



Eötvös Loránd Tudományegyetem
Informatikai Kar
Informatikatudományi intézet
Programozási Nyelvek és
Fordítóprogramok Tanszék

Háromdimenziós űrhajós játék C++ nyelven

Témavezető:

Pataki Norbert
Adjunktus, Ph.D.

Szerző:

Nagy Dániel
Programtervező informatikus BSc.

Budapest, 2024

1. Köszönet nyilvánítás	4
2. Bevezetés	5
2.1 Motiváció a témaválasztás mellett.....	5
2.2 A szakdolgozat témája.....	5
3. Felhasználói dokumentáció.....	6
3.1 A program telepítése és futtatása.....	6
3.2 Rendszerkövetelmények	6
3.3 Kezdőképernyő.....	7
3.4 Játékmenet	7
3.5 Irányítás és játék ablak	9
3.6 Ellenségek.....	11
3.7 Játékterek	13
3.8 Fejlesztési Rendszer	13
3.8.1 Fegyverek.....	15
3.8.2 Speciális felszerelések.....	16
4. Fejlesztői dokumentáció	18
4.1 Megoldási terv	18
4.2 Mappaszerkezet	18
4.3 Felhasználói esetek	19
4.4 Az OGLBase projekt / OpenGL alapok	20
4.4.1 BufferObject.....	20
4.4.2 VertexArrayObject	22
4.4.3 Mesh	23
4.4.4 ProgramObject	23
4.4.5 Camera	24
4.5 Központi fájlok	25
4.5.1 Main.cpp.....	26

4.5.2	CMyApp osztály	26
4.6	Időbélyeg	29
4.7	Ütközésvizsgálat	29
4.7.1	Axis-Aligned Bounding Boxes	29
4.7.2	A Gilbert-Johnson-Keerthi algoritmus	31
4.8	Az Entity osztály	35
4.9	Színterek	37
4.9.1	Talaj	39
4.10	Ellenségek	41
4.10.1	Az Update módszer	42
4.11	A játékos	49
4.12	Fejlesztési Rendszer	51
4.12.1	Fegyverek	52
4.12.2	Speciális felszerelések	53
4.12.3	Adattárolók	54
4.13	Részecskerendszer	55
4.14	Hangeffektek	57
4.15	A játék tesztelése	59
4.15.1	Egység tesztek	59
5.	Fejlesztési lehetőségek	61
6.	Összefoglalás	62
7.	Hivatkozások	63

1. Köszönet nyilvánítás

Ezúton szeretnék köszönetet mondani mindazoknak, akiknek segítsége és támogatása hozzájárult a szakdolgozatom elkészítéséhez.

Elsősorban a témavezetőmnek, Pataki Norbertnek tartozom köszönettel, aki segítőkész hozzáállásával, szakmai tanácsaival és a rendszeres konzultációk szorgalmazásával végig motiváltan tartott.

Hálával tartozom a családomnak, akik önzetlenül támogattak és hittek bennem az egyetemi éveim alatt. Szeretném megköszönni szüleimnek azt a rengeteg szeretetet és segítséget, amire mindig támaszkodhattam tanulmányaim során. Az ő támogatásuk nélkül nem készülhetett volna el ez a dolgozat.

Szeretném megköszönni páromnak a folyamatost biztatást és támogatást, amivel segített átvészelni a dolgozós hétköznapiakat. Köszönöm neki, hogy türelemmel és érdeklődéssel kísérte végig a munkámat.

Végül, de nem utolsó sorban, hálával tartozom a barátaimnak, akik rengeteg hasznos tanáccsal láttak el a dolgozat készítése során.

2. Bevezetés

2.1 Motiváció a témaválasztás mellett

A XXI. század gyermekei számára valószínűleg többnyire ugyanaz a dolog jelentette az első bepillantást a szoftverek világába, a videójátékok. Még ma is tisztán él a fejemben, hogy kisgyermekkoromban mekkora örömforrás volt, mikor hosszas könyörgés után édesapám odaadta nekem régi Nokia telefonját, hogy játszhaszak egy pár kört a Space Impact nevű ikonikus játékkal.

Akkoriban ez még varázslatnak tűnt számomra, nem tudtam mást elképzelni, hogy ez mégis hogyan működhet, csak a varázslatot. Többek között ez is vezetett el engem a szoftverfejlesztés irányába. Egyetemi éveim során megtanulhattam, hogy mi is van igazából a varázslat mögött. Tanulmányaim alatt lehetőségem nyílt bepillantani a számítógépes grafika világába is, amibe azonnal beleszerettem. Lenyűgöző volt először látni, hogy hogyan tudunk pusztán program kódból, a matematika segítségével élethű képeket kirajzolni a monitorra. Mindent meg akartam tanulni a témáról és ez a motivációm azóta se hagyott alább.

A témaválasztásnál nem volt kérdés számomra, hogy ezen a területen belül szeretnék valamit alkotni, ezzel is bővítve a tudásomat és tapasztalatomat ebben a témában. Már csak magán a konkrét programom kellett gondolkodnom, amikor eszembe jutottak a régi Space Impact-os emlékeim. Ez az emlék adta az ihletet a szakdolgozatom témájának kiválasztásához. Egy grafikus szoftver, ami feleleveníti ennek a nagy múltú videójátéknak az emlékét, modern köntösbe bújtatva. A játék neve Intergalactic.

2.2 A szakdolgozat témája

A szakdolgozat célja egy háromdimenziós megjelenéssel bíró űrhajós játék megvalósítása. A játékosnak egy űrhajót vezérelve kell különféle ellenséges objektumokat elpusztítani és minél tovább életben maradnia. Az ellenségek elpusztításával a játékos pontokat tud szerezni, amit felhasználhat az űrhajó és a fegyverzetének a javítására és fejlesztésére. Ezen fejlesztések az űrhajó megjelenését is módosíthatják. A felhasználónak lehetősége van választani több különböző játéktér közül. A játékban külső nézetten keresztül látjuk az űrhajót és az elpusztítandó objektumokat. A szoftvert C++ programozási nyelven implementálom, felhasználom az OpenGL könyvtárat a grafikai elemek megjelenítéséhez.

3. Felhasználói dokumentáció

3.1 A program telepítése és futtatása

A program Windows operációs rendszeren portable, azaz telepítés nélkül is futtatható. A mellékelt feltöltött zip fájlban található egy „Portable” mappa, ami tartalmazza a futtatható állományt („Intergalactic.exe”), melyre rákattintva a játék automatikusan elindul. Indítás előtt érdemes meggyőződni róla, hogy a számítógép grafikus kártyájának illesztőprogramja naprakész.

3.2 Rendszerkövetelmények

Az alkalmazás a legtöbb mai, átlagos teljesítményű, 64-bites Windows operációs rendszerrel rendelkező számítógépen gond nélkül futtatható, amennyiben a grafikus kártya támogatja legalább az OpenGL 3.0-ás verzióját.

A program az alábbi táblázatban látható hardver konfigurációkon lett tesztelve:

Processzor	Videókártya	Memória	Kijelző képfrissítési ráta	Átlagos FPS
Intel Core i7 12. Gen	NVIDIA GeForce RTX 3070 (24GB)	32 GB	165Hz	165
Intel Core i7 10.Gen	Intel integrált laptop GPU	8GB	60Hz	60
Intel Core i7 9.Gen	Intel integrált laptop GPU (4GB)	8GB	60Hz	60
Intel Core i5 7.Gen	NVIDIA GeForce GTX 1050Ti (8GB)	8GB	60Hz	60

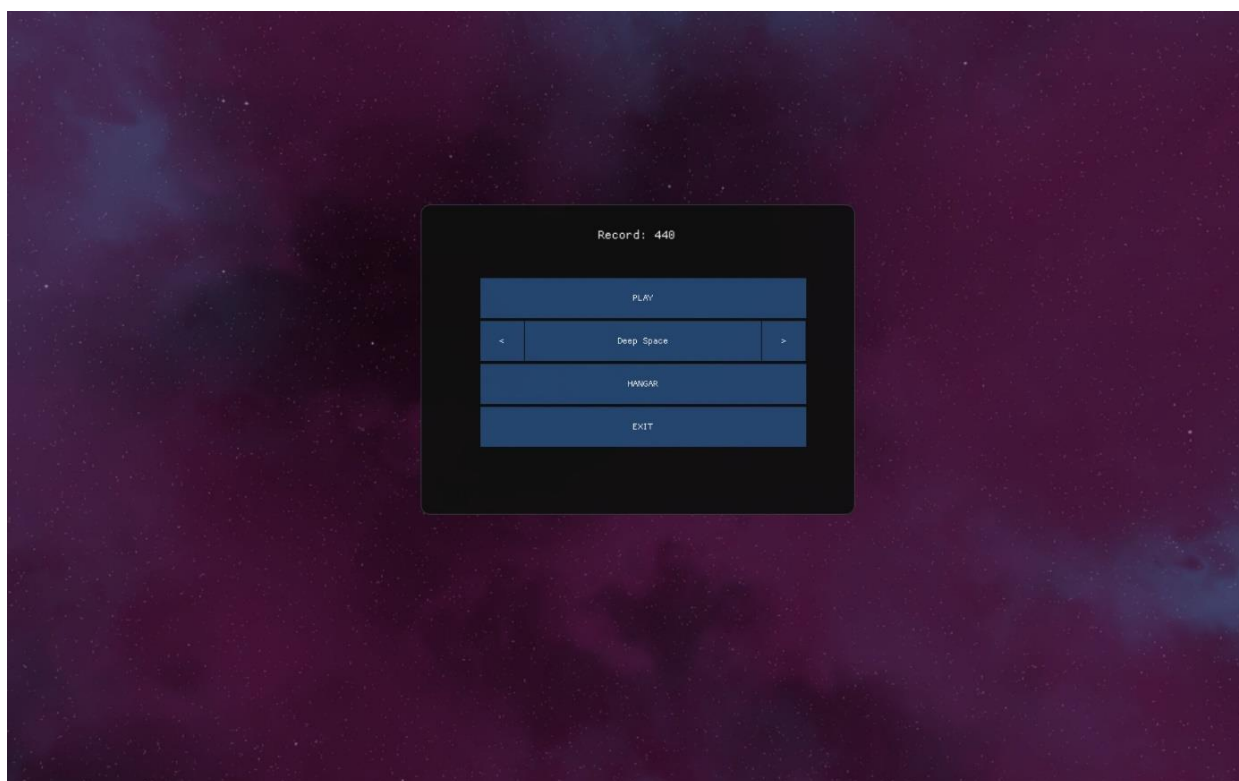
A szoftver alapértelmezetten vsync¹ mellett működik. A táblázat utolsó oszlopa a tesztek során megfigyelt átlagos FPS számot mutatja egy játék kör futása során. A tesztelésnél jóval több ellenséges objektumot indítottam, mint amivel egy rendes játék kör során találkozhatunk. A

¹ Kijelző technológia, ami szinkronizálja az alkalmazás kirajzolási sebességét a monitor képfrissítési rátájával.

teszteredményekben látható, hogy még ilyen körülmények mellett is maximális FPS-el futott a program minden kipróbált konfiguráción.

3.3 Kezdőképernyő

A játék elindítása után a főmenü fogadja a felhasználót. A kezdő ablak tetején látható a játékos által eddig elért maximális pontszám.



1. ábra: A kezdőképernyő

A főmenüben lehetősége van a felhasználónak kiválasztani az aktuális játékteret, és elindítani a játékot a választott pályán. A „HANGAR” gombra kattintva a program átnavigálja a felhasználót egy másik képernyőre, ahol az űrhajók fejlesztésére és testreszabására van lehetőség (3.8 Fejlesztési Rendszer). Az alkalmazást az „Exit” gomb megnyomásával tudjuk bezárni.

3.4 Játékmenet

A „PLAY” gomb megnyomása után elindul egy játék kör az aktuálisan kiválasztott pályán. Ebben a részben arról lesz szó, hogy mi a menete egy ilyen játék körnek.

A játék célja egyszerű, minél több ellenséges űrhajót kell a felhasználónak elpusztítania mielőtt a saját űrhajója megsemmisülne. Minden elpusztított ellenfél után 10 pontot kap a játékos. A felhasználó nem csak akkor kapja meg a pontot, ha ő maga lőtte le az ellenfelet. Előfordulhat

például, hogy szűk helyeken manőverezve az ellenség hibát vét, és nekiütközik egy akadálnak. A 10 pont ekkor is jár.

Minden játéktérben található valamiféle bázis, ahonnan bizonyos időközönként új ellenséges űrhajók fognak elindulni. Ezeket a bázisokat számos lövegtorony védi, amik automatikusan tüzet nyitnak a játékosra, ha az túl közel merészkedik hozzájuk.

A kör kezdete után egy folyamatosan szűkülő időkeret határozza meg, hogy mikor induljon a következő ellenséges űrhajó a bázisról. Kezdetben ez 20 másodperc, majd minden egyes ellenfél indulása után 1 másodperccel csökken. Tehát a második ellenfél 20, a harmadik 19 másodperc múlva fog indulni, és így tovább. Ha sokáig életben marad a játékos, akkor ez az időkeret el fogja érni az 5 másodpercet, ami után már nem fog tovább csökkeni.

Ez a dinamikus időkeret 4 szakaszra osztja a játékot. Minél későbbi szakaszában vagyunk a körnek, annál nagyobb az esélye, hogy gyorsabb, veszélyesebb ellenséges űrhajók induljanak a bázisról.

Az alábbi táblázat azt mutatja, hogy az ellenséges űrhajók indulási időkeretéhez mérten mikor fog új szakaszába érni a játék.

Időkeret	Játék szakasz
20-16	1.
15-11	2.
10-6	3.
5	4.

Ahogy telik az idő a játék közben, úgy lesz egyre nehezebb dolga a játékosnak. Egyre veszélyesebb és egyre több ellenséges űrhajó fog a nyomába eredni, így előbb vagy utóbb a túlerőben lévő ellenségek el fogják pusztítani a hajónkat és véget ér a kör.

A kör végeztével a felhasználó megkapja az összegyűjtött pontokat játékbéli fizetőeszköz („Credit”) formájában. Fontos azonban, hogy ez csak akkor teljesül, ha a felhasználó végig játszotta a kört. A pontok nem fognak jóváírásra kerülni, ha a játékos megszakította a játékot és visszalépett a főmenübe.

3.5 Irányítás és játék ablak

Játék közben az elsődleges bemeneti periféria az egér. A játékos az egér segítségével tudja kormányozni az űrhajót. A kurzor tulajdonképpen egy botkormány szerepét tölti be. Az oldalirányú mozgítás az űrhajó hosszanti tengelyére mért forgatást eredményez (roll), míg a kurzor fel- és lehúzása a keresztbemenő tengelyre mérve forgatja a hajót (pitch).

Amennyiben a játékos hirtelen, radikális irányváltást szeretne elérni, a következő billentyűket használhatja az irányváltáshoz (az alábbi magyarázatoknál minden irány a hajó saját tengelyeihez képes értendő):

- W: Lefelé dőlés
- S: Felfelé dőlés
- A: Balra forgatás
- D: Jobbra forgatás

A jobb egér gomb nyomva tartása közben szabadabban körbe tudunk nézni az űrhajónk körül, ez megkönnyíti az ellenségek felkutatását és követését.

Egy harmadik kamera módot is használhat a játékos a „V” billentyű nyomva tartásával. Ekkor a kamera pozíció megfordul és hátrafelé tud tekinteni a felhasználó. Ezáltal szemmel tudja tartani az őt üldöző ellenfeleket.

A „Q” billentyű nyomva tartása közben a hajó egyenesen fog tovább repülni, nem fogja követni az egér mozgást. Ez abban az esetben lehet hasznos, amikor a felhasználó a hajó haladási irányának megváltoztatása nélkül szeretne körbe nézni.

A játék kör közbeni felhasználói felület elrendezését a 2. ábra szemlélteti.

A képernyő felső részén láthatjuk rendre az FPS² számot, a játékidőt, illetve az aktuális pont állást.

A képernyő alsó részén lévő piros folyamatjelző sáv mutatja a hajónk állapotát/életerejét százalékos formában. Egy körnek akkor van vége, ha az itt látott érték 0%-ra csökken.

² Frame per Second: Az egy másodperc alatt kirajzolt képek száma.



2. ábra: A játék ablak

Az életerő sáv alatt láthatjuk az éppen felszerelt fegyvereket, piros kerettel és nagyobb mérettel megjelölve azt, ami aktív. A felhasználó az „1”, a „2” és a „3” billentyűk lenyomásával tudja kiválasztani, hogy melyik fegyver legyen aktív. Az aktív fegyvert a bal egérgombbal lehet elsűtni vagy aktiválni. Az egyes fegyverek különböző módon viselkednek, ez a (3.8.1) részben van kifejtve. Minden fegyver rendelkezik egy időkerettel, ami azt jelzi, hogy az elsűtést követően mennyi időre van szükség ahhoz, hogy újra használni tudjuk. A fegyverek képei alatt látható sárga folyamatjelző sávok jelzik a felhasználónak, hogy mikor tudja használni újra az adott fegyvert.

A képernyő bal alsó sarkában láthatjuk az űrhajónk aktuális sebességét, illetve a gázkar állását. A hajó sebességét kétféle módon tudja módosítani a játékos. Az egér görgő segítségével maradandóan tudja állítani a sebességet, felfelé görgetéssel növeli, lefelé görgetéssel csökkenti azt. A hajónk minimális sebessége 80, a maximális alapból 90, de utóbbit van lehetőség fejleszteni. A másik lehetőség a sebesség manipulálására a bal oldali CTRL billentyű nyomva tartása. A billentyű lenyomása után az űrhajó lelassul 80-ra, felengedése után visszagyorsul az eredeti sebességére. Minél lassabban halad az űrhajó, annál kisebb ívben tud megfordulni, így ezt a gyors manőverezésre lehet kihasználni.

A jobb alsó sarokban látható az adott speciális felszerelés ikonja. Amennyiben a felszerelés használható (az ikon alatt látható folyamatjelző sáv 100%-on van), akkor a bal oldali SHIFT billentyű lenyomásával aktiválhatja azt a játékos.

A játékot az ESC billentyű lenyomásával lehet szüneteltetni. Ekkor megjelenik egy ablak, amin a megfelelő gombra kattintva a játékos folytathatja a kört, vagy visszaléphet a főmenübe.

Az alábbi táblázat összefoglalja a hajó irányításához szükséges információkat.

egér	hajó kormányzása
bal egér gomb	tüzelés
jobb egér gomb (tart)	körülnézés
Egér görgő	sebesség állítása
W	lefelé dőlés
A	ballra forgatás
S	felfelé dőlés
D	Jobbra forgatás
Q	egyenesen repülés
V (tart)	Hátra nézés
LCTRL (tart)	lassulás
LSHIFT	speciális felszerelés aktiválása
1	első fegyver
2	második fegyver
3	harmadik fegyver


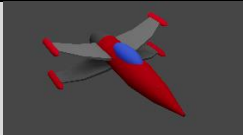
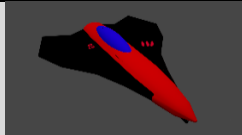
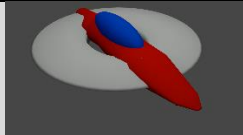
3.6 Ellenségek

Játék közben 4 fajta ellenséges űrhajóval találhatja szembe magát a felhasználó. Ezeket a kinézetükön kívül más tulajdonságok is megkülönböztetik egymástól:

- Életerő
- Sebesség
- Sebzés
- Mozgékonyosság (Mekkora ívben képesek fordulni)
- Tüzelési szög (Haladási irányuktól milyen mértékben képesek eltérni amikor céloznak)
- Tüzelési távolság (Milyen messziről képesek becélozni a játékost)
- Tüzelési ráta (Milyen gyorsan képesek tüzelni)
- Lövedék fajtája

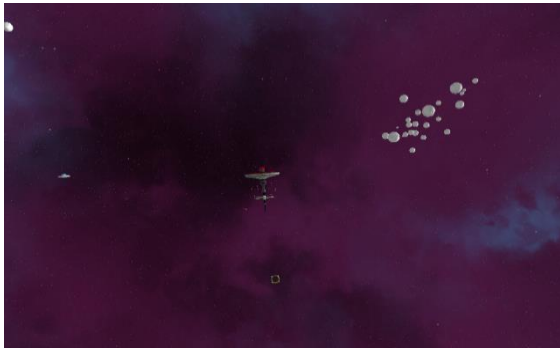
Az ellenségek viselkedése típustól függetlenül megegyezik. Alap esetben a játékos felé tartanak és megpróbálnak mögé kerülni. Majd, ha elég közel kerülnek hozzá és be tudják célozni, akkor tüzet nyitnak rá. Verthelyzetbe kerülve megpróbálják kicselezni a játékost, illetve, ha akadály kerül az útjukba, akkor legjobb tudásuk szerint próbálják kikerülni azt.

A következő táblázat bemutatja a 4 fajta ellenségek űrhajót:

	 3. ábra: SaR	 4. ábra: Falcon	 5. ábra: Raptor	 6. ábra: Rocketeer
Életerő	100	150	200	150
Sebesség	100	120	180	100
Sebzés	10	10	20	100
Mozgékonyosság	8	10	13	9
Tüzelési szög (Radián)	0.1	0.2	0.3	0.5
Tüzelési távolság	200	250	300	400
Tüzelési ráta (1 / mp)	4	4	5	0.1
Lövedék	Lézer	Lézer	Lézer	Hőkövető rakéta

3.7 Játékterek

A játékos a főmenüben kiválaszthatja, hogy melyik játéktérben szeretne játszani. Bár a különböző pályák főleg esztétikai célt szolgálnak, az azokon lévő objektumok és a színtér kialakítása lehetőséget nyújthat egyedi taktikák használatára is. Például a 7. ábrán látható „DeepSpace” nevű pályán az aszteroidamező nyújthat fedezéket a játékos számára, a 8. ábrán látható „Planet Earth” játéktérben pedig a földfelszínhez közeli repülés segítheti ki a felhasználót, hiszen az ellenséges űrhajók nem merészkednek túl közel a talajhoz.



7. ábra: DeepSpace pálya



8. ábra: Planet Earth pálya

3.8 Fejlesztési Rendszer

A felhasználó a játék során összegyűjtött kreditjeit felhasználhatja az űrhajójának a fejlesztésére és módosítására. Ezt a 9. ábrán látható hangár ablakon keresztül teheti meg. Itt a játékos a bal egér gombot nyomva tartva és az egeret mozgatva tudja körbe nézni a hajót. A jobb felső sarokban látható a játékos kreditjeinek a száma.

A játékosnak kétféleképpen van lehetősége fejleszteni a repülőjét. Fejlesztési pontokkal, amikkel a hajó egyes tulajdonságait tudja feljavítani, illetve új fegyverek és speciális felszerelések megvásárlásával és felhelyezésével.

