



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Rittgasszer Ákos

OPENGL GRAFIKUS KÖNYVTÁRRA ÉPÜLŐ JÁTÉKMOTOR FEJLESZTÉSE

KONZULENS

Dr. Tóth Balázs György

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
2 Játékmotor funkcionalitása	11
2.1 Unreal Engine	11
2.1.1 Funkciók	12
2.2 Unity	13
2.2.1 Funkciók	13
2.3 Saját játékmotor	14
2.3.1 Általános funkciók.....	15
2.3.2 Labirintus specifikus funkciók.....	15
3 Objektum orientált elvek, tervezési minták és fejlesztési folyamatok	16
3.1 Objektum orientált tervezési elvek	17
3.2 Clean Code elvek	17
3.3 Objektum orientált tervezési minták.....	18
3.3.1 Game Loop	18
3.3.2 Update Method	19
3.3.3 Command.....	20
3.3.4 Event Queue.....	22
3.3.5 Observer.....	23
3.3.6 Singleton	24
3.3.7 State	25
3.4 Fejlesztési folyamatok	25
3.5 Tesztelés.....	27
4 Felhasznált külső könyvtárak és programok	29
4.1 Felhasznált programok.....	29
4.1.1 Git	29
4.1.2 Premake	30
4.2 Felhasznált könyvtárak	31
4.2.1 OpenGL (glad, GLFW).....	31
4.2.2 spdlog.....	31

4.2.3 ImGui	32
4.2.4 glm	32
4.2.5 stb_image	33
4.2.6 Bullet Physics SDK	33
5 Program felépítése és implementálása	34
5.1 Layer	35
5.2 Window	35
5.3 Renderer	36
5.3.1 Shader	36
5.3.2 Textúra	37
5.3.3 Camera	38
5.4 Command	38
5.5 Observer	39
5.5.1 Achievement	39
5.6 GameObject	39
5.7 Labirintus	40
5.8 Physics	42
6 Labirintus demo játék	43
7 Továbbfejlesztési lehetőségek	45
8 Összegzés.....	48
Irodalomjegyzék.....	49

HALLGATÓI NYILATKOZAT

Alulírott **Rittgasszer Ákos**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 08.

.....
Rittgasszer Ákos

Összefoglaló

A cél egy egyszerű játékmotor fejlesztése, ami labirintus játékok kezelését és megjelenítését beépítetten támogatja. Ezek mellett természetesen megvalósítja egy alapszintű motor funkcionalitását is.

Funkcióinak alapját a modernkori motorok képességei határozzák meg. Felhasználási területe eléggé specifikus, így az ehhez szükséges funkciók megvalósítására lett fektetve nagy hangsúly. Ezek alapján a motor elsősorban 2D játékok készítéséhez szükséges eszközkészlettel rendelkezik.

A motor fejlesztésekor nagy hangsúly lett fektetve arra, hogy minél könnyebben tovább fejleszthető legyen a kód. Ennek eléréséhez nagy segítséget adtak a tervezési minták, objektum orientált programozási elvek és clean code elvek.

A megvalósítás OpenGL felhasználásával történt C++ nyelven, úgy, hogy a grafikus, platform függő rész minél egyszerűbben lecserélhető legyen más megoldásra. A megjelenítés egyszerűsítéséhez külső könyvtárak is felhasználásra kerültek, amik a hatékonyságot segítik elő.

A motor képességeinek bemutatásához elkészült egy egyszerű játék, ami kizárólag ezt a motort felhasználva készült.

Abstract

The goal is to develop a simple game engine designed to manage labyrinth games and display them. In addition, of course, it implements the functionality of a basic engine.

Its functions were based on the capabilities of modern-day engines. Its field of application is quite specific, so the implementation of the necessary functions has been placed with great emphasis. Based on this, the engine primarily has a toolkit for making 2D games.

When developing the engine, great emphasis was placed on making it as easy as possible to develop the code. To achieve this, design patterns, object-oriented programming principles and clean code principles have been greatly helped.

It was implemented using OpenGL in C++ language, making it as easy as possible to replace the graphical part with another interface. External libraries have also been used to simplify the implementation, which greatly promote efficiency.

To demonstrate the capabilities of the engine, a simple game was created using only this engine.

1 Bevezetés

Napjaink játékeinak alapjai nagyon hasonlítanak egymáshoz. Hasonló a grafikai, fizikai, vagy éppen hang motor, ami lehetővé teszi a programok magas szintű felhasználói élményét. A közös funkciók miatt használnak játékmotorokat, amik egy egységes alapot biztosítanak a játékfejlesztéshez.

Ezek a motorok általánosságban három fő komponensből állnak. Ezek a grafikai megjelenítés, a fizikai szimuláció és az élethű hangképzés. Ezeken az általános részekeken kívül készítenek sokkal specifikusabb modulokat is, amik például folyadékok szimulációjáért, vagy éppen az emberi haj vagy ruházat élethű megjelenítéséért felelnek.

Az említett komponenseknek szorosan együtt kell tudniuk működni. Például két test találkozásakor a játékfizikának megfelelően változik a mozgásuk, amit a képernyőn történő megjelenítésnek le kell tudnia követni. Az ütközések járhatnak hanghatásokkal is, amik tulajdonságai függhetnek a tér fajtájától és méretétől, de a mozgó tárgyak kamerához viszonyított sebességétől is.

A modern játékmotorokkal úgy lehet új játékot létrehozni, hogy a grafikai megjelenítéshez, a fizikai motorhoz és a hang motorhoz hozzá se kell feltétlenül nyúlni, mert azok működése belül implementálva van. Ezen kívül lehetőség van ezeknek a moduloknak a parametrizálásához, aminek köszönhetően teljesen eltérő környezetben levő játékokhoz is lehet ugyanazt a motort használni. Például, ha szeretnénk egy egyszerű, a Holdhoz hasonlatos környezetet létrehozni, akkor azt úgy tehetjük meg, hogy csökkentjük a testekre ható gravitációt és a légkör sűrűségét. Ezekhez a módosításokhoz automatikusan alkalmazkodik a hang és a mozgás grafikus megjelenítése. Lényegében az utóbbi két eszközzel érzékeltetjük az fizikában eszközölt változtatásokat.

Fontos, hogy ezek a motorok sokszor lehetőséget adnak előre elkészített modelleknek, textúráknak és animációknak a felhasználására. Ezek segítségével akár úgy is létre lehet hozni egyszerűbb játékokat, hogy azok elkészítése semmilyen kódolást nem igényel.

Előny, ha egy motor több platformot is támogat. Ezzel megelőzve azt, hogy többször, különböző platformokra kelljen elvégezni a fejlesztést.

Egy játékmotor egyszerű működését úgy kell elképzelni, hogy az adatszerkezet a felhasználói bemenetek, a világra jellemző fizika és a játéklogika alapján módosul, amit megjelenít a grafikus motor és a hang motor. Mindezt úgy kell megvalósítani, hogy elegendően gyors legyen ahhoz, hogy az ember számára élvezhető legyen. Ez a megjelenítés szempontjából azt jelenti, hogy legalább 30 képnek kell láthatóvá válnia másodpercenként, de inkább 60-nak. Ez a szám az úgy nevezett fps¹, ami minél nagyobb, annál kellemesebbnek látja az emberi szem. Az emberi agy másodpercenként 30 képtől látja folyamatosnak a mozgást.

Elmondható, hogy a motor nagyon fontos jellemzője a teljesítmény. Ennek a teljesítménynek a szűk keresztmetszete a grafikai megjelenítés. Ahhoz, hogy megfelelő sebességgel tudjuk feldolgozni az adatokat számos eszközhöz lehet folyamodni. Az, hogy grafikus kártyákat használunk alapvető eszköz, ami nélkül nem is lehet játékmotort készíteni. A teljesítmény növeléséhez hatékony megoldás lehet még a többszálúsítás. Ez azt jelenti, hogy az egymástól függetlenül végezhető műveleteket a processor különböző szálaival végeztetjük el. Bizonyos esetekben az is hatékony megoldás lehet, hogy nem grafikai megjelenítéseket is a grafikus kártyával végeztetünk el, mivel annak teljesítménye nagyságrendekkel nagyobb, mint a processor-é.

Az, hogy grafikus kártyákat szeretnénk programozni felveti azt a problémát, hogy számos különböző gyártó létezik, amik más architektúrájú kártyákat gyártanak. Ezért szükség van egy egységes interfészre, amin keresztül tudjuk őket programozni. Ehhez számos eszköz létezik, de a legelterjedtebb talán az OpenGL és a DirectX. Ezek közül az OpenGL nagy előnye, hogy platformfüggetlen, míg a DirectX csak Windows rendszereken működik.

A játékmotorok még egy fontos jellemzője, hogy 2D vagy 3D játékok készítésére készültek elsősorban. A nagyobb motorok mindkettőt támogatják, de eltérő eszközkészletet biztosítanak az elkészítendő játék típusától függően.

A könnyebb munkához előnyös, ha a motor rendelkezik valamilyen grafikus felhasználói felülettel, amin keresztül egyszerűbben, akár programozási tudás nélkül is egészen bonyolult játékokat lehet elkészíteni.

¹ fps: frames per second – a másodpercenként megjelenített képek száma

A szakdolgozat keretében elkészítendő játékmotor elsődleges célja, hogy egy olyan eszköz legyen, amivel labirintusokat egyszerűen lehessen megjeleníteni és kezelni, amellett, hogy ellássa egy alapvető motor funkcióit is. Mindezt úgy legyen megvalósítva, hogy minél könnyebben lehessen bővíteni és a több platformra való kiterjesztést is könnyen lehessen kivitelezni. Maga a bemutató játék Windows operációs rendszerre készül. A motor elsősorban 2D játékok elkészítéséhez készül, de cél, hogy egyszerűen tovább lehessen fejleszteni 3D megjelenítéshez szükséges funkciókkal.

Egy nagyobb projekt, aminek fontos tulajdonsága a könnyű továbbfejlesztés körülmények közötti tervezést igényel. Ezt tovább nehezíti a több külső fejlesztésű könyvtár, amiknek esetleges módosulása bármikor megtörténhet. Ezeket úgy célszerű beintegrálni a projektbe, hogy minél kevesebb függőség alakuljon ki, ezzel az esetleges módosításokat elősegítve.

Az előbb említett szempontok függvényében mindenképp elengedhetetlen az alapos tervezés, aminek hála egyszerűbb fenntartható kódot készíteni. Ehhez nagy segítséget adnak az objektum orientált programozás tervezési elvei[1]. Ezek az elvek azért lettek kitalálva, hogy jól karbantartható kód keletkezzen, ami jó minőségű és ez a fejlesztés során nem romlik. Ezen kívül az esetleges módosítások egyszerű refraktorokon keresztül megvalósíthatóak. A Clean Code elvek[2] figyelembevétele is segíti a fejlesztést. Ezek alkalmazásával nő a kód olvashatósága, ami könnyebbé teszi a továbbfejlesztést és a mások általi felhasználást.

