egy elég időigényes művelet, ezért csak egyszer számoljuk ki, és ezt tárolni kell a memóriában.

3.11 Program futtatása forráskódból

A Python egy scriptnyelv, ezért nem feltétlen van egy bináris futtatható fájlunk, amit két kattintással futtathatunk. Ezért szükséges a forráskódból indítani a programot. Mivel a Python interpretált nyelv, ezért futásidőben fogja úgyis kiértékelni a változókat. Amire szükségünk van a programom futtatásához, az egy Python 2.7-es interpreter. (Ezt megszerezhetjük a Python hivatalos oldaláról. [4]) A Python csomag tartalmazza a PIP csomagkezelőt. Linuxon és MacOS rendszeren nem biztos, hogy megtalálható a natívan telepített Python mellett, ezért Terminal segítségével megfelelő paranccsal telepíteni kell azt. A programot úgy terveztem, hogy minél kevesebb csomagot kelljen beszerezni. A végső változat két külső könyvtárra hivatkozik, amit a PIP segítségével kell feltelepíteni. Ezek a Pygame és a Pillow (Python Imaging Library). Telepítéshez használt parancs:

- **$ (sudo) pip install pygame**
- **$ (sudo) pip install pillow**

Miután rendelkezünk a csomagokkal, már futtatható a program. Menjünk a mappába, ahol található a VisualLogo.py fájl. Windowson adjuk hozzá a Python könyvtárát a környezeti változókhoz, hogy tudjuk majd használni parancssorban. Futtatáshoz adjuk ki a következő parancsot:

- **python VisualLogo.py**

Ha nem indul a program, próbáljuk törölni a UserData mappa tartalmát, majd próbálkozzunk újra.

A programot le lehet fordítani bináris állományra. Mivel ez nem része az alap Python-nak, ezért le kell töltenünk valamit, amivel ezt el tudjuk majd készíteni. Az általam használt csomag erre a Pyinstaller. Telepítése PIP segítségével történik. Több lehetőségünk van, hogyha szeretnénk létrehozni a futtatható állományt. Lehet statikus és dinamikus linkeléssel. A dinamikus az alapbeállítás, statikushoz használjuk a --onefile kapcsolót.

- **$ pyinstaller VisualLogo.py --onefile**

A Pyinstaller támogat Windows, Linux, és MacOS rendszereket. Figyeljünk arra, hogy ne legyen ékezetes betű az elérési útvonalban a lefordított állományhoz, mert furcsa hibákat fogunk kapni. Arra ügyeljünk még, hogy nem mindegy melyik utasításkészlettel fordítjuk le. Raspberry Pi ARM alapú, ezért nem fog rajta futni az X86 se a X64 alatt lefordított program.

3.12. Manuális kódírás

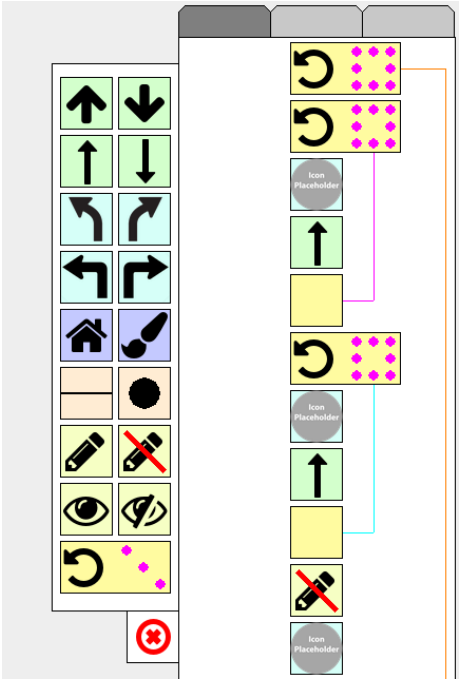
A program tervezése során úgy lett kialakítva, hogy akár kódolva is lehessen készíteni benne forráskódot, ne csak vizuálisan. Első lépésünk, ha nem ismerjük a szintaktikáját a programnak, akkor készítsünk egy fájlt úgy, hogy minden parancs szerepel benne, és mentsük ki fájlba azt. Ezután menjünk bele a UserData mappába és keressük meg az előzőleg készített fájlt. Csomagoljuk ki a .zip tömörített fájlt. Ekkor rendelkezésünkre áll két fájl, a .dat és a .jpg fájlok. A .dat fájl tartalmazza a parancsokat szövegesen leírva, míg a .jpg a program által megjelenített képet. Nyissuk meg a .dat fájlt bármilyen szövegszerkesztővel.

```
1  <class=Forward, mul=1>
2  <class=Backward, mul=1>
3  <class=Left, mul=2>
4  <class=Right, mul=2>
5  <class=Home>
6  <class=FloodFill>
7  <class=PenWidth, penwidth=0>
8  <class=PenColor, pencolor=(0, 0, 0, 255)>
9  <class=PenDown>
10 <class=PenUp>
11 <class=ShowTurtle>
12 <class=HideTurtle>
13 <class=Loop, loop_index=0, cycle_number=3>
14 <class=LoopEnd, loop_index=0>
15
```