Az ablak bal oldalán találhatóak a fejlesztési pontokhoz tartozó műveletek. A játékos 200 kreditért tud vásárolni 1 fejlesztési pontot az „upgrade points” melletti gombra kattintva. Amit aztán felhasználhat az alábbi tulajdonságok egyikének fejlesztésére:

- SPEED: Az űrhajó maximális sebessége
- MOBILITY: Fordulási sebesség
- HEALTH: Maximális életerő
- DAMAGE: Az alap fegyver sebzése
- FIRE RATE: Az alap fegyver tüzelési sebessége

Minden attribútumra legfeljebb 10 fejlesztési pont rakható. A felhasználónak meg kell fontolnia azonban, hogy melyik tulajdonságokat akarja fejleszteni, mert fejlesztési pontokból maximum 25 darab vásárolható. Ez azt jelenti, hogy az elméletben elkölthető 50 pontnak csak a felét tudja kihasználni a játékos. A felhasználónak tehát el kell döntenie, hogy milyen irányba szeretné a hajóját fejleszteni. Egy gyors, mozgékony, de kisebb tűzerővel rendelkező és sebezhetőbb repülőt szeretne, vagy inkább egy lassabb, de strapabíróbbat. Ez a dizájn lehetőséget ad a kísérletezésre, és a játékos saját játékstílusának felfedezésére.

Ezek a fejlesztések pusztán a „motorháztető alatt” fejtik ki hatásukat, vagyis a hajó megjelenését nem befolyásolják.



9. ábra: Hangár ablak

Az 9. ábrán látható ablak jobb oldalán találhatóak a megvásárolható fegyverek és speciális kiegészítők, amiket lehetőség van felszerelni a hajóra. Egy felszerelés ikonja fölé helyezve a kurzort, megjelenik annak rövid leírása és attribútumai. Ha egy fegyver, vagy kiegészítő ikonja szürke, az azt indikálja, hogy még nem birtokolja azt a játékos. Ebben az esetben az ikonra kattintást követően megjelenik egy ablak, ahol a felhasználó láthatja a felszerelés árát, illetve megvásárolhatja azt.

Amennyiben egy fegyvert vagy kiegészítőt már megvásárolt a játékos, akkor „drag and drop” módszerrel beillesztheti a képernyő alján lévő kék keretes rekeszek egyikébe. A fegyvereket a

középső két helyre, a kiegészítőket a jobb oldali rekeszbe lehet beilleszteni. A két cserélhető fegyver rekesze között látható egy lézerlövedéket ábrázoló ikon, ami a 10. ábrán látható. Ez a hajó elsődleges és alap fegyvere, amit nem lehet lecserélni. A másik két fegyver szabadon választható, sőt akár még ugyanazt a típusú fegyvert is felszerelheti a játékos mind a két helyre.

Ennek több előnye is lehet, az egyik a kihűlési idő, amivel minden fegyver rendelkezik. Ez határozza meg, hogy mennyi időnek szükséges elteltie, amíg újra lehet tüzelni egy adott fegyverrel. Így tehát ha ugyanazt a típusú fegyvert szereli fel a játékos mind a két lehetséges helyre, akkor lehetősége van akár rögtön egymás után elsütni őket.

Ezek a fejlesztések vizuálisan is megjelennek a hajón, a fegyverek esetén a bal oldali rekesz a bal, a jobb oldali rekesz a jobb szárny alá szereli fel az adott fegyvert. A speciális kiegészítők elhelyezkedése egyedileg változik. Egyes fegyverek képesek a hajótól függetlenül is mozogni/célozni, ezeknél taktikai különbséget is jelenthet, hogy melyik szárny alá vannak felszerelve, hiszen a mozgási tartományuk le van korlátozva olyan szinten, hogy a felhasználó saját hajóját ne tudják eltalálni. Ez lehet még egy ok arra, hogy ugyanazt a fegyvert szerelje fel mind a két oldalra.

Egy felszerelt kiegészítőt a rekeszre kattintva lehet eltávolítani a hajóról.

3.8.1 Fegyverek

- **Alap fegyver:** Az űrhajó elsődleges fegyvere két darab lézerágyú, amik a szárnyak



10. ábra: Alap fegyver ikonja

tővénél helyezkednek el. A felhasználói felületen a középső fegyver rekeszben helyezkedik el. Ez a fegyver az egyetlen, ami nem távolítható el a hajóról. Kezdetben 4 lövés/másodperces tüzelési rátával és lövedékenként 10-es sebzéssel rendelkezik, viszont ezt fejlesztési pontokkal lehet javítani a hangárban. A két ágyú szinkronban működik, elsütésnél mind a két oldal tüzelni fog. Ennél a fegyvernél nincs lehetőség szabad célzásra, tehát a hajónk aktuális orientációja határozza meg a tüzelési irányt.

- **Thermal Rocket Launcher:** Rakétavető, ami egy hőkövető rakétát lő ki a célba vett

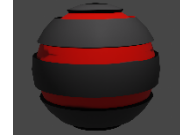


11. ábra: Rakétavető ikonja

ellenséges űrhajó irányába. A fegyver aktív állapotában automatikusan megpróbál bemérni egy előtte lévő, közeli ellenfelet. Amennyiben ez sikerül, a bemért ellenfél körül megjelenik egy piros téglalap szegély. A fegyvert csak akkor lehet elsütni, ha van bemért ellenfél. A kilőtt rakéta követni fogja a célpontot, azonban

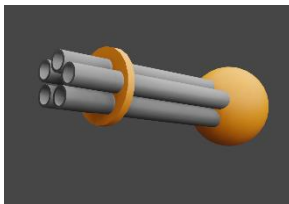
ez nem jelent garantált találatot, hiszen a lövedék nem tud akármilyen kis ívben megfordulni. Találat esetén 100 életerő pontot vesz el az ellenségtől. Elsütést követően 15 másodperc múlva tudunk újra tüzelni vele. A fegyver ikonja a 11. ábrán látható.

- **Mine Placer:** Elsütést követően egy aknát hagy maga mögött, ami felrobban, ha egy ellenséges űrhajó a közelébe ér. Az akna 150-es sebzéssel rendelkezik. A fegyvert tüzelés után 20 másodperccel lehet újra használni. A 12. ábrán látható ennek a fegyvernek az ikonja.



12. ábra: Akna ikonja

- **Machine Gun:** Egy gyors tüzelésű lézerfegyver szabad célzással, ikonja a 13. ábrán látható. Ez a fegyver abban különbözik az alap lézerágyútól, hogy a tüzelési iránya nem függ a hajó orientációjától. A kurzor aktuális helyzete felé tüzel, így lehetőség van szabadabban célozni vele, akár hátrafelé is. A tüzelési irány azonban nem teljes mértékben független a hajó helyzetétől, le van korlátozva annyira, hogy a felhasználó saját repülőjét ne tudja vele eltalálni. Másodpercenként 10 lövést képes leadni és a lövedékek egyenként 10 életerő pontot vonnak le az eltalált ellenféltől.



13. ábra: Machine gun ikonja

- **Turret:** Automata lövegtorony a hajóra szerelve. A többi fegyverrel ellentétben ezt nem elsütni kell, hanem aktiválni. Aktiválást követően 15 másodpercig automatikusan tüzelni fog a legközelebbi ellenségre. Természetesen ennek a fegyvernek is le van korlátozva a mozgása annyira, hogy a játékos űrhajóját még véletlenül se tudja eltalálni. Nagy előnye ennek az eszköznek, hogy miután aktiválja a felhasználó, át tud váltani egy másik fegyverre és szimultán használhatja azt. A fegyver 4 lézer lövedéket lő ki egy másodperc alatt, lövedékenként 10-es sebzéssel. A 15 másodperc lejárta után egy percet követően van lehetőség ismét aktiválni ezt a fegyvert. A fegyver ikonja a 14. ábrán látható.



14. ábra: Turret ikonja

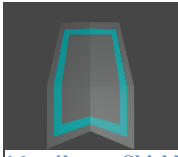
3.8.2 Speciális felszerelések

- **Speed Booster:** Aktiválást követően az űrhajó 5 másodpercig az aktuális végsebességének kétszeresével fog haladni. Az 5 másodperc lejárta után a sebesség visszaáll az eredeti végsebességre. Hasznos lehet, ha megszorodtak az ellenfelek a felhasználó mögött. A felszerelés 20 másodpercenként használható, ikonja a 15. ábrán látható.



15. ábra: Speed Booster ikonja

- **Shield:** Aktiválást követően 10 másodperc erejéig egy áttetsző energia pajzsot helyez



16. ábra: Shield ikonja

az űrhajó köré. Ez a pajzs blokkolni fog minden beérkező lövedéket, így lényegében sebezhetetlenné téve a játékost. Nagy taktikai előnyt jelent, hiszen a pajzs nem blokkolja a kifelé menő lövedékeket, így a felhasználó továbbra is képes lesz tüzelni az ellenségekre. Fontos azonban, hogy ez az eszköz csak a lövedékek ellen véd, az ütközés ellen nem. A 10 másodperc lejárta után fél perc elteltével vethető be újra a pajzs. A pajzs ikonja a 16. ábrán látható.

- **Stealth Coat:** Álcázó üzemmód, melynek ikonja a 17. ábrán látható. Miután aktiválja a felhasználó, 8 másodpercig a hajója áttetszővé válik, és láthatatlan lesz az ellenfelek számára. Az ellenséges űrhajók abba az irányba haladnak tovább, amerre az eltűnés előtt látták tartani a felhasználót. Ez idő alatt könnyedén az ellenségek mögé lehet kerülni. A 8 másodperc lejárta után a hajó újra láthatóvá válik, és az ellenfelek ismét célba tudják venni azt. A felszerelést 40 másodpercenként lehet használatba venni.



17. ábra: Stealth Coat ikonja

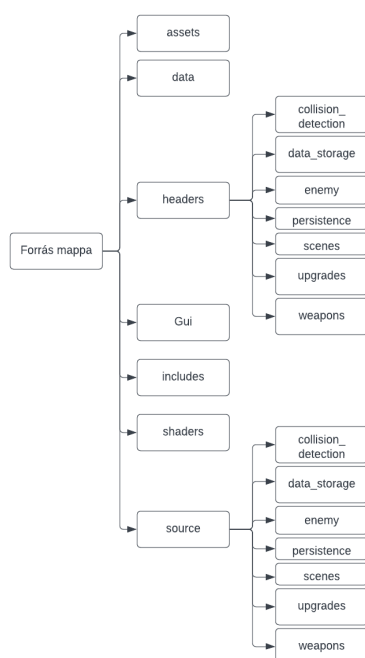
4. Fejlesztői dokumentáció

4.1 Megoldási terv

Az alkalmazás elsősorban 64 bites Windows operációs rendszeren futtatható. A fejlesztést Windows 11-en végeztem felhasználva a Visual Studio Community 2022 fejlesztői környezetet. A program implementálásához nem használtam játékmotort, a grafikus elemek megjelenítéséhez az OpenGL grafikus könyvtár 4.6-os verzióját használtam. A hardverhez való hozzáférést az SDL program könyvtár biztosította. A játékban lévő hangfájlok kezelésére és lejátszására SDL_mixer³ használok. A felhasználói felületet a nyílt forráskódú Dear ImGui könyvtár segítségével implementáltam. A játékban lévő modelleket és textúrákat a Blender modellező programban készítettem. A teszteléshez a Visual Studioba integrált CppUnitTestFramework keretrendszert használtam fel.

4.2 Mappaszerkezet

A 18. ábrán látható a fő projekt mappaszerkezete a forrás mappából (A Visual Studio projekthez képes egy szinttel mélyebben) kiindulva.



- **assets:** A játékban felhasznált modellek, textúrák, képek és hangeffektek mappája.
- **data:** Ide kerül a perzisztencia réteg által elmentett játékállás.
- **headers:** A C++ header fájlok mappája, értelemszerűen további mappákra bontva a nagyobb logikai egységek szerint.
- **Gui:** ImGui forrásfájlok.
- **includes:** Az OGL_base kiinduló projekt által definiált osztályok mappája.
- **shaders:** Az OpenGL által használt shader³ programok mappája.
- **source:** A C++ forrásfájlok mappája (.cpp). A headers mappához hasonlóan további részekre bontva

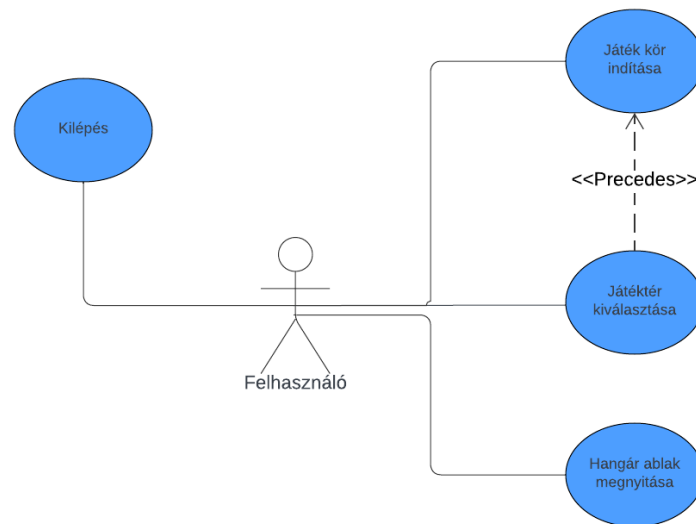
18. ábra: Mappaszerkezet

³ A grafikus kártyán futó program. Többnyire a grafikus szerelőszag programozható részeinek implementációja.

4.3 Felhasználói esetek

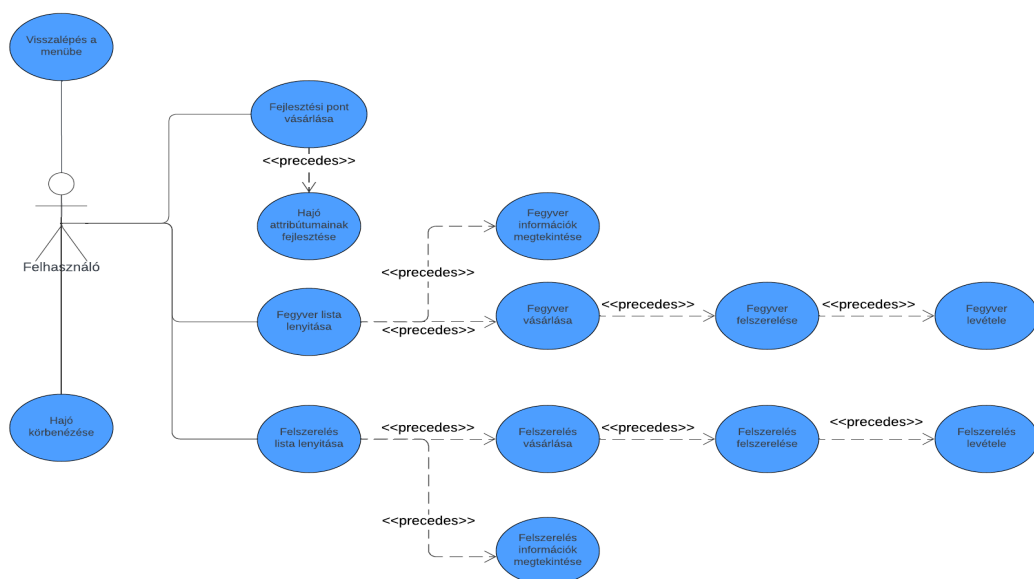
A felhasználó döntési lehetőségeit három főbb esetre lehet felbontani, lefedve ezzel a legfontosabb ablakokat a programban.

Az első eset a főmenü, a szoftver elindítása után ez az első dolog, amivel találkozik a felhasználó. A főmenü felhasználói eset diagramját a 19. ábra szemlélteti.



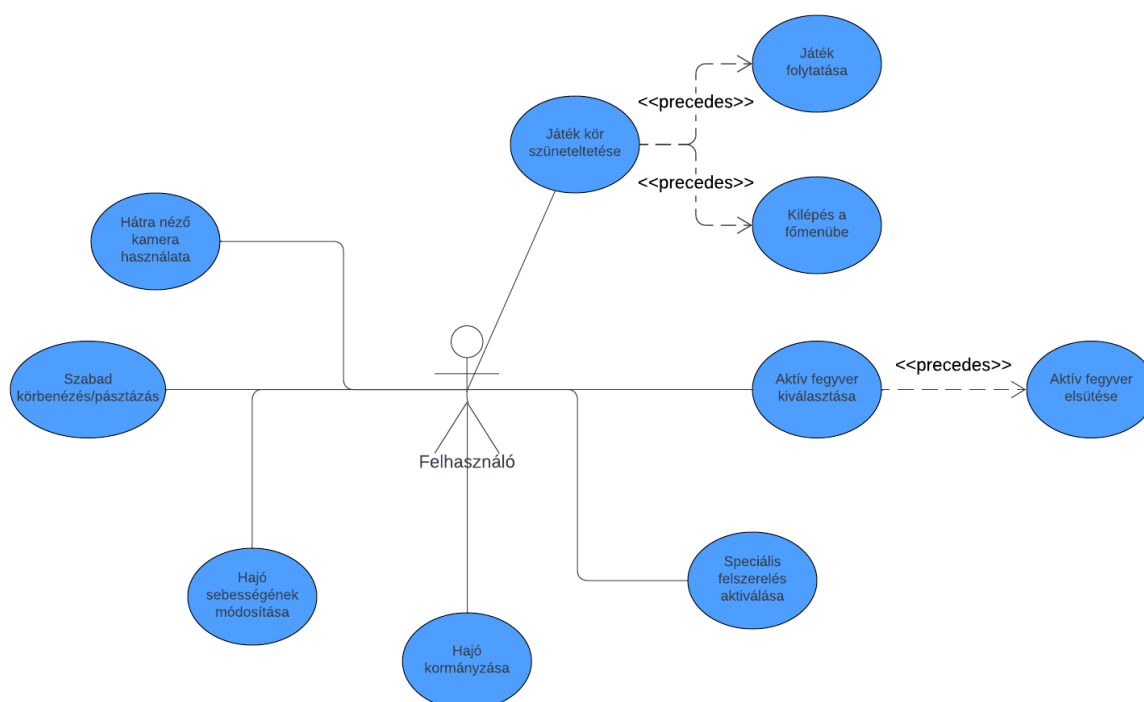
19. ábra: A főmenü felhasználói esetei

A főmenüből a felhasználó átnavigálhat a „Hangar” ablakra, ahol az űrhajó fejlesztésére van lehetőség. A hangár felhasználói eset diagramja a 20. ábrán látható.



20. ábra: A hangár felhasználói esetei

A 21. ábrán látható diagram magának a játék körnek felhasználói eseteit mutatja be.



21. ábra: A játék kör felhasználói esetei

4.4 Az OGLBase projekt / OpenGL alapok

Az alkalmazás kiindulási alapját az informatika kar számítógépes grafika laborja által készített OGLBase⁴ projekt adta. A projekt lényegében csomagoló osztályokat hoz létre a legalapvetőbb OpenGL-es funkciók köré, így megkönnyítve azok használatát. Ezen funkciók használata alapesetben eléggé körülményes és nagyban rontják a program olvashatóságát, így bármilyen komolyabb grafikus projekt készítésénél elengedhetetlen ezen funkciók absztraktálása. Ebben a részben az OGLBase projekt által definiált csomagoló osztályokat tárgyalom.

A projekt mappaszerkezetét tekintve (4.2) az „include” mappában találhatóak az OGLBase projekt osztályai.

4.4.1 BufferObject

Az OpenGL különböző típusú memóriabuffereket használ ahhoz, hogy adatot továbbítson a GPU felé. A buffer típusa határozza meg, hogy a grafikus kártya hogyan értelmezze a benne lévő adatot. A BufferObject egy template osztály, aminek paraméterként meg lehet adni a buffer típusát és használati módját. Az osztály konstruktorait használva akár egy sorral létre lehet

⁴ <http://cg.elte.hu/index.php/grafika-bsc-gyakorlat-anyagok-regi-bef2023/>

hozni tetszőleges buffereket. A két leggyakrabban használt típusra a könnyebb olvashatóság kedvéért template paraméterek nélkül is lehet hivatkozni. Ezek az ArrayBuffer és az IndexBuffer.

BufferObject
- m_id : GLuint - m_sizeInBytes : GLsizeiptr - g_lastbound : static GLuint
+ BufferObject() + ~BufferObject() + BufferObject(BufferObject&&) + operator=(BufferObject&&) : BufferObject& + BufferObject(vector<T>&) + BufferObject(array<T>&) + Clean() : void + Bind() const: void + BufferData(GLsizeiptr, const GLvoid*) + BufferSubData(GLintptr, GLsizeiptr, const GLvoid*);

Az ArrayBuffer, ahogy neve is sugallja, adatok egy tömbjét tartalmazza. Ez az egyik legáltalánosabb buffer és ezért talán a leggyakrabban használt is. Legtöbbször ezt a típust használjuk, amikor egy adott objektum vertexeit⁵ szeretnénk feltölteni a grafikus kártyára. Ahogy említettem, ez a buffer típus önmagában eléggé általános, és ez szükséges is, hiszen eltérő lehet, hogy milyen adatokat

szeretnénk egy darab vertexben eltárolni. Emiatt a buffer objektum létrehozása után specifikálni kell a tömb felépítését. Meg kell adni, hogy egy vertex hány attribútumból áll és ezek milyen típusú adatok. Egy ilyen attribútum specifikálására látható példa az alábbi kódrészletben:

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), 0);
```

A példa egy vertex első (0. indexű) attribútumait állítja be, ami jelen esetben a pozíció. Az első paraméter az attribútum index, majd meg kell adni, hogy hány, és milyen típusú adatról áll. Jelen esetben ez három darab float változót jelent (x, y, z koordináta). A harmadik paraméterben beállítjuk, hogy normalizált legyen-e az adat. Az utolsó két paraméter a tömbben való indexelést segíti. Itt kell megadnunk, hogy egy vertex mekkora (hány byte után következik a következő), illetve, hogy egy vertexen belül hány bájtot kell haladnunk, hogy ezt az attribútumot kiolvassuk.

Az ArrayBuffer csak azt az információt továbbítja, hogy egy adott objektum milyen vertexekből áll. Ez azonban még kevés a kirajzoláshoz, hiszen a grafikus kártya még nem tudja, hogy ezeket a csúcspontokat milyen sorrendben rajzolja ki, kösse össze. Ennek a specifikálására szolgál az IndexBuffer, ami csupán annyit közöl a GPU-val, hogy a feltöltött ArrayBufferban milyen sorrendben haladjon. Ez a buffer csupán egész számokból (intekből) áll, így jóval

⁵ Csúcspontok, amikből egy például 3D-s objektum felépül. Információt hordoznak az objektum adott pontjáról (szín, pozíció, normálvektor, textúra koordináta stb..)

memória hatékonyabb. Hiszen egy adott vertexet általában többször is felhasználunk a kirajzolás során.

4.4.2 VertexArrayObject

Az előző részben tárgyaltam azt a két buffer típust, ami alap esetben minimálisan szükséges egy objektum kirajzolásához. Az OpenGL egy állapotgép. Tehát amikor ki szeretnénk rajzolni valamit, előtte csatolni kell (bind-olni) a szükséges buffereket, kirajzolás után pedig lecsatolni őket. Amikor több ilyen buffert is használunk, akkor kényelmetlenné válhat, és nagyobb teret ad a hibázásra, ha mindegyiket egyenként kell kezelnünk. Ennek a folyamatnak a

leegyszerűsítése érdekében jött létre a VertexArrayObject, ami lehetővé teszi, hogy összekapcsoljuk az egy objektum kirajzolásához szükséges buffereket, és egyszerre kezeljük őket a VertexArrayObject-en keresztül. Ezután egy adott geometria/modell kirajzolása előtt elég csak ezt csatlakoztatni. Az OGLBase projekt ennek a használatát is

VertexArrayObject
- m_id : GLuint
+ VertexArrayObject() + ~VertexArrayObject() + VertexArrayObject(VertexArrayObject&&) + operator=(VertexArrayObject&&) : VertexArrayObject& + Init(Initializer_list<AttributeData, ArrayBuffer&>) : void + Init(Initializer_list<AttributeData, ArrayBuffer&>, const IndexBuffer&) : void + Bind() : VertexArrayObject& + Unbind() : void + AddAttribute(AttributeData, BufferObject&) : VertexArrayObject& + SetIndices(const IndexBuffer&) : VertexArrayObject&

egyszerűbbé teszi. A 22. ábrán látható kódrészlet egy példát mutat arra, hogy hogyan lehet használni ezt az osztályt arra, hogy a létrehozott buffereket összekapcsoljuk. A könnyebb olvashatóság kedvéért lehetővé teszi azt is, hogy rajta keresztül specifikáljuk az ArrayBuffer felépítését:

```
// geometria VAO-ban való regisztrálása
m_CubeVao.Init(
{
    { CreateAttribute< 0,
      glm::vec3,
      0,
      sizeof(Vertex)
    >, m_CubeVertexBuffer },
    { CreateAttribute<1, glm::vec3, (sizeof(glm::vec3)), sizeof(Vertex)>, m_CubeVertexBuffer },
    { CreateAttribute<2, glm::vec2, (2 * sizeof(glm::vec3)), sizeof(Vertex)>, m_CubeVertexBuffer },
  },
  m_CubeIndices
);
```

22. ábra: Vertex Array Object inicializálása

4.4.3 Mesh

A Mesh osztály összefogja az eddig tárgyalt különböző buffer objektumokat, hogy egy tetszőlegesen bonyolult háromdimenziós objektumot egyszerűen lehessen kezelni és kirajzolni.

Ez az osztály definiálja a programban használatos Vertex struktúrát is, ami ebben az esetben 3 féle adatot tartalmaz. Ezek a pozíció (3D vektor), normálvektor (3D vektor) és textúra

koordináta (2D vektor). Az osztályhoz

akár külön-külön is társítani lehet

Vertex (Array) és Index buffert, amit

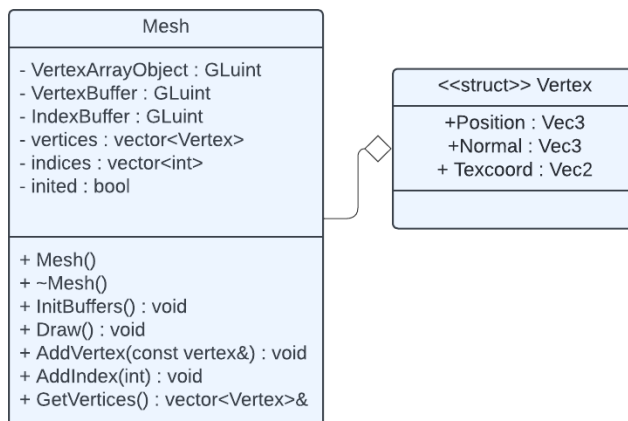
aztán az „InitBuffers” függvény segítségével lehet inicializálni. Ezt

követően a „Draw” függvény hívására

az osztály csatolja a hozzá tartozó

VertexArrayObject-et és meghívja a

megfelelő kirajzoló parancsot.



A Mesh osztállyal szorosan együttműködik az ObjParser osztály. Ez lehetővé teszi, hogy egy 3D modellező programban elkészített modellt betöltsünk egy Mesh objektumba. A parser osztály „obj” fájlkiterjesztésű objektumleíró dokumentumokat tud értelmezni. A „parse” függvénynek paraméterként megadhatjuk az adott obj fájl elérési útját, majd a függvény feltölti a különböző buffereket és összekapcsolja azokat egy VertexArrayObject-ben.

4.4.4 ProgramObject

OpenGL-t használva egy objektum kirajzolásához meg kell adnunk egy program objektumot, ami lényegében azt specifikálja, hogy milyen shader programokat használjon a grafikus szerelőszalag a kirajzolás során, illetve milyen bemeneti adatokra van ezeknek szüksége. A

buffer objektumokhoz hasonlóan

alap esetben viszonylag sok kód

szükséges egy ilyen program

objektum létrehozásához, majd

használatba vételéhez. Először is

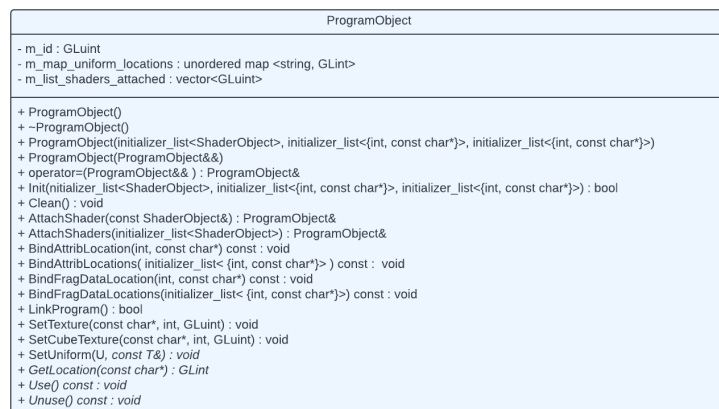
be kell töltenünk az adott

shaderek forráskódját, majd létre

kell hoznunk egy program

objektumot, amihez aztán hozzá

tudjuk társítani a betöltött shadereket. Aztán specifikálhatjuk a bemenő paramétereket. Majd,



ha ez mind megvan, még linkelnünk is kell a program objektumot. A ProgramObject osztály segítségével azonban akár két függvényhívással megtehetjük mind ezt, erre látható példa az alábbi kódrészletben:

```
m_programSkybox.Init(  
    {  
        { GL_VERTEX_SHADER, "skybox.vert" },  
        { GL_FRAGMENT_SHADER, "skybox.frag" }  
    },  
    {  
        { 0, "vs_in_pos" },  
    }  
);  
m_programSkybox.LinkProgram();
```

Miután inicializáltuk, és linkeltük a programot, a „Use” függvényt meghívva tudjuk aktiválni. A függvény meghívása után minden rajzoló parancs az adott programot fogja használni egészen addig, amíg meg nem hívjuk az „Unuse” függvényt, vagy aktiválunk egy másik program objektumot.

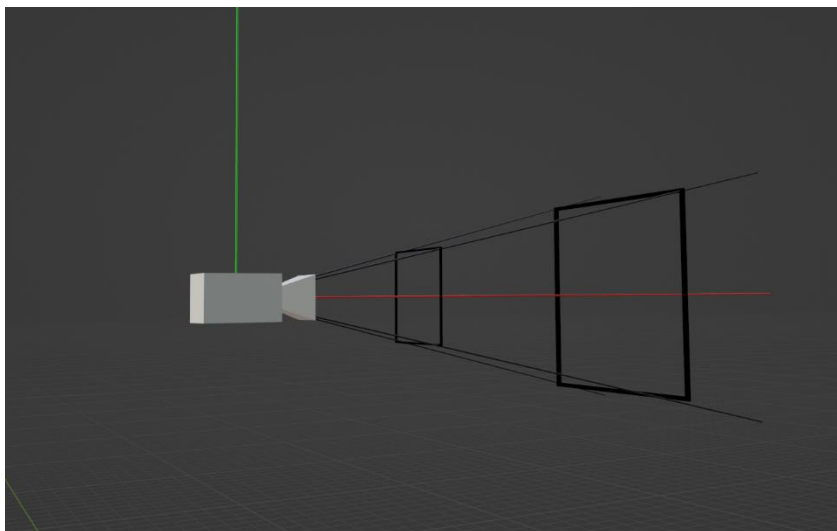
4.4.5 Camera

Egy grafikus program elengedhetetlen tartozéka egy virtuális kamera is. Ez határozza meg, hogy mit és mennyit látunk az általunk definiált színtérből. Mivel kirajzoláskor a kamera közeli vágósíkjára vetítjük rá a kirajzolandó objektumokat, így a kamera feladatai közé tartozik még a projekciós és nézet mátrix előállítása is, amiket a kirajzolásnál történő koordináta-rendszer váltásoknál használunk fel.

A szoftverben a kamerát a következő tulajdonságok jellemzik:

- pozíció
- előre mutató vektor (nézeti irány)
- felfelé mutató vektor
- látószög
- képarány
- távoli vágósík távolsága
- közeli vágósík távolsága

A 23. ábra szemléletesen ábrázol egy ilyen virtuális kamerát.



23. ábra: Virtuális kamera szemléltetés

A camera osztály a korábban tárgyalt objektumoktól eltérően valójában nem egy OpenGL-es

gCamera
<ul style="list-style-type: none"> - m_app : CMyApp* - m_speed : float - m_viewMatrix : mat4 - m_matViewProj : mat4 - m_slow : bool - m_eye : vec3 - m_at : vec3 - m_up : vec3 - m_u : float - m_v : float - m_fw : vec3 - m_st : vec3 - m_matProj : mat4 - m_goFw : float - m_goRight : float
<ul style="list-style-type: none"> + gCamera() + ~gCamera() + gCamera(vec3, vec3, vec3) + GetViewMatrix() : mat4 + Update(const float) : void + LinkToApp(CMyApp*) : void + SetView(vec3, vec3, vec3) : void + SetProj(float, float, float, float) : void + LookAt(vec3) : void + Resize(int, int) : void + FocusOnPosition(vec3) : void + KeyboardDown(SDL_KeyboardEvent&) : void + KeyboardUp(SDL_KeyboardEvent&) : void + MouseMove(SDL_MouseMotionEvent&) : void + UpdateUV(float, float) : void

funkciót absztraktál, sokkal inkább mondható egy kötelező elemnek minden grafikai alkalmazásnál. Mivel a kamera viselkedése, orientációja, pozíciója az adott alkalmazástól függ, így ennek az osztálynak az implementációját valamelyest módosítottam az eredeti OGLBase projekthez képest. Az osztály konstruktorának meghívása után hozzá tudok csatolni egy MyApp osztályt (4.5.2). A kamera ezután aszerint fog viselkedni, hogy az alkalmazás milyen stádiumában van éppen. Például egy játék kör közben a kamera alap esetben a játékos úrhajója mögött lesz rögzítve, míg a menü ablakban egy folyamatos körbenéző animációt fog alkalmazni.

4.5 Központi fájlok

A játék felépítésének magját a „main.cpp” fájl és a „CMyApp” osztály alkotja. Ebben a részben ezek működését és feladatát mutatom be.

4.5.1 Main.cpp

A main.cpp a program belépési pontja. A mappaszerkezetet (4.2) tekintve a „source” mappában található. Feladata, hogy létrehozza és inicializálja a program működéséhez szükséges kontextusokat, illetve az alkalmazás leállása után törölje azokat. Az SDL könyvtár segítségével beállítom az OpenGL futtatásához szükséges paramétereket, illetve létrehozok egy Windows ablakot. Majd elkészítem az OpenGL kontextust. Az OpenGL mellett a felhasználói felülethez felhasznált ImGui könyvtárat is inicializálni kell.

A main.cpp példányosít egy CMyApp objektumot is „app” néven, ez fogja a játéklógika központi elemét alkotni.

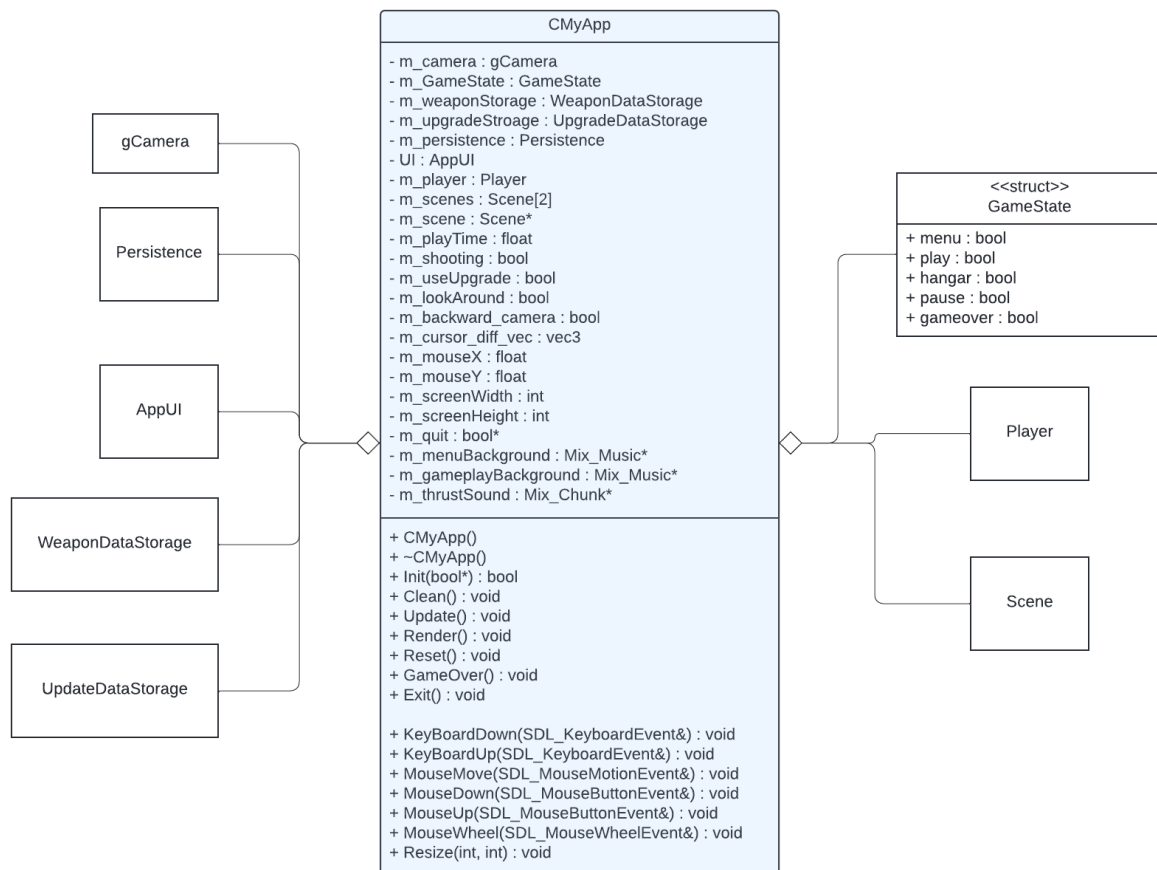
Ebben a forrásfájlban található maga a játékciklus is, amin belül három kiemelendő feladatot lát el a main.cpp. Egyrészt, az SDL könyvtárat felhasználva detektálom a különböző felhasználói inputokat. A detektálást követően meghívom a példányosított CMyApp objektum megfelelő eseménykezelő függvényeit, paraméterként továbbítva az adott felhasználói inputot. Az eseménykezelés után rendre meghívom az „app” objektum „update” és „render” függvényét. Végül a felhasználói felület kirajzolását végzem el az ImGui::Render() parancs meghívásával.

4.5.2 CMyApp osztály

A CMyApp osztály (röviden: app) feladata, hogy az alkalmazás főbb mozzanatait kezelje és felügyelje. Itt zajlik az eseménykezelés érdemi része és a program mentéséért és betöltéséért felelős perzisztencia osztály példányosítása és kezelése, illetve a játék aktuális állapotának nyomon követése.

A játék állapotát egy „GameState” nevű struktúrában tárolom és ennek függvényében kezelem a szoftver működésében résztvevő objektumok frissítését és kirajzolását.

A játéklógika megvalósítását 2 fő osztály végzi, ezek a játékos (Player) osztály és a jelenet (Scene) osztály. Ezeket az objektumokat a CMyApp osztályon belül példányosítom és kezelem. A játékban lévő két különböző játéktérhez egy-egy külön Scene objektum tartozik, amiket az app osztályban egy tömbben tárolok, illetve egy mutatót állítok ezek közül az éppen aktuálisra.



24. ábra: CMyApp osztálydiagramja

Ahogy a 24. ábrán látható osztálydiagramon is látható, a CMyApp osztályban implementálom az „update” és „render” függvényeket. Ez a két függvény minden animációval rendelkező program alappillére. A fejlesztés során szem előtt tartottam, hogy az app osztály valóban csak magas szinten, átfogóan vegyen részt a játéklogika megvalósításában. Így az update metódus csupán az aktuális játékállapot függvényében frissíti azokat az objektumokat, amik a tényleges implementációt tartalmazzák, a render metódusban pedig meghívom ezek kirajzoló függvényeit.

Az alkalmazás átfogó működtetéséért a CMyApp osztályon kívül még másik három objektum felel. Ezek a kamera, az AppUI és a Persistence osztályok. Mivel ezek szoros együttműködésben dolgoznak az app osztállyal, így mind a három „barát” objektumként (friend class) van deklarálva a CMyApp header-ben. Ez lehetővé teszi ezeknek az osztályoknak a közvetlen hozzáférést az app osztály azon adataihoz is, amikhez más osztályoknak nincs hozzáférése.

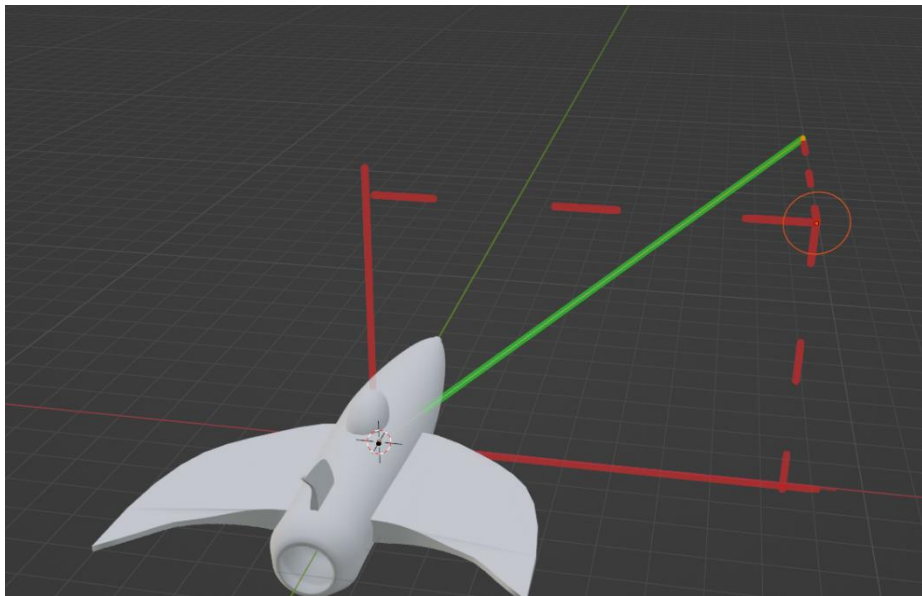
Bár a CMyApp osztály alapból csak egy magas szintű játéklógikát implementál, egy fontos számítás elvégzésének felelőssége mégis erre a típusra hárul. Meg kell határoznia, hogy a kurzor aktuális helyzete milyen pozíciót jelöl a háromdimenziós térben. Ezt majd a játékos osztály fogja felhasználni a haladási irány meghatározásához.

A kurzor világ koordinátáját (3D-s pozícióját) az „m_cursor_diff_vec” változó tárolja. Ennek kiszámítása két lépésből áll.

Először is a kurzor mozgatás eseményének kezelésénél frissítenem kell az „m_mouseX” és „m_mouseY” változókat. Az SDL könyvtár által észlelt „mouse move” esemény egy 2 dimenziós vektorban adja át a kurzor új helyzetét, ez a koordináta a képernyő bal felső sarkához képest adja meg a pozíciót, azaz az „x” komponens 0-tól az ablak szélességének pixelben mért nagyságáig terjedhet. Az „y” komponens pedig 0-tól az ablak magasságáig. Ez kényelmetlen és nehezen használható ebben az esetben, ezért ezt a koordinátát átkonvertálom olyan módon, hogy az „x” és „y” komponens a képernyő közepéhez képesti távolságot mutassa -1 és 1 közé szorítva. A konverziót a következő módon hajtom végre az „x” komponens esetében:

$$x = \frac{x}{\text{képernyő szélessége} * 0.5} - 1.$$

Ha megvannak a -1 és 1 közé szorított értékek, akkor a háromdimenziós origóból kiindulva vektor összeadással kiszámolom az „m_cursor_diff_vec” értékét. Ehhez felhasználom a játékos aktuális orientációját, így lényegében a játékos űrhajójához képesti kurzor pozíciót kapom. Tehát az origóból kiindulva rá mérem a játékos keresztirányú vektorára a kurzor pozíció x komponensét, majd a felfelé mutató vektorra az y komponensét. Majd „előre” lépek egyet a



25. ábra: Cursor_diff_vec

játékos előre mutató vektorját felhasználva, így megkapva az `m_cursor_diff_vec` kiszámított értékét. A folyamatot a 25. ábra szemlélteti.

4.6 Időbélyeg

Az időbélyeg (angol nevén: time step vagy delta time) méri az eltelt időt 2 frame kirajzolása között. Időbélyeget minden olyan grafikus, vagy animációt tartalmazó szoftvernek tartalmaznia kell, amiben fontos a konzisztencia. Segítségével tudjuk, hogy mennyi idő telt el az előző frame óta. A játékban az időbélyeg az `app` osztály „update” metódusának elején frissül. Könnyebbséget jelent, hogy az `ImGui` alapértelmezetten kiszámolja ezt az időbélyeget, amit az „`ImGui::GetIO().DeltaTime`” hivatkozás segítségével le tudok kérdezni. Ezt az értéket eltárolom a „delta_time” nevű változóban, amit aztán minden játékbéli objektum megkap az „Update” metódusa paramétereként.

Erre azért van szükség, hogy az objektumok haladási sebessége független legyen attól, hogy éppen milyen FPS számmal fut az alkalmazás. Könnyű belegondolni, hogy ha például egy konstans sebesség értékkel mozgatunk előre egy objektumot minden frameben, akkor a kétszeres FPS szám egyben kétszeres sebességet is jelent. Ez egyszerűen orvosolható, csupán az elmozdulás mértékét kell beszorozni/súlyozni az előző képkocka kirajzolása óta eltelt idővel.

A programban minden mozgó objektum bármilyen mozgása (rotációja is) a `delta_time` értékével van súlyozva, így erre a részletre a későbbiekben nem térek ki.

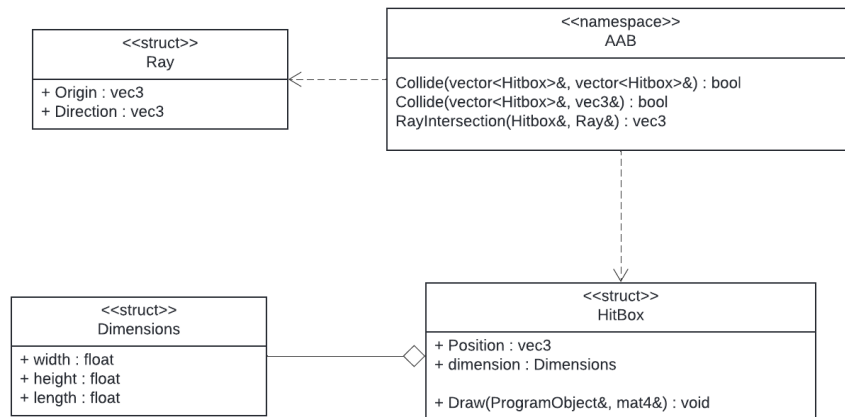
4.7 Ütközésvizsgálat

Az alkalmazásban a játékbéli objektumok ütközésének vizsgálatához két módszert használok fel. Ebben a részben ezeket a módszereket tárgyalom részletesen.

4.7.1 Axis-Aligned Bounding Boxes

Az alkalmazásban a játékoson kívül minden más objektum ütközéseit axis-aligned bounding boxok (röviden: AAB) segítségével vizsgálom. Egy AAB lényegében egy téglatest, ami magába foglal egy jelen esetben háromdimenziós objektumot. Legfontosabb tulajdonsága, hogy az egyes élei párhuzamosak a koordináta-rendszer tengelyeivel, ezt kihasználva nagyon egyszerű és gyors ütközésvizsgálatot lehet velük végezni. Mivel a saját elképzelésem alapján implementáltam ezt a módszert, így előfordulhat, hogy néhány helyen eltér a megszokott AAB ütközésvizsgálatától. Alap esetben egy AAB memóriában való eltárolásához csupán két átlós csúcsot szokás felhasználni, azonban esetemben ez a megoldás nem lett volna praktikus egyes

számítások elvégzéséhez. Ehelyett egy axis-aligned boxot a programban egy „HitBox” nevű struktúra szimbolizál. Egy hitboxot a középpontjával és a méreteivel tárolok el. Egy objektumhoz akár több ilyen hitbox is tartozhat.

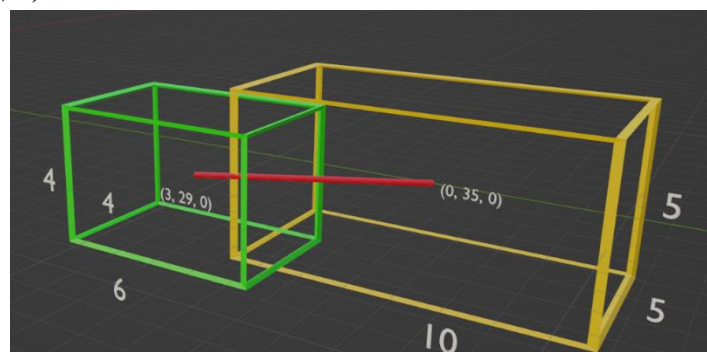


26. ábra: AAB -hoz tartozó osztálydiagram

A hitboxokkal történő ütközésvizsgálatot az „AAB” névtérben található „Collide” metódusokkal végzem két esetre bontva. Az egyik esetben két objektum hitboxainak ütközését, a másikban egy hitbox tömb és egy háromdimenziós pont metszését vizsgálom.

Az első esetben az algoritmus egy egymásba ágyazott ciklusban megvizsgálja, hogy a két tömb bármelyik eleme metszésben van-e a másik tömb bármelyik elemével.

A két hitbox közötti metszésvizsgálat első lépése, hogy kiszámolom a középpontokat összekötő vektort (a 2 középpont különbsége). Majd megnézem, hogy az így kapott vektor „x” koordinátájának abszolút értéke kisebb-e, mint a két hitbox szélességének az összegének fele. Ugyanezt a műveletet elvégzem az „y” koordináta és a magasság, illetve a „z” koordináta és a hosszúság viszonyában is. Amennyiben mind a három eset igazra értékelődik ki, akkor a két hitbox metszi egymást, tehát ütköznek. Az algoritmus szemléltetése a 27. ábrán látható, ahol a piros vektor: (3, -6, 0).



27. ábra: AAB szemléltetése

A hitbox és pont esete is nagyon hasonló. Azzal a különbséggel, hogy itt a pont helyzete és a hitbox középpontja közötti vektort vizsgálom. Ebben az esetben természetesen csak a hitbox dimenzióival kell elvégezni az összehasonlítást.

Egy hitboxot viszonylag könnyű úgy megadni, hogy a lehető legjobban illeszkedjen az objektumra abban az esetben, ha az adott objektum orientációja fix. Azoknál az objektumoknál, amik képesek rotációra, már egy kicsit bonyolultabb a helyzet. Ebben az esetben, például az ellenséges űrhajónál, az objektum orientációja szerinti lineáris interpolációval számolom újra a hitbox dimenzióit, hogy a lehető legjobban illeszkedjenek az aktuális helyzethez.

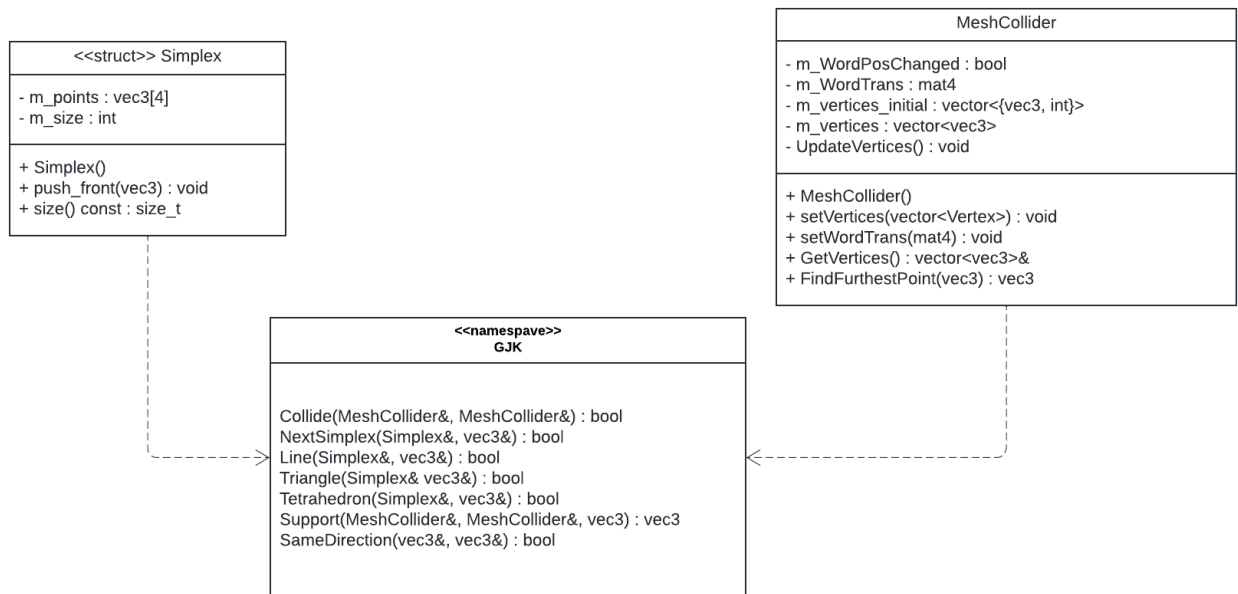
A játékban az objektumok és lövedékek közötti ütközést is ezeknek a hitboxoknak a felhasználásával vizsgálom. A lövedékek tipikusan nagyon gyorsan haladó és viszonylag kis méretű objektumok, így feleslegesnek tartottam hitboxokat illeszteni körük. Az ütközésvizsgálatnál csak az aktuális pozíciójukat (középpontjukat) veszem figyelembe. Erre szolgál a 26. ábrán látható AAB névtér második „collide” metódusa. Két lövedék ütközésének esetét nem vizsgálom a programban.

4.7.2 A Gilbert-Johnson-Keerthi algoritmus

Az AAB ütközésvizsgálat egy nagyon gyorsan számolható módszer annak a megállapítására, hogy két objektum metszi-e egymást. Pont ezért előszeretettel használják valós idejű alkalmazásokban. Viszont többnyire csak nagyon egyszerű objektumokra. Ennek az egyértelmű indoka, hogy egy bonyolultabb objektumra (Mesh-re) nagyon pontatlan eredményt tud csak nyújtani és akkor is ütközést jelezhet, amikor még szemmel láthatóan nem metszik egymást a modellek. Az ellenséges űrhajók esetében ez nem jelent akkora problémát a játékélményre nézve. A játékost illetően azonban már más a helyzet, itt szükségesnek láttam egy pontosabb ütközést vizsgáló algoritmust is alkalmazni.

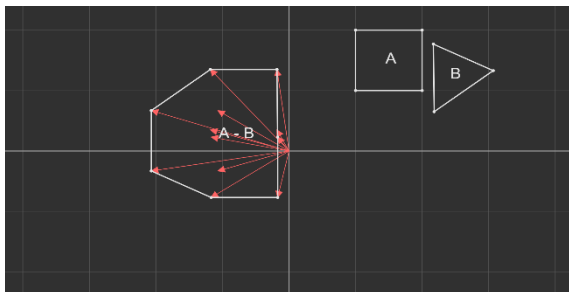
A Gilbert-Johnson-Keerthi (röviden: GJK) algoritmus képes mesh-re pontosan megállapítani, hogy két tetszőlegesen bonyolult objektum metszi-e egymást, teszi mindezt közel konstans időben. Egy megkötés van csupán a modell bonyolultságát illetően, mégpedig, hogy a GJK algoritmus csak konvex objektumokra tud pontos eredményt adni. Ha ténylegesen tetszőleges objektumot szeretnénk vizsgálni, akkor fel kell bontanunk a konkáv modelleket konvex alakzatokra. Amennyiben ezt nem tesszük meg, akkor a GJK algoritmus az objektumot körülvéző konvex burokkal (Convex hull) tud ütközést vizsgálni. Én ez utóbbi lehetőség mellett döntöttem.

Az algoritmust „Winterdev” e témáról szóló cikke alapján implementáltam.⁶

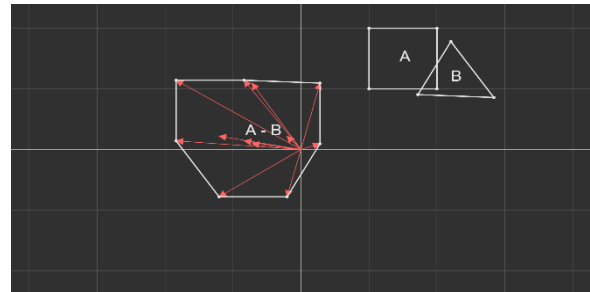


28. ábra: GJK névtér osztálydiagramja

A GJK algoritmus két objektum Minkowski összegét használja fel az ütközés megállapításához. két alakzat Minkowski összegét úgy kapjuk, hogy az egyik alakzat minden egyes pontját hozzáadjuk a másik alakzat minden egyes pontjához. Az így kapott eredmény szintén egy



30. ábra: Minkowski különbség nem metsző alakzatokra⁶



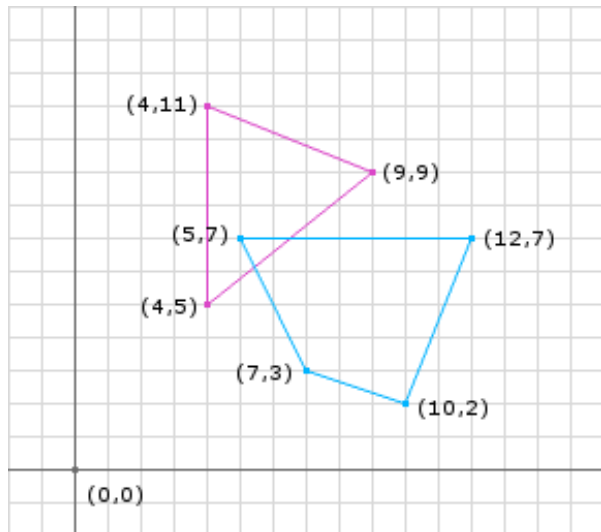
29. ábra: Minkowski különbség metsző alakzatokra⁶