A komplex rendszerek megvalósítása során általánosságban ugyanazok, vagy nagyon hasonló helyzetek és problémák jönnek elő. Ez kiemelten igaz a grafikus felhasználói felülettel rendelkező programok esetén. Ezek egyszerűbb és egységes megoldására találták ki az objektum orientált tervezési mintákat[3]. Ezek a minták, olyan megoldást kínálnak, amik felhasználásával az előbbieken említett elvek betartásával lehet megoldani a felmerülő problémákat. Használatuk további előnye, hogy széles körben el vannak terjedve, ezért más fejlesztők is könnyen megértik őket és szakszerűen tudják használni.

A motor fejlesztése során fontos a tesztelés. Mivel a program funkciójából adódóan nehéz egyszerűen, jó tesztek készíteni, készíték egy demo játékot, ami reprezentálja a motor képességeit. A játék fejlesztése párhuzamosan folyik a motoréval, ezzel megpróbálva kiküszöbölni a tesztek hiányából adódó hibákat. A demo játék, mind a motor általános, mind a labirintus specifikus funkciókat hivatott demonstrálni, úgy,

hogy közben egy felhasználási példát is nyújt. Ez segít másik fejlesztőknek a hatékony felhasználásban és kiegészíti a dokumentációt. Cél, hogy a játék és a motor fejlesztése párhuzamosan folyjon, úgy, hogy az előbbi mindig tükrözze az utóbbi képességeit.

A bemutatott példakódok a C++ szemantikáját követik, de sokszor csupán a könnyebb érthetőség miatt vannak használva. Ez azt jelenti, hogy a kódok egy része le van egyszerűsítve (pl.: hiányzó függvény vagy változó deklarációk) és sokszor nem is fordítás készek.

2 Játékmotor funkcionalitása

Napjainkban számos különböző képességű és célú játékmotor létezik. Több játékfejlesztő cég elkészíti a saját motorját, amit a saját játécai igényeire szab. Léteznek olyan motorok, ami bizonyos feltételekkel bárki számára elérhetőek és fel lehet használni saját játék készítéséhez.

A két legelterjedtebb játékmotor, ami a feltételek betartásával bárki számára szabadon felhasználható a Unity[4] és az Unreal Engine[5]. Ennek a két motornak vizsgálatával lesz meghatározva a saját motorom funkcionalitása. Mindkét említett motor fejlesztési nyelve a C++.

2.1 Unreal Engine

Az Unreal Engine nagy hangsúlyt fektet a grafikai megjelenítésre. Lehetőség van minél élethűbb tereket megalkotni, amiket számos modellel lehet feltölteni. Rendelkezik modell szerkesztővel is, aminek köszönhetően egészen komplex tárgyak és karakterek elkészítése is lehetséges a motoron belül.

Alapértelmezetten tudunk különböző típusú játékokat, különböző platformra készíteni, amik egy alap kiindulási vázat nyújtanak. Ilyen típus például az fps² játék vagy autós játék. Támogatott platformok a személyi számítógép, a mobiltelefon, különböző konzolok és VR³ eszközök.

A motor rendelkezik egy vizuális script nevű lehetőséggel, amit Blueprint-nek neveznek. Ennek lényege, hogy egy grafikus felületen tudunk dobozok és élek segítségével scripteket készíteni, amik teljes mértékig helyettesíthetik a hagyományos program írását.

A motor kizárólag 3D játékok készítéséhez készült. 2D játékok készítése körülményes és nincsenek hozzá megfelelő eszközök.

² first person shooter – belső nézetű lövöldözős játék

³ virtual reality – virtuális valóság

Rendelkezik egy komplex grafikus felülettel, aminek célja, hogy minél kényelmesebben, minél kevesebb kódolással el lehessen készíteni tetszőleges programot.

2.1.1 Funkciók

A motor számos beépített funkcióval és eszközzel rendelkezik, amik megkönnyítik a fejlesztés folyamatát:

- Előre elkészített mintaprogramok felhasználása kiindulásként
- Modellek importálása, készítése és szerkesztése
- Textúrák importálása, kezelése és szerkesztése
- Shaderek készítése és felhasználása
- Komplex grafikus felhasználói felület, ami szinkronban van a mögötte levő adatszerkezettel
- Hangok kezelése
- Felhasználói bementek kezelése
- Fények és kamerák paraméterezése
- Tetszőleges fizikai környezet kialakítása és paraméterezhetősége
- Terep importálása, szerkesztése és készítése
- Komplex modellszerkesztő komponens
- Előre elkészített modellek, textúrák és animációk
- Blueprint script rendszer
- 3D játékok fejlesztésére használható

A felsorolt funkciók többsége nem a játékmotor funkcionalitásáért felel, hanem a kényelemért. Előre elkészített asset-ekkel és a paraméterek felhasználó felületre történő kivezetésével egyszerűen lehet játékokat készíteni.

Előnye továbbá, hogy a forráskódját bárki megtekintheti és készíthet hozzá addon-okat. Lehetőség van továbbá a motor továbbfejlesztésének, aminek köszönhetően játékspecifikus funkciókat is lehet elkészíteni.

2.2 Unity

A Unity játékmotor elsődleges célja, hogy egyszerűen lehessen elkészíteni egyszerű játékokat különböző platformokra. A legnagyobb hangsúly az egyszerű felhasználhatóságon van.

Az Unreal Engine-hez képest nem rendelkezik olyan komplex modell szerkesztő rendszerrel, de itt is létezik rengeteg ingyenes és fizetős asset, amik szabadon felhasználhatók a fejlesztendő játékokhoz.

Nem rendelkezik komplex grafikus script rendszerrel, de a C# nyelvű scripteken keresztül egyszerűen lehet fejleszteni. Rengeteg beállítás és paraméter ki van vezetve a grafikus felhasználói felületre, de nem tudunk komolyabb játékokat készíteni csupán ezek felhasználásával. Ezt ellensúlyozza, a rengeteg egyszerűen használható könyvtár, amik egyszerűvé teszik a programozást

Platformok közül támogatott a számítógép, a mobil telefon, különböző konzolok és a virtuális valóság eszközök is.

Előnye a számtalan mások által készített modell, példa program vagy akár addon, amik nagy száma annak köszönhető, hogy a Unity nyílt forráskódú.

A rendelkezik 2D és 3D résszel is, amiknek köszönhetően minden fajta játékhoz megtalálhatjuk a megfelelő kiindulási alap és eszközkészlet.

2.2.1 Funkciók

A motor beépített funkcióinak elsődleges célja a minél egyszerűbb felhasználás:

- Kiindulási projektek a különböző típusú játékokhoz
- Modellek betöltése és kezelése
- Textúrák betöltése és kezelése
- Shaderek használata
- Egyszerű grafikus felhasználói felület
- C# scriptelési lehetőség
- Hangok kezelése
- Felhasználói bementek kezelése

- Fények és kamerák paraméterezése
- Tetszőleges fizikai környezet kialakítása és paraméterezhetősége
- Előre elkészített modellek, textúrák és animációk
- 2D játékok fejlesztésére használható
- 2.5D játékok fejlesztésére használható
- 3D játékok fejlesztésére használható

2.3 Saját játékmotor

Az elkészítendő motor funkció az előbbieken ismertetett két komoly képességű motor alapján lett megalkotva, a szükséges egyszerűsítések eszközölésével.

A cél egy olyan játékmotor elkészítése, ami egy olyan eszközkészletet biztosít, amivel egyszerűen lehet labirintus specifikus játékokat létrehozni. Ez alapján a funkciók két részre lettek osztva. Az egyik az általános, a másik pedig a labirintus specifikus funkciók.

A funkciók meghatározása során legnagyobb hangsúly az egyszerű felhasználhatóságon volt, mivel a motor nem, vagy csak minimálisan rendelkezik grafikus felhasználói felülettel. Ez azt jelenti, hogy a motor lényegében egy C++ könyvtár, ami egy játék elkészítéséhez szükséges típusokat, függvényeket és algoritmusokat tartalmaz.

A funkciók egy 2D játék készítését segítik elsősorban. Ez alapján elsősorban a Unity 2D motor képességei lettek felhasználva a funkciók kialakításához.

2.3.1 Általános funkciók

Az általános funkciók, azok, amik tetszőleges játék készítéséhez használhatóak.

Ezek a motor alap képességei:

- 2D objektumok megjelenítése
- Felhasználói bemenetek kezelése
- Fizikai jelenségek modellezése
 - Ütközés detektálása, kezelése
 - Súrlódás és légellenállás
 - Fények

A motor kényelmi funkciói:

- Beépített grafikus objektumok
- Layerek
- Előre elkészített objektumok

2.3.2 Labirintus specifikus funkciók

- Labirintust leíró file kezelésé
- Labirintus generálása
- Labirintus készítése grafikus felületen kézzel
- Labirintus játékhoz köthető alapvető játéklógika

3 Objektum orientált elvek, tervezési minták és fejlesztési folyamatok

Egy összetett program implementálását a hatékonyság és minőség érdekében egy tervezési folyamatnak kell megelőzni. A tervezéskor rengeteg tényezőt kell figyelembe venni. Ilyen például az átláthatóság, a továbbfejleszthetőség, vagy az újra felhasználhatóság.

Mivel az említett tényezők minden szoftverre értelmezhetőek, célszerű egységesen kezelni őket. Ehhez léteznek objektum orientált elvek, amiket előnyös figyelembe venni akár tervezés, akár fejlesztés vagy akár refaktorálás során.

A legelterjedtebb elvek a SOLID[6] elvek, melynél a betűk egy-egy törvényt jelölnek. Ezen elvek megfogalmazzák az alap elvárásokat, de számos egyéb elv[7] kiegészítésével válnak teljessé.

Másik elterjedt eszköz, a tervezési minták [8]. Ezek olyan egységes megoldást nyújtanak objektum orientált tervezési problémákra, amiket szinte mindenki ismer és alkalmaz. Ennek köszönhetően egységes és átlátható megoldásokat lehet alkalmazni, ami könnyebbé teszi mindenki számára a további fejlesztést. Az egyes minták felhasználhatósága sokrétű, de úgy lettek megalkotva, hogy minél inkább eleget tegyenek az objektum orientált programozás elveinek.

Az említett elvek és minták elsősorban abban segítenek, hogy a rendszer jól legyen megtervezve, jól legyenek felépítve és összeillesztve a különböző részek. Ezen kívül léteznek olyan elvek, amik arra lettek kitalálva, hogy magát a kódot tegyék olvashatóbbá és könnyebben értelmezhetővé. Ilyenek a Clean Code elvek.

Az előbbieken említett eszközök használatával lehet legegyszerűbben olyan programot tervezni és készíteni, aminek minősége hosszú távon is fenntartható. Ehhez azonban nem elég az elején figyelembe venni az elveket és mintákat, hanem azokat folyamatosan alkalmazni kell és nem szabad sajnálni azt a munkát, amit miattuk fektetünk bele a fejlesztésbe. Ez a többletenergia már nem is olyan hosszú távon megtérül, például már a hibajavításokat is könnyebbé teszi.

3.1 Objektum orientált tervezési elvek

Ezen elvek alkalmazása során fontos az észszerűség, mivel olykor előfordulhat, hogy egymásnak ellentétes dolgokat állítanak és az adott körülmény ismeretében kell meghozni a döntést, hogy melyik élvez nagyobb prioritást.

