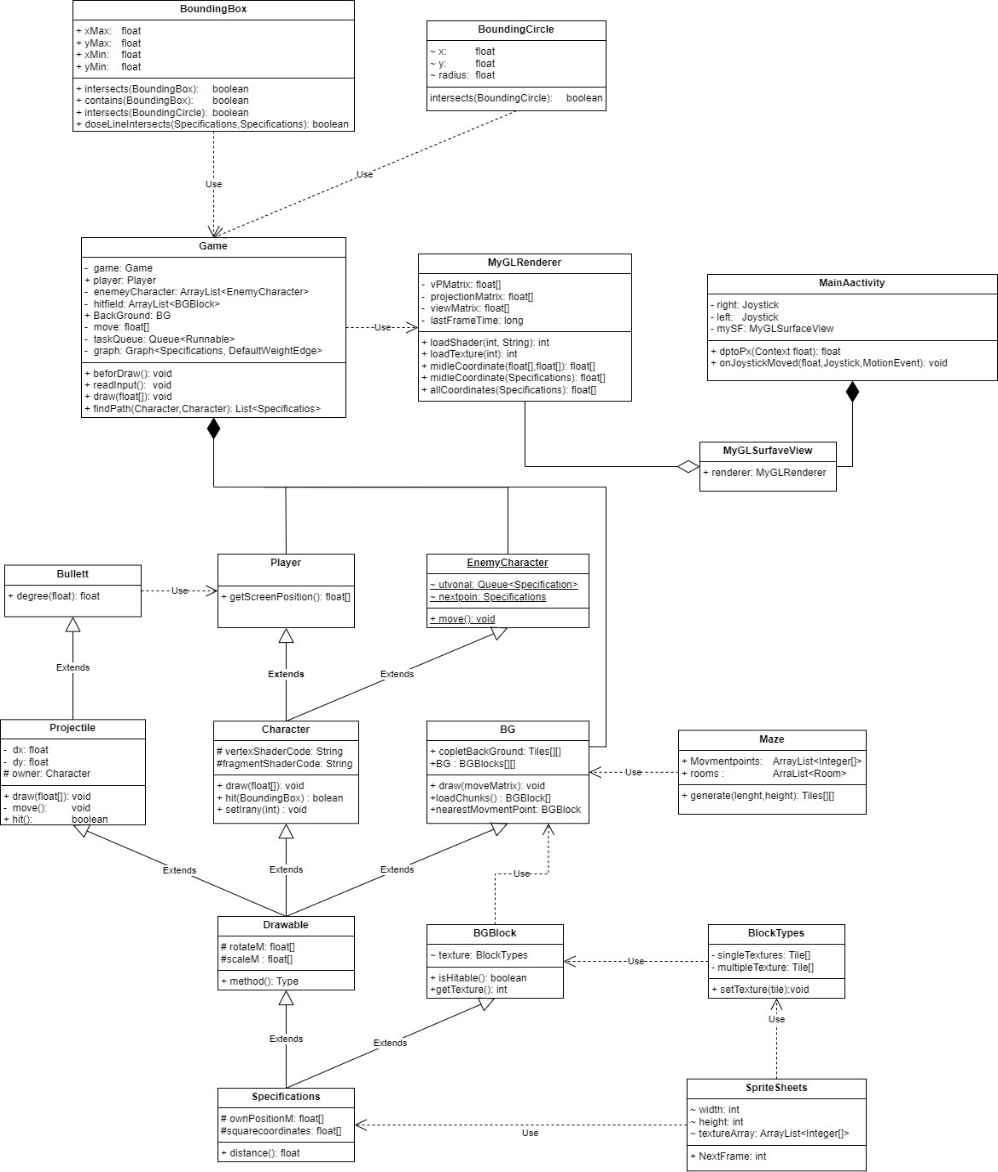
## -kód bemutatása

Az itt látott kódrészlet a Specific osztály 2 függvényé az egyik a distnace a másik a dxdy nevet viseli. A distance függvény, mint már említettem az objektumtól való másik objektum légvonal béli távolságot hivatott kiszámolni mind ezt az által, hogy a MyGLRendererben található midleCoordinate függvényt használva, ami visszatér a 2 objektum x és y koordinátán való elhelyezkedésével és ezek alapján egy egyszerű távolság számító függvénnyel kiszámítja ezen 2 objektum távolságát. Ezen függvény természetesen az öröklődés miatt működik mindegyik játékbéli objektumon. A distance metódus szolgált a fejlesztés elején az ütközések kezelésére viszont ezt idővel leváltotta BoundingBox illetve a BoundingCircle osztályok melyek precízebb és hibamentesebb megoldást nyújtanak. A másik függvény, ami még a képen látható az nem más, mint a dxdy függvény, ami főleg az ellenfeles objektumok miatt lett beépítve. Ugyan is ezen függvény főbb célja az, hogy a 2 dimenziós térben, ahol a játék zajlik. Ezen függvényen keresztül képesek legyünk megmondani, hogy a többi objektum milyen irányba helyezkednek el az adott objektumhoz képest. Kérdezhetnénk, hogy „Miért volt erre szükség?” amire csak annyi a válasz, hogy ezen a függvényen keresztül képesek vagyunk megmondani az ellenfeleknek, hogy merre találják a játékost, ha arra lenne szükség vagy esetleg a játékosnak kéne valami külső eseményre reagálni, ami megint csak ez alapján lenne megoldható. Ami még fontos függvény lehet ebben az osztályban   
az a getHeight függvény lehet, amit azon indokból írtam meg hogy ha a háttér és a benne található négyzetese elemek által teremtett négyzetrácsos rendszerben a programnak valamilyen eltolásra lenne szükség, viszont az eltolás akkor se teremtsen problémát, ha az adott négyzetes elemek mérete esetlegesen a fejlesztés folyamán elkezdene változni. A függvényben látható allCoordinates metódus, ami segít ezen megoldás eléréséhaz. Mert maga az előbb említett metódus annyit csinál, hogy a kapott objektumon végrehajtja a programban felirt átméreteződést ami ugye kihatással lehet az objektum méreteire. Az ebből a függvényből visszatérő koordinátákból kivontam azokat, amik a négyzet tetejének és aljának felelt meg majd ennek vettem abszolút értékét arra az esetre, ha ezen objektum valamilyen forgatási művelet folyamán fejjel lefele helyezkedne. A következő képen már a SpriteSheets osztály konstruktora található, ami 4 bemeneti paraméterrel rendelkezik, amik nem lennének mást hivatott reprezentálni, mint a spritesheet-hez való elérési utat az ezen található textúrák szélességét, magasságát és az ehhez tartozó FPS számot, ami egyszerűen csak annyit jelentene, hogy milyen sebességgel történjenek ezen textúrák közötti váltások a játék futása során. Ezen adatok letárolását követően történik az adott spritesheet beolvasása a BitmapFactory és Bitmap osztályok segítségével. Amikkel a folyamat úgy néz ki pontosan, hogy a bitmap nevű változóban