alakzat lesz. Az algoritmus ennek a műveletnek az ellentétjét használva, a Minkowski „különbséget”. Ebben az esetben minden pontból kivonjuk (egy vektorként) a másik alakzat minden pontját. Gondoljunk bele, ha két objektum metszi egymást, akkor definíció szerint lennie kell legalább 1 olyan pontnak a térben, ami mindkét alakzatnak a része. Ha egy pontot kivonunk önmagából, akkor az origót fogjuk kapni. Innen már talán érződik az elképzelés lényege. Ahogy az a 29. és 30. ábrán is látható, ha két objektum metszi egymást, azaz ütköznek, akkor a Minkowski különbségüknek tartalmaznia kell az origót. A GJK algoritmus alap ötlete

⁶https://winter.dev/articles/gjk-algorithm?fbclid=IwAR02rYdZ7ulFy5TfeXOYw_rcbuLbc4CnspoShjP7QblYLM6gBJfQzzL9OUI

tehát, hogy két tetszőleges objektum metszés vizsgálatának a problémáját visszavezeti arra, hogy egy darab alakzat tartalmazza-e az origót.

A probléma már csak az, hogy minden alakzat végtelen sok pontból áll, amit egy számítógépen természetesen nem tudunk kezelni. Itt jön be a képbe az a korábban tárgyalt feltétel, hogy csak konvex alakzatokra működik az algoritmus. A konvex alakzatoknak azt a tulajdonságát használja ki, hogy elég azokat a pontokat ismerni, amik az alakzat középpontjához képest valamilyen irányba a legtávolabb esnek, a továbbiakban ezeket a pontokat extrém pontoknak fogom nevezni. Az így kapott pontok együtt megalkotják az alakzat konvex burkát, ami egy konvex alakzat esetén persze megegyezik magának az objektumnak a felületével.



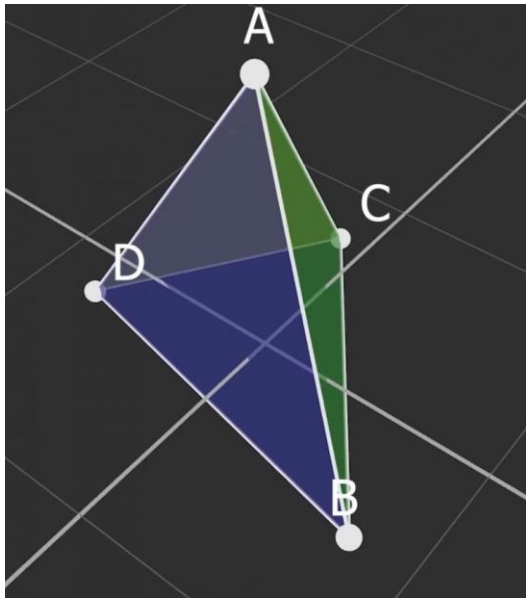
31. ábra: Extrém pontok szemléltetése

Innentől kezdve tetszőlegesen bonyolult konvex alakzattal tudok metszést vizsgálni, hiszen magáról az objektumról pusztán annyit kell tudni, hogy egy adott irány mentén melyik a legtávolabbi pontja. Ha egy objektumhoz készítek egy függvényt, ami megadja ezt a pontot, akkor utána azt használhatom ütközésvizsgálatra. Az én esetemben ezt a függvény a 28. ábrán látható „MeshCollider” osztályban implementálom a „FindFurthestPoint” nevű függvényben. A függvény tetszőleges Mesh objektumra működik, mégpedig úgy, hogy végig iterálok az objektum csúcspontjain, és kiválasztom azt, aminek a megadott iránnyal vett skalárszorzata a legnagyobb. Ez még csak az objektumok külön-külön vett extrém pontjait határozza meg, az algoritmusnak viszont a Minkowski különbség extrém pontjaira van szüksége. Ehhez végezzünk el pár algebrai átalakítást:

$$\max\{D * (A - B)\} = \max\{(D * A) - (D * B)\} = \max\{D * A\} - \max\{(-D) * B\}^6$$

A fenti képletben a „D” azt az irányvektort jelöli, amerre keresem a legtávolabbi pontot, „A” és „B” pedig a két alakzatot jelöli. Látható, hogy ha a Minkowski különbségben szeretnénk megtalálni egy extrém pontot egy „D” irányban, akkor csak ki kell vonni az egyik alakzat „D” irányába lévő extrém pontjából a másik alakzat ellenkező („-D”) irányába lévő extrém pontot. Ezt az implementációban a GJK névtér „Support” függvényében számolom ki.

Most, hogy ez megvan, a következő lépés, hogy eldöntsem csupán ezeket az extrém pontokat felhasználva, hogy a különbség alakzat tartalmazza-e az origót. Ezt a GJK algoritmus úgy vizsgálja, hogy folyamatosan próbálja egy úgynevezett „Simplex” alakzattal körül venni azt. Egy simplex lényegében a legegyszerűbb alakzat, ami egy adott dimenzióban körül tud határolni egy térrészt. Ez egy dimenzióban egy vonal, kettő dimenzióban egy háromszög, illetve három dimenzióban egy tetraéder. Ezeknek az alakzatoknak az előnye, hogy egyszerű



32. ábra: Tetraéder az origó körül⁶

skalárszorzatok segítségével meglehetősen állapítani, hogy melyik csúcspont, vonal, vagy lap van a legközelebb az origóhoz, vagyis, hogy a simplexhez képest az origó melyik térrészben található. Ahogy az algoritmus előre halad, úgy fogja a simplexeket frissíteni ezeket a skalárszorzos teszteket figyelembe véve olyan módon, hogy egyre „közelebb” kerüljön az origóhoz. Teszi ezt úgy, hogy veszi az aktuálisan vizsgált simplex felől az origóba mutató vektort, majd a „Support” függvény segítségével meghatározza az erre a vektorra nézve legtávolabbi pontot. Amennyiben azt találom, hogy az újonnan felvett pontot már megvizsgáltam, de

még ez is túl közel van ahhoz, hogy az origó „túloldalára” kerüljön, akkor biztosan tudhatom, hogy nem történt ütközés. A metszés megállapításához a végső konklúziót három dimenzióban csak egy újonnan felépített tetraéder vizsgálatánál lehet levonni. Ehhez meg kell vizsgálni, hogy a tetraéder melyik lapján „túl” található az origó, hogy tovább tudjon haladni az algoritmus. Amennyiben már az összes lehetséges térrészt megvizsgáltam, és azt találtam, hogy az origó egyikben sem lehet, akkor a simplexek felvételének módja miatt az origó már csak kizárólag a vizsgált tetraéderen belül helyezkedhet el. Ekkor megállapítottam, hogy a két vizsgált alakzat Minkowski különbsége magába foglalja az origót, tehát ütköznek. A 32. ábrán egy ilyen esetre látható példa.

Ahogy már említettem, a GJK algoritmus közel konstans időben képes megállapítani az ütközés tényét. Viszont az algoritmus működéséhez tudnom kell a résztvevő objektumok összes csúcspontjának a világ koordinátáit⁷. Ez problémát vet fel, hiszen alapesetben egy Mesh

⁷ Az általunk generált színtérben lévő pozíció

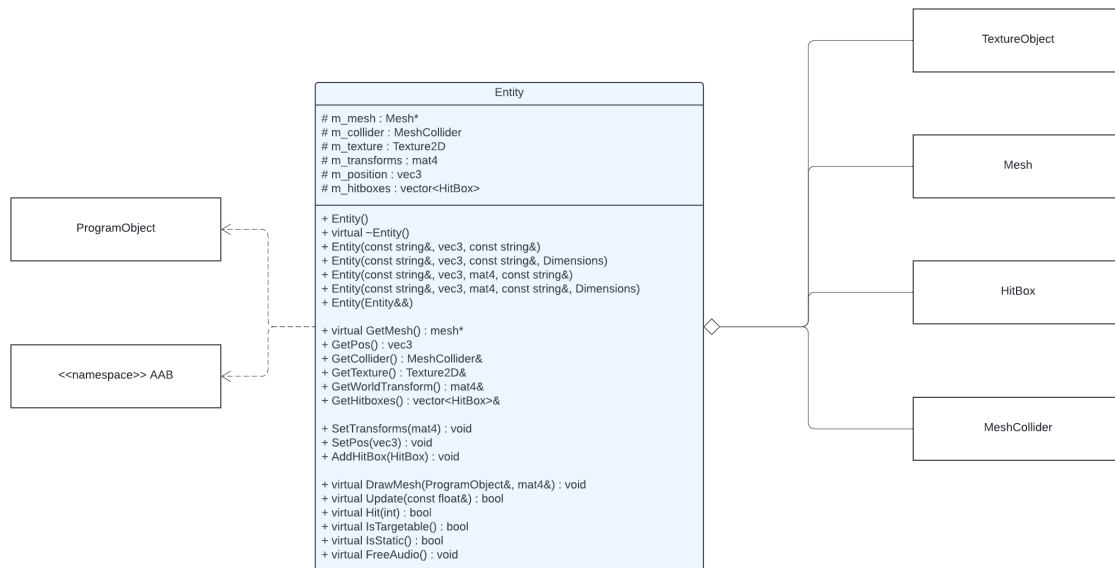
objektum kirajzolásához CPU oldalon pusztán a vertexek modell koordináta-rendszerben⁸ vett pozícióit, illetve az objektumot a színtérben elhelyező transzformációs mátrixot kell ismerni. A vertexek transzformációját már a grafikus szerelőszalag végzi. Ennek a problémának a megoldására született a 28. ábrán látható „MeshCollider” osztály. Ennek az osztálynak a feladata, hogy eltárolja a CPU számára is egy adott modell csúcspontjainak a világ koordinátáit. Ezeknek a pozícióknak a meghatározása azonban a processzor számára nagyon megterhelő még úgy is, hogy a koordináták frissítését párhuzamos módon végzem. Ezért a MeshCollider osztály gondoskodik róla, hogy csak abban az esetben frissítse ezeket a koordinátákat, amikor arra ténylegesen szükség van. Az „m_WordPosChanged” változóban számontartom, hogy a legutóbbi vizsgálat óta változott-e a modell pozíciója, orientációja. Amikor egy ütközésvizsgálat keretein belül meghívódik a Collider „FindFurthestPoint” metódusa, megnézem, hogy változott-e a modell pozíciója, és ha igen, akkor frissítem a csúcspontokat.

Az imént említett teljesítménybéli problémák miatt a GJK algoritmust csak a játékos ütközésvizsgálatához használom az AAB módszerrel együtt. Ha a játékos objektumnál azt találom, hogy az AAB ütközésvizsgálat igazgal tért vissza, abban az esetben megvizsgálom az esetet a GJK algoritmussal is, és ennek eredménye alapján állapítom meg, hogy történt-e ütközés. Ennek köszönhetően maga a GJK algoritmus és a vertexek frissítése csak akkor fog lefutni, amikor arra ténylegesen szükség van.

4.8 Az Entity osztály

Az „Entity” osztály az ősosztálya minden háromdimenziós objektumnak, amit egy adott színtérben láthatunk. Az olyan adattagok és metódusok, amikkel minden objektumnak rendelkeznie kell, itt találhatóak. Ezek főleg a színtérben való megjelenítéshez és az ütközések vizsgálatához kapcsolódó adattagok. Az objektumokat a színtérben egy Mesh és egy ehhez kapcsolódó textúra segítségével jelenítem meg, így ezek az adattagok az Entity osztályba kerültek. Ezen kívül a kétféle ütközésvizsgálati módszer által használt objektumok is itt találhatóak, ez 1 darab MeshCollider objektumot és hitboxoknak egy tömbjét jelenti. Amennyiben egy objektummal valamilyen okból nem szeretnék ütközést vizsgálni, szimplán üresen hagyom a hitboxok tömbjét. Ezeken kívül ebben az osztályban tárolom az objektum színtérben való pozícióját (ez többnyire a mesh objektum középpontját jelöli), illetve a kirajzoláshoz szükséges transzformációs mátrixot is.

⁸ A modell megalkotásánál a mesh középpontjához képesti koordináta-rendszer



33. ábra: Entity osztály osztálydiagramja

A 33. ábrán látható, hogy az Entity osztályban csak a legáltalánosabb metódusokat implementálom. Azokat, amikre minden objektumnak szüksége lehet. A különböző konstruktorok segítségével létre tudok hozni bármilyen meshel és textúrával rendelkező objektumot akármilyen pozícióval és transzformációkkal. Ezáltal az egyes színterek feltöltésénél akár egy függvény hívással el tudok helyezni bárhol egy objektumot a világban. Ez akkor hasznos, amikor a különböző akadályokat, logikával nem rendelkező modelleket szeretnék elhelyezni egy színtérben. Ekkor a létrehozott objektum típusa pusztán Entity lesz. Az Entity osztály rendelkezik ugyan Update metódussal, de csak virtuális formában, hiszen az adott objektum logikájának implementációja már a konkrét leszármazott objektum típusától függ. Ez, és a további virtuális metódusok lehetőséget adnak arra, hogy a polimorfizmust kihasználva egy adatgyűjteményen belül tudjak tárolni Entity objektumokat és ezek mégis eltérő viselkedést mutassanak. Ezek között a metódusok között találhatóak olyanok is, amik pusztán valamiféle információt közölnek a leszármazott objektumról. Például az „IsStatic” metódus arra a kérdésre válaszol, hogy az adott entitás mozog-e a színtérben vagy sem. Ez a függvény alapértelmezetten igazzal tér vissza, viszont például az ellenségek esetén felüldefiniálom ezt a metódust. Az Entitások közötti különbségtételt meg lehetett volna oldani másképp is, például amikor meg szeretném tudni, hogy egy adott entitás ellenségnek számít-e, megpróbálhatnám átkonvertálni az adott objektumot Enemy típusúra. Véleményem szerint valamivel elegánsabb megoldás, hogy a polimorfizmust kihasználva rákérdezhetek az adott entitás valamely tulajdonságára.

Grafikus alkalmazás lévén érdemes pár szót ejteni a „DrawMesh” függvény működéséről, hiszen ez az a metódus, ami az objektum megjelenítéséért felel, ráadásul ezt a függvényt csak kevés leszármazott típus definiálja felül. Ez a metódus csupán annyit csinál, hogy a paraméterében megkapott ProgramObject-en keresztül feltölti a textúrát, illetve a shaderek által felhasznált transzformációs mátrixot a grafikus szerelőszalag „uniform”⁹ változóiba. Ezután meghívja a Mesh objektumnak „Draw” parancsát.

A teljesítményt szem előtt tartva a program fejlesztése közben kifejezetten figyeltem arra, hogy nagy méretű objektumokat még véletlenül se másoljak egyik memória területről a másikra feleslegesen. Emiatt az Entity osztály nem rendelkezik másoló konstruktorral.

4.9 Színterek

A szoftver működését tekintve a színterek vagy más szóval jelenetek központi szerepet töltenek be. Az alkalmazásban a „Scene” osztály testesíti meg ezeket. A scene az, amin belül egy játék kör történései zajlanak. Itt található az adott színtérben/jelenetben szereplő entitások tömbje, itt tartom számon a jelenetben éppen „részt vevő” lövedékeket is, illetve a Scene osztály felelőssége az olyan funkcionalitások implementálása, amik az egész játékteret érintik.

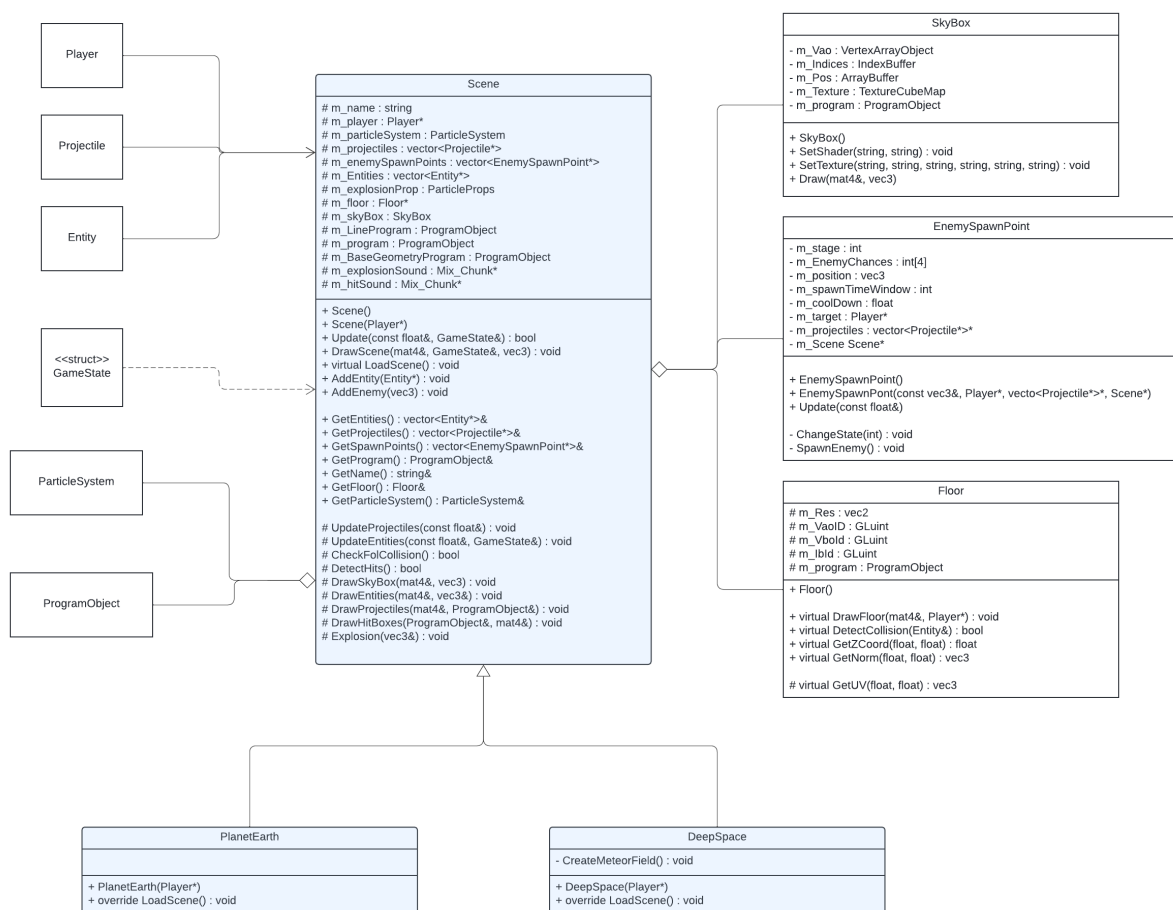
A játékban a különböző pályák (jelenleg „DeepSpace” és „PlanetEarth”) a Scene ősosztály leszármazottjai. Bár valójában a CMyApp osztályban csak ezeket a leszármazott színtereket példányosítom, ezek feladata csupán, hogy a „LoadScene” metódusban feltöltsék az entitások tömbjét a rájuk jellemző, speciális objektumokkal. Így megalkotva egy pályát. Magát az entitás tömb karbantartását végző műveleteket és az egyéb játék logikával kapcsolatos teendőket a Scene ősosztályban implementálom. Ezáltal könnyedén hozzá lehet adni új pályákat a játékhoz, mindössze egy osztályt kell leszármaztatni a Scene-ből, és a „LoadScene” metódusban feltölteni azt a kívánt entitásokkal, illetve hozzá kell adnom az új színtér típus egy példányát az app objektum „m_scenes” tömbjéhez.

A jelenet osztály felelős a benne lévő objektumok és magának a színtérnek a kirajzolásáért is. Ezért minden Scene objektum rendelkezik egy ProgramObject adattaggal, amihez a konstruktorban hozzá tudom csatolni az adott pálya által használt alap shader programokat. Ezután minden entitás alap esetben ezeket a shadereket fogja használni kivéve, ha valamilyen okból (például áttetsző objektumról van szó) az adott entitásnak van saját, speciális shader konfigurációja. Szükség van azonban még egy ProgramObject adattagra, mégpedig azért, mert

⁹ A CPU felől a GPU-ra feltöltött adatok, amiket a grafikus szerelőszalag felhasznál a megjelenítéshez.

performancia okokból kifolyólag a lézer lövedékek kirajzolását nem Mesh objektummal, hanem egyszerű téglatestekkel oldottam meg. Ehhez másfajta shader programok szükségesek, így ezek tárolására kell a másik, „m_BaseGeometryProgram” névre hallgató ProgramObject adattag.

A szintér egy másik nagyon fontos alkotó eleme a skybox, ami lényegében az adott jelenet háttérét fogja biztosítani. Egy skybox inicializálását, illetve kirajzolását a „SkyBox” osztályban implementáltam. Egy skybox lényegében egy kocka a kamera körül, aminek minden pixelének a távolsága „végtelen” a kamerához képest, belső oldalain egy-egy textúrával. A skyboxok működését ebben a dokumentumban nem tárgyalom részletesebben, az ELTE számítógépes labor honlapján azonban található egy pontos és részletes leírás róluk¹⁰.



34. ábra: Scene osztály osztálydiagramja

A 34. ábrán látható, hogy a Scene osztály rendelkezik a szintérben szereplő Entitások és lövedékek tömbjével. Mivel mind a két tömb/vektor esetében ki szeretném használni a

¹⁰<https://docs.google.com/document/d/16lsAVF8vOmYr9gh54MZF6j0iURGIYUUr6-y6WCX8YiA/edit#heading=h.e6jevuqkcrum>

polimorfizmust, így ezek az gyűjtemények csak mutató objektumokat tartalmaznak a heapen allokált Entitásokra és lövedékekre. A konkrét entitásokat és lövedékeket „unique pointer” -ek segítségével hozom létre, így élettartamuk az őket tartalmazó jelenet élettartamától függ, tehát lényegében a Scene-en belül léteznek. Más a helyzet a játékos objektummal. A Scene osztály csak egy mutatót tartalmaz a játékos objektumra, amit a konstruktora paramétereként kaphat meg. Ennek az az oka, hogy míg például az ellenséges űrhajók vagy akadályok logikailag nem létezhetnek egy jeleneten kívül, addig a játékos igen.

A jelenetek tartalmaznak még „EnemySpawnPoint” objektumokat is. Ezek valósítják meg az ellenséges űrhajók indítását a „[JÁTEKMENET](#)” című rész leírása szerint.

Mivel a Scene osztály felelős magának a játéktérnek a megjelenítéséért, ezért a különböző részecske effektek kirajzolása is ezeken az objektumokon keresztül történik a „ParticleSystem” osztály felhasználásával. Erről a ([4.13 RÉSZECSEKERENDSZER](#)) részben írok részletesebben.