A projekt során leghasznosabbnak ítélt tervezési elvek:

- Single-responsibility principle: Az osztályoknak egy felelőssége legyen és az legyen egyértelmű.
- Open-closed principle: Az objektumok legyenek nyíltak a kiterjesztésre, de zártak a módosításra.
- Liskov substitution principle: Minden osztály legyen helyettesíthető a leszármazottaival, úgy, hogy a helyettesítés nem változtatja a működést.
- Encapsulation: A komponensek nem mutassák a belső adataikat a kívüllágnak, módosítása ellenőrzött függvényeken keresztül legyen lehetséges.
- Inheritance: Ne használjuk túl a leszármazást, csak akkor alkalmazzuk, ha indokolt.

3.2 Clean Code elvek

A Clean Code olyan elvárásokat állít a kód felé, amik teljesítésével a kód könnyen olvasható lesz és megkönnyíti a fejlesztők munkáját.

A leghasznosabb elvek:

- Meaningful Names: Mindent úgy nevezzünk el, hogy a neve utaljon a felelősségére és feladatára. A jó kód magát magyarázza, nincs szükség kommentekre. Alkalmazzuk a projektre jellemző elnevezési konvenciót⁴.
- Functions: A függvények minél rövidebbek legyenek és lehetőleg egy dolgot csináljanak. Ha valahol kódismétlés található, akkor azt emeljük ki külön függvénybe. Ne legyen sok paramétere a függvényeknek és megfelelően használjuk a visszatérési értéket.

⁴ Szabálykészlet a forráskód alkotóinak elnevezéséhez

- Unit Tests: Unit tesztekkel jól nyomon követhető, hogy jól működik-e a program az eszközölt változtatások után. Unit testből legyen a legtöbb. Hasznos lehet a Test-driven Development[10].
- Classes: Az osztályok rendelkezzenek a lehető legkisebb felelősséggel, emiatt lehetőleg legyenek minél rövidebbek. Kerüljük a túl mély öröklési láncot. Alacsony legyen az osztályok közötti függőség.
- Refactoring: Fontos a gyakori refaktorálás, amik folyamatosan szinten tartják vagy javítják a kód minőségét.
- Code Smells [11]: A Code Smell olyan részlet a kódban, ami azt sejteti, hogy valamilyen elv megsérült. Ezek figyelembevétele nagyon fontos a refaktorálás során.

3.3 Objektum orientált tervezési minták

A tervezési minták [3] adják a program vázát. Ezeket felhasználva és ezekre építve lehet fenntartható és szép programot fejleszteni. Elsősorban azokra a mintákra szeretnék részletesebben kitérni, amik leghasznosabbnak bizonyultak a játékmotor fejlesztése során. A legtöbb mintát egészen sokáig lehet bonyolítani és optimalizálni, de az alábbiakban csupán az alapok lesznek bemutatva. Ezen kívül meg lesz említve a felhasználási területük is.

Nagyon fontos, hogy mint ahogy a programozás és programtervezés más területein, itt is nagyon fontos az észszerűség. Óriási könnyebbséget tudnak jelenteni a tervezési minták, de a nem megfelelő használatuk még ennél is nagyobb károkat okozhat. Ezért fontos többször átgondolni, hogy használatuk indokolt-e.

3.3.1 Game Loop

[12] Ez a minta képezi a játék törzsét, vagyis alapidinamikáját. A lényege, hogy egy végtelen ciklus futása maga a játék folyama. Ebből abban az esetben lépünk csupán ki, amikor bezárjuk a játékot.

A ciklus törzsében a játéktól függően sok minden történhet. Leegyszerűsítve a dolgokat, itt történik meg az adatszerkezet frissítése a felhasználói input-ok alapján és a kép újrarajzolása, a változott adatszerkezet alapján.

Maga a minta elég egyszerű. Kicsit bonyolultabbá teszi a dolgot, ha figyelembe vesszük azt, hogy különböző számítógépek eltérő hardverrel rendelkeznek. Illetve még ha ugyanazon az eszközön futtatjuk a játékot kétszer akkor is lehet eltérés a játék által felhasználható erőforrások méretében. Ez azt eredményezi, hogy egy gyorsabb gépen, vagy ha kevesebb egyéb program fut a háttérben, a ciklus törzse gyorsabban ismétlődik, vagyis a játék „gyorsabb” lesz. Ezt a felhasználó úgy veheti például észre, hogy az objektumok gyorsabban mozognak.

Ennek a problémának a kiküszöbölésére be kell vezetni az időt, mint paramétert, amittől függ az adatszerkezet és a megjelenés változása. Ez a paraméter, az előző ciklus lefutása óta eltelt idő lesz. Az, hogy hányszor fut le a ciklus másodpercenként a már említett fps érték, ami minél nagyobb annál jobb.

A minta leegyszerűsítve:

```
while (gameIsRunning ())
{
    float elapsedTime = currentTime - previousTime;
    handleInput ();
    update (elapsedTime);
    render ();
}
```

Látható, hogy az adatszerkezet frissítését az eltelt időtől tettük függővé. A mintát szükséges esetben lehet továbbfejleszteni, hogy optimálisabban működhessen.

3.3.2 Update Method

[13] Szorosan együttműködve a leghatékonyabban a Game Loop mintával. Lényegében ez a minta az előbb említett update függvényről szó, pontosabban arról, hogy mi történik a belsejében.

Azt szeretnénk, hogy a játéktérben lévő összes objektum frissüljön. Ehhez először az adatszerkezetben kell végrehajtani a szükséges változtatásokat. Ennek elvégzésére hivatott az update függvény. A nehézség az, hogy az egyes objektumok különbözőképpen reagálnak az idő múlására.

Ezt úgy oldja meg a minta, hogy minden objektumnak implementálnia kell egy update metódust. Ezt az update metódust lehet minden egyes alkalommal hívni amikor frissíteni szeretnénk az adatszerkezetet. Az előzőleg felvázolt okokból célszerű, ha az update függvény megkapja paraméterül az eltelt időt az előző frissítés óta.

A kivitelezéshez valahogy garantálni kell, hogy minden objektumnak legyen update függvénye. Ezt egy virtuális függvénnyel lehet egyszerűen megoldani.

```
class GameObject
{
    virtual void update (float elapsedTime) = 0;
}
```

Ezt az interfészt megvalósító függvények lesznek a játék dinamikus objektumai.

```
class Car : GameObject
{
public:
    virtual void update (float elapsedTime) override
    {
        move ();
    }
}
```

Ezek után már csak arra van szükség, hogy eltároljuk a frissítendő objektumokat és azokat mindig frissítsük. A tároláshoz heterogén kollekcióra van szükség, mivel nem ismerjük az objektumok pontos típusát, csak, hogy frissíthetők. Ez egyszerűsítve így néz ki:

```
List<GameObject*> gameObjects;
gameObject.Add (new Car ());

while (true)
{
    for (GameObject* gameObject : gameObjects
        gameObject->update (elapsedTime);
}
```

Ez a minta gondoskodik arról, hogy az adatszerkezet mindig frissüljön.

3.3.3 Command

[14] Ez a minta lényegében a callback-ek objektum orientált helyettesítői. Sajnos a motorban nem lehet teljesen mellőzni a callback-ek használatát, mivel számos könyvtár és az OpenGL is használ callback függvényeket.

Lényegében a minta célja, hogy a felhasználó által adott inputokat átalakítsuk a program által végrehajtandó utasításokká, függvényekké. Ezenkívül szeretnénk azt elérni, hogy minél könnyebben le lehessen cserélni az egyes inputokra történő reakciókat, úgy, hogy ezt akár a felhasználó is megtehesse.

Ehhez elsőként szükségünk van egy absztrakt osztályra, ami rendelkezik egy virtuális execute metódussal.

```

class Command
{
public:
    virtual execute (Actor actor) = 0;
}

```

Arra, hogy paraméterül megkapjon egy actor-t azért van szükség, hogy tudja min kell végrehajtani a függvényeket. Ezek után létrehozhatunk különböző parancsokat, amiket majd inputokhoz tudunk kötni.

```

class JumpCommand : Command
{
public:
    virtual void execute (Actor actor) override
    {
        actor.jump ();
    }
}

```

Meg kell mondanunk valahogy, hogy milyen input hatására melyik parancsnak kell végrehajtódnia. Ehhez egy függvényre van szükség.

```

Command* handleInput ()
{
    if (spacePressed ())
        return new JumpCommand ();
    return nullptr;
}

```

Ez a függvény felel a parancsok és inputok összekapcsolásáért, vagyis ezt kell módosítani (vagy a hozzá tartozó osztályt) ha máshogy szeretnénk kezelni a bemenetet. A példában látható spacePressed függvény elrejt a platformfüggő bemenetkezelést.

Ezek után a maga a végrehajtás már egyszerűen megvalósítható.

```

handleInput ()->execute(actor);

```

A minta továbbfejleszthető a Command osztály okosításával, úgy, hogy a végrehajtott műveleteket vissza is lehessen vonni. Ehhez a végrehajtások ellentétjeit kell implementálni. Ennek a funkciónak elsősorban a tervező programoknál van nagy jelentősége és nem a játékoknál.

Hasznos továbbfejlesztés továbbá, hogy a végrehajtandó parancsokat eltároljuk, hogy az esetleges lassulások esetén se vesszenek el. Vagyis, ha a program éppen nem tudja végrehajtani a parancsot, akkor berakja egy tárolóba. Ebből a tárolóból

folyamatosan hajtódnak végre a parancsok, úgy, hogy mindig az vesszük, amelyik legrégebben került bele (FIFO⁵).

3.3.4 Event Queue

[15] Lényege, hogy a felhasználó bemenetről érkező inputok megfelelően legyenek kezelve. Elsősorban az a lényege, hogy ne vesszen el bemenet és azok megfelelően legyenek feldolgozva.

Annak, hogy egy bemenet ne vesszen el viszonylag egyszerű a megoldása. Lényegében ugyanúgy kell megoldani, mint a fentebb említett Command minta esetén. A bemenetek akkor veszhetnek el, ha éppen nem tudjuk őket feldolgozni és ezért nem vesszük őket figyelembe. Ehhez szükség van egy FIFO típusú tárolóra, amibe rakjuk be folyamatosan a beérkező adatokat és hajtjuk végre a legrégebben bent levőket.

Itt az lehet érdekes, hogy vannak-e olyan jelek, amik egymás után ismétlődnek, de igazából összevonhatóak egyé. Például, ha előre szeretnénk menni akkor folyamatosan nyomjuk az előre nyilat. Normál esetben ez azt jelenti, hogy a számítógépnek rengeteg “előre” inputot kell lekezelnie. Ezt optimalizálhatjuk úgy, hogy az egymást követő ilyen jeleket összevonjuk egyé és csak az számít mennyi ideig tartott, nem pedig, hogy hányszor lett elküldve a jel. Itt arra kell figyelni, hogy mikor tehető ez meg és mikor kellen külön kezelni a bemenetek.

A mások fontos szempont, hogy az egyes bemenetek feldolgozását követő végrehajtandó parancsok végrehajtása be is legyen fejezve. Itt az alapján lehet kétféle parancsot meghatározni, hogy probléma-e, ha átlapolódnak. Például, ha hangot szeretnénk többször egymás után lejátszani, és nincs idejük befejeződni, akkor az a felhasználónak értelmetlen zaj lesz. Ellenben, ha mozgatunk valamit, ott nem kell erre akkor figyelmet fordítani.