letárolódik az egész spritesheet majd ezen bitmapből a magasságnak és szélességnek megfelelő négyzeteket vágunk ki, amiket azonnal be is töltjük az OpenGLES-be majd ezt követően az OpenGLES által hozzá generált id-t letároljuk soronként a spriteSheetArray-bea fent látható módon. Ezen módszert eredetileg helyettesíteni tudtam azzal, hogy az adott textúrákat kivagdostam külön png fájlokba majd ezeket betöltve az Android Studio-ba egyesével használtam őket. De az egyre növekedő textura mennyiségek és a konzulensem tanácsára elkészítettem ezen osztályt, amely megszüntette a további hosszadalmas és aprólékos megoldás szükségletét. A lejjeb(ábraszám) látható képen továbbra is a Spritesheet osztályt láthatjuk viszont ezek már a hozzá tartozó függvényeket, amik az adott spritesheet-ből kinyert képeket kezeli az előbb említett időleges textura váltogatással. Mint látható a képen az a nevű változóba töltődik be az előbb említett id-k ból a kívánt textura méghozzá a szerint hogy a paraméter listában kapott irany nevű változó 0,1,2,3 értékkel rendelkezik ha ezen változó olyan értékkel érkezik ami nem felel meg a betöltött textúrák mennyiségének akkor a függvény egyszerűen ignorálja az id lecserélését és az inicializált értékkel halad tovább ami nem lehet üres hiszen az osztály konstruktoraiból egyik sem rendelkezik olyan verzióval amely ezt megengedné hisz nem megfelelő sourceId-val indított inicializáció futásközben hibát dob. A metódusban láthatjuk még továbbá az előbb említett időleges váltakozást, ami úgy valósul meg hogy az adott texturán eltöltött időt számolja a függvény és ha ez meghaladj a konstruktorban megadott időt akkor a counter érték növekszik, ami által a következő textura történik betöltésre. Lejjebb még látható az ezen metódus paraméter lista nélküli változatát, ami csak a legelső textura elemet mutatja. Ezen függvényre azért volt szükség mert a tesztelés folyamán nem minden objektum kapott irányra és mozgásra megfelelő textúrát igy ezen osztály ilyenkor egy általános képpel töltődik be aminek ugye nincs szüksége az előbb említett funkciókra. A következő képen egy új osztály található már ami nem más, mint a Character osztály ezen osztály fő célja hogy a játékban létező olyan objektumoknak teremtsek egy olyan alapot ami alapjára leágaztatva többféle különböző karaktert tudok készíteni a játékomhoz egy igazán egyszerű módon. A konstruktorban a drawban megalapozott metódusok láthatóak, amik az OpenGLES futattásához és azzal való műveletvégzéshez készíti elő a környezetet. Továbbá látható még a képen a draw metódus, ami egy általános alapot képez az objektumok kirajzolásához. Ami ezekből a függvényekből talán fontos kiemelni az nem lenne más, mint a setvPMatrixHandel fügvény lenne ami azon mátrixok beállitására szolgál amikről már meséltem ide értve az objektum mátrixát amiben tárolva van az elmozdulása az átméretezés és esetélesegesn az elforgási paraméterek. Természetesen ezen mátrixok itt nem láthatóak hosz ezek mind el vannak rejtve ezen