Az osztály „Update” metódusa végzi a munka nagy részét. Ez az a függvény, amit az app objektum meghív a saját Update metódusában. A 34. ábrán látható, hogy ez a metódus paraméterként megkapja az app osztálytól az aktuális státuszát az alkalmazásnak. Ezt felhasználom arra, hogy eldöntsem, hogy az adott frameben mely objektumokat kell és melyeket nem kell frissíteni. Ez a függvény végzi a privát segédmetódusokat meghívva a játékbéli ütközések és találatok kezelését is. Miután frissítettem a kellő entitásokat, megvizsgálom, hogy a játékos ütközött-e bármilyen más objektummal ([4.7](#)), illetve érte-e végzetes találat. Amennyiben igen, az „Update” függvény igazzal tér vissza ezzel jelezve, hogy véget ért a kör. Az ellenséges űrhajók ütközéseit nem a Scene-osztályban vizsgálom, mivel az ellenséges objektumok „Update” metódusában szükségszerűen amúgy is végig iterálok az aktuális jelenet objektumain, így az ütközésvizsgálat is itt történik. Az ellenségek az Update metódusuk visszatérési értékével jelzik, hogy ütköztek-e. A Scene osztály egy verem adatszerkezetbe gyűjti azoknak az entitásoknak az indexét, amik jelezték, hogy megsemmisültek, majd a frissítésekét követően törli ezeket az objektumokat.

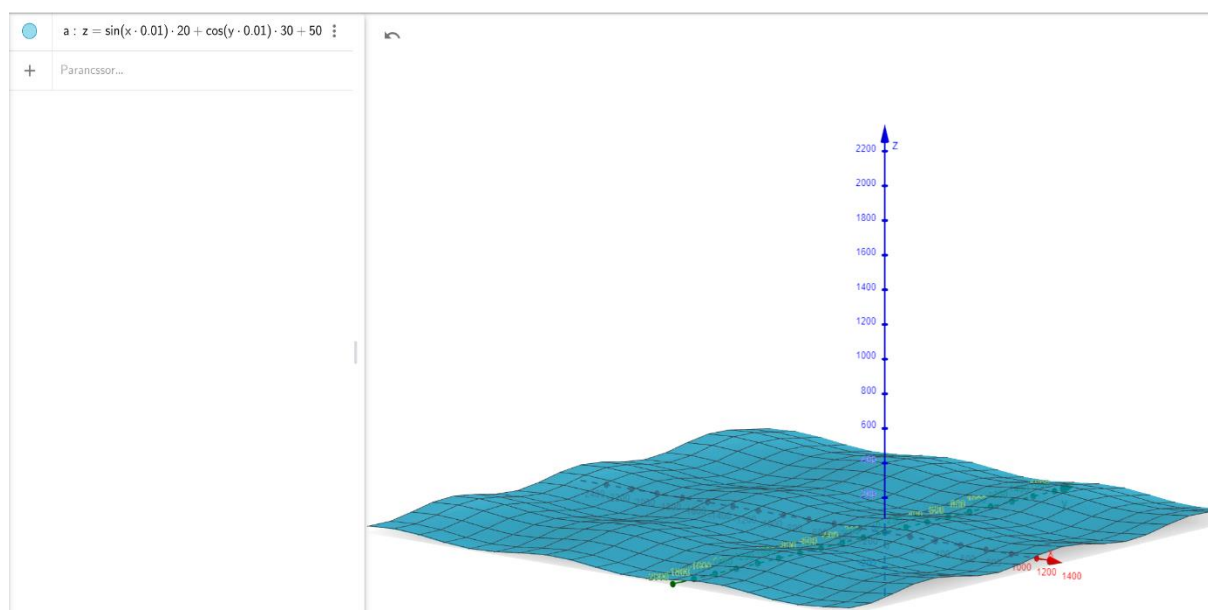
4.9.1 Talaj

A 34. ábrán látható egy „Floor” nevű osztály is. A jeleneteknek ugyanis része lehet egy talaj is, ahogy az a „PlanetEarth” pályán látható. A talaj objektumot kissé kivételesen kell kezelni minden tekintetben. Mivel a játékban egy szintérnek nincs határa, a játékos olyan messzire repülhet, amilyen messze csak akar. Ez a „DeepSpace” pályán nem jelent gondot, hiszen a skybox működése miatt az mindig követni fogja a kamerát. Viszont egy talajjal rendelkező

pályán meg kell oldani, hogy akármerre is repül a játékos, mindig ott legyen alatta ez a talaj. Emiatt ennek kirajzolását nem lehet egy darab Mesh-el megoldani, hiszen az végtelen sok csúcspontból kéne, hogy álljon. Ennek megoldására egy, a játék iparban előszeretettel használt megoldásnak az alapötletét használtam fel. Lényegében a „Floor” osztályban csak egy darab lapot hozok létre egy bizonyos mérettel, ami a földfelszínt fogja szimbolizálni. Majd ebből a lapból rajzolok ki hat darabot. Először is kiszámolom, hogy melyik lap fölött helyezkedne el a játékos, ha végtelen sok ilyet pakolnék le a szintérre úgy, hogy teljes egészében lefedjék azt. Ha ez megvan, akkor eltranszformálok az eredeti lapot az így megállapított pozícióba, majd kirajzolom még az ezt közvetlenül körülvevő lapokat is. Ezáltal ahogy mozog a játékos, úgy fog a talaj is vele együtt mozogni.

Ez felvet még egy problémát, ha a játékos nagyon magasan repül a talajhoz képest, akkor könnyen rájöhetne a csalásra, hiszen látná a kirajzolt lapok határát. Ennek orvoslására a „PlanetEarth” szintér alap shaderében egy egyszerű lineáris interpolációval köd/homokvihar effektust készítek, így megakadályozva, hogy túl messzire lásson a játékos.

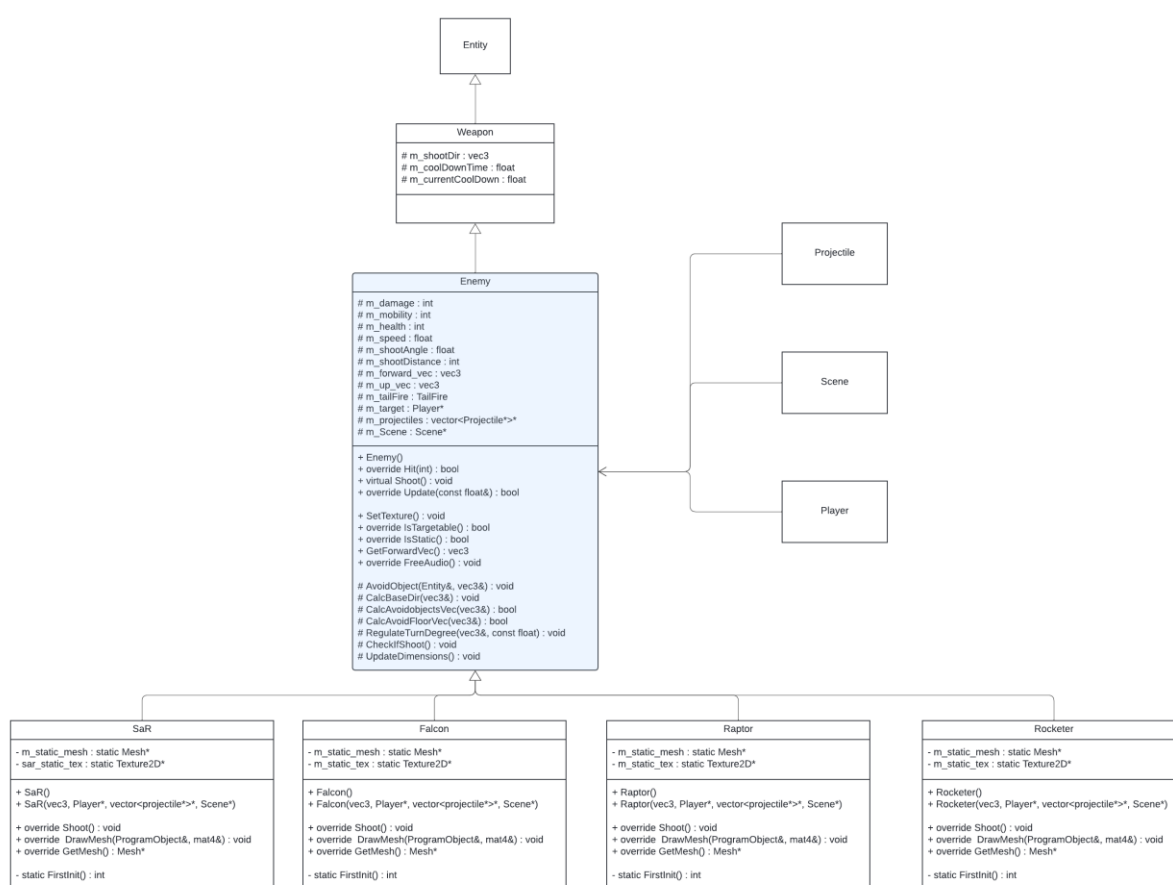
Egy talaj vagy lap kirajzolásához nem használom fel a Mesh objektumot. Pusztán létrehozok egy csúcspontokból álló lapos négyzetrács hálót, majd a csúcspontjait transzformálok valamilyen függvény segítségével felhasználva az adott csúcspont indexét paraméterként. A „PlanetEarth” pályán látható dűnék kirajzolásához például a 35. ábrán látható függvényt használom.



35. ábra: Floor példa

4.10 Ellenségek

Az ellenséges űrhajók viselkedése van talán a legnagyobb hatással a játékelményre. Nagyon sok kísérletezés és konstansokkal való játszadozás kellett ahhoz, hogy az ellenséges objektumok realiztikusan viselkedjenek, kihívást jelentsenek, ugyanakkor ne túl nagy kihívást. Emellett fontos volt az is, hogy hatékonyan hozzák meg a döntéseket és ne terheljék le túlságosan a hardvert, hiszen a játékban elvben akármennyi ellenség lehet egyszerre a pályán. Pontosan ezek miatt ez az a része az alkalmazásnak, ami a legtöbbször lett újraírva és gondolva. Ebben a részben az ellenséges űrhajók viselkedését fogom tárgyalni.



36. ábra: Enemy osztály osztálydiagramja

Az ellenségek implementálásánál is szem előtt tartottam, hogy a későbbiekben könnyedén lehessen újféle ellenséges űrhajókat hozzáadni a játékhoz. Ezért a viselkedésük logikáját az „Enemy” ősosztályban implementálom, míg a leszármazott osztályokban csak az adott típusú ellenség attribútumait és megjelenését kell specifikálni a konstruktorban. Ezek az attribútumok az életerő, sebesség, mobilitás, sebzés, lövési távolság, maximum tüzelési szög és a tüzelési

rát. Ha egy adott ellenség típus nem lézer lövedéket lő ki, akkor az őosztály „Shoot” metódusát felüldefiniálva tetszőleges lövedék típust tudunk beállítani neki.

A 36. ábrán látható, hogy minden leszármazott osztály rendelkezik egy statikus Mesh, illetve textúra objektummal. A leszármazott ellenségek példányai tehát egy közös, statikus modellt és textúrát használnak a „DrawMesh” függvényeikben. Ezt meg lehet tenni, hiszen az ugyan olyan típusú ellenségek megjelenése megegyezik, csupán a pozíciójuk és orientációjuk más, így a kirajzolást tekintve elég csupán a transzformációs mátrixot eltárolni példányonként. Így minden egyes leszármazott Enemy típusnak csak az első példányosítása fogja betölteni az adott modellt, illetve textúrát. Azért csak az első példányosításkor történik meg a betöltés, mert ezekhez a műveletekhez szükséges egy működő OpenGL kontextus megléte. A modellek és textúrák betöltése meglehetősen lassú folyamat, így nagyban rontaná az alkalmazás teljesítményét, ha minden egyes ellenség példányosításánál újra be kéne tölteni ezeket.

Egy ellenséget a játékmódel szemponájából a pozíciója, előre mutató vektorja és felfelé mutató vektorja reprezentál.

Ennél az osztálynál is az „Update” metódus lesz az, amiben a logika nagy része implementálva van. Mielőtt azonban rátérnék erre, röviden összefoglalom az osztály többi, fontosabb metódusának működését és feladatát:

- **bool Hit(int):** Ezzel a függvénnyel kezelem le azt, hogy mi történjen, ha az adott ellenséget eltalálja egy lövedék. A paraméterében kapott egész szám a lövedék sebzését jelenti. A függvény levonja az ellenség életerejéből a lövedék által okozott sebzést. A visszatérési érték azt jelzi, hogy az ellenséges űrhajó megsemmisült-e a találat következtében.
- **void Shoot():** Létrehoz egy, az adott ellenség típus által használt lövedéket, majd hozzáadja ezt a jelenet lövedékeinek tömbjéhez.
- **void UpdateDimensions():** Az ellenség hitboxának dimenzióit módosítja lineáris interpolációval az aktuális orientáció szerint (4.7.1).

4.10.1 Az Update metódus

Ebben az alrészben végig vezetem, hogy mi történik minden frameben, amikor egy ellenséges űrhajónak meghívódik az „Update” metódusa.

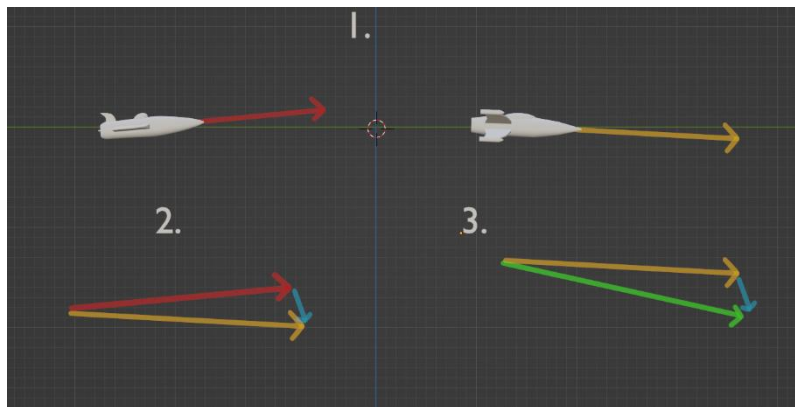
Először is az ellenséges hajó előre mozog az előző frameben kiszámolt előre mutató vektorjának irányába. Az elmozdulás mértéke az ellenség sebességével van súlyozva.

$$position = position + forwardVec * speed$$

Ha az aktuális lehülési idő nem 0, azaz nemrég tüzelt az ellenség, akkor a lehülési időből levonom a „deltaTime” -ot.

A most következő rész a legfontosabb, ki kell számolni, hogy merre haladjon tovább az űrhajó a következő frameben. Ehhez először is létrehozok egy vektor típusú változót, amiben a számítások ideiglenes eredményét fogom tárolni. A haladási irány kiszámításának négy fő lépése van. Mind a négy lépést külön segédfüggvényben implementáltam. Az aktuális irány kiszámítása alatt az ideiglenes vektort mindig normalizáltan tartom, így erre nem térek ki külön.

Az első lépés, hogy kiszámoljam merre menne a hajó, ha minden akadály és korlátozás nélkül kéne választani egy irányt. Ezt a számítás a „CalcBaseDir” metódussal végzem, ami az imént említett ideiglenes irányt fogja manipulálni. Ez az irány pusztán a célpont (m_target), vagyis a játékos helyzetétől függ. Mivel egy ellenséges hajó célja, hogy le vadássa a célpontot, ez a vektor egyszerűen az ellenség pozíciójából a játékos felé mutató vektor lesz ($target\ position - m_position$). Más a helyzet azonban, ha az ellenség „vert helyzetben” van, azaz a játékos közvetlenül mögötte repül, követi őt. Ekkor az előző esethez képest éppen



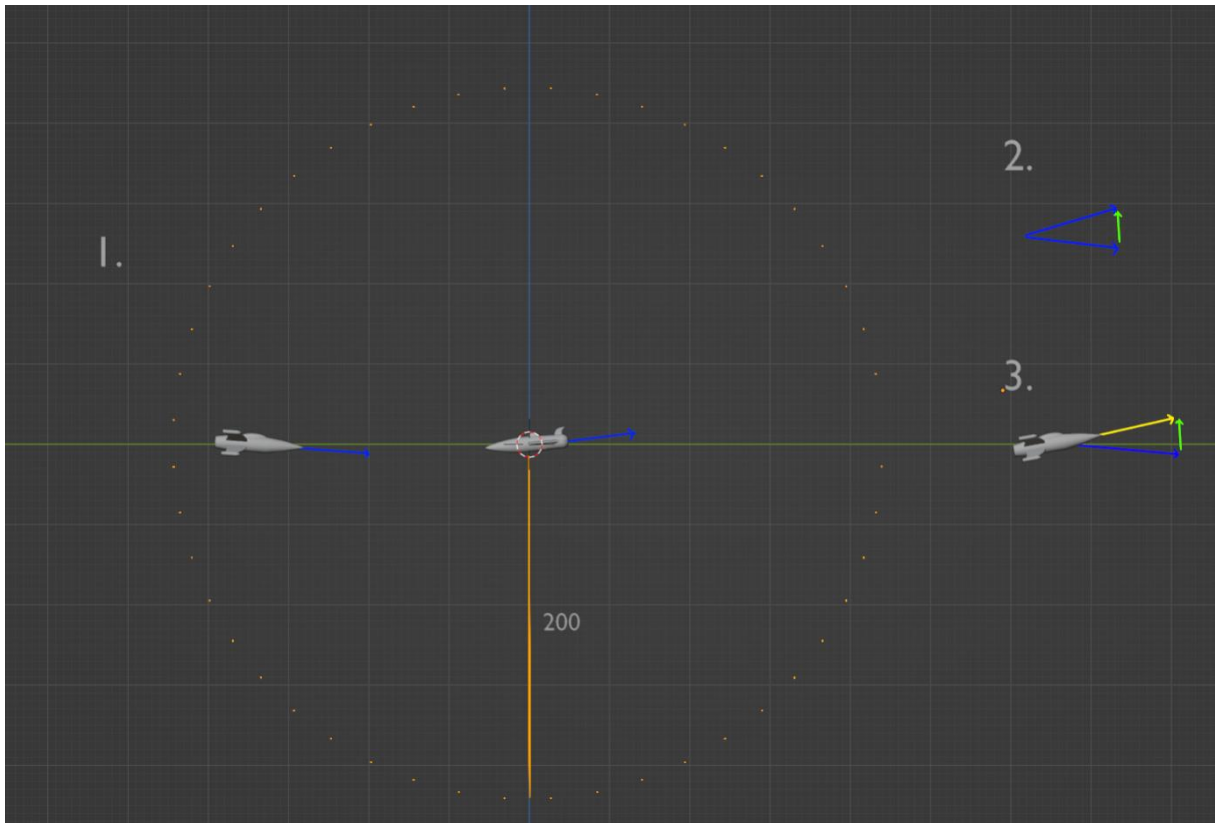
37. ábra: Ellenség vert helyzetben

ellenkezőleg fog dönteni, ki akar térni a játékos útjából és el akar távolodni tőle. Azt, hogy a játékos éppen az ellenség mögött van-e, úgy döntöm el, hogy veszem a skalárszorzatát a játékos helyzetétől az ellenség felé mutató vektornak és az

ellenség előre mutató vektorjának. Ha ez a skalárszorzat pozitív, akkor a játékos az ellenség mögött van. Persze ettől még akármilyen messze lehet. Ezért még egy ellenőrzést elvégzek, mégpedig, hogy a játékos közelebb van-e mint 100 egység. Ha ez is teljesül, akkor az ellenséges űrhajó észleli, hogy vert helyzetben van. Ekkor veszem az ellenséges űrhajó előre mutató vektorját, kivonom ebből a játékos előre mutató vektorját és az így kapott keresztirányú vektor

normalizáltját hozzáadom az ellenséges űrhajó irányához. Erről a 37. ábrán látható egy szemléltetés, ahol a zöld vektor mutatja az új irányt.

Ha megvan ez az alap irány, a következő lépés, hogy az ellenséges hajó megpróbálja kikerülni a veszélyt jelentő akadályokat. Ezt a feladatot „CalcAvoidObjectsVec” metódus látja el. Először is a játékosal való ütközést szeretném elkerülni. Itt két eshetőséget vizsgállok. Az első eset, amikor a játékos és az ellenség egymás felé repülnek és közelebb vannak egymáshoz, mint 200 egység. Ekkor az előre mutató vektorhoz hozzáadom a játékos előre mutató vektorjának



38. ábra: Ellenség a játékosal szemben

negáltjának és az ellenség előre mutató vektorjának különbségét. Ezt a keresztirányú vektort még súlyozom a két objektum bezárt szögével. Tehát minél kisebb szögben haladnak egymás felé, annál drasztikusabb lesz a kikerülő manőver. Ezt a 38. ábra szemlélteti. A másik eset amikor az ellenség a játékos mögött van és nagyon közel hozzá (50 egységen belül). Ekkor az előző esethez nagyon hasonlóan járok el, a különbség csak annyi, hogy ebben az esetben nem negálom a játékos irányát.

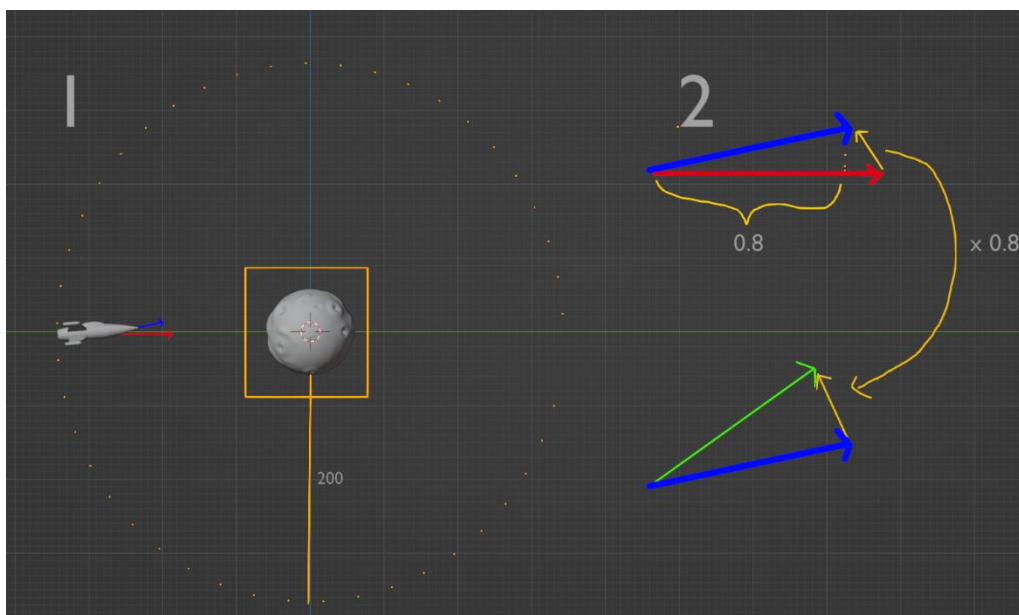
A következő lépés, hogy a szintér többi akadályát vizsgáljam. Ehhez végig iterálok az aktuális Scene objektum Entitásain és mindegyikkel meghívom az „AvoidObject” metódust. Itt megjegyzem, hogy az ellenségek ütközésvizsgálata is itt történik. Minden entitásnál először

megnézem, hogy ütközik-e vele az aktuális ellenség. Ha igen, akkor az Update metódus igazzal tér vissza ezzel jelezve a tartalmazó Scene objektumnak, hogy az adott ellenség megsemmisült.

A kikerülendő objektumokat három csoportra bontottam. Az első csoport a mozgó entitások, amik lényegében a játék jelenlegi állapotában a többi ellenséget jelentik. Itt felhasználom az ellenség ősosztályában felül definiált „IsStatic” metódust, amit az Entity osztálytól örököl. Ez az eset egyszerű, csupán annyit teszek, hogy ha a másik ellenséges hajó közelebb van mint 40 egység, akkor az átmeneti irányhoz hozzáadom a két ellenséget összekötő vektor normalizáltját. Tehát lényegében az „elfelé” mutató vektort.

A másik 2 csoportba olyan entitások tartoznak, amik statikusak. Így akár több hitboxal is rendelkezhetnek, így ezután lényegében a hitboxokat próbálja kikerülni az ellenség.

A második csoportba a „kicsi” hitboxok tartoznak. Egy hitbox akkor számít kicsinek, ha a legnagyobb dimenziója is kisebb mint 100 egység. Egy ilyen hitboxot akkor próbál meg kikerülni az ellenség, ha közelebb van, mint 200 egység, és a hitbox középpontja felé mutató egység vektor és az ellenség irányának skalárszorzata nagyobb, mint 0.7. Ez lényegében azt jelenti, hogy az akadály felé halad az ellenséges űrhajó. Ekkor veszem az előbb említett két vektor különbségét, és az így kapott vektort hozzáadom az új irányhoz. A keresztirányú vektort itt az előbb említett skalárszorzattal súlyozom. Ez egy kicsit realisztikusabb viselkedést eredményez, hiszen így minél kisebb ez a skalárszorzat, annál kevésbé fog módosítani az alap irányán az ellenség. Ezt az eljárást a 39. ábrán szemléltetem.



39. ábra: Ellenséges hajó kis objektumot kerül

Az előbb tárgyalt módszer egészen jól működik kicsi objektumok, illetve hitboxok esetén. Viszont mivel nem veszem számításba a hitbox konkrét méretét, így nagy méretű hitboxokat nem kerülne ki az ellenséges hajó ezzel a módszerrel megbízhatóan.

A nagy hitboxok esetén egy bonyolultabb módszer kell, ami akármekkora objektumra működni tud. Ebben az esetben az ellenség csak akkor kerül ki ezt a nagy hitboxot, ha a jelenlegi pályáján tovább haladva ütközne vele. Ezt úgy állapítom meg, hogy indítok egy „sugarat” az ellenség pozíciójából az éppen aktuális átmeneti előre mutató vektor irányába, majd megnézem, hogy ez a sugár metszi-e a vizsgált hitbox valamelyik lapját, és ha igen, akkor melyik az a lap, amit legelőször metszett el a sugár. Ennek kiszámításához az AAB (4.7.1) névtér „RayIntersection” segéd metódusát használom. Mivel egy hitbox élei párhuzamosak a koordináta-rendszer tengelyeivel, így előre tudni lehet egy adott lap normálvektorát. Ezt kihasználva a sugárral való metszés megállapításához felhasználom a lapokkal egybeeső sík normálvektoros egyenletét.

Vegyük például a hitbox azon lapját, amelyik az x tengely mentén pozitív irányba néz. Ekkor a lap egy pontja (nevezzük p-nek): *hitbox középpontja* + $\frac{\text{hitbox szélessége}}{2}$. A lap normálvektora pedig (1, 0, 0). Innen a sík normálvektoros egyenlete:

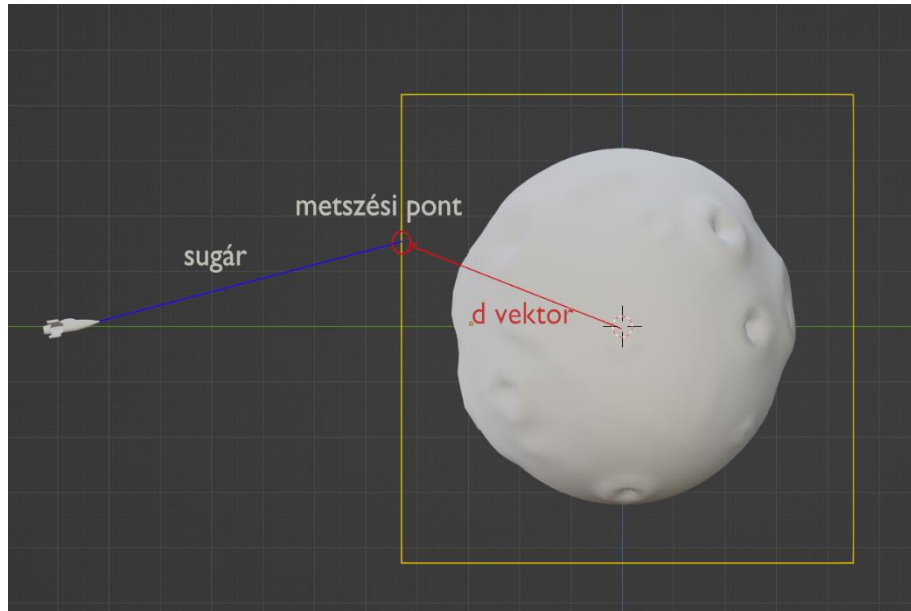
$$1 * (x - p_x) + 0 * (y - p_y) + 0 * (z - p_z) = 0$$

Jól láthatóan csak az X-es tag nem nullázódik ki, így elég ezzel folytatni a számítást. Az ellenség pozíciójából indított sugár parametrikus egyenlete: *ellenség pozíciója* (x) + t * *ellenség irány vektora* (v), ahol „t” a paramétert jelöli. A metszési pont kiszámításához ezt beillesztem a sík egyenletébe: $1 * (x_x + t * v_x - p_x) = 0$. Innen „t” kifejezve: $t = \frac{p_x - x_x}{v_x}$.

A kódban tehát csak ezt az utolsó lépést kell kiszámolnom ahhoz, hogy megkapjam a t – paramétert. Persze még le kell ellenőrizni, hogy a sugár irányának x koordinátája ne legyen 0. Ebben az esetben amúgy sem történhetne metszés. Ha az így kapott t paraméter negatív, az azt jelenti, hogy a hitbox adott lapja az ellenség mögött van, ekkor nem történt metszés. Ha t pozitív, akkor kiszámolom a metszési pontot (beillesztem a t paramétert a sugár egyenletébe), Mivel nem magával a lappal, hanem az azzal egybeeső síkkal végeztem a metszés vizsgálatot, így még le kell ellenőrizni, hogy ez a pont még a hitbox dimenzióin belülre esik-e. Ehhez veszem a metszési pont, és a hitbox középpontjának különbségét (**d vektor**). Majd ebben az esetben a következő vizsgálatokat végzem el ezzel a vektorral:

$$|d_y| < \text{hitbox magassága} * 0.75 \ \&\& \ |d_z| < \text{hitbox hossza} * 0.75$$

Láthatóan egy kicsivel nagyobb dimenziókkal számolok, mint a hitbox tényleges mérete (0.5 - el kéne szorozni), erre azért van szükség, mert előfordulhatna olyan eset, amikor maga a sugár már nem metszi a hitboxot, de a két objektum hitboxa még metszené egymást. A metszési pont keresését a 40. ábra szemlélteti.



40. ábra: sugárral való metszés

Amennyiben azt találom, hogy a metszési pont még a lap dimenzióin belül van, akkor kiszámolom azt az irányt, amerre a legkényelmesebb kikerülni az adott lapot, majd a függvény visszatérési értékében átadom ezt a vektort az ellenségnek.