Ezt úgy tudjuk megoldani, hogy jelezzük mikor hajtódott végre egy parancs és csak azután vesszük a következőt.

Ez a minta annyira nem bírt nagy jelentőséggel ennek a motornak az elkészítésekor, de a továbbfejlesztések miatt fontosnak tartottam kitérni rá.

⁵ FIFO: First In First Out tároló, az kerül ki belőle, ami legelőször (legrégebben) került bele

3.3.5 Observer

[16] A minta az egyik leggyakoribb a grafikus felhasználói felülettel rendelkező programok körében. Azokon belül is a Model-View-Control architektúrát[19] használóknál a legelterjedtebb.

Abban könnyíti meg a dolgunkat, hogy ha valamilyen programbeli eseményt követően szeretnénk arra valahogy reagálni, akkor nem kell zavaró függőségekkel és nehezen érthető feltételekkel teleszórni a kódot.

Úgy működik, hogy azokhoz az osztályokhoz, amiket értesíteni szeretnénk létrehozunk úgynevezett observer-eket, amik egy közös őssel rendelkeznek.

```
class Observer
{
public:
    virtual void onNotify (const Entity& entity, Event event) = 0;
}
```

Az egyes observer-ek értesítése után, a paraméterek segítségével el tudják végezni az adatszerkezeti módosításokat. Az olyan osztályokban, amik értesíteni szeretnénk másikat, el kell tárolni az értesítendő observerek-et, egy heterogén kollekcióban.

```
List<Observer*> observers;
```

Ezeknek utána meg kell hívni a megfelelő metódusát, hogy megtörténjék az értesítés.

```
for (Observer* observer : observers)
    observer->onNotify (const Entity& entity, Event event);
```

Az egyes observerek-et össze kell kötni az adatszerkezettel, hogy tudja azt módosítani. Ezt tehetjük leszármazással (az Observer osztályból) vagy tartalmazással. Célszerű tartalmazást használni, mivel a leszármazáskor sérül a Single Responsibility elve.

Az observer minta a Model-View-Controller architektúrán kívül achievement-ek kezelésére is rendkívül alkalmas. Az achievement-ek arra használatosak, hogy nyomon tudjuk követni és visszajelezni a játékos teljesítményét.

3.3.6 Singleton

[17] A minták bevezetésekor volt szó arról, hogy ésszel kel használni őket. Ez erre a mintára hatványozottan igaz. Ennek oka, hogy ez a minta nagyon egyszerű és nagyon sokszor a rossz tervezést követően egyszerűen lehet vele hibákat javítani.

A lényeg, hogy vannak olyan objektumok, amikből csupán egy létezése indokolt. Ilyenkor felmerül, hogy ezt az egy példányt minél több helyről szeretnénk elérni. Ez könnyen ahhoz vezethet, hogy van egy globális isten osztályunk, amit minden más használ és módosít. Ez számtalan objektum orientált elvet sért meg.

Implementálás szempontjából ez statikus függvényeket jelent, amiken keresztül lehet a singleton osztályt módosítani.

```
class Singleton
{
public:
    static Singleton* instance ()
    {
        if (instance == nullptr)
            instance = new Singleton ();
        return singleton;
    }
private:
    static Singleton* instance;
}

Singleton::instance ();
```

Gyakori használat például, ha egy ablakunk van, amire minden ki szeretne rajzolni. Ekkor helytelen több ablak példányt létrehozni, mert konfliktusban lennének egymással. Az említett példa nem minden környezetben állja meg a helyét, van amikor indokolt több ablak példány független létezése.

Másik példa egy log rendszer. Ezt a program minden részéről szeretnénk használni. Ekkor lehetséges, hogy a többszöri példányosítás nem okozna konfliktust, de teljes mértékig felesleg, mivel egy közös erőforrás is el tudja látni a feladatot, tartalmazni a szükséges adatokat. Ez azért is előnyös, mert ha valahol módosítunk egy beállítást, az azt eredményezi, hogy mindenhol a módosításnak megfelelően fog működni, ez pedig megőrzi az egységességét (fontos szem előtt tartani, hogy ez valóban célunk-e).

Fontos leszögezni, hogy ha felmerül egy Singleton osztály gondolata, gondoljuk át kétszer is, hogy tényleg ez-e a helyes megoldás. Ha rosszul használjuk, az óriási gondokat tud okozni.

3.3.7 State

[18] Ez a minta lényegében az állapotgépekről szól. A játékoknál gyakran előfordul, hogy az adott állapottól függ, hogy mit tudunk csinálni és mit nem. Például, ha ugrani szeretnénk akkor figyelembe kell venni, hogy nem vagyunk-e a levegőben. Mert ha igen akkor nem tudunk megint ugrani (kivéve, ha olyan a játék).

Az állapotgépek legszemléletesebben gráfként vagy táblázatként ábrázolhatóak. A lényegük, hogy vannak állapotok és események. Az határoz meg egy állapotgépet, hogy milyen állapotból milyen esemény hatására milyen állapotba tudunk eljutni. Például a “földön” állapotról az ugrás parancs hatására eljuthatunk a “levegőben” állapotba, ahol viszont az ugrás parancsot kiadva nem történik semmi.

Az implementálás rengeteg feltételt tartalmaz ezért nagyon fontos a beszédes elnevezés és a megfelelő típushasználat.

3.4 Fejlesztési folyamatok

A fejlesztési folyamatok szétdarabolása elősegíti a hatékony munkát és a kód minőségének fenntartását.

A feladatok a végrehajtás sorrendjében:

1. Program vázának kialakítása

A program váza magában foglalja a program belépési pontját és a motor által keletkező könyvtár bekötését a példa alkalmazásba. Azoknak az osztályoknak az implementálását takarja, amelyek reprezentálják magát a grafikus alkalmazást, annak specifikált példányát és az egész program magját, ahol található a belépési pont. Mindez úgy történik, hogy a motor minél jobban elkülönüljön a játéktól.

2. Külső források, könyvtárak integrálása

A fejlesztés egyik legnehezebb folyamata. A nehézsége abban rejlik, hogy úgy kell a könyvtárakat integrálni a projektbe, hogy azok könnyen lecserélhetőek legyenek. Például az OpenGL meghajtásáért felelős függvényeket, úgy be kell becsomagolni, hogy minél kevesebb munkával meg lehessen oldani egy új platform bevezetését.

A különböző könyvtárakat egymással is és a motorral is össze kell kötni, úgy, hogy a használatuk minél egyszerűbb és kényelmesebb legyen, anélkül, hogy összeakadjanak egymással.

3. Ablak elkészítése

Az ablak fejlesztése az a pont, ahol látható eredménye van az eddigi munkának. Itt is nehézség, hogy úgy kell létrehozni, az ablakot, hogy az könnyen lecserélhető lehessen más platform vagy technológia ablakára.

Itt még csak a felhasználótól független tartalmak megjelenítése lehetséges, tekintve, hogy a felhasználói bemenetek még nincsenek lekezelve. Ezen kívül egy egyszerű és akár testre is szabható felhasználói felület elkészítése is cél.

4. Felhasználói bemenetek kezelése

A felhasználói bemenetek bekötése is nehézséget jelent abból a szempontból, hogy eltérő platformokon eltérően kell megvalósítani. Ennek megoldására ez is elrejtésre kerül, a motor egy interfésze mögé, ahova bármikor új eseménykezelést lehet bekötni.

Ezután már lehetséges, hogy a felhasználó befolyásolja a kijelző tartalmát, de még csak direkt grafikus függvények hívásával. Ennek oka, hogy a játék logikája és adatszerkezete csak később készül el.

5. Labirintus kezelést támogató modul fejlesztése

Lehetővé teszi véletlen útvesztők generálását és megoldását. Illetve ezek adatszerkezetben történő eltárolását, mentését és beolvasását.

A motor még nem képes megjeleníteni az útvesztőket.

6. Labirintus modul bekötése a motorba

A motorba bekerül a labirintus kezelő modul. Ezek után már képes a labirintus adatszerkezetével dolgozni és grafikusan megjeleníteni azt.

7. Alapvető fizika bekötése

Az alapvető fizikai törvények megvalósítása. 2D motor révén elsősorban súrlódás és ütközésetektálás része az alap fizikának.

A fizikát össze kell kötni a grafikus elemek adatszerkezetével, a játéklogikán keresztül. Mivel a játéklogika csak később kerül lefejlesztésre, itt még csak a megfelelő interfész osztályok létrehozására van lehetőség.

8. Alapvető játéklógika implementálása

A játéklógika elkészítésével létrejön az adatszerkezet. Elkészülnek a játékbeli parancsok és összeköttetés alakul ki a felhasználói bemenettel. A grafikus felület a mindenkor adatszerkezetet tükrözi. Kapcsolat épül fel a fizika és az adatszerkezet között, amivel a fizikai törvények láthatóvá válnak a képernyőn.

A cél, hogy a logika csak a minimális funkcionalitást valósítsa meg, és a fejlesztők által testreszabható és továbbfejleszthető legyen.

9. Példa játék készítése

A demo játék minden más implementációs feladat elkészülte után lehetett kész. Mindemellett fontos szerepet játszott a projekt fejlődésében, abból a szempontból, hogy vele párhuzamosan volt fejlesztve. Ezzel segítette annak ellenőrzését, hogy jól működnek-e a különálló részek egyben.

Célja az, hogy megmutassa a motor képességeit, lehetőleg minél jobban lefedve annak funkcióit. Egy példaként is szolgál jövőbeli projektek számára.

A játék elkészítésével kiderül mik a motor esetleg hiányosságai és hogy mennyire használható alkalmazás hozható létre az elkészített motorral

3.5 Tesztelés

A tesztelés nagyon fontos eleme minden fejlesztésnek. Ez garantálja azt, hogy ami jó az jó is marad. Sajnos a grafikus alkalmazások tesztelése egy bonyolultabb eszköztárat és bonyolultabb tesztek igényel.

Tesztelés szempontjából nagyon fontosak a unit tesztek. Ezek a tesztek minél kisebb egységek megfelelő működésért felelnek. Miután elkészültek azzal lehet számolni, hogy ha sikertelen a futásuk, akkor valami el lett rontva. Ez alapján nyomon lehet követni, hogy az eddig még jól működő részek továbbra is jól működnek-e.

Grafikus felületek tesztelésénél felmerül, hogy szükség lenne referencia képek és aktuális képek összehasonlítására. Erre sajnos nem találtam megfelelő eszközt, ezért nem készültek ilyen típusú tesztek.

Jelen motornál leginkább az adatszerkezeti változásokon lehetett unit tesztek végezni. Itt lebutított előre elkészített objektumokon lehet műveleteket végezni, amiknél azt lehet vizsgálni, hogy az adatszerkezet az elvártak megfelelően változott-e.

Egy eszköz, ami elősegíti a hibamentes fejlesztést, a tesztelés vezérelt fejlesztést (Test-driven-development). Ennek alapkonceptiója az, hogy előbb készülnek el a tesztek és utána a program. Ez azt jelenti, hogy a tesztek eleinte hibát jeleznek és akkor lehet készre nyilvánítani egy részt, ha a hozzá tartozó teszt hibátlanul lefutott. Ezt ebben az esetben a demo program jelentette.