16. ábra, Összes parancs szöveges ábrázolása, data0.dat

A 15. ábrán látható az összes parancs szintaxisa. Némelyikhez tartozik pár paraméter, amit lehet változtatni.

1. Forward, Backward parancsok „mul” paramétere adja meg, mennyit menjen előre a rajzoló nyíl. A program futása során a távot megkapjuk a mul * 10px képletből.
2. A Left és Right parancsok is tartalmaznak egy „mul” paramétert. Itt azonban ez fordulást jelent. A parancsok 15° többszörösei, és ezt lehet állítani a paraméterrel.
3. A PenWidth paramétere egy hozzávetőleges szám. A képet, amivel a program fogja kiszámolni a valódi vastagságot, a következő: (PenWidth × 2) + 1
4. A következő parancs a PenColor. Itt a paraméter egy RGBA kódot vár értékként, és azt direkt adja át a programnak. Itt bármilyen színt ki tudunk keverni. Előzőleg említve lett már, hogy tapasztalatom szerint az Alpha nem változtatja meg a színt, ezért ajánlott azt az értéket 255-ön hagyni.
5. Következő a Ciklus. Ezzel már több paraméterünk is van. Első a „loop_index”. Ezt helyesen kell beírni, mert különben a program szét fog hullani. Ez az azonosító az, ami segítségével meg tudja találni a hozzá tartozó befejező párt. A másik paraméter a „cycle_number”. Ez adja meg hányszor fog a ciklus lefutni. Ide ne írjunk 9-nél nagyobb számot, mert a program összeomlik.
6. A végső a LoopEnd. Itt csak a „loop_index” paraméter van, amit már előtte említettem. Fontos, hogy helyes legyen ez a szám, mert a program nem várt működést produkálhat.
7. Ha így létrehozott kódot szeretnénk a programban használni, csomagoljuk vissza a tömörített fájlba a képpel együtt, és már használható is. Ilyenkor megeshet, hogy pár ikon nem úgy fog kinézni, mint elvárnánk.



17. ábra, Saját kezűleg készített kód

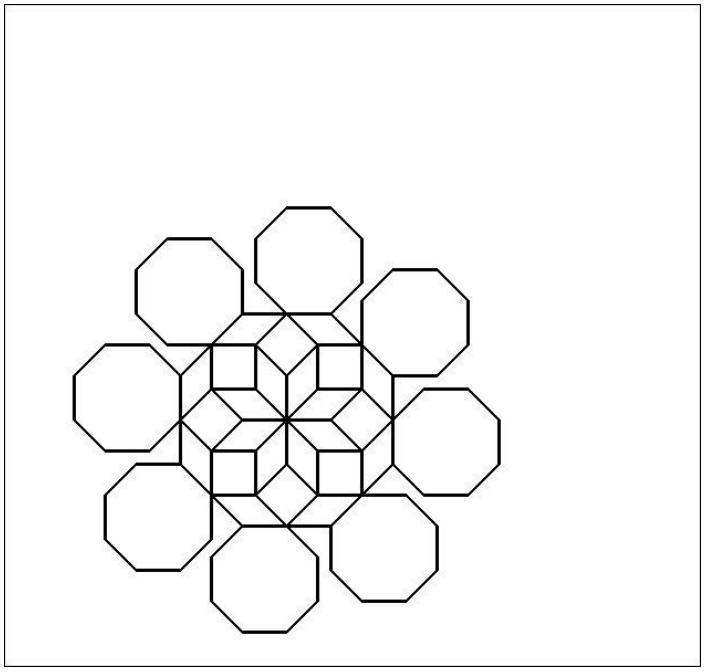
A 16. ábrán látható egy kép erre az esetre. Egy átmeneti kép kerül a helyére, de attól még le tudja majd futtatni a kódot a program.

3.13. Továbbfejlesztési lehetőségek

A program első verziója rámutatott arra, hogy miért nem Pythonban fejlesztenek játékokat. Az a tény, hogy futásidőben történik a változók kiértékelése, nagyon lelassítja a program futását.

