EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatika Kar

Programozási Nyelvek és Fordítóprogramok Tanszék

Játékprogramozás C-ben

Témavezetők:

Dr. Porkoláb Zoltán
Egyetemi docens, PhD.

Dr. Horváth Gábor
Software Development Engineer 2,
Programtervező Informatikus MSc

Készítette:

Vápár András
programtervező informatikus szak

Budapest, 2021. 05. 28.

1 Bevezetés

Szakdolgozatom keretében C programozási nyelvben készítettem el egy játékprogramozásra alkalmas keretrendszert, illetve annak funkcióit bemutató demó alkalmazást. Szinte mindent magam implementáltam, a külső könyvtárak, amiket használok:

- SDL2 (Simple DirectMedia Layer): Ablakok kezelését teszi lehetővé, illetve rájuk való rajzolást. Billentyűzetről inputkezelés.
- doctest: Egy c++ egységteszt keretrendszer.

Az erőforrásaim túlnyomó részét az alábbi modulok fejlesztésébe fektettem, ezért dolgozatomban a nagyobb hangsúly az említett keretrendszeren lesz.

Szakdolgozatom így két fő részre bontható: a megvalósított modulokra, amikre a játék épül, illetve az alkalmazás.

Munkám során a következő nagyobb modulokat kellett elkészítenem:

- Entity Component System: Tetszőleges adattal rendelkező entitások hatékony tárolása, lekérdezése.
- Collision: Egy alakzat (pont, vektor, mozgó vagy statikus téglalap) ütközésének detektálása egy téglalappal.
- Pathfinding: Út találása két pont között.

Ezeket a főbb modulokat emeltem ki az elején, de természetesen a fejlesztői dokumentációban róluk, illetve az itt nem említett modulokról többet is lehet olvasni. Egyes modulok között függőség áll fenn, ezekről részletesebben a fejlesztői dokumentációban lehet olvasni.

Ahogy említettem, készült továbbá egy egyszerű demó alkalmazás, ezen keresztül fogom megmutatni a keretrendszerem funkcióit.

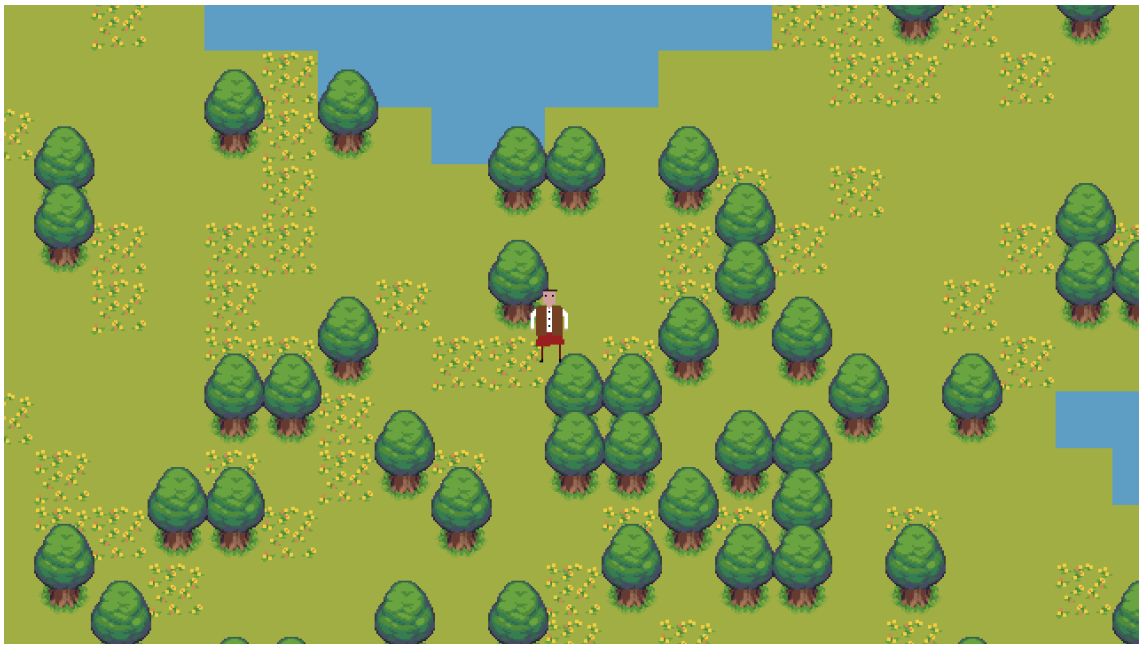
Megemlíteném, hogy sok triviálisnak tűnő feladatom volt, ami nem is olyan egyértelmű. Íme két példa: az első, a monitor koordináta rendszere, és a világ logikájának koordinátája között valamilyen kapcsolat megteremtése, konvertálása. A másik pedig az entitások textúráinak helyes sorrendben történő kirajzolása.