A program sok komponensből áll, ezért nagyon fontosak az olyan tesztek, amik az egész egységes működését vizsgálják. Ehhez test rendszer híján manuális tesztek lettek használva. Ezek egy része a játék tesztelésére irányult, másik része pedig a motorra, mint könyvtár.

4 Felhasznált külső könyvtárak és programok

A fejlesztés gyorsabbá tételének érdekében használtam harmadik fél által készített könyvtárakat. Ezen kívül használtam olyan programokat, amik segítették a fejlesztés folyamatát és hibák keresését.

4.1 Felhasznált programok

Itt elsősorban az olyan programokat szeretném bemutatni, amik használata nagyban befolyásolta a fejlesztést. A fejlesztői környezetekre és más editorokra nem térek ki.

4.1.1 Git

A Git [21] egy nyílt forráskódú elosztott verziókezelő rendszer. Felhasználóbarát használatához léteznek különböző kliensek. Ezek közül a GitHub [22] az egyik legelterjedtebb. Én is ezt használtam a projekthez.

A git lényege, hogy a projektek változásának nyomon követését egyszerűbbé teszi. Eltárolja magát a forráskódot és annak változásait, ezzel könnyebbé téve a módosítások nyomon követését. Lehetőség van különböző ágak létrehozására, ezzel a több fejlesztő által egyszerre történő fejlesztést teszi egyszerűbbé.

A GitHub kliens emellett rendelkezik számos más funkcióval, mint például issue-k kezelésével, vagy a merge konfliktusok feloldását segítő eszközzel.

Legnagyobb előnye, hogy számos használt könyvtár forráskódja git-en található. Így egyszerűbben lehet beintegrálni őket a motorba, submodule-ként. Ez azért nagyon hasznos, mert a submodule-ok frissítésével mindig a könyvtár legfrissebb verzióját lehet használni.

A git hátránya, hogy forráskód tárolására van kitalálva, ezért a projekt file-ok (pl.: Visual Studio solution file) nem kezeli jól. Ilyenkor előfordulhat olyan, hogy a git-ről való leszedést követően összekavarodik az előzőleg létrehozott projekt. Ez úgy jelenik meg leginkább, hogy a linkelés és az include-olás működésképtelenné válik. Ez elsősorban az összetettebb felépítésű projekteknél fordul elő. Orvoslására célszerű, valamilyen projekt generáló program használata.

4.1.2 Premake

A premake [20] egy parancssori alkalmazás, ami arra képes, hogy a megadott leírás alapján legeneráljon egy projektet, létező forráskódból. Komplexebb projekteknél azért van rá szükség, mert a verziókezelő rendszerek a forráskód tárolására lettek kitalálva és elveszhetnek a projekt beállításai. Ezen kívül számos más ilyen program létezik.

Választásom azért esett a Premake-re, mert nem bonyolult és elterjedt. Ezen kívül előnye, hogy multiplatform és sok fejlesztőkörnyezetet támogat. Ez azt jelenti, hogy egyszerűen tudok váltani környezetet, mert csak a generálás paramétereit kell átvariálnom hozzá.

Használatához script-ekre van szükség, amik leírják a projekt felépítését. Ezeket a scripteket lua⁶ nyelven kell elkészíteni.

Meg lehet adni, hogy milyen projektekből áll a program. Ezeket a projekteket tovább lehet részletezni. Például, hogy mivé fordítsa a fordító (pl.: exe⁷, dll⁸, lib⁹), vagy, hogy használjon-e precompiled header-t¹⁰ és melyik file legyen az.

Meg lehet adni a különböző fájlokhoz szükséges include-okat, és a linkeléshez szükséges adatokat. Ezeket egyszer kell megadni és utána nem kell a beállításokkal foglalkozni, csak egy egyszerű paranccsal legenerálni a projektet.

Az egyszerűbb használatához létre lehet hozni python¹¹ scripteket és batch¹² fájlokat. Így egyszerűen újra lehet generálni a projektet és még a git submoduleok frissítése is automatikusan megtörténik.

⁶ Széles körben elterjed, egyszerű script nyelv

⁷ Futtatható file

⁸ Dinamikus könyvtár

⁹ Statikus könyvtár

¹⁰ Olyan header file, aminek fordítása csak egyszer történik meg

¹¹ Elterjed, magas szintű programozási nyelv

¹² Indításával végrehajtja a benne levő parancssori parancsokat

4.2 Felhasznált könyvtárak

A felhasznált könyvtárak könnyebbítették a fejlesztés folyamatát és optimálisabbá tették az elkészült programot. Ezen kívül egységesebbé is tették azt.

4.2.1 OpenGL (glad, GLFW)

Fontos az elején leszögezni, hogy az OpenGL [23] nem egy könyvtár. Az OpenGL egy szabvány, ami egy olyan függvényhalmazt (API) szolgáltat, amivel meg lehet hajtani a grafikus kártyákat. Ahhoz, hogy használhassuk szükséges, hogy a grafikus kártya támogassa azt, implementálva legyenek a megfelelő műveletek. Ezen kívül szükség van könyvtárakra, amik segítségével meg lehet hívni az OpenGL API parancsait.

Az egyik ilyen könyvtár a GLFW [25]. Ez egy nyílt forráskódú könyvtár, ami eszközkészletet biztosít, az OpenGL alapszintű meghajtásához. Lehetőség van benne ablakokat létrehozni és kezelni, illetve felhasználói inputok detektálására is képes. Ezen kívül tud context-eket kezelni. Előnye egyszerűsége és kis mérete.

A glad [24] a GLFW kiegészítéseként használható. Arra képes, hogy a grafikus meghajtó függvényeit betölti, hogy használhatóak legyenek a C++ kódból. Ezzel lehet kihasználni a modern OpenGL függvényeit.

Az előbbi két könyvtár elengedhetetlen a grafikus megjelenítéshez és a felhasználói bemenetek detektálásához.

4.2.2 spdlog

[26] Ahhoz, hogy tudjuk jelezni a fejlesztő felé, hogy milyen problémák léptek fel, milyen folyamatok fejeződtek be sikeresen, hasznos egy log rendszer. Ennek lényege, hogy szeretnénk fontosabb dolgokról tájékoztatni a programozókat, de a játékosokat nem szeretnénk fölöslegesen zavarni. Ez elsősorban hibajavításnál és fejlesztésnél lehet fontos.

Az információ közlés legegyszerűbb módja, ha simán kiírjuk azt a standard kimenetre. Ez azonban nehezen értelmezhető. Itt jön képbe az spdlog, ami testreszabható formátummal jeleníti meg az információkat.

Az spdlog könyvtár képes tetszőleges szöveget a kívánt formátumban kiírni. Megkülönböztet többféle információ fajtát. Ilyen például a sima információ, a figyelmeztetés, a hiba vagy a debug üzenet. Ezek formátuma egységes, de színben

megkülönböztethetők. Be lehet állítani mondjuk úgy, hogy a hibákat pirossal, a sima információkat pedig zölddel jelezze. Ez segít az esetleges hiba esetén megtalálni a hibához tartozó log bejegyzést.

```
### [00:06:04] Engine: This is info!  
### [00:06:04] Engine: This is error!  
### [00:06:04] Engine: This is warning!  
### [00:06:04] Engine: This is debug!
```

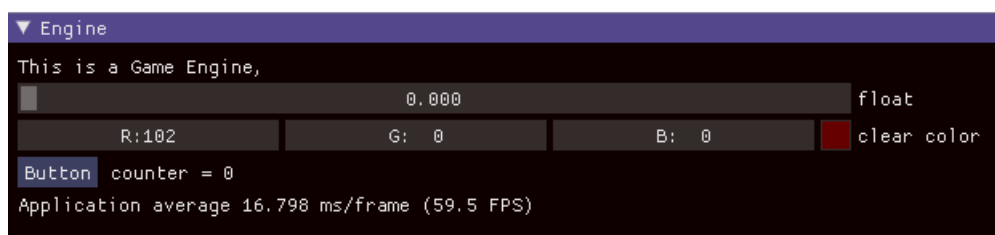
1. Így jelennek meg a különböző log bejegyzések a motor kimenetén

4.2.3 ImGui

Az ImGui [27] egy nyílt forráskódú könyvtár, ami a renderelés optimalizálását hívatott elvégezni. Elsősorban játékok, valós idejű 3D alkalmazások fejlesztéséhez készítették.

Önállóan nem tud működni, szükség van hozzá valamilyen OpenGL könyvtárra (GLFW, glad). Ezek segítségével egyszerűen lehet összerakni egészen bonyolult grafikus felületeket, úgy, hogy mellette nagy hangsúly van a renderelés optimalizálásán.

Tartalmaz előre elkészített objektumokat, például jól kezelhető dialógokat, különböző input mezőket, gombokat és sok más grafikus elemet. Ezek segítségével egyszerűen tudunk összerakni saját ablakokat.



2. Egyszerű egyedi ImGui dialóg

4.2.4 glm

A glm [28] az OpenGL-hez készített matematikai könyvtár. Legnagyobb előnye, hogy készen adja a szükséges matematikai típusokat, amik kompatibilisek az OpenGL könyvtárakkal.

Ebben az esetben a leghasznosabb típusai a vektorok és a mátrixok. Az említett típusokkal végzett műveletek elvégzését is támogatja a könyvtár. Ezen kívül a geometriai

transzformációk végrehajtásához is vannak beépített függvényei. Ilyen többek között az eltolás, a forgatás, vagy a skálázás.

4.2.5 stb_image

Az `stb_image` [29] egy olyan könyvtár, ami textúrákhoz szükséges képek betöltését és a textúrák megalkotását segíti.

Elvégzi azt a folyamatot, amikor a képet át kell konvertálni olyan formában, hogy a számítógép tudja használni. Ebben az átalakított formátumban lehet odaadni a grafikus meghajtónak. Előnye, hogy kis méretű és egyszerű használni.

4.2.6 Bullet Physics SDK

[30] Nagytudású fizikai könyvtár. Rengeteg komplex funkcióval rendelkezik, de számomra az alapvető fizikai törvények érvényesítése fontos.

A leghasznosabb funkciója az ütközésetektálás, ami elengedhetetlen a játék megfelelő működéséhez. Ezen kívül hasznos lehet még a súrlódás és az ütközéseket követő mozgásváltozások, pattogások.

5 Program felépítése és implementálása

A program felépítése során azon volt a fő hangsúly, hogy minél kisebb összefüggő részekre legyen bontva és egy egységes, jól felhasználható eszközöket biztosítson a játék számára. Ehhez alapvetően két projekt létrehozása szükséges. Az egyik a játékmotor, ennek belső logikájához csak minimálisan szeretnénk, ha hozzáférhetne a játékfejlesztő. A másik maga a játék, aminek egész része a játékfejlesztő által készül.

A két projekt létrehozása felveti azt a kérdést, hogy hogyan szeretnénk, ha összefüggés legyen a kettő között. Mivel a motor méretben és így fordulási időben is nagy, nem szeretnénk, ha ezzel lassítaná a játék fordulását. Emiatt kézenfekvő, hogy a motor projektből valamiféle könyvtár generálódjon, amit aztán használhat a játék.