osztály ősosztályában. Ami még érdekes lehet ebben az osztályban az nem más mint az OpenGLES-ben használt vertex illetva fragment shader kódo ezen kódok hivatottak arra hogy az OpenGLES-es backend rendszer tudja értelmezni azt hogy az általa kirajzolt vertecies milyen pozicióban helyezkedjenek el és hogyan viselkedjenek az általaunk megadott paraméterek által. A fragmentShaderben lévő kód érteleme annyit jelent, hogy azon vertex-ek amiket ezen osztály generált azokon a megjelenő fragmentek valójában az általunk betöltött textúrák fragmentjei legyenek. Következő osztály, amit beszertnék mutatni ne legyen más, mint a Maze osztály melynek létrejöttét nem indokolja más mint hogy a játékomban való pálya generállást nem akartam mással csinálni mint egy labirintus generátorral viszont hogy ezt meg tudjam valósitani szükségem volt egy labirintus generálási algoritmusra. Amikből az interneten keresve rengeteget lehet találni ilyenek példásul az iteratív randomizált Kruskal algoritmus vagy az ugyan csak iteratív randomizált Prím algoritmusok, amik meglehetősen egyszerű utasitáso alapján hoznak létre különbövő labirintusokat mégis valami még ennél is egyszerűbbet szerettem volna, ami ez által remélhetőleg gyorsabb is lesz. Igy születtett a választásom a Wilson algoritmusra, aminek a használata részemről a legegyszerűbbnek tűnt. Hiszen ezen algoritmus ugy működik, hogy miután felvázoltunk egy négyzetrácsos rendszert, aminek az elemeit tekinthetjük a labirintus nem bejárható elemeinek. Ezt követően lehetőleg véletlenszerűen kiveszünk ezen elemekből egyet, amit a végleges labirintus elemévé teszünk. Majd ezt követően egy bejáró algoritmust használunk, aminek annyi feladata lesz, hogy a még nem látogatott elemekből egy kezdőelemet kiválasztva elinduljon egy általa véletlenszerűen kiválasztott irányba, amit addig folytat míg 2 lehetősségbe nem ütközik az egyik, hogy abba az útba fut bele, amin ő maga haladt végén ilyenkor a legutóbbi irányváltást újra sorsolja amíg nem olyan lépést választ, ami megfelelő. Vagy nem ütközik azon elembe, ami a hivatalos labirintusba tartozik. Ezt követően a bejárt utat hozzáadjuk a hivatalos labirintushoz majd a bejáró algoritmust ismételgetjük amíg nem fogy el az összes elem a nem bejárt elemek közül. Ezen algoritmust implementálva már csak egy fontos feladat volt számomra méghozzá az, hogy az igy megvalósított labirintus egy bejárható labirintussá tegyem, amit úgy képzeltem el, hogy az egyes blokkok helyét reprezentálja végleges labirintus úgy, hogy minden ilyen blokk helyén valójában egy n\*n-es szoba szerepeljen. Amihez természetesen megszüntette azon lehetőséget, hogy a programban egy statikus szoba másolásával történjen az elemek befejezése. Ezen probléma megoldására készült el a Room nevű osztálya, aminek az osztályváltozóit az (ábraszám) -on láthatnak. Amik között megtalálható egy 7\*7-es szoba beégetett változatát, ami a program korai változatában volt hasznos számomra hiszen akkor még nem volt teljesen kész az osztály viszont