A tanulás volt a szakdolgozat fő célja, hogy egy alacsonyabb programozási nyelvben milyen lehet játékot, vagy üzleti alkalmazást fejleszteni, és tudtam, hogy ebből sokat fogok tudni fejlődni. Valóban sok új ismeretre szert tettem.

A (Robert, Game Programming Patterns, 2014) egy kifejezetten érdekes olvasmány volt, amiből sokat merítettem.

2 Felhasználói dokumentáció

Az elkészített játék lényegében egy demó alkalmazás, amin keresztül be tudom mutatni a könyvtáramban elkészített funkcionalitásokat.
Ebben a programban egy piros-gúnyás embert irányítunk, egy füves, virágos, fákkal teli szigeten.
A szigetet nem lehet elhagyni, de egyébként szabadon mozoghatunk rajt. Fákon nem tudunk keresztülsétálni.



1. ábra: A demó alkalmazás

2.1.1 Célközönség

A szoftvert azoknak ajánlom, akik C programozási nyelvben szeretnének kétdimenziós játékot fejleszteni.
A 4 Fejlesztői dokumentáció részben további információ szerepel a könyvtárak működéséről, s használatukról.

2.2 Funkciók

A főhőst az AWSD billentyűkkel lehet a szokásos módon irányítani: A-val balra, W-fel felfelé, S-sel lefelé, D-vel jobbra. A játékos a kamera közepén indul, mozgás esetén követi őt. A baloldali Ctrl-t nyomva tartva megnő a főhős sebessége.
A világban vannak fák, illetve a zöld, füves szigetet víz veszi körül, ezeken nem tud a játékos keresztül futni, hanem megakad rajtuk.

Ezeken túl, ha a bal egérgombot nyomva tartjuk a képernyőn a játékos karakter egy bizonyos környezetén belül, akkor megjelenik egy piros útvonal, ami a hős hitboxának bal felső pontjától utat keres oda, ahol éppen az egeret lenyomtuk. A hitbox az egy láthatatlan téglalap, ami az entitások azon részét definiálja, amik ütközni tudnak más entitások hitboxával.

Ez az útvonal természetesen kikerüli az akadályokat.

2. ábra: Útkeresés.

A játékból az escape billentyű lenyomásával tudunk kilépni.

2.3 Telepítés

Töltsd le a [https://github.com/vapandris/Archipelago] oldalról az Archipelago.zip állományt. Csomagold ki, és futtathatod is az Archipelago.exe file-t.

A könyvtár telepítéséről a 3.1 Fejlesztői környezet megteremtése című fejezetben lehet olvasni.

A letöltött .zip állomány 61.2 KB helyet foglal.

A futtatható állomány 64 bites Windows operációs rendszerre lett fejlesztve.

Futás közben nagyjából 10 MB memóriát használ.

3 Fejlesztői dokumentáció

Ebben a fejezetben a demó alkalmazás, illetve a keretrendszer fejlesztőinek fontos információk találhatóak meg.
A fejlesztői dokumentáció szerkezete a következőkben leírtaknak megfelelően épül fel. Először, hogy hogyan lehet a szakdolgozatom kódját megszerezni, illetve a használt fordító rendszert, és a használt unit teszt rendszert honnan lehet elérni, s továbbá hogyan kell beüzemelni. Az egymásra épülő modulokról fogok beszélni. Hogyan kell használni őket, illetve hogyan működnek. Írok még a demó alkalmazás működéséről is. Ez két egymásra épülő modul lesz, a nézet, illetve a logika. Végül a munkám teszteléséről is írok.
Alapvetően Windows 10 operációs rendszerben fejlesztettem de apró módosításokkal Arch Linux-on is sikerült beütemezni.