Kétféle könyvtár típus jöhet szóba. Az egyik a statikus könyvtár¹³, a másik a dinamikus könyvtár¹⁴. A könyvtáraknak az alapvető funkciója, hogy a gyakran használt kódrészeket ne kelljen minden egyes használatkor újra lefordítani. A statikus könyvtárak ezt relatíve egyszerűen teszik meg. Eltárolják a futtatható állományokat és azok a fordító és a linker segítségével felhasználódnak amikor szükséges. A dinamikus könyvtárak ezt annyiban bonyolítják tovább, hogy míg a statikus könyvtárból annyi példányra van szükség ahány program fut, itt egy könyvtár van és azt használják a futó programok.

Ebben az esetben nem rendelkeznek számottevő előnnyel a dinamikus könyvtárak a statikus könyvtárakkal szemben. Ellenben statikus könyvtárak készítése, beállítása és használata egyszerűbb.

A motor úgy lett kitalálva, hogy minden megoldható legyen az előre definiált típusok felhasználásával és a belőlük történő leszármazásra. Ez alapján a belépési pont¹⁵ a motorban található.

Ahhoz, hogy létre tudjunk hozni egy játékot, le kell származni a GraphicsProgram osztályból. És implementálni kell a CreateGraphicsProgram függvényt.

¹³ static libraries: lib kiterjesztésű file-ok

¹⁴ dynamic libraries: dll kiterjesztésű file-ok

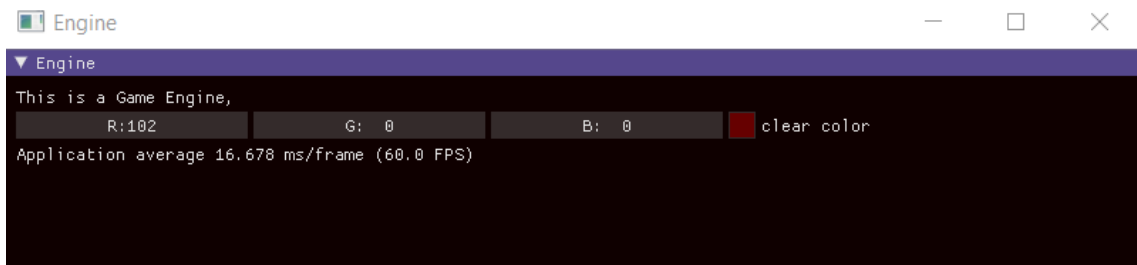
¹⁵ main függvény

```

Engine::GraphicsProgram* CreateGraphicsProgram()
{
    return new LabyrinthGame();
}

```

A leszármazott osztály (LabyrinthGame) példányosítását követően létrejön egy játék, amit aztán testre tudunk szabni. A testreszabáshoz a motor osztályait és azok leszármazottait kell használni. Alapértelmezetten létrejön egy ablak, amiben van egy ImGui-s menü. Ezt az ablakot lehet utána használni a további objektumok megjelenítésére. Alapértelmezetten be lehet állítani a háttér színét és megjelenik a képfrissítés mértéke (fps).



3. Előre elkészített ablak

5.1 Layer

Egy játék elkészítésekor a legfontosabb objektum a Layer (réteg). A Layer-ek segítségével tudunk megjeleníteni dolgokat és kezelni azokat. A játék beépített rétege, az ImGuiLayer. Ez felelős az alapértelmezett ablak megjelenítéséért. Ezt is személyre lehet szabni.

A játék központi része az a ciklus, ami addig fut, amíg be nem zárjuk az ablakot. Ebben történnek az adatszerkezeti és grafikus frissítések, valamint a felhasználói bemenetek kezelése. Ez a Game Loop tervezési minta megvalósítása.

A játék összes layer-e ebben a függvényben kerül frissítésre. Ezen kívül minden más a layer-eken történik, illetve azok objektumain.

5.2 Window

Az ablakokon lehet megjeleníteni az adatszerkezeti változásokat. Az ablakért a Window osztály felelős. Ez az osztály nem képes önálló működésre, szüksége van egy platformtól függő, natív ablakra (esetünkben OpenGL).

Ez a natív ablak végzi a tényleges munkát. A Window osztály, csupán egy felületet nyújt az egyszerű használatához és elrejt, valamint könnyen lecserélhetővé teszi a platformfüggő részeket.

5.3 Renderer

A rendererbe tartozó osztályok és függvények célja, hogy egyszerűsítsék a megjelenítést, valamint elrejtsék a platformfüggő parancsokat és típusokat. Ez azt eredményezi, hogy a platformváltás egyszerűbb, csak ezeket a wrapper¹⁶ osztályokat és függvényeket kell módosítani.

Ehhez szükséges a választott platformon (ez esetben OpenGL) implementálni a szükséges folyamatokat.

Így lehetséges sima objektumokat megjeleníteni, azokhoz shadereket¹⁷ készíteni és textúrázni.

5.3.1 Shader

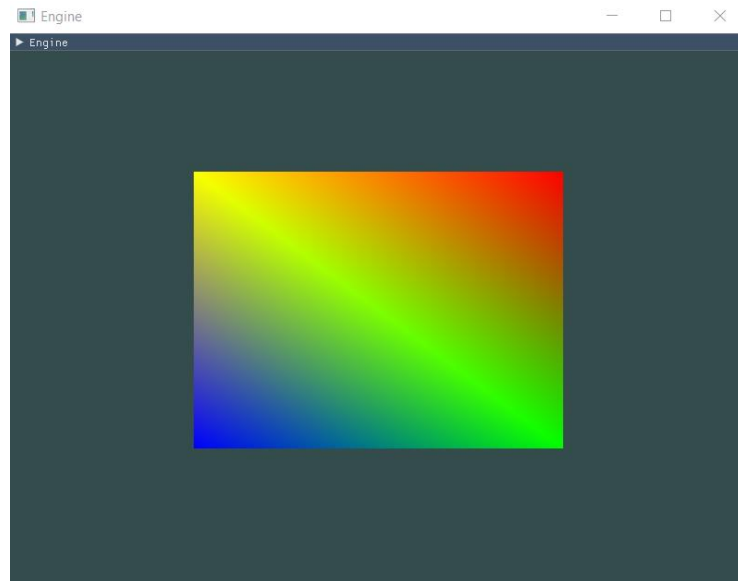
A shaderek felhasználása a motorból, így egészen egyszerűvé vált. A Shader osztály által könnyen végrehajtható minden művelet, anélkül, hogy a natív függvényeket kellene használnunk.

```
shader = Shader::Create("vs.vs", "fs.fs");  
shader->use();
```

A Shader osztály elrejt egy OpenGLShader osztályt, ami valójában hajtja végre a platformfüggő műveleteket. A Shader osztályra azért van, szükség, hogy ne lássa a fejlesztő a platform specifikus megoldásokat és ezt a részt könnyen le lehessen cserélni, anélkül, hogy módosítani kéne nagyobb mennyiségű kódot.

¹⁶ Becsomagolják a platform függő részeket egy egységes környezetbe

¹⁷ Olyan program, amit a GPU hajt végre



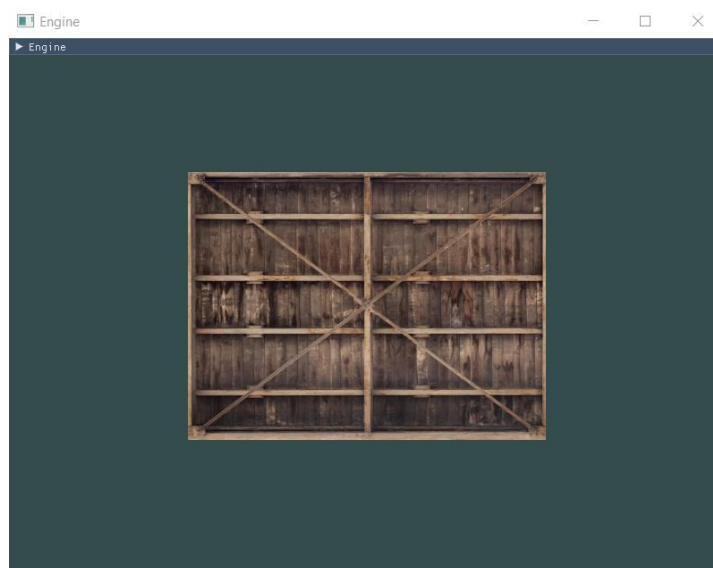
4. Shaderrel színezett téglalap

5.3.2 Textúra

Az alakzatok textúrázása is egyszerűbbé vált, a megfelelő osztálynak köszönhetően.

```
texture = Texture2D::Create(std::string("texture.png"));
```

Itt is létezik a Shader-hez hasonlóan és azonos okokból egy OpenGLTexture2D osztály, ami a valós munkát végzi. Ahhoz, hogy textúrázni tudjunk, szükség van a megfelelő shader-ekre.



5. Textúrázott téglalap

5.3.3 Camera

A kamera osztály platform független. Ennek lényege, hogy úgy tudja transzformálni az objektumokat, mintha egy paraméterezett kamerából látnánk azokat. Az emögötti mátrixműveletek szerencsére nem függenek a felhasznált technológiától, így csak a megfelelő shaderek kellene ahhoz, hogy a kamera osztály működhessen.

A kamerának számos paramétere van, ami szükséges a megfelelő mátrix előállításához. Ilyenek a kamera pozíciója, iránya, dőlése és világban a fentet jellemző irány. Ezen kívül szükség van egy olyan jellemzőre, ami a képbe való zoomoltásgot írja le.

Ezek alapján két mátrix megalkotása szükséges. Az egyik a projection mátrix, ez a zoomoltság és a képernyő mérete alapján kerül kiszámolásra. A másik a view mátrix, ami pedig a kamera pozíciójából és irányából kreálható. A már említett glm könyvtár rendelkezik olyan függvényekkel, amik az említett paraméterek alapján meghatározzák a kívánt mátrixokat.

A további műveleteket már a vertex shader végzi el.

```
gl_Position = projection * view * transform * vec4(aPos, 1.0);
```

A képletben szerepel még a transzformációs mátrix, ami a nagyítást, eltolást és forgatást írja le, a kamerától függetlenül.

5.4 Command

A Command osztály a végrehajtható parancsokat reprezentálja. Ez azt jelenti, hogy egy olyan interfész, aminek implementálásával lehet új parancsot létrehozni.

Ahhoz, hogy tudjuk mikor milyen parancsot kell végrehajtani szükség van egy bemenet kezelőre, ami a bemenethez köti a megfelelő parancsot. Ehhez az InputHandler osztály lett létrehozva, amiben vannak alapértelmezetten parancsok és bemenetek, de a fejlesztő adhat meg sajátot is.

A felhasználói bemenetek kezelése és az azokhoz tartozó parancsok végrehajtása minden ciklus iterációban egyszer megtörténik. Nincsen semmiféle pufferek és csővezeték (pipeline), így előfordulhatnak hibák.

5.5 Observer

Az Observer tervezési minta segítségével arra van lehetőség, hogy több akár egymástól független esemény közös hatására végrehajtsunk valamit. Például, ha új mezőre lépek és az a mező a cél, akkor vége a játéknak.