valamilyen módon tesztelnem kellet, hogy a további elemek, amik a szoba elhelyezkedésétől függtek rendesen tudtak működni. Ez viszont idővel átalakult mivel nem akartam törölni ezen elemet azon indokból, hogy ha fejlesztés során valamilyen olyan probléma jönne létre a háttérrel, amiben nem vagyok biztos, hogy maga a generálás lenne a hibás érte akkor egyszerűen elkezdtem ezen 7\*7-es szoba változót használni, ami alapján a hiba könnyeben megtalálhatóbb volt. Ami még észrevehető az ábrán, hogy tartalmaz egy vertical- horizontalConnection változót amire a kódban később lesz szükség. Illetve van még 1 matrix, corners és blocks változók, amikre majd a szoba betöltésekor lesz szükség ugyan is ezeket felhasználva határozza meg a program, hogy a játékos karakter ütközik-e a szobában lévő falakkal, de erről majd később. A következő ábrán a Room osztály egyik függvénye található, ami arra szolgál, hogy a Maze nevezetű osztályban a véglegesen elkészült szobák között megfelelő átmenetet tudjak létre hozni attól függetlenül, hogy valójában mekkora a szobák mérete. Ugyan is eredetileg a háttérnek szolgáló 2 dimenziós mátrix létrehozatalakor az első lépés nem más, mint hogy az adott szobák belekerüljenek ezen szobák, ami valahogy úgy néz ki, mint ami az (ábra) bal oldalán látható. Majd ezen szobákból a program a már említett függvény alapján felül írja azon elemeket a szobának, amelyekre nincs szükség igy kapja meg a (ábra) jobb oldalán látható végeredményt. Ezzel párhuzamosan a folyosók 4 sarkában lévő 4 háttérobjektumnak az elmozdulási mátrixát a program elmenti azon indokból, hogy majd ezeket felhasználva a játékban lévő ellenfelek útvonalat tudjanak majd keresni a játékkos által irányított karakterhez. Ezen pontok úgy lettek kijelölve, hogy futás közben az ellenfelek karakterek ne ütközzenek semmilyen objektummal viszont ez a megoldás nem problémamentes hiszen a mozgásba nincs betervezve az, hogy olyan objektumokat is ki tudjon kerülni az ellenfél amelyet a szobában megtalálhatóak. A következő ábrán látható azon kódrészlet, amely azért felelős, hogy a futás közbeni változó szoba mérettől független legyen a szoba generálása. Ezen probléma megoldására olyan rendszert hoztam létre, amely a szoba négyzetrácsos rendszeréhez indexeket ad majd ezeket maradékos osztással leosztva megkapjuk a szobában szükséges elemeket, amiket a feltételek alapján beállítunk. Majd ezeket a generált sorokat hozzáadjuk a roomNxN változóhoz, ami futás közben tárolja a szoba reprezentációját. A Room osztállyal kapcsolatban, ami még felmerülhet kérdés az a Tile enum használatát érintheti. Aminek a hozzárendeléshez szükséges kód részletek a BlockType osztályban találhatóak meg. A BlockType osztáylról amit szükséges tudni az az hogy ezen osztályon belül fel vannak sorolva az android stúdió által fenntartott resource id-k amikethez a hozzárendelést egy switch case használatával tudtam megoldani. Hiszem az android stúdió nem tud szolgálni olyan megoldással a resource-ök beolvasására, ami