- Ezért az első feladat az lenne, hogy az egész kódbázist átírjuk C++ nyelvre. Ott a fordító is segít, hogy minél gyorsabban fusson a kód.
- Következő lépés a FloodFill algoritmus gyorsítása lenne. Ez egy nagyon sok erőforrást és számítást igénylő algoritmus. Mivel tárhely részéről a program nem fogyaszt sokat, ilyen megközelítésből lehetne megközelíteni.
- Az Időzítőt is fejleszteni kellene, hogy ne a kirajzolt képkockák számától legyen függő a program gyorsasága. A Delta Time alapú megközelítést kéne alkalmazni.
- A funkcionalitását is lehetne továbbvinni, pl. fejtörőket lehetne készíteni. Az egyik felhasználó készít egy képet, és a másik feladata, hogy rajzolja ki ugyanazt.

3.14. Galéria



18. ábra, Egyik szebb kép, amit készítettem a program segítségével

3.15. Végső gondolatok

A munkám során arra a kérdésre kerestem a választ, miért nem elterjedtek a vizuális programozások. Ahogy egyre jobban kezdtem dolgozni a programon és a megvalósításán, érezhető volt, hogy sokkal több fejlesztés és munka kellett ahhoz, hogy egy minimális környezetet létrehozzak. Amíg a legtöbb nyelvhez elég egy szövegszerkesztő, addig egy vizuális programozáshoz nem csak a fejlesztőkörnyezetet kell elkészíteni, hanem egy egészen más típusú értelmezőt is. Ha ezek készen vannak, jön a következő lépés. A jelenlegi nyelvet bővíteni kell. Az általam készített program skálázhatósága elég alacsony. A program ott bukik meg, ha új parancsot kell hozzáadni, akkor nem elég csak létrehozni a hozzátartozó osztályt, hanem mindenhol az eseménykezelőt is ki kell egészíteni.

Az eseménykezelő dönti el, hogy mi fog történni, miután azonosította a parancsot. Ez egy olyan megoldás része, ahol a parancsok nem ismerik az útvonalat a modellhez, hanem muszáj egy másik osztályra hagyatkozni, ami megoldja ezeket.

Ebben a témában elég foghíjas az irodalomjegyzék, mivel nincs konkrét módszer, ami alapján el tudtam indulni, ezért volt szükséges, hogy a program írása előtt minél több működő megoldást vizsgáljak meg. Elemezni, hogy azok milyen módszereket használnak a különböző problémák megoldására. Miután elkezdtem fejleszteni a programot, sokszor megesett, hogy újra kellett tervezni a program struktúráját, mivel valahol túlságosan meggátolta az előrehaladást. Minden elismerésem az MIT és az Epic Games fejlesztőcsapatainak, mivel amit ők letettek az asztalra a programjaikkal, az kisebb csodának tekinthető. Az ő munkájuk nélkül az én programom sem jöhetett volna létre így ebben a formában.

A programom Python verzióját véglegesnek nyilvánítom a 1.2.2. verzióval. További fejlesztések már C++ nyelven fognak történni ebben a témában. Egy újragondolt és fejlettebb környezet létrehozása a cél, ami nem merül ki a Logó nyelvben, hanem további más nyelvek adottságait és megoldásait felhasználja. A Python nyelv egy nagyon jó nyelv a modellek tesztelésére, ebből tanulva már könnyebben lehet továbblépni.

4. Irodalomjegyzék

[1] http://stackoverflow.com/questions/38645391/python-pygame-error-failed-loading-libpng-dylib-dlopenlibpng-dylib-2-imag/39425923#39425923

[2] GitHub link

[3] Bitbucket link

[4] https://www.python.org/

[] Carol Vorderman: Programozás gyerekeknek, HVG könyvek, 2016, [369],

ISBN-978-963-304-320-2

[] Martin Flower: Refactoring Kódjavítás újratervezéssel, Kiskapu Kft., 2006, [224],

ISBN-963-9637-13-0

[] https://docs.python.org/2.7/ , Python dokumentáció

[] http://effbot.org/ , PIL és Tkinter segédanyag

[] http://www.pygame.org/docs/ , Pygame dokumentáció

5. Köszönetnyilvánítás

Szeretnék köszönetet nyilvánítani Dévai Gergely tanársegéd témavezetőmnek, amiért felajánlotta és vezetett a téma kidolgozásában és megvalósításában. Továbbá szeretném megköszönni Toma Zsófiának, Megyesi Attilának, Szitár Gergőnek és Bodrogi Zsófiának, hogy tesztelték a programot, és ötletekkel láttak el, hogy hol lehet csiszolni a részleteket.