Observerek segítségével tudja értesíteni a fizikai modul, az adatszerkezet osztályait, a történt eseményekről.

5.5.1 Achievement

Azok az osztályok, amik megvalósítják a közös Achievement interfészt rendelkeznek egy közös `onNotify` metódussal. Ez a metódus mindig valami meghatározott esemény hatására kerül meghívásra. Ezek a meghatározott események a játék logikájától függenek.

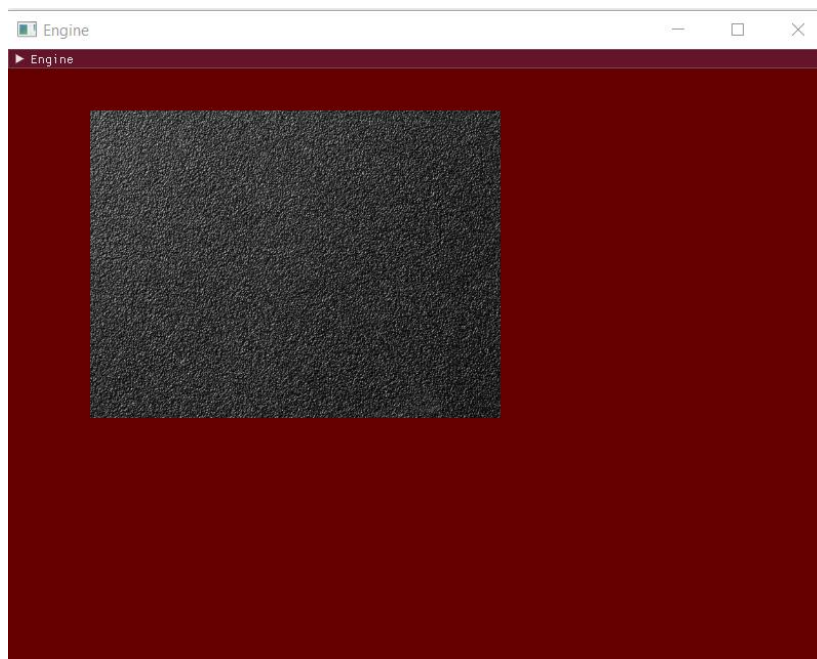
Amikor kiértékelünk egy observer-t (Achievement-et), akkor az végre hajtja a megfelelő utasításokat, például kiírja, hogy nyert a játékos, vagy csak eltárolja, hogy melyik mezőkön jártunk már.

Használata nagyban függ attól, hogy milyen játékhoz használjuk, ezért a motor csupán az interfészeket foglalja magába, azok megvalósításáról a játékfejlesztőnek kell gondoskodnia.

5.6 GameObject

Ezen osztály leszármazottai a játék objektumai. Ki lehet őket rajzolni és lehet velük műveleteket végezni. Ha szeretnénk egy saját objektumot a játékban, akkor ebből kell leszármazni és ezt kell testre szabni.

Ilyen például a motorba épített textúrázható téglalap, amit tetszőleges helyre lehet elhelyezni, tetszőleges méretben és tetszőleges mintával.



6. TexturedRectangle rajzolása, úttest mintával

5.7 Labirintus

A labirintusért felelős modul önállóan is hasznos lehet, de a motorba integrálással egyszerűen lehet labirintust készíteni a játékban.

A Labyrinth osztály összefoglalja a labirintus különböző részeit és egy olyan típus keletkezik így ami a játékmotorhoz lett kitalálva. Ez magában foglalja a labirintus adatszerkezeti részét, különböző megjelenítését és perzisztens¹⁸ tárolását.

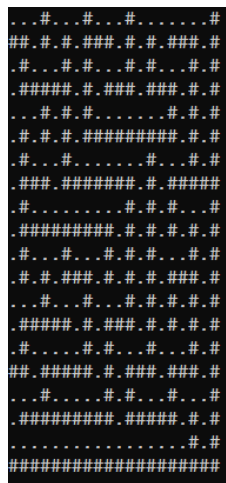
A Labyrinth osztály egy template osztály, aminek template paramétere a generátor típusa. Ennek van egy alapértelmezett értéke, ami az általam implementált algoritmust megvalósító osztály. Ennek a paraméternek a módosításával lehet új, a fejlesztő által készített algoritmusokkal generálni a labirintust.

Az osztálynak továbbá része egy rajzoló, ami platform függően képes kirajzolni a labirintust. Illetve ez képes a standard kimenetre és fájlba kiírni az adatszerkezetet. A labirintust leíró fájlok lb kiterjesztésűek, a könnyebb kezelhetőség érdekében.

¹⁸ Eltárolható, esetünkben fájlba írható és onnan beolvasható

A labirintust generáló algoritmusnak a megfelelő formátumban kell létrehoznia az adatszerkezetet. Az útvesztő adatszerkezeti szinten tartalmazza a méretét (egy négyzetrácsként kell elképzelni, aminek van magassága és szélessége). Ezen kívül tartalmaz cellákat, amik a labirintus egyes mezőit képviselik. Ezeknek a celláknak a generáláshoz szükséges adatokon kívül részük, hogy melyik irányban tartalmazznak falat. Ezek segítségével lehet megjeleníteni grafikusán a labirintust.

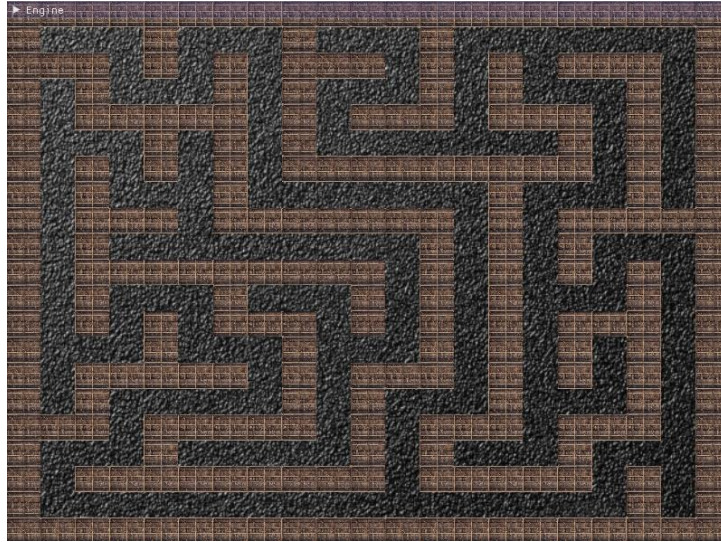
A labirintus generálása véletlenszerűen történik, de a jobb kezelhetőség érdekében lehet fix seed-del¹⁹ generálni, ami eredményeképpen mindig ugyanazt az útvesztőt kapjuk.



7. Labirintus kiírása a konzolra

Az ablakra történő kirajzolás egy platform függő művelet és a játékfejlesztőknek lehetőségük van saját megjelenítést kitalálni és lefejleszteni.

¹⁹ seed: a véletlenszerű generálás paramétere, egyezés esetén a kiment is azonos



8. Labirintus megjelenítése a grafikus felületen

5.8 Physics

A fizika modul az alapvető fizikáért felelős. Ez a játék mivoltából jelenleg csak 2D fizikát tartalmaz magában.

A legfontosabb eleme az ütközésérzékelés. Ez teszi lehetetlenné, hogy a játékos át tudjon menni a falakon. Vagy ezzel lehet megvalósítani, hogy tudjuk, mikor értük el célt.

6 Labirintus demo játék

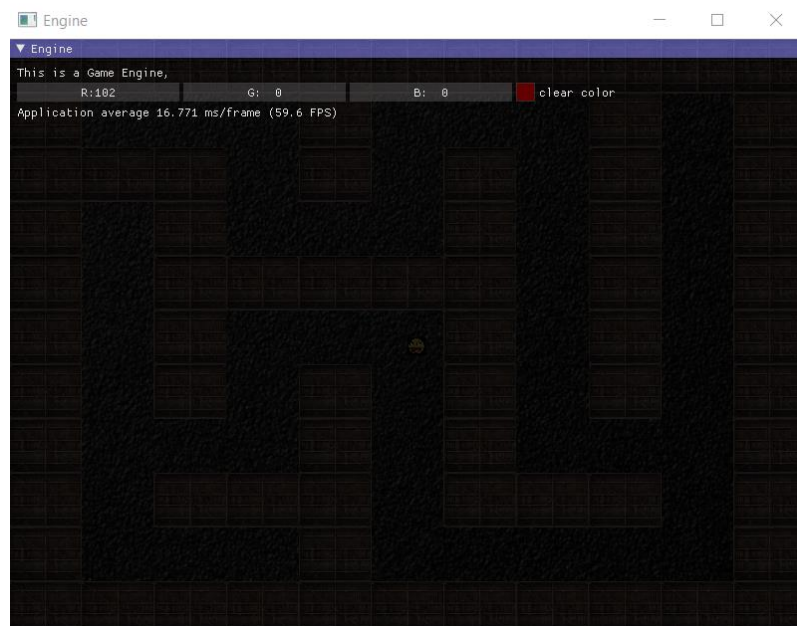
A játék egy egyszerű demonstrációja annak, hogy mit lehet csinálni a motorral. Létrehozása nagyon egyszerű. Csak létre kell hozni egy osztályt, aminek őse a motor GraphicsProgram osztálya.

```
class LabyrinthGame final : public Engine::GraphicsProgram { ... };
```

Ezután meg kell valósítani a motor CreateGraphicsProgram függvényét, ami visszaadja milyen típusú a játék.

```
Engine::GraphicsProgram* CreateGraphicsProgram()
{
    return new LabyrinthGame();
}
```

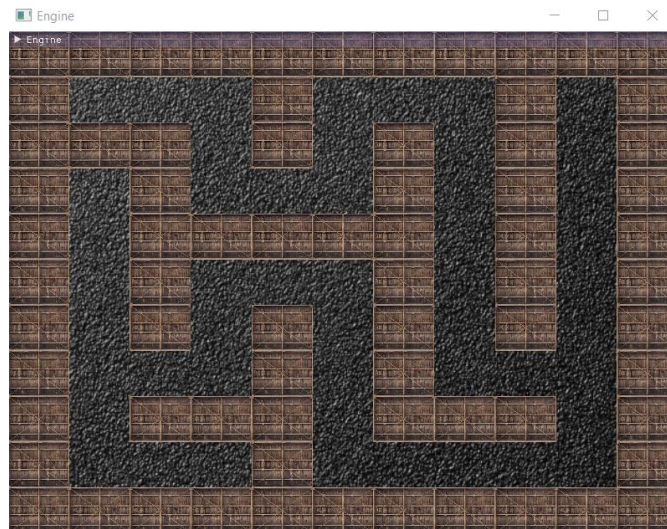
Ekkor kerül példányosításra maga a játékért felelős objektum. Ennek konstruktorában hozzá tudunk adni layer-eket és meg tudjuk adni melyik layer-ek között vizsgáljon ütközést. Van lehetőség az ImGuiLayer testreszabására.