Ezt a vektort, az éppen vizsgált lap normálvektorára merőleges vektorok (lényegében a másik 2 tengely a koordináta-rendszerben) egy súlyozott összegeként számolom ki. Például, ha a metszés lapjának normálvektora (1, 0, 0), akkor a kikerülési irány:

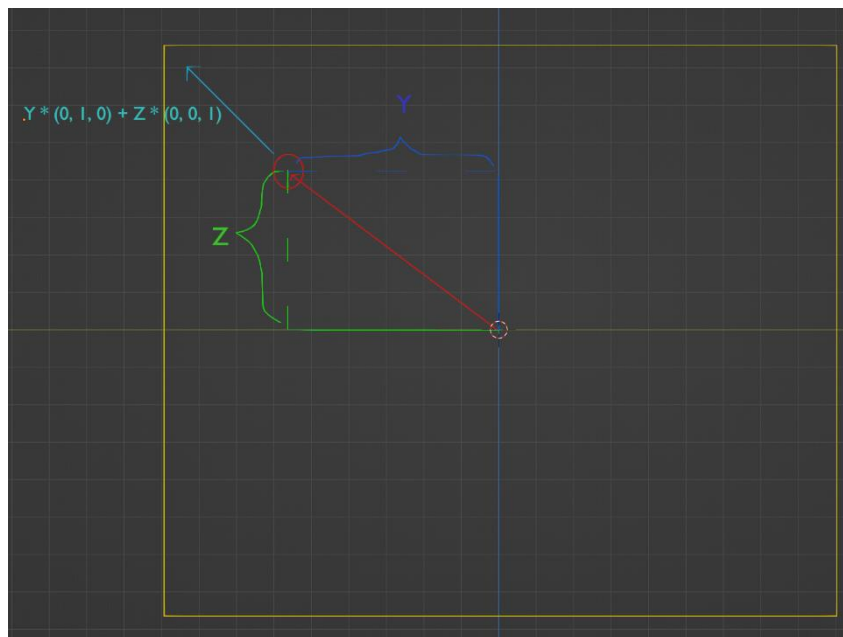
$$Y * (0, 1, 0) + Z * (0, 0, 1) \text{ ahol}$$

$$Y = \text{signum}(d_y) * \frac{|d_y|}{\text{hitbox magassága} * 0.75}$$

$$Z = \text{signum}(d_z) * \frac{|d_z|}{\text{hitbox hossza} * 0.75}$$

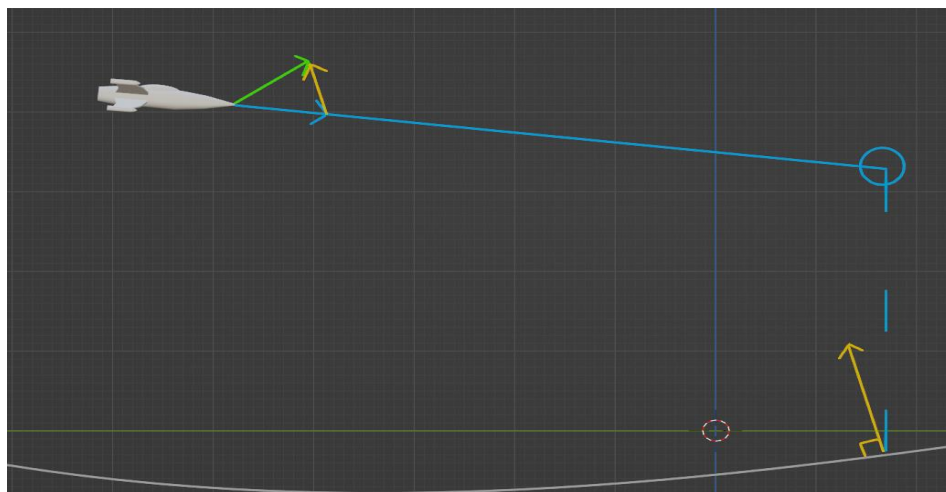
Lényegében tehát megnézem, hogy a metszési pont milyen messze van a hitbox középpontjától az y és x tengelyekre nézve, majd veszem ezek százalékos értékét a hitbox teljes dimenzióihoz

képest. Ennek szemléltetése a 41. ábrán látható. Az így kapott vektort normalizálva visszaadom az ellenséges űrhajónak, ami aztán hozzáadja ezt az előre mutató vektorjához.



42. ábra: Kikerülési vektor kiszámítása

Ezeket az eseteket lefedve már majdnem minden objektumot megvizsgáltam a színtérben. Egy kivétel van még, a talaj. Ahogy azt említettem, a talaj egy kicsit más bánásmódot igényel minden szempontból. Ebben az esetben azonban egyszerű a helyzet. Ha az ellenség észleli, hogy közel repül a talajhoz, egyszerűen hozzáadom az aktuális irányhoz a talaj azon pontjának normálvektorját, ami fölött az ellenség repül. Pontosabban először elmozgatom a vizsgált pozíciót az ellenség haladási irányába 100 egységgel, így az ellenséges hajó „előre” fog tekinteni. A normálvektort még súlyozom az ellenség talajtól való távolságával, és a normálvektorral bezárt szögével, olyan módon, hogy minél közelebb halad a talajhoz és minél meredekebb szögben tart felé, annál drasztikusabb legyen az irányváltás.

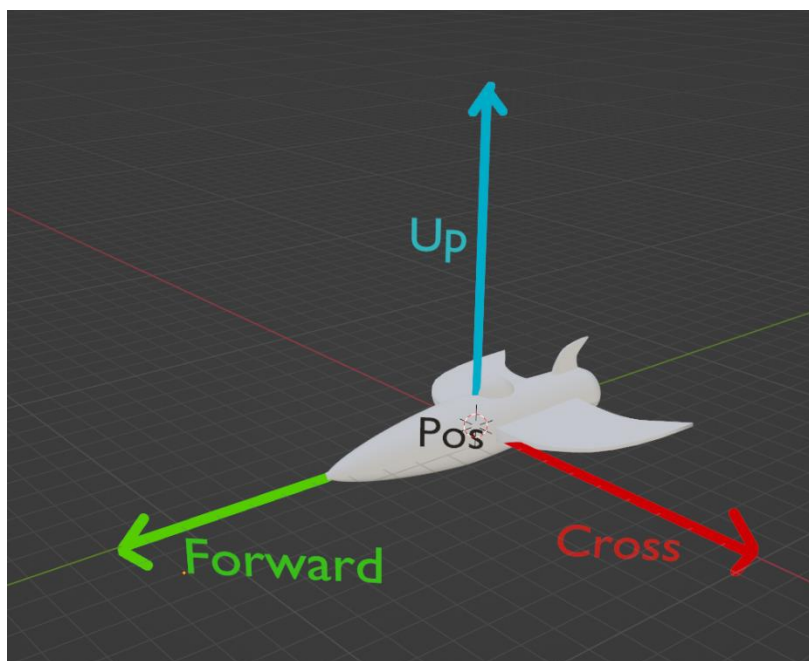


41. ábra: Ellenség a talajt kerüli

Az így elvégzett számítások tehát egy súlyozott összegként előállítják az új haladási irányt egy átmeneti változóban. A probléma már csak az, hogy ez a vektor még jelenleg bármerre állhat, akár teljes mértékben az aktuális haladási iránnyal szemben is. Így ezt a vektort még le kell korlátozni aszerint, hogy az adott ellenség mennyire mobilis, mekkora szögben tud elfordulni adott idő alatt. Így elkerülve azt, hogy például 1 frame alatt 180 fokot forduljon az ellenséges űrhajó, ami finoman szólva nem lenne túl realisztikus. Ezt a módosítást a „RegulateTurnDegree” metódussal végzem. Itt pusztán annyit csinálok, hogy veszem az újonnan kiszámolt irány és a régi irány különbségét. Ha az így kapott keresztirányú vektor hossza nagyobb, mint amit az ellenséges hajó típusának mobilitása megenged, akkor egyszerűen ennek a keresztirányú vektornak egy rövidebb alakját hozzáadom a régi irányhoz, majd normalizálom az így kapott vektort.

4.11 A játékos

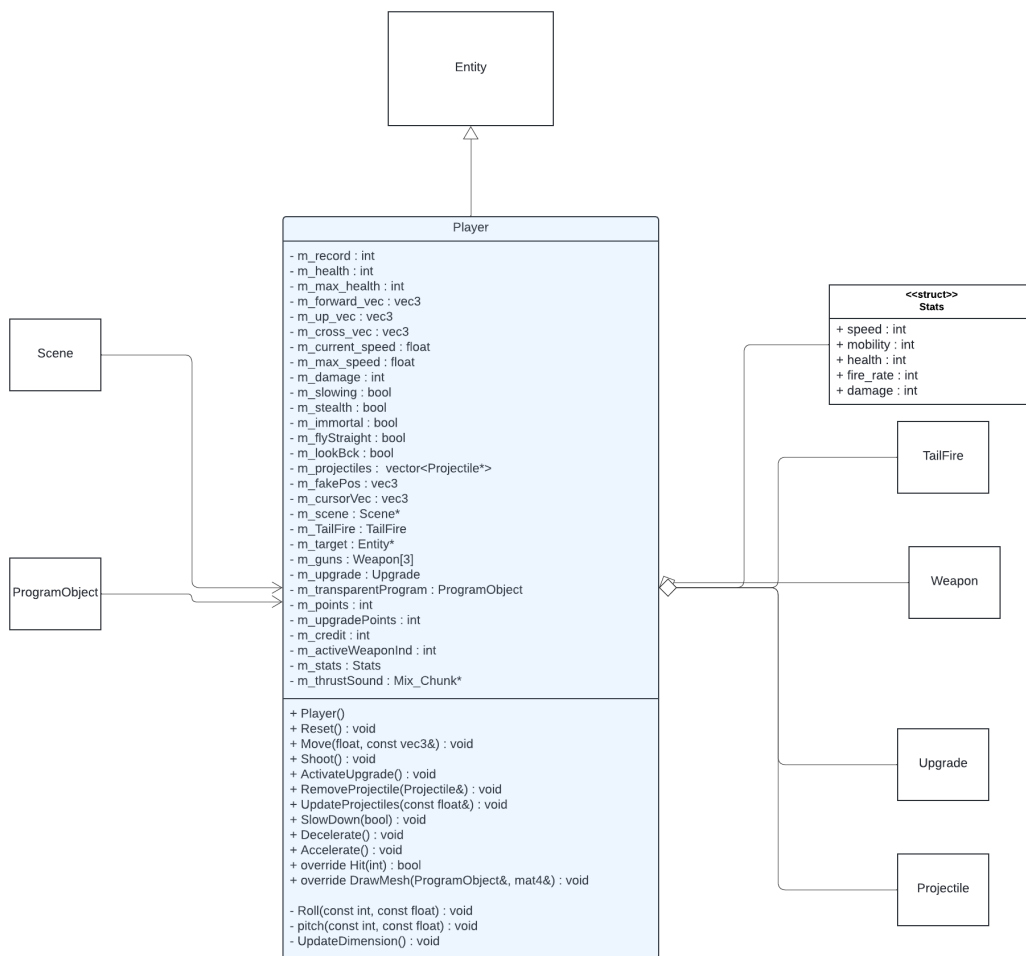
A „Player” osztály feladata a játékos és űrhajójának reprezentálása. Legfőbb felelőssége, hogy kezelje az űrhajó irányításával kapcsolatos felhasználói inputokat. A felhasználó űrhajóját a játékban a 43. ábrán látható módon a pozíciója, előre mutató vektorja, felfelé mutató vektorja és keresztirányú vektorja reprezentálja.



43. ábra: A játékos reprezentációja

A Player osztály esetén a „Move” nevű metódusban implementálom azokat a történéseket, amik minden frameben megtörténnek a játékos űrhajójával. Lényegében ez felel meg a többi

objektum „Update” metódusának. Ez a függvény a delta időn kívül paraméterként megkapja azt az app objektumban kiszámolt vektort, ami a kurzor jelenlegi háromdimenziós pozíciója felé mutat (4.5.2). A függvény először is előre mozgatja az űrhajót a játékos sebességéhez mérten annak előre mutató vektorja szerint. Ha ez megvan, meg kell állapítanom a kurzorpozíció szerinti következő haladási irányt. Ehhez először skalárszorzattal rá vetítem a „cursor_diff_vec” vektort az éppen aktuális felfelé mutató, valamint keresztirányú vektorra. Majd az így kapott vektorokat súlyozom a játékos aktuális mobilitási attribútumával, és hozzáadom ezeket az előre mutató vektorhoz. A keresztirányú vektort a felfelé mutató vektorhoz is hozzáadom (ez is súlyozva van a skalárszorzattal). Ezzel érem el a kurzor botkormány szerű viselkedését. A felfelé mutató, és keresztirányú vektor kiszámításánál is gondoskodni kell róla, hogy a hajó tengelyei egymásra merőlegesek maradjanak. Ezt a vektorok keresztbe szorzásával érem el. Mivel a játékos esetében a fegyverek és a speciális felszerelés mind különálló objektumok, így ebben a metódusban ezeket is frissíteni kell (meg kell hívni az „update” metódusukat).



44. ábra: Player osztály osztálydiagramja

A 44. ábrán láthatóak a Player osztály további metódusai. Szinte minden adattaghoz vannak setter és getter metódusok definiálva, ezeket nem jeleztem külön a diagramban. Mivel a Player osztály további metódusai működésüket tekintve viszonylag egyszerűek, és nagyrészt az inputkezelés a feladatuk, így ezekről csak röviden számolok be az alábbi felsorolásban.

- **Reset:** Visszaállítja a játékost az eredeti kiindulási állapotába. Minden új kör kezdete előtt lefut.
- **Shoot:** Meghívja az aktív fegyver „Shoot” függvényét. Az éppen aktív fegyver indexét az „m_activeWeaponInd” változó tárolja.
- **ActivateUpgrade:** Amennyiben a hajó fel van szerelve egy speciális felszereléssel, meghívja annak „Activate” metódusát.
- **RemoveProjectile:** A játékos saját maga tartja számon, hogy milyen aktív lövedékei vannak éppen, ez a metódus az „m_projectiles” vektorból törli az adott lövedéket.
- **UpdateProjectiles:** Meghívja a játékos lövedékeinek „Update” metódusát.
- **SlowDown:** Beállítja az „m_slowdown” adattag értékét. Amennyiben ennek a változónak az értéke igaz, minden frameben csökken a játékos sebessége. Ezt a LCTRL billentyű nyomva tartása eredményezi (3.5).
- **Decelerate:** Csökkenti a játékos sebességét 2-vel. (felhasználó lefelé görget)
- **Accelerate:** Növeli a játékos sebességét 2-vel. (felhasználó felfelé görget)
- **Roll/Pitch:** A billentyűkkel való kormányzás megvalósítását végzik.

4.12 Fejlesztési Rendszer

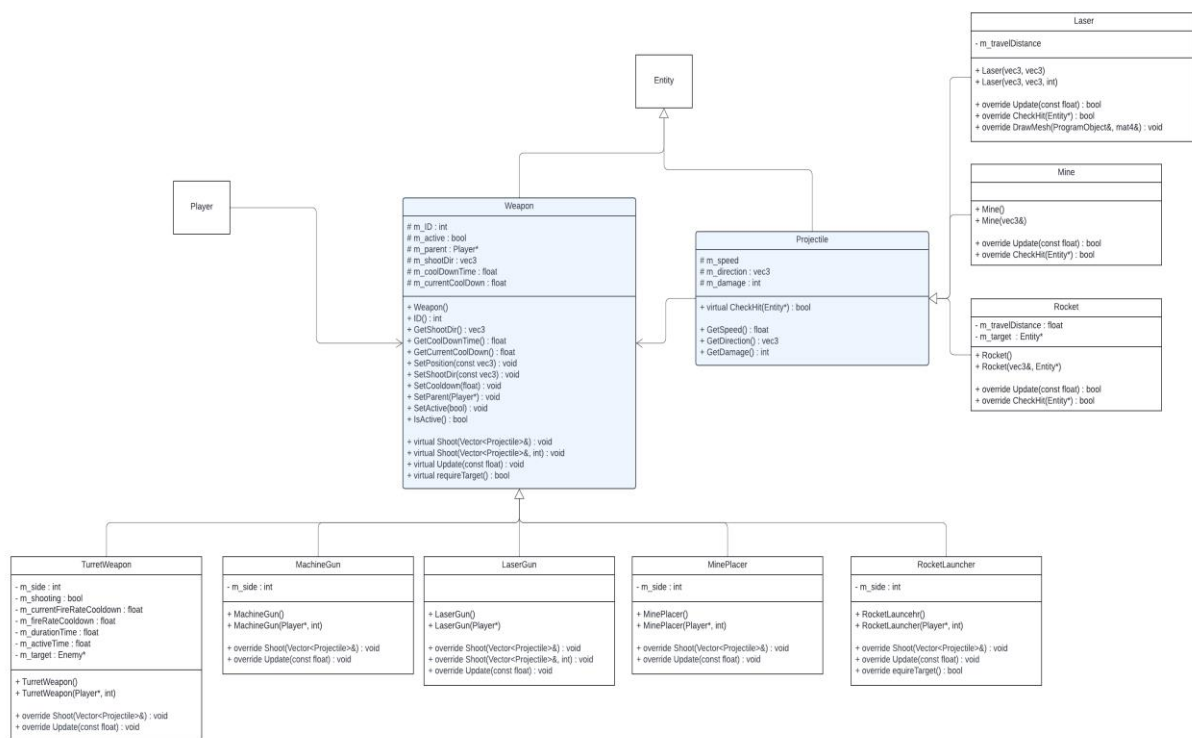
A 44. ábrán látható, hogy a játékos osztály rendelkezik egy „Stats” típusú változóval. A Stats egy struktúra, ami a fejlesztési rendszerben beállított attribútumokat tárolja. Amikor a felhasználó a hangárban módosít egy attribútumot fejlesztési pontokkal, akkor igazából ennek az adattagnak az értékeit változtatja. Ezeket az attribútumokat felhasználom a játékos Reset metódusában. Az érintett attribútumok kiszámítási módja a következő:

- $\text{életerő} = 100 + 10 * m_stats.health$
- $\text{végsebesség} = 90 + 10 * m_stats.speed$
- $\text{sebzés} = 10 + 5 * m_stats.damage$

A mobilitásra vonatkozó attribútumot a „Move” metódusban használom fel, a tüzelési sebességet pedig közvetlenül az elsődleges fegyver megfelelő setter metódusában állítom be.

4.12.1 Fegyverek

A Játékos rendelkezik fegyverek egy 3 elemű tömbjével. Ebben az esetben is kihasználom a polimorfizmust, hiszen a fegyvereknek is van egy őosztálya, ami az alapvető funkcionalitásokat látja el, de a konkrét megvalósítást a leszármazott, konkrét fegyverek végzik. A fegyverek legfontosabb két adattagja a pozíció, és a tüzelési irány. Ezek mellett minden fegyver rendelkezik még egy lehűlési idővel, ami azt mondja meg, hogy mennyi idő eltelte után lehet újra elsütni őket. A fegyver őosztályban deklarálva van egy Player* típusú adattag is. Egy fegyvert tehát hozzá lehet csatolni egy player objektumhoz, ám ez nem kötelező, hiszen ahogy például az ellenségek esetében említettem, a fegyver őosztályból nem csak a játékosra felszerelhető fegyverek származhatnak le. Egy másik példa erre a lövegtorony, ami az ellenséges bázist védi. Az alkalmazás szintjén az is egy fegyvernek számít. Ha azonban egy fegyvert felszerelünk a játékos űrhajójára, akkor az Update metódusban a fegyver pozícióját a játékoséhoz képest számolom ki.



45. ábra: Fegyverekhez és lövedékekhez tartozó osztálydiagram

A 45. ábrán látható, hogy az alap fegyver, azaz a „LaserGun”-on kívül mindegyik fegyverfajta rendelkezik egy „m_side” adattaggal. Ez a változó azt jelöli, hogy a szülő objektum, tehát a játékos melyik szárnyára van felszerelve az adott fegyver. Ezt a fegyver a saját pozíciójának kiszámolásához használom, illetve vannak olyan típusú fegyverek is, amiknek a működését is befolyásolja, hogy melyik oldalon helyezkednek el. Például a TurretWeaponnek „vigyáznia

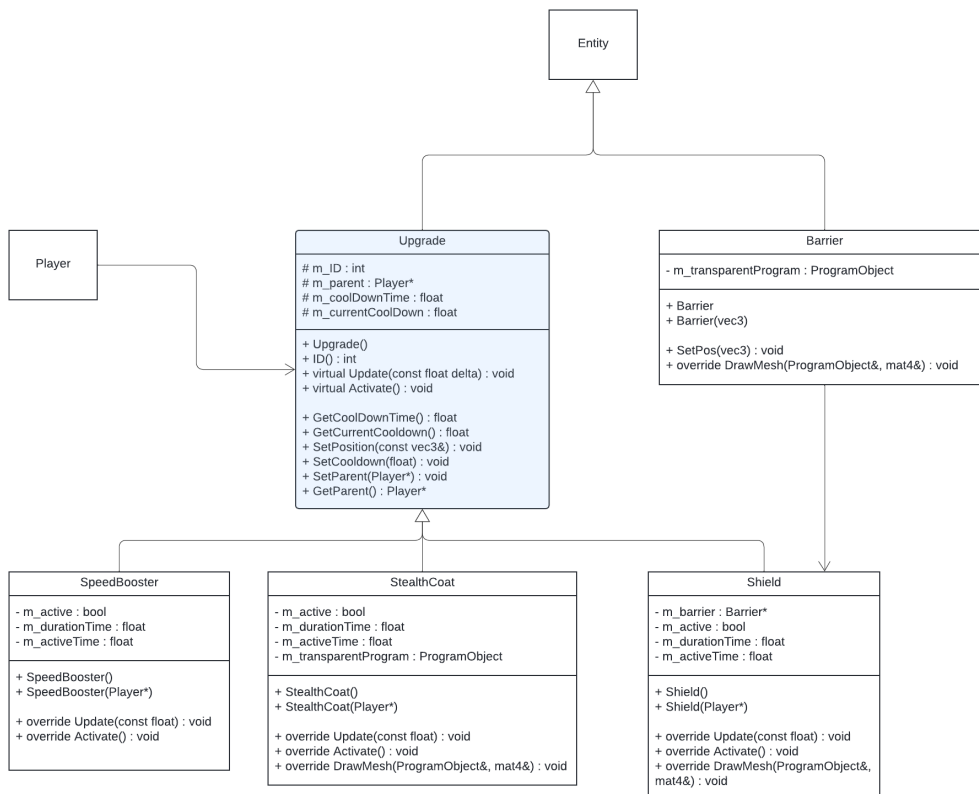
kell”, nehogy eltalálja a játékos hajóját. A fegyverek legfontosabb módszere természetesen a Shoot. Amikor a játékos elsüt egy fegyvert, először az adott leszármazott fegyver objektumban ellenőrzöm, hogy letelt-e már a lehűlési idő, tehát tüzelhet-e újra a fegyver. Ha igen, akkor a Shoot módszerben a paraméterben megkapott lövedék vektorhoz hozzáadok egy új lövedéket. A 45. ábrán szintén látható, hogy ez a Shoot módszer túl van terhelve. A második variációja megkap egy int típusú változót is a paramétereinek között. A lézerlövedékeknek ugyanis meg lehet adni, hogy mekkora legyen a sebességük. Ebben a második paraméterben specifikálhatom, hogy a kilőtt lézerlövedék mekkorát sebezzen. Alap esetben 10-es sebességgel rendelkezik.

Azt, hogy egy lövedék eltalált-e egy Entitást, magában a lövedék objektumban vizsgálom a „CheckHit” módszerben. Azért választottam ezt a megközelítést, hiszen eltérhet, hogy a különböző lövedékek esetén mi számít találatnak. Például egy akna már akkor is felrobban, ha egy ellenség a közelébe repül, míg egy lézerlövedéknek ütköznie kell az objektummal. Na de mi történik, ha egy lövedék nem talál el semmit? Az akna lövedéke kívül minden leszármazott lövedék osztálynak van egy „m_travelDistance” adattagja. Ebben tartom számon, hogy mekkora utat tett meg a lövedék a kilövés óta. Ha ez elér egy bizonyos határt (változhat típustól függően), akkor megsemmisül. Így egy elkészült lövedék nem fog a végtelenségig repülni a színtérben.

4.12.2 Speciális felszerelések

A speciális felszerelések valamelyest különböznek a fegyverektől. Mivel ezek lényegében bármit csinálhatnak, így nincs lehetőség sok funkcionalitást kiemelni az ősoosztályba. Pusztán a virtuális „Activate” módszer felüldefiniálásával tudom specifikálni, hogy mi legyen az adott leszármazott osztály funkcionalitása. A speciális felszerelések aktiválása után is van egy meghatározott időkeret, amin belül nem lehet újra használatba venni azokat. A felszerelések színtérbeli pozícióját szintén a játékos objektumhoz képest számolom ki.

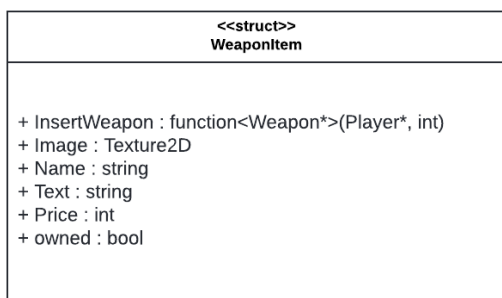
Ezek a felszerelések többnyire a játékos valamely tulajdonságát módosítják. Ezekhez sokszor külön segédmetódusok is vannak definiálva a Player osztályban. Például a Shield fejlesztés aktív ideje alatt a játékost nem sebezhetik meg a lövedékek. Ehhez meghívja a játékos „SetImmortal” függvényét, ami igazra állítja az „m_immortal” értéket. Ekkor, ha a játékost találat éri, tehát meghívódik a „Hit” módszere, akkor nem fogja levonni az életerejéből a sebzést. A speciális felszerelések osztálydiagramja a 46. ábrán látható.



46. ábra: Speciális felszerelésekhez tartozó osztálydiagram

4.12.3 Adattárolók

A program részét képezik még a „WeaponDataStorage” illetve az „UpgradeDataStorage” osztályok is. Ezek lényegében csak csomagoló osztályok egy-egy map adatszerkezet körül. Ezekben a mapekben tartom nyilván, hogy milyen típusú fegyverek és speciális felszerelések



47. ábra: Fegyver tételhez tartozó osztálydiagram

léteznek a játékban. Az Upgrade és Weapon osztályok is rendelkeznek egy-egy „ID” nevű adattaggal. Ennek a változónak a segítségével tudom beazonosítani az egyes leszármazott fegyver vagy felszerelés típusokat ebben a map adatszerkezetben. Ezeket az adat tárolókat főleg a felhasználói felület, és a perzisztencia

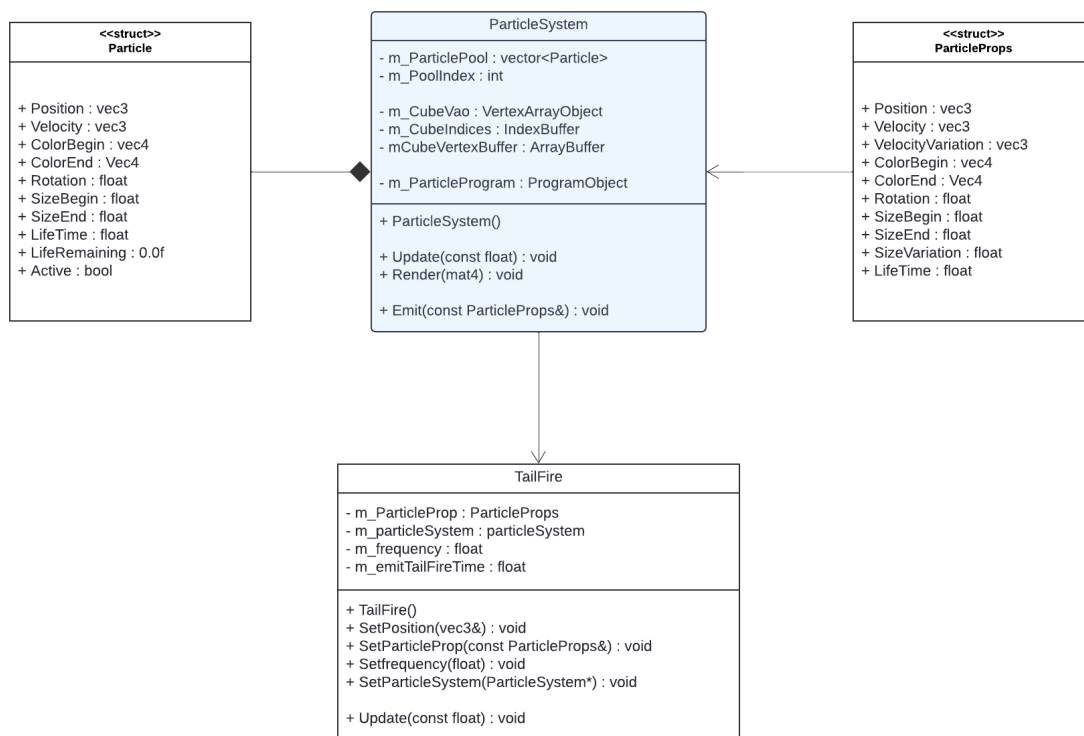
használja. Ezért olyan információkat tárolok bennük, amiket ezek felhasználhatnak. A fegyverek esetén a map adatszerkezetben a 47. ábrán látható struktúra típusú elemeket tárolok. A felszerelések tárolása nagyon hasonló. A 47. ábrán látható, hogy ez a struktúra csupa olyan adattagból áll, amik a felhasználói felület számára fontosak lehetnek. Talán csak az első adattag szorul egy kis magyarázatra. Az „InsertWeapon” adattag egy függvény pointer, ami

meghívásakor létrehoz egy példányt az adott fegyver típusból. Ezt a hangár ablakban használom ki. Amikor a felhasználó felszerel egy adott fegyvert az űrhajóra, akkor a háttérben ennek a függvény pointernek a segítségével a játékos fegyvereket tároló tömbjében helyben példányosítok egy fegyvert.

4.13 Részecskerendszer

A játékban a különböző robbanás és tűz effektek megvalósítására egy egyszerű, CPU oldalon működő részecskerendszert (Particle system) implementáltam. A megvalósításhoz felhasználtam és alapul vettem „The Chernobyl” videóját¹¹, amiben ennek a részecskerendszernek a 2D-s implementálása látható.

Az alkalmazásban minden Scene objektum tartalmaz egy példányt ebből a részecskerendszer osztályból. A Scene osztály feladata tehát, hogy ezt kezelje és frissítse.



48. ábra: Részecskerendszerhez tartozó osztálydiagram

A 48. ábrán látható „ParticleSystem” valósítja meg a részecskerendszer működését. Az osztály tartalmaz egy „m_ParticlePool” nevű vektort. Itt tárolja a rendszer a színtérben szereplő részecskéket. Na de miből is áll egy részecske? Ezt a 48. ábrán látható „Particle” struktúra mutatja be. Egy részecske rendelkezik pozícióval, sebességgel, színnel, rotációval, mérettel és

¹¹ <https://www.youtube.com/watch?v=GK0jHlv3e3w&t=1296s>

élettartammal. Vannak ezek között olyan tulajdonságok, amiket a részecske élettartama alatt interpolálva számolok ki. Ezért ezekből a tulajdonságokból el kell tárolni a kezdő és végső értéket is. Ilyen például a méret, ami tipikusan csökken a részecske életének előre haladtával.

Van még egy adattag a Particle nevű struktúrában, mégpedig az „Active” nevű logikai változó. Ez pontosan azt tárolja, amire először gondolhatnánk, hogy az adott részecske aktív-e, ki kell-e azt rajzolni, illetve kell-e számolni vele. Erre azért van szükség, mert részecskéből egyszerre rengeteg lehet a színtérben (Az én alkalmazásom esetén maximum 1000) és szörnyen lassú lenne, ha menet közben kéne ezeket a részecskéket példányosítani és hozzá adni a ParticlePoolhoz. Ehelyett az osztály konstruktorában előre létrehozok 1000 darab részecskét, mindegyiket inaktív állapotban. Innentől az Update és Render metódusokban csak azokkal a részecskékkal foglalkozok, amelyek aktívak.

Egy részecskét az „Emit” függvény hívásával lehet aktívvá tenni. Ez a függvény paraméterül vár egy „ParticleProps” struktúrát. Ez a struktúra lényegében egy sablon, amit felhasználva beállítom az újonnan aktivált részecske tulajdonságait. A struktúra tartalmaz két plusz adattagot is, ezek a VelocityVariation és a SizeVariation. Ezek azt mondják meg, hogy az újonnan aktivált részecske mekkora mértékben térhet el a sablonban meghatározott sebesség és méret tulajdonságoktól. Ezt kihasználva aktiválni tudok több részecskét is ugyanazzal a sablonnal, és ezek viselkedése mégis más lesz. Például a Scene osztály rendelkezik egy „m_explosionProp” nevű adattaggal. Ebben a sablonban olyan nagyra van állítva a VelocityVariation, hogy lényegében teljesen véletlenszerű irányba fog haladni az aktivált részecske. Így felhasználva ezt a sablont, ha aktiválok például 50 darab részecskét, azok mind különböző irányba fognak haladni ezzel egy robbanás effektet előidézve.

Azt, hogy az Emit függvény a Poolon belül melyik részecskét fogja aktiválni, az m_poolIndex változó dönti el. Ez egy futó index, ami minden aktiválást követően tovább halad a vektor végéről az elejére, majd visszakerül a végére. Így biztos lehetek abban, hogy nem fogok felülírni egy másik aktív részecskét (persze ez csak addig igaz amíg kevesebb aktív részecske van, mint 1000).

Egy részecskét a színtérben egy kis kockával reprezentálok, aminek a mérete, színe és orientációja az adott részecske tulajdonságaitól függ. A kockát a részecskerendszer konstruktorában hozom létre, és különböző transzformációs mátrixokat felhasználva rajzolom ki mindegyik aktív részecske megjelenítéséhez.

A Scene osztályban használt robbanás sablon mellett van még egy gyakran használt ParticleProps, ami köré egy csomagoló osztályt is írtam. Ez a „TailFire”, ami az űrhajók meghajtóiból kiáradó tüzet jeleníti meg. Ez az Update metódusában mindig aktivál pár részecskét abban a részecskerendszerben, aminek a referenciáját megkapta a konstruktorában. Az, hogy mennyit aktiválok egyszerre, az „m_frequency” adattagtól függ. Magukat a részecskéket hasonló módon gyártom, mint a robbanás esetében, pusztán annyi a különbség, hogy kisebb lesz a részecskék sebessége, így nem fognak annyira hirtelen „szétszéledni”. Mivel maga a TailFire objektum folyamatosan halad a szülő objektumával együtt, és mindig aktivál pár darab részecskét, ezért lényegében a részecskéknek nem is nagyon kéne mozogniuk semerre, elég, ha például az űrhajó „maga mögött hagyja” azokat.

4.14 Hangeffektek

A játékban hallható hangeffektek és aláfestő zenék lejátszásához az SDL_mixer könyvtár használtam fel. Az alkalmazásban csak olyan hangfájlokat használtam, amik szerzői jog által nem védettek, szabadon használhatók. A hangeffekteket és zenéket a „pixabay.com”¹² weboldaltól szereztem be.

Az SDL_mixer hivatkozásait és típusait a programkódban a „Mix” prefix alapján lehet megkülönböztetni.

Az SDL_mixer könyvtár hangcsatornákat használ az audio effektek lejátszásához és kezeléséhez. A könyvtár inicializálásakor specifikálni kell, hogy mennyi hangcsatornát használhat a program. A csatornák képesek egymással párhuzamosan működni, így ezt a számot aszerint kellett megválasztanom, hogy mennyi hangeffektet szeretnék maximum egyidőben lejátszani. A könyvtár inicializálását a main.cpp fájlban végzem, ahol lefoglallok 6 darab hangcsatornát. Az alkalmazás során igazából csak ötre lenne szükség, viszont az SDL_mixer esetén csak páros számú csatorna lefoglalására van lehetőség.

A könyvtár különbséget tesz hangeffektek és zenék között. A zenék alapból egy speciális hangcsatornát használnak (-1 -es indexű) és alapértelmezetten hurokban játszódnak le. Az alkalmazásban a háttérzene lejátszása a CMyApp osztály feladata. A zenéket az SDL_mixer által definiált „Mix_Music” típusban lehet tárolni, míg a hangeffekteket típusa „Mix_Chunk”. Az app objektumban példányosítok két Mix_Music típusú változót, ezek lesznek a menü, illetve a játék kör alatt hallható aláfestő zenék. A különböző hangeffektek kezelése a hangot kiadó

¹² <https://pixabay.com/sound-effects/>

típusok feladata. Például minden fegyver rendelkezik egy `m_shootSound` nevű, `Mix_Chunk` típusú változóval, ami az elsütéskor kiadott hangfájlt kezeli.

Bár az lenne a legrealisztikusabb, ha minden hangot kiadó objektumnak saját csatornája lenne, ezt feleslegesnek és erőforrás pazarlónak tartottam. Így a lejátszandó hangeffekteket 5 csoportra osztottam. Ezáltal az ugyanazt a csatornát használó hangeffektek félbeszakíthatják egymást, ám a csoportokat úgy alakítottam ki, hogy ez ne legyen feltűnő vagy zavaró a felhasználó számára. Az alábbi táblázat azt magyarázza, hogy milyen indexű csatornákon milyen típusú hangokat játszik le:

0	Fegyverek elsütésének hangja
1	Robbanások hangja
2	Találatok hangja (lövedék objektummal ütközik)
3	Játékos hajtóművének hangja
4	Ellenségek hajtóműveinek hangja

A játékban minden hangeffekt hangerejét interpoláltam aszerint, hogy a hang forrása milyen messze van a játékostól. A játékos űrhajója, vagy annak fegyverei által kiadott hangok persze mindig „maximális” hangerővel játszódnak le. Az, hogy egy adott hangeffekt milyen messziről hallható már a csatornáktól függ. Például egy robbanás távolabbról kivehető, mint egy lövedék általi találat.

A hajtóművek hangja folyamatos, így ezeknek muszáj volt külön-külön hangcsatornát foglalni. Az ellenséges űrhajók hajtóműveinek hangját egy kicsit trükkösen oldottam meg. Minden egyes ellenséghez új hangcsatornát rendelni bonyolult és erőforrás pazarló lett volna, és igazából felesleges is. Az alkalmazás pusztán monó hangzást biztosít a lejátszott hangfájloknak, azaz a hang forrásának iránya nem kivehető. Így tehát külön-külön csatornák esetén is maximum hangosabb lenne az ellenséges hajtóművek hangja, ahogy közel érnek a játékoshoz. Ezt a viselkedést azonban meg lehet valósítani egy darab csatornával is. A `CMyApp` osztályban lényegében folyamatos hurokban játszom le az ellenséges hajtóművek hangeffektjét, azonban minden `update` metódus futásának elején 0-ra állítom a 4-es csatorna hangerejét. Majd az `Enemy` osztály `update` metódusában folyamatosan hangosabbra veszem azt. Lényegében minden ellenséges objektum hozzáadja a saját hangerejét a 4-es csatornához. Így minél több ellenség van a játékos közelében, annál hangosabban hallható a hajtóműveik által kiadott hang.

4.15 A játék tesztelése

A játék tesztelése alapvetően két módszerrel zajlott.

A grafikus elemek megjelenítését és a shader programok működésének tesztelését nem igazán lehet automatizálni, így ezeknél a manuális tesztelés volt az egyetlen lehetőségem. Megfigyeltem, hogy helyesen töltődnek-e be a textúrák és modellek, ezek orientációja megfelelő-e. Sok esetben pusztán tesztelési céllal definiáltam függvényeket a forráskódban, hogy egyes funkcionálisok vizualizálják. Erre a legjobb példa a hitboxok működése. Mivel egy hitbox a színtérben láthatatlan, így írnom kellett a Scene osztályban egy segédmetódust, ami megjeleníti őket. Ezzel ellenőrizve, hogy jó helyen vannak-e, megfelelőek-e a dimenziók és a mozgó objektumoknál helyesen működik-e az interpoláció.

A grafikus elemeken kívül egy másik fontos, gyakran tesztelendő aspektusa a játéknak a teljesítmény. Ennek tesztelésére külön osztályt definiáltam „Timer” néven. Ebben az osztályban a chrono könyvtár részét képező „high_resolution_clock” objektumokat használok arra a célra, hogy megvizsgáljam egy-egy kódrészlet futási idejét. A Timer osztály konstruktorában elindítom az idő mérését, majd a destruktorban leállítom azt és kiírom a konzolra az eltelt időt. Így egy adott kódblokk méréséhez csak létre kell hozni egy ilyen Timer objektumot a blokk elején. Az alkalmazás leginkább teljesítmény kritikus része az ellenséges objektumok „Update” metódusa, így ezt a tesztelési formát ott használtam a legtöbbször.

4.15.1 Egység tesztek

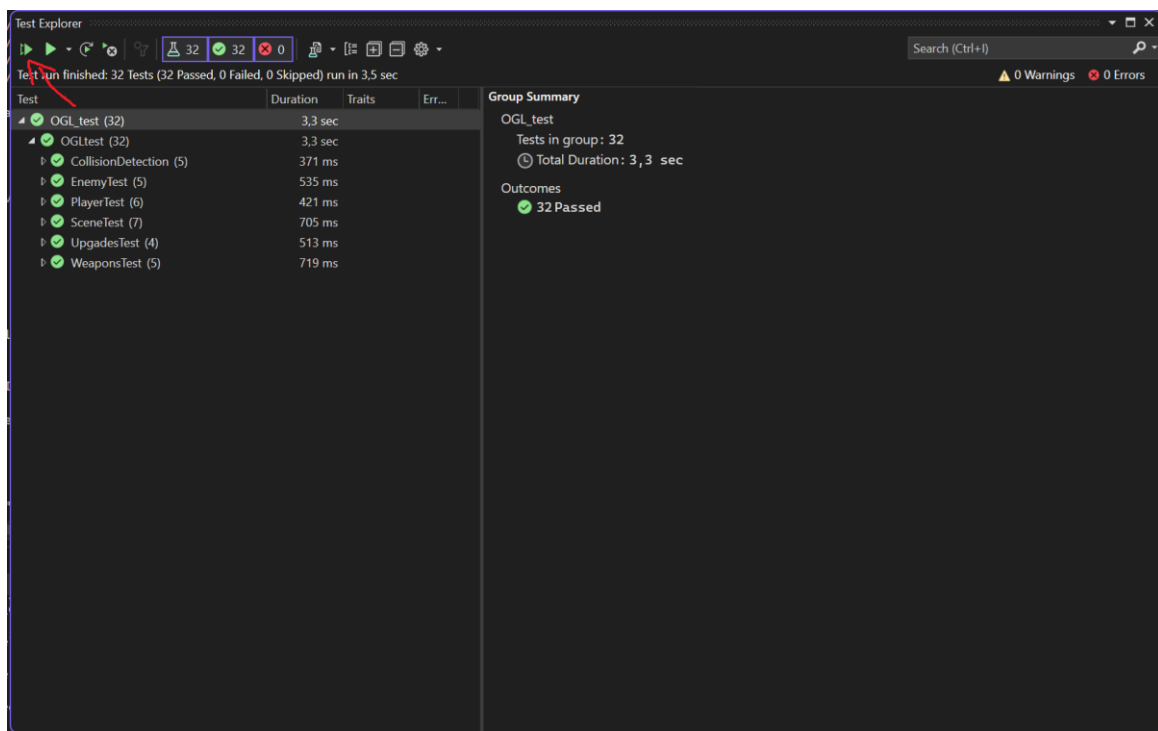
A játéklogika teszteléséhez hozzacsatoltam az alkalmazáshoz egy tesztprojektet is. Az egységtesztek implementálásához a Visual Studio fejlesztő környezetbe integrált CppUnitTestFramework keretrendszert használtam fel. Az alkalmazásban az objektumok általában maguk felelnek a saját modelljeik kirajzolásáért, ehhez persze felhasználnak OpenGL-es függvényeket és funkciókat. Bár a tesztekben nyilván nem rajzolok ki semmit a képernyőre, mégis sok objektum a példányosításakor is használ OpenGL függvényeket. Emiatt a tesztek lefutása előtt egy inicializáló metódusban szükséges volt létrehozni egy OpenGL kontextust, majd a lefutásuk után törölni azt.

Az egység tesztekben a játék fontosabb alap alkotóelemeit teszteltem, külön csoportokra bontva őket. Ezek a tesztcsoportok a következők:

- **Ütközésvizsgálat**
- **Ellenségek viselkedése**

- Játékos viselkedése
- Scene osztály átfogó működése
- Speciális fejlesztések funkcionalitásai
- Fegyverek működése

A teszteseteket Visual Studioban a **Test** → **Run All Tests** menüpontra kattintva lehet futtatni. Amennyiben ez a menüpont nem elérhető, akkor a **Test** → **Test Explorer** menüpontra kattintva lehet futtatni a teszteseteket a megjelenő kis ablak bal felső sarkában található zöld nyíl segítségével, ahogy azt a 49. ábra szemlélteti. A teszteket Debug módban kell futtatni.



49. ábra: Tesztesetek futtatása

5. Fejlesztési lehetőségek

- **Többjátékos mód:** A játék alap elképzelése könnyedén átültethető többjátékos módba, sőt, véleményem szerint maga a játékmenet nagyon jól illeszkedne a mai multiplayer játékok formátumához. A játék körök izgalmasabbak lennének és nagyobb sikerélményt nyújtana a felhasználónak egy-egy ellenfél elpusztítása. Ennek az implementálása már nem fért bele ennek a szakdolgozatnak a keretibe, de a jövőben szeretném megvalósítani ezt az elképzelést.
- **Vásárolható kozmetikai kiegészítők:** A felhasználónak lehetősége lenne a kreditjeit a hajó kinézetének módosítására költeni. Gondolok itt például textúrákra, különlegesebb, más színű hajtómű lángokra és társaira. Az ilyen jellegű kiegészítők is nagy hírnévnek örvendenek a mai játék iparban.
- **Több szintér:** A játék megalkotásánál szem előtt tartottam, hogy könnyedén bővíthető legyen. Több, érdekesebb pálya hozzá adásával nagy mértékben lehetne növelni a játék változatosságát.
- **Több fegyver/felszerelés:** Az előző ponthoz hasonlóan minél többféle fegyver közül tud válogatni a felhasználó, annál inkább nő a játék újrajátszhatósága. Például lehetne implementálni egy felszerelést, ami aktiváláskor feljavítja a hajónk állapotát, vagy egy fegyvert, ami mozgásképtelenné teszi a körülöttünk lévő ellenségeket.
- **Material system:** A játékban jelenleg minden szintérhez tartozik egy alap shader konfiguráció. Ezek a shaderek beégetett változókat használnak például a lokális megvilágítás kiszámításához. Ha azonban minden objektum rendelkezne a saját anyag tulajdonságaival, akkor sokkal szebb lehetne a játék megjelenése.
- **GPU oldali részecskerendszer:** Ahogy azt korábban említettem, a játékban lévő részecskerendszert a CPU kezeli. Általános célú Compute Shaderek használatával azonban sokkal részletgazdagabb robbanásokat, részecske effekteket lehetne elérni.
- **Batch rendering:** A batch rendering egy olyan technika, ami lehetővé teszi azt, hogy több, hasonló objektumot egyszerre, kötegelve rajzoljunk ki. Ezt felhasználva a játék ciklus render metódusa jóval gyorsabban lefutna, ezzel növelve az alkalmazás általános teljesítményét.

6. Összefoglalás

Úgy gondolom, hogy ennek a szakdolgozatnak az elkészítése örökre egy meghatározó emlék lesz számomra. Ez az eddigi legnagyobb és legösszetettebb projektem. A munka közben rengeteget fejlődtem szakmailag és a hasznos tapasztalatok, amiket szereztem örökre velem maradnak. Az az elképzelésem, hogy játékmotor nélkül készítem el a dolgozatot kifizetődőnek bizonyult, hiszen így igazán beleáshattam magam a játékfejlesztés és a grafikus programozás részleteibe.

Egy ilyen jellegű projektnél az eredetileg kigondolt elképzelés sokszor hatalmas változásokon, torzulásokon megy keresztül mire megszületik a végeredmény. Az ember gyakran csak az első suhintás után jön rá, hogy mibe is vágta a fejszáját, így ez teljesen normális. Azonban véleményem szerint nekem sikerült elérnem a kitűzött céljaimat ezzel a dolgozattal kapcsolatban. Legalábbis nagy részben.

Az egyetemi éveim alatt szerzett tudás nélkül természetesen elképzelhetetlen lett volna, hogy ez a dolgozat elkészüljön. A fejlesztés során sok múltbéli tárgyam tananyagával találtam szembe magam. Nyilvánvalóan főként a C++ nyelvről, számítógépes grafikáról és az objektum orientált programozásról szerzett tudásom segített ki leginkább. De a konkurens programozástól kezdve az esemény vezérelt alkalmazások fejlesztéséig rengeteg más ismeretre is szükségem volt a dolgozat megírása közben.

Persze a jó programozó holtig tanul, így nekem is számtalan dolgot kellett magamtól megtanuljak, hogy ez a program megszülethessen. Különösen nagy kihívást jelentett kezdetben a 3D modellek megalkotása. Ahogy már említettem, én készítettem minden modellt, ami a játékban szerepel így a végére talán már egész jól bele is jöttem, sőt mondhatom, hogy egy új hobbival gazdagodtam. Rengeteget tanultam a számítógépes grafikáról is. Könyveket, cikkeket olvastam, videókat néztem a témáról és a dolgozat írása közben még jobban belopta magát a szívembe ez az irány.

Összességében azt mondhatom, hogy sikerült elérnem a kitűzött céljaimat a szakdolgozatot tekintve és egy ilyen játékprogram elkészítésével egy régi álmom vált valóra.

7. Hivatkozások

- **Felhasznált irodalom:**

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman: Real-Time Rendering (Fourth Edition), 2018, [1198], ISBN-9781-13862-7.

- **Online források:**

[2] <https://winter.dev/articles/gjk-algorithm> (2023.11.03)

[3] <https://learnopengl.com> (2023.09.10)

[4] <https://www.opengl.org/Documentation/Specs.html> (2023.9.05)

[5] <https://github.com/ocornut/imgui/wiki> (2023.10.25)

- **Youtube videók és lejátszási listák:**

[6] <https://www.youtube.com/playlist?list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb> (2023.06.15)

[7] https://www.youtube.com/playlist?list=PLlrATfBNZ98foTJPI_Ev03o2oq3-GGOS2 (2023.07.20)

[8] <https://www.youtube.com/watch?v=GK0jHlv3e3w&t=917s> (2023.11.16)

[9] <https://www.youtube.com/watch?v=MDusDn8oTSE> (2023.11.03)

[10] <https://www.youtube.com/watch?v=ajv46BSqcK4> (2023.11.02)

- **Felhasznált zenék és hangeffektek:**

[11] <https://pixabay.com/sound-effects/laser-gun-shot-sound-future-sci-fi-lazer-wobble-chakongaudio-174883/> (2023.11.28)

[12] <https://pixabay.com/sound-effects/medium-explosion-40472/> (2023.11.28)

[13] <https://pixabay.com/sound-effects/missile-blast-2-95177/> (2023.11.28)

[14] <https://pixabay.com/sound-effects/080998-bullet-hit-39870/> (2023.11.28)

[15] <https://pixabay.com/sound-effects/loopingthrust-95548/> (2023.11.28)

- [16] <https://pixabay.com/sound-effects/rocket-loop-99748/> (2023.11.28)
- [17] <https://pixabay.com/sound-effects/mechanicalclamp-6217/> (2023.11.28)
- [18] <https://pixabay.com/sound-effects/sci-fi-music-loop-01-57623/> (2023.11.28)
- [19] <https://pixabay.com/sound-effects/summon-them-now-114743/> (2023.11.28)