által ezen PNG fájlok beolvasása automatizálható lehetne. Ami ebben az osztályban még érdekes lehetne az nem más minthogy itt kapnak különböző jelzőket az adott textúrák úgy, mint a későbbi kód részletekben észrevehető hitable változó, ami alapján a program képes megkülönböztetni azt, hogy a játékos karakter milyen háttér objektumokkal képes ütközni. Természetesen ezen változók listája bővülhet a szerint, hogy milyen további igényre lesz szükség a játék fejlesztése során. A következő osztály, ami igencsak fontos a projekt szempontjából az nem más, mint a BG osztály. Ezen osztállyal valóbb elvárás nem más, mint hogy az eddig bemutatott BGblock osztályt úgy legyen képes használni, hogy ezen osztályból készített mátrix megfelelő reprezentációja legyen a játékban használt háttérnek. Ezen elvárás egyik nagyobb problémája nem más, mint az egyes háttér elemek mozgatása és adott esetben való betöltése az ütközési változókhoz. A következő ábrán észrevehető, hogy ez nem más mint a már említett BG osztály konstrukcióra ami kezdet kezdetében megegyezik a karakter aztán kezdetével ezen ismétlődés indoka nem más mint hogy az osztály a Dowable leszármazottja. Viszont a további kód igencsak érdekes hiszen az itt található sorok veszik át a használatban lévő maze osztályból szükséges elemeket, amik nem mások, mint a Movementpoints a room-ok és a sizeup paraméter, amikre a program további részében szükségünk lesz. Ami még továbbá észrevehető az nem más, mint az adott szobáknak való sarkainak a beállítása ezen sarkok beállítása nem történik máshogy, mint a szobának 4 sarkának megfelelő 4 elemhez való háttér objektum pozíciójának hozzárendelésével. Továbbá ami még itt történik az nem más, mint az ezen szobákhoz való blocks ArrayList feltöltése. Majd ezt követően a játék az osztályban való Move mátrix pozíciójának beállítása, ami a kezdő értékhez állítja be a hátteret. Amit még talán fomntos meglitani az nem más, mint a LoadUpBG függvény, ami nem más, mint a maze osztályból kinyert háttér adatokból t örténő objektum generálásért felel. Ezen metódus több más belső függvényt takar, amelyre a kód átláthatóságáért volt szükség. Kezdetben, mint láthatjuk a metódus végighalad a teljes kétdimenziós háttér mátrixon mindegyik elemre meghívva a setTexture függvényt. Ezen függvény ezek után létrehoz egy BGblock elemet, amit aminek ezután a mátrixát beállítja a getHeight függvénnyel. Ezen függvényben azért van szükségünk mind az i illetve a j változóra hogy ezek alapján el tudjuk mozdítani az objektumot az OpenGLES által használt koordináta rendszeren majd ezt követően be is állítjuk a textúráját. Ezen függvények megvalósítását láthatjuk (ábrán). A további függvények, amik még szerepelnek ebben az osztályban az nem más, mint a loadChunks, nearestMomentPoint, draw illetve a LoadGraph. A loadChunks függvény látható (ábrán) észrevehetünk egy egész nagy ki kommentett részt ami az eredeti ütközés érzékeléséhez tartozott. Ezen függvény megértéséhez először is meg kell említenem hogy