9. Testreszabott ImGuiLayer

A demo játéknak két layer-e van. Az egyik a labirintusé, a másik a játékosé. A két réteg között figyelni kell az ütközésetektálásra, hogy ne tudjon a játékos átmenni a falakon.

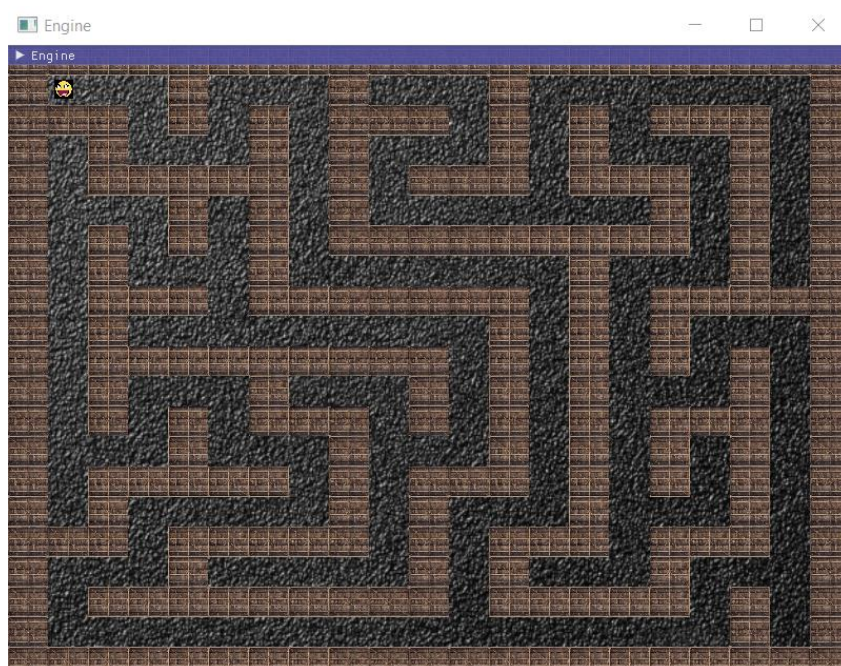
A labirintus rétegén egy tetszőleges méretű útvesztő található. A falak reagálnak a játékoskal való ütközésre.



10. 5x5-ös útvesztő a labirintus layer-en

A játékos által irányított objektum egy másik rétegen van. Ez a réteg a gameLayer. A játékoshoz létre lett hozva egy objektum, ami a motor Player osztályának leszármazottja. Ez az elem a nyilakkal irányítható és ütközik a labirintus falaival.

A játék igyekszik lefedni a motor összes tudását, de ezt a lehető legegyszerűbben teszi. Emiatt nem rendelkezik komolyabb játék logikával, csupán egy váz, ami más játékok alapjaként szolgálhat



11. Demo játék

7 Továbbfejlesztési lehetőségek

A motor egy alapvető játékhoz szükséges funkciókkal rendelkezik. De mivel nagy hangsúly volt az objektum orientált felépítésen és a modulok egyszerű lecserélhetőségén volt, bőven van még olyan funkció, ami hozzáadható. Számos funkciónak már elő van készítve minden, csupán egy kevés munkát igényel a beszerelésük. Azonban vannak olyanok is, amik esetlegesen nagyobb módosítást igényelhetnek. Az alábbiakban szeretnék néhány részben vagy egészben hiányzó funkciókat szeretnék megemlíteni.

- **Inputkezelés**

Jelenleg a motor rendelkezik, egy jól működő inputkezelő rendszerrel, ami mögött a platform egyszerűen módosítható. A hiányossága az, hogy semmiféle előfeldolgozás nem történik a bemenet észlelése és végrehajtása között. Azzal lehetne jobbá tenni, ha a bemenet egy csővezetéken átmenve feldolgozásra kerülne. Ezzel hatékonyabb és biztonságosabb lenne a folyamat.

- **3D**

A motor jelenleg 2D beépített objektumokat tartalmaz, de semmi nem akadályozza a játékfejlesztőt, hogy 3D játékot készítsen. Az egyetlen akadály, hogy a labirintus nem rendelkezik olyan rajzoló algoritmussal, ami térben ábrázolná azt és ennek elkészítéséhez a motor kódját kellene kiegészíteni.

- **Labirintus megoldásának megjelenítése**

Jelenleg a labirintus nem jeleníti meg a grafikus felületen a megoldási útvonalát. Az útkereső algoritmus implementálva van és használható is a konzolos felületén az útvesztőnek. Ahhoz, hogy egy játékban is használható lehessen nincs szükség nagy módosításokra, de azokat a motorban kell végrehajtani.

- **Fizika**

A fizika egy egyszerű 2D játékhoz lett készítve. Ez pedig az ütközés detektálásban kimerül. Ahhoz, hogy nagyobb tudású fizikai motorunk legyen vagy a most létezőt kell tovább fejleszteni, vagy egy külső fizikai motor is felhasználható. Mindkét opció nagyobb módosításokkal jár és lehet, hogy nagyobb munkára van hozzá szükség. Mindazonáltal a már létező kódok nagyobb mértékű módosítására nem lenne szükség, csupán azok kiegészítésére.

- **Párhuzamosítás**

A motor jelenleg egy szálon működik. Már egész egyszerű esetekben előjöhethet teljesítmény probléma, ami párhuzamosítással megoldható lenne. Ennek kivitelezése helyenként nem lenne bonyolult, máshol viszont nagy nehézségeket tudna okozni.

- **Renderer**

A mostani renderer képes minden szükséges funkciót ellátni. Használata esetenként macerás lehet, ezt pedig még több metódussal és típussal lehetne segíteni. Ezek elkészítése nem jelent nagy nehézséget. Ezen kívül még több előre elkészített grafikus objektum is segítené a fejlesztést.

A másik hiányossága, hogy nem tud külső forrásból modelleket beolvasni. Itt maga a beolvasás jelent nagyobb munkát, a kezelése utána már egyszerűbben megoldható.

- **Animációk**

Jelenleg a motor semmilyen formában nem támogat animációkat. Ezek bevezetése, a motor kiegészítésével járna.

- **Hangok**

A hangok kezelése sem része a motornak. Bevezetésük egy külső könyvtár segítségével könnyen elintézhető. Ezután már csak az egyszerű használathoz szükséges objektumokat kell létrehozni.

- **Grafikus felület**

A jelenlegi eszközökkel egészen egyszerűen létre lehetne hozni, egy olyan grafikus felhasználói felületet, ami a motor használatát egyszerűsítene.

- **Fények**

A motor nem képes komplex fényeket kezelni. Ezen kívül nem tesz különbséget az objektumok anyagában. Ezek mind megvalósíthatóak a játék szintjén, de egyszerűen felvihetők a motorba.

- **Scriptek**

Nincsen kitalált script rendszere a motornak. Ennek bevezetése egy nagyobb tervezési feladatot jelent, valamint a motor nagymértékű kiegészítése szükséges.

- **Egyéb**

Az említetteken kívül még számos olyan funkció létezik, amikkel bővíthető a motor tudása. Ezek nagyja könnyedén vagy legalábbis a jelenlegi kódállomány módosítása nélkül kivitelezhető lenne. Ennek oka az alapos tervezés és az Open-Closed törvényre fektetett nagy hangsúly.

8 Összegzés

Az elkészített motor egy jó alap egy komplexebb eszköz készítéséhez, azonban tudása nagyban elmarad napjaink legnagyobb motorjaitól. Fejlesztése során többször bizonyosságot nyert, hogy megérte akkora energiát fektetni az alapos tervezésbe és objektum orientált tervezési elvek és minták használatába.

Megtapasztaltam, hogy egy program soha nem készül el és mindig van rajta mit továbbfejleszteni. Sokszor egészen apró részletekbe lehet rengeteg munkát beleölni. Ezen kívül sokszor a láthatatlan részek a legfontosabbak.

Irodalomjegyzék

- [1] *OOP elvek*: https://en.wikipedia.org/wiki/Object-oriented_programming#SOLID_and_GRASP_guidelines (2021. okt.)
- [2] *Clean Code elvek*: <https://dzone.com/articles/clean-code-summary-and-key-points/> (2021. okt.)
- [3] *Game Programming Patterns*: <https://gameprogrammingpatterns.com/contents.html/> (2021. okt.)
- [4] *Unity*: <https://unity.com/> (2021. okt.)
- [5] *Unreal Engine*: <https://www.unrealengine.com/en-US/> (2021. okt.)
- [6] *SOLID Principles*: <https://en.wikipedia.org/wiki/SOLID/> (2021. okt.)
- [7] *OOP Principles*: <https://training.by/#!/News/275?lang=en> (2021. okt.)
- [8] *Design Patterns*: <https://refactoring.guru/design-patterns/> (2021. okt.)
- [9] Robert C. Martin: *Clean Code*, ISBN 0-13-235088-2, 2008, <https://enos.itcollege.ee/~jpoial/oop/naited/Clean%20Code.pdf> (2021. okt)
- [10] *Test driven development – TDD*: https://en.wikipedia.org/wiki/Test-driven_development/ (2021. okt.)
- [11] *Code Smell*: https://en.wikipedia.org/wiki/Code_smell/ (2021. okt.)
- [12] *Game Loop Pattern*: <https://gameprogrammingpatterns.com/game-loop.html/> (2021. nov.)
- [13] *Update Method Pattern*: <https://gameprogrammingpatterns.com/update-method.html/> (2021. nov.)
- [14] *Command Pattern*: <https://gameprogrammingpatterns.com/command.html/> (2021. nov.)
- [15] *Event Queue Pattern*: <https://gameprogrammingpatterns.com/event-queue.html/> (2021. nov.)
- [16] *Observer Pattern*: <https://gameprogrammingpatterns.com/observer.html/> (2021. nov.)
- [17] *Singleton Pattern*: <https://gameprogrammingpatterns.com/singleton.html/> (2021. nov.)
- [18] *State Pattern*: <https://gameprogrammingpatterns.com/state.html/> (2021. nov.)

- [19] *Modell View Controller architecture*:
<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller/>
(2021. nov.)
- [20] *Premake*: <https://premake.github.io/> (2021. nov.)
- [21] *Git*: <https://git-scm.com/> (2021. nov.)
- [22] *GitHub*: <https://hu.wikipedia.org/wiki/GitHub/> (2021. nov.)
- [23] *OpenGL*: <https://www.opengl.org/> (2021. nov.)
- [24] *glad*:
https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library#glad_.28Multi-Language_GL.2FGLES.2FEGL.2FGLX.2FWGL_Loader-Generator.29/ (2021. nov)
- [25] *GLFW*: <https://www.glfw.org/> (2021. nov.)
- [26] *spdlog*: <https://github.com/gabime/spdlog/> (2021. nov.)
- [27] *ImGui*: <https://github.com/ocornut/imgui/> (2021. nov.)
- [28] *glm*: <https://github.com/g-truc/glm/> (2021. nov)
- [29] *stb_image*: <https://github.com/nothings/stb/> (2021. nov)
- [30] *bullet physics*: <https://github.com/bulletphysics/bullet3/> (2021. nov)
- [31] *OpenGL guide*: <https://learnopengl.com/> (2021. nov)
- [32] *GitHub repository*: <https://github.com/rittakos/LabyrinthGameEngine/> (2021. nov)