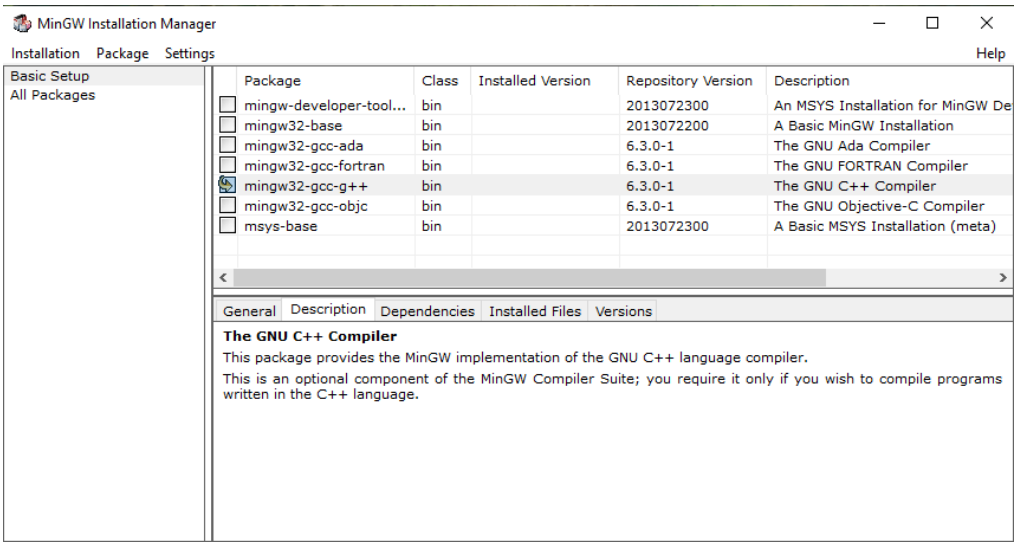
3.1 Fejlesztői környezet megteremtése

3.1.1 Git telepítése, forráskód másolása

1. A szakdolgozatom forráskódja a [https://github.com/vapandris/Archipelago] GitHub oldalról érhető el.
2. A Git Bash-nek a 2.30.1-es verzióját használtam, a telepítőt erről a linkről lehet megszerezni: [https://git-scm.com/download/win].
3. Kövesd a telepítő utasításait.
4. Indítsd el a Git Bash-t.
5. A cd {mappa neve} utasítással lépj el oda, ahova a forráskódot másolni szeretnéd, majd add ki a *git clone https://github.com/vapandris/Archipelago* parancsot.

3.1.2 MinGW telepítése

1. Tölts le a [https://sourceforge.net/projects/mingw/files/latest/download] linkről a MinGW telepítőjét.
2. Indítsd el, majd hagyd, hogy végig fusson.
3. Utána a lenti ábrán látott ablakon pipálja be a mingw32-gcc-g++ opciót
4. Az Installation fülben pedig válassza ki a telepítés opciót.



3. ábra: MinGW telepítője.

3.1.3 SDL2 telepítése

1. Töltsd le az SDL2 könyvtárat a következő linkről: [http://libsdl.org/release/SDL2-devel-2.0.14-mingw.tar.gz].
2. Csomagold ki, majd nyisd meg benne az i686-w64-mingw32 mappát, ha 32 bites rendszerben dolgozol, ha 64 bitesben, akkor pedig az x86_64-w64-mingw32 nyisd meg. Mind a kettőben ugyan olyan nevű mappákat fogsz találni.
3. Az include mappa tartalmát másold át a MinGW mappában található include mappába.
4. Hasonlóan a lib mappában talált .a, .la állományokat másold át a MinGW lib mappájába
5. Ugyan így a bin mappában található SDL2.dll állományt másold át a MinGW bin mappájába. Abban a mappában fogod találni a gcc.exe-t is.
6. Ezek után az SDL_image könyvtárat is külön kell telepíteni teljesen azonos módon, a megfelelő mappa include, lib, bin tartalmát át kell másolni a MinGW include, lib, bin mappájába. Ezt a könyvtárat a következő linkről lehet letölteni: [https://www.libsdl.org/projects/SDL_image/release/SDL2_image-devel-2.0.5-mingw.tar.gz].

3.1.4 Meson telepítése

A meson fordító keretrendszer telepítéséről részletesen az alábbi linken található egy útmutató, ez minden szükséges információt tartalmaz: [https://mesonbuild.com/Getting-meson.html].

3.1.5 doctest telepítése

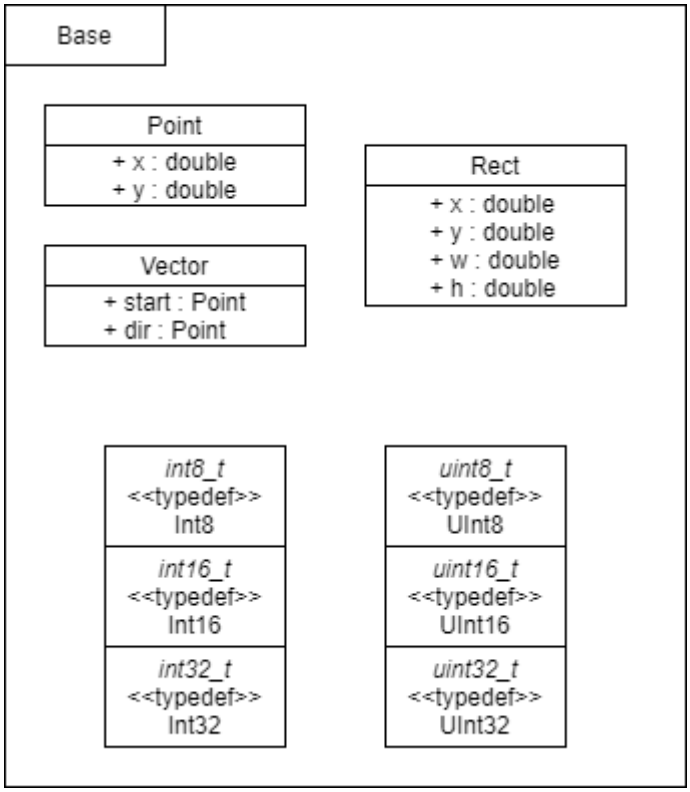
Ez egy header-only könyvtár, így nagyon egyszerű az üzembehelyezése.
A következő oldalról [https://github.com/onqtam/doctest/blob/master/doctest/doctest.h] töltsük le, vagy másoljuk ki a file tartalmát, majd a c++ állományokat tartalmazó mappában hozzunk létre egy doctest nevű file-t a letöltött tartalommal.

3.2 Alkalmazott modulok

3.2.1 Base modul

Ez a modul egyszerű adatstruktúrákat tartalmaz, illetve typedef-eket tetszetősebb kód érdekében. Csak header fileokat tartalmaz.
Az adatstruktúrák egyszerű, kétdimenziós geometriai alakok:

- Pont
- Téglalap
- Vektor



4. ábra: UML a Base modulról

3.2.2 Entity Component System modul

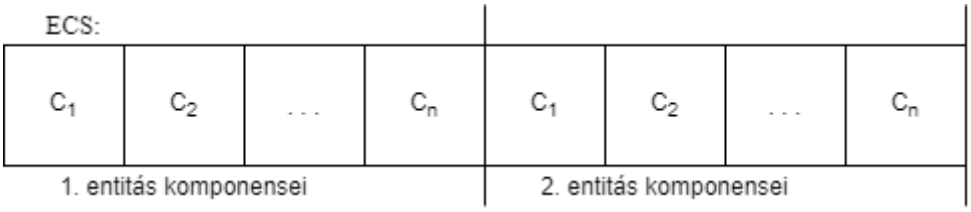
Az ECS (Entity Component System) egy programozási minta, amit többnyire játékprogramoknál alkalmaznak, viszont egyéb szoftverekben is használható, ahol az ECS nyújtotta előnyöket ki lehet használni. (Austin, 2019)

Az ECS egy Data-Oriented dizájn, ami kompozíciót alkalmaz az objektum-orientált öröklődés helyett. Ez nagy rugalmasságot enged meg.

De hogy mit értünk kompozíció alatt? A kódban definiálhatunk több, különböző komponenst, és ezekből egyet, vagy többet hozzá rendelhetünk az entitásokhoz. Fontos megjegyezni, hogy a komponensek csupán adatok, viselkedést majd az ECS-nek a System része fogja szolgáltatni.

Egy példa: Tegyük fel, hogy a játékunkban van a játékos, virágok és fák. Mind a hármat ki akarjuk rajzolni, azonban csak a fával akarunk tudni a játékossal összeütközni. Ekkor a fának, illetve a virágoknak is lesz egy grafikus-komponense, ami a kirajzolásukhoz szükséges adatot tárolja, viszont hitbox-komponenssel csak a fa, illetve a játékos fog rendelkezni.

Azt látjuk, hogy ez a megközelítés rugalmas, hiszen egy entitást akármilyen komponensekből össze tudunk rakni, viszont, ez hogyan garantálja a gyorsaságot? Ez a jó cache kihasználtságából ered. Az entitásokhoz tartozó komponensek adatait az ECS memóriában egymás után eltárolja, ezt a következő ábra szemlélteti.



5. ábra: Ci i∈[1,...,n] n ∈N, Ci egy komponens tetszőleges adattal

Amikor az operációs rendszertől memóriát kérünk, akkor az a kért mennyiségnél többet ad vissza, illetve többet is másol át a jelentősen gyorsabb elérésű cache-be. Így ha az adatok a memóriában egymás mellett helyezkednek el, akkor a cache-be is jobban tudjuk őket bemásolni.

A fenti ábrán látszik, hogy egy entitás n darab komponensből állhat össze. Ez implementációtól függ, de jelen esetben minden entitásnak lefoglalunk az összes komponensnek elegendő helyet, legfeljebb nem fog mindegyikből összeállni az adott entitásunk.

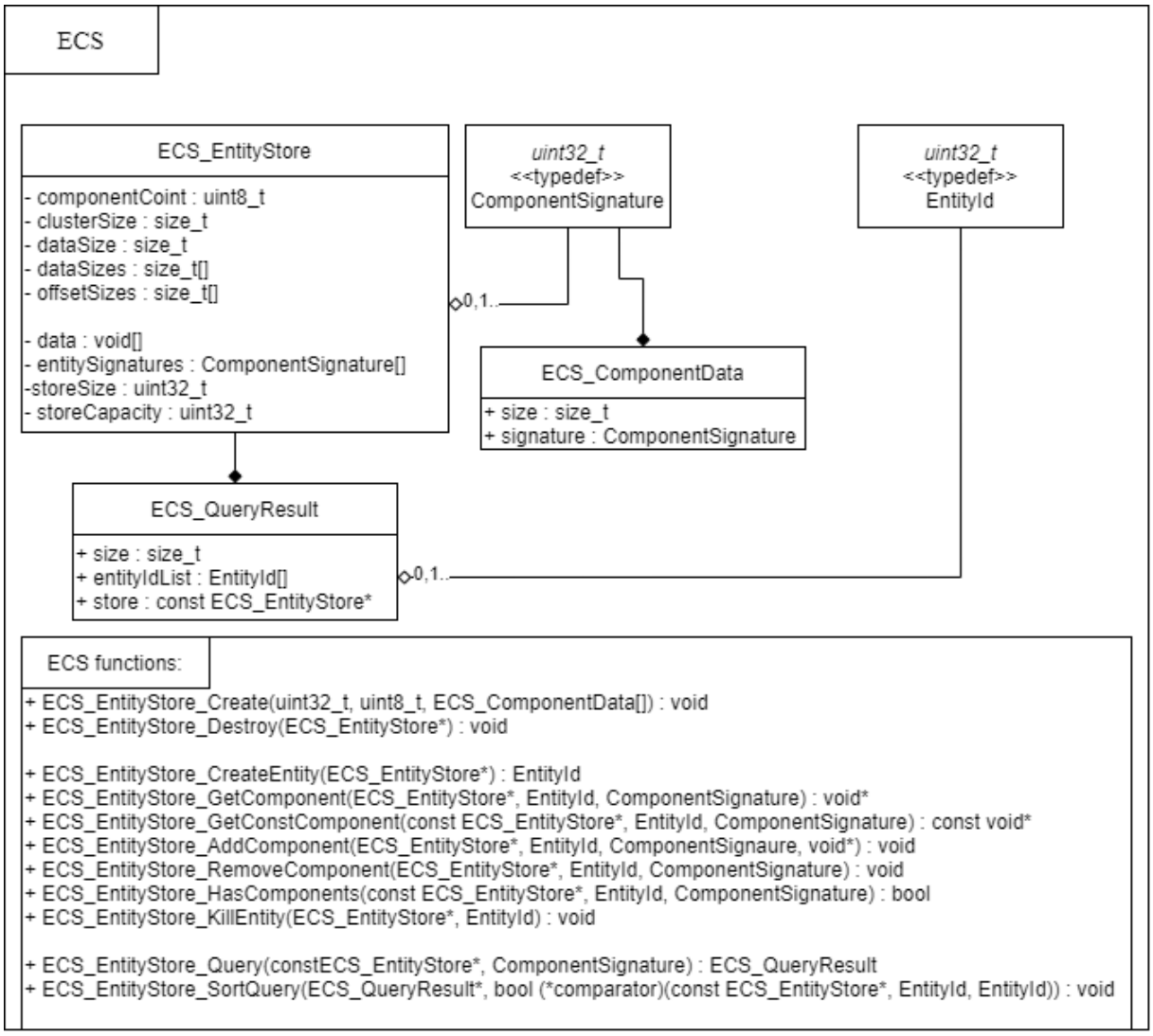
Egy entitást teljes mértékben csak a komponensei határozzák meg, és így egy entitásra tekinthetünk úgy, mint egy egyszerű index, vagy azonosító, amin keresztül el tudjuk érni az adott entitás komponenseit.
Ezen túl érdemes eltárolni egy szignatúra-tömböt, ebbe speciális számokat tárolunk, amikkel meg lehet határozni, hogy egy adott entitás mely komponenseket használja. Ha ezeket a speciális szignatúrákat binárisan reprezentáljuk, akkor a 0 bit azt jelenti, hogy az adott komponenst nem használja, ha 1-es a bit, akkor pedig használja.

6. ábra: ECS szignatúrák

Az ECS-be olyan elemek fognak kerülni, amiken nagyon sűrűn végig akarunk iterálni. Így, mivel az entitások komponensei a cache-be kerülnek, sokkal gyorsabb lesz az elérésük, mint ha memóriában szétszórva lennének.
A megoldásom sokat merített Dylan Falconer implementációjából. (Dylan, 2020)

3.2.2.1 Interfész

Ebben a részben az általam megvalósított Entity Component System interfészét mutatom be.



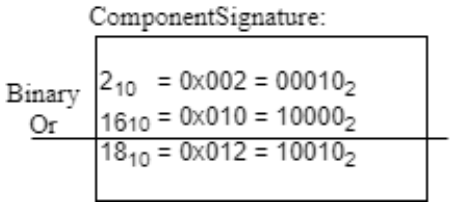
7. ábra: Az ECS-modul szerkezete

Ahogy látható, az EntityId, amin keresztül az entitásokat elérhetjük csupán egy azonosító, pontosabban egy index. A ComponentSignature is egy azonosító lesz, amivel egy, vagy több komponensre tudunk majd hivatkozni.

3.2.2.1.1 Adatstruktúrák

3.2.2.1.1.1 ComponentSignature

Minden komponenshez kell, hogy tartozzon egy egyedi pozitív egész szám. Ezzel tudjuk az adott komponenst azonosítani. Ezen túl ennek a számnak kettő hatványnak kell lennie, így a bináris alakjában csak egy darab egyes lesz, a többi bináris helyiérték nulla lesz. Ezáltal az adott komponenst valójában csak egy bit azonosít. Kettő különböző komponens szignatúrájából a bitenkénti vagy művelet képzett érték mind a két komponensre hivatkozik.



8. ábra: Szignatúra szemléltetése, összevagyolása

3.2.2.1.1.2 EntityId

Ez egy pozitív egész szám, amin keresztül el tudunk érni egy entitást az Entity Component Systemből. Fontos megjegyezni, hogy ha törlünk az ECS-ből egy entitást, akkor ezek az azonosítók már más elemre mutathatnak, ezt a jelenséget invalidációnak hívják. Nem érdemes ezeket az azonosítókat elmenteni hosszú távra, különösen, ha törlünk az ECS-ből.

3.2.2.1.1.3 ECS_EntityStore

Rajta keresztül érhetjük el az Entity Component Systemet. Többet róla az 4.2.2.2 Implementáció című alfejezetben.

3.2.2.1.1.4 ECS_QueryResult

Amikor az ECS-en szűrni szeretnék bizonyos komponenssel rendelkező entitásokra, akkor egy ECS_QueryResult típusú változót kapunk vissza.
size: Elárulja, hogy hány elemre teljesült a szűrés.
entityIdList: Ebből tudjuk, hogy mely EntityId-jű entitásokra teljesül a szűrt feltétel.
store: Arra az ECS_EntityStore-ra egy mutató, amire a szűrést végrehajtottuk.

3.2.2.1.1.5 ECS_ComponentData

Erre az ECS_EntityStore_Create függvényben lesz szükség. Ezen keresztül meg kell mondani, hogy az adott komponens, amit regisztrálni akarunk az ECS-be, annak mekkora a mérete (ezt a sizeof operátorral célszerű), illetve meg kell adni neki egy ComponentSignature-t, amin keresztül hivatkozhatunk rá.

3.2.2.1.1.6 ECS_EntityStore_Comparator

Egy függvény-mutató, amit az ECS_QueryResult rendezésénél fogunk használni. Ezzel határozhatjuk meg, hogy két, azonos ECS_EntityStore-beli EntityId-vel rendelkező entitást hogyan, esetleg mely komponensek adatai szerint hasonlítsunk össze.

3.2.2.1.2 Függvények

Definíció	**Leírás**
ECS_EntityStore* ECS_EntityStore_Create (uint32_t, uint8_t, ECS_ComponentData...)	Visszaad egy megadott kapacitású ECS_EntityStore-t. A paraméterül megadott komponenseket lehet az Entitásokhoz kötni, illetve a megadott ComponentSignature-rel lehet rá majd hivatkozni.
void ECS_EntityStore_Destroy (ECS_EntityStore*)	Felszabadítja az adott ECS_EntityStore által lefoglalt erőforrásokat.
EntityId ECS_EntityStore_CreateEntity (ECS_EntityStore*)	Létrehoz egy új entitást az ECS-ben, és visszaad egy Id.-t, amivel elérhetjük azt. Ha a maximum kapacitáson túl akarunk új entitást létrehozni, akkor a kapott ECS_EntityStore kapacitása nő.
void* ECS_EntityStore_GetComponent (ECS_EntityStore*, EntityId, ComponentSignature)	Visszaadja a megadott entitáshoz tartozó kért komponenst.
const void* ECS_EntityStore_GetConstComponent (const ECS_EntityStore*, EntityId, ComponentSignature)	Visszaadja a megadott entitáshoz tartozó kért komponenst konstansként. Erre azért van szükség külön, mert így a paraméterül kapott ECS_EntityStore lehet konstans.

void ECS_EntityStore_AddComponent (ECS_EntityStore*, EntityId, ComponentSignature, void*)	A kért entitáshoz hozzárendel egy komponenst megadott adattal, amit lemásolunk.
void ECS_EntityStore_RemoveComponent (ECS_EntityStore, EntityId, ComponentSignature)	A kért entitásnak törli egy komponensét.
bool ECS_EntityStore_HasComponents (const ECS_EntityStore*, EntityId, ComponentSignature)	Megmondja, hogy egy entitásnak van-e megadott szignatúrájú komponense/komponensei, utóbbi esetben bináris vagy-gyal lehet őket összekötni.
void ECS_EntityStore_KillEntity (ECS_EntityStore*, EntityId)	Törli az ECS-ből a kért entitást. Ennek a műveletnek a hatásául invalidálódhatnak korábban elmentett EntityId-k.
ECS_QueryResult* ECS_EntityStore_Query (const ECS_EntityStore*, ComponentSignature)	A kért szignatúrájú entitások Id.-it tartalmazó ECS_QueryResult-ot adja vissza. A ECS_EntityStore_HasComponents függvénnyel hasonlóan lehet a paraméterben bináris vagy-ot használni.
void ECS_EntityStore_SortQuery (ECS_QueryResult*, ECS_EntityStore_Comparator)	Az kért ECS_EntityStore_Query-t rendezi a megadott logika szerint.
void ECS_QueryResult_Destroy (ECS_QueryResult*)	Felszabadítja a megadott ECS_EntityStore_Query-t

3.2.2.2 Implementáció

Ebben a részben ismertetem, én hogyan valósítottam meg az Entity Component Systemet, azaz, hogy az interfészben ismertetett függvények hogyan működnek. Szó lesz az ECS_EntityStore belső működéséről, hogyan működik, illetve a használt statikus (csak a forrásfile-ban látható) függvényeket is ismertetni fogom.

3.2.2.2.1 Void mutató

Mindenekelőtt a void*-t ismertetem, mert ez nem feltétlenül ismert mindenki számára.
A többi típusos mutatóval ellentétben, void mutatók nem árulják el a típusukat. Csupán annyit tudunk, hogy hova mutat a mutató, illetve, hogy mennyi memóriát foglaltunk le. C-ben a mutatók adatsorozatra is mutathatnak, így, ha nem tudjuk, hogy mennyi, illetve mekkora méretű elemekre mutat egy void*, akkor azt nem tudjuk használni.
Továbbá, ha érdekel minket egy void* mutatott értéke, akkor azt először a kívánt típusú mutatóvá kell konvertálni, és csak azt követően tudjuk dereferálni.
Ha void*-ot tömbként használjuk, akkor nem tudjuk a szögleteszárójellel lekérdezni a kívánt indexű elemet, hiszen nem tudjuk egy elem méretét, hogy tudjuk, hogy a memóriában mennyit kell arrébb lépni a kezdőponttól, hanem (vitathatóan) a legkényelmesebb megoldás, ha 1 byte-os uint8_t*-vé kasztoljuk, majd a + operátorral kell a megfelelő elem elejére navigálni. Egy alternatív megoldás, ha a mutatót típusra kasztoljuk, akkor már alkalmazható a szögleteszárójel operátor.

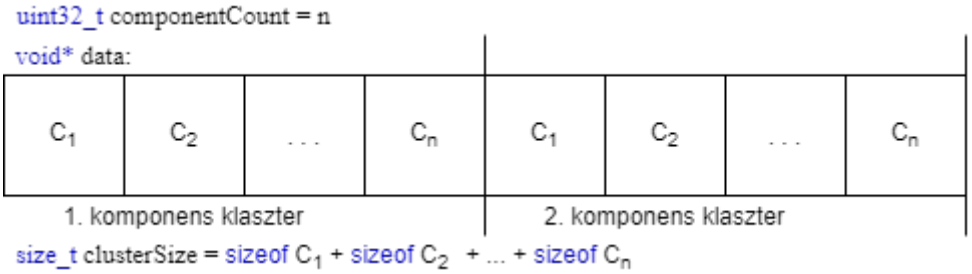
3.2.2.2.2 ECS_EntityStore

Ez az adatstruktúra felelős az Entity Component Systemünk belső állapotának a tárolására. Most elmagyarázom adattagjainak szerepét, és hogy előre tekintve, a függvények hogyan módosítják őket.

3.2.2.2.2.1 Adattagok

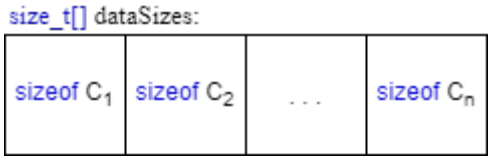
uint8_t componentCount: Eltárolja, hogy mennyi különböző komponenstípust regisztráltak.
Az implementációmban, mivel a komponens-szignatúra egy 32-bites szám. A szignatúra definíciója szerint egy szignatúrának egyedinek kell lennie az adott Entity Component Systemben, illetve pontosan egy darab bitnek kell egyesnek lennie, így a maximális komponensek száma: 32.

size_t clusterSize: Mennyi memóriát foglal el egy komponens-klaszter. Ez azonos az összes komponens méretének az összegével.
Ennek segítségével el lehet képzelni, hogy a memóriában egy elemhez egy ilyen clusterSize méretű részt foglalunk le, ami lefoglal az összes lehetséges komponensnek helyet.