a már bemutatott Maze osztályban nem csak a mozgási pontok állítottam be hanem minden egyes szoba valójában tartalmazott egy betöltési pont gyanánt szolgáló koordinátát. Majd ezen koordináták alapján és a már felvázolt distance függvény segítségével határoztam meg hogy melyek azok a fali elemek amikre szükségem volt. Viszont ezen megoldás rengeteg problémát vetett fel először is az adott karakternek egy adott körön belül kellett lennie hogy maga a szoba betöltődni viszont ez azt is jelentette hogy maga a marikának nagyobbnak kell lennie mint az említett szobának hiszen a sarkokban inkonzisztens vállik a betöltés és olyan szobák töltődhetnek be játék közben amire nincsen szükségünk. Ezen fő probléma megoldására született a nem ki kommentedet részben észrevehető függvény ami az adott szobákban már legenrált sarkok alapján érzékelődik hogy a játékos melyik szobában vagy illetve szobákban tartózkodik majd amint ezen feltételeknek bekövetkeztek egy egyszerű függvénnyel kiválogatjuk azon elemeket a szobában amik a falakat reprezentálják. Ezen betöltési módszert alapötletét a játékokban használt BVH betöltési módszer nyomán találtam ki. Ugyanis a BVH vagy más néven Bounding Volume Hierarchy úgy működik hogy a kívánt játéktérben való keresést mindig ellenőrizzük egy nagyobb kezdeti négyszögből hogy azon négyszög tartalmazza-e a kívánt objektumot majd ezt követően ezen négyszöget megfelezzük és tovább ellenőrizzük ezen négyszögek tartalmazzák-e a kívánt objektumot. Ezt a keresés formát egy nagyon gyors és egyszerű módszernek tartják hiszen a terület felezésével halad tovább hogy megtalálja az adott objektumot, ami által a függvény O(log2(n)) komplexitást ér el. A következő metódus amit még az osztály tart almaz és látható az (ábrán) az nem más mint a LoadGraph függvény. Ezen metódus alapjául nem szolgál más mint a JGraphT ami egy külső Java könyvtár aminek a főszerepe hogy a gráfokal való munkát megkönnyítse. Azt ez alapján inicializált gráfot megtaláljuk az osztálynak a tetején a többi változóval együtt. Ezen inicializáció erre a gráfra súlyozást vezet be az élekre a csomópontok távolsága alapján. Majd ezt követően az adott movemtnpointok között történik egy vizsgálat azon indokból hogy ellenőrizze ezen csomópontok között fenn áll-e a falakkal való ütközés. Amint ezen feltételről megbizonyosodik a program a 2 csomópont között kapcsolatot alakít ki. Fontos megemlíteni hogy ezen gráfban még nem szerepelnek sem a játékos sem az ellenfelek hiszen ezen objektumok mozgó objektumoknak minősülnek ami által a gráfban való reprezentációja nem lenne pontos hiszen a csomópontokkal való összekötés során a széleknek nem lenne konzisztens súlyozása, a távolság változása miatt. Ezért ezen objektumok beillesztése futás közben történik amikor ezen objektumok útvonalat keresnek az adott végponthoz és miután ezen végponthoz való útvonalat megtalálták azt lementik és utána törlik magukat illetve végpontjukat. Viszont ezen függvényekre csak később esik bemutatás. Ami még fontos ebben az osztályban az nem más mint a nearestMovmentPoint metódus amit a (ábrán) láthatunk. Ezen metódus szükségességét az előbb említett probléma igényli hiszen futás közben az adott objektumoknak meg kell találnunk a legközelebbi mozgási pontját ahhoz hogy be tudjuk őket illeszteni a gráfban. Ezen függvény igen egyszerű hiszen csak így minimum elem kiválasztás függvény amiben alkalmazzuk a már említett Specification osztályban létrehozott distance függvényt. Az eljárás a kiválasztás követően vissza tér az adott elemmel amit a további fügvény felhasználhat. Ezt követően szeretném bemutatni a program egy meghatározó akár a program gerincének is nevezhető osztályt ami felelős a már többször is emlitett ütközés érzékelésért. Ez az osztály nem lenne más mint a BoundingBox osztály amiről felsznesen csak annyit tudok elmondani hogy a már bemutatott Specification és Room osztályokat tudja kezelni annak érdekében hogy a programban lévő összes elemet képes legyen használni. Az osztályt ugy kellene elkézelni mintha egy képzeletbeli négyzetett rajzolnánk a szükséges objektum köré és ezen objektumok egymásba esését vizsgálná meg.Ezen elképzelés megkönyitése érdekében egy vizuálisabb megközelitésre lehetünk figyelmesek a (ábrán). Az (árbrán) a BoundingBox osztály egyik konstruktora látható és az osztályváltozók : xMax, yMax, xMin, xMin,

# Kódolási kezdetek

Az első lépések a programozás elkezdésekor, mint ahogy tanultuk nem más, mint az igények felmérése. A saját igényeim a projekttel kapcsolatban nem voltak mások, mint hogy egy olyan rendszert használjak, ami nagyon minimális szinten könnyíti meg a célom elérését így rengeteget tudtam tanulni ezen rendszerek alapjairól miközben fejlesztek és ezzel párhuzamosan egész nagy mértékű befolyásom maradt a kódolásban is. Ezen szempontok alapján az egyik opció az volt, hogy a fejlesztést letudom C nyelven viszont ott már inkább a fejlesztés nagy része azzal telt volna, hogy a grafikai elemkért felelős részeket összeállítsam. Ezt pedig ki szerettem volna kerülni teljesen így születtet az az ötlet, hogy a projektet Androidos környezetben valósítom meg ahol beépítve van hozzáférésem az OpenGLES-hez ami egy elég jó minőségű grafikai segédprogram és ezen felül tudok használni Java-t mint programozási nyelvet is amiben a tapasztalataimat szerezem ez elmúlt egyetemi éveim során. Így nagyban letudtam csökkenteni a jövőbeli tanulási szükségleteimet az által, hogy a fejlesztési nyelv nem lesz teljesen új egyedi szintaktikai elemekkel és egyéb aprócska részletekkel, amik a fejlesztés során problémát jelenthetnek. Mind ezen döntéseket követően jöhettek a projekt érdemibb része, ami a grafikai részek megértését és használatának betanulást jelentené. Ezen részekben rengeteg segítséget kaptam konzulensemtől így ezen részek is letudható volt. Ami még a kezdeti kérdések közé tartozhatna az nem lenne más mint hogy az alap ötlet valóságban való átültetése aminek a végeredménye nem lenne más mint az (ábrán) található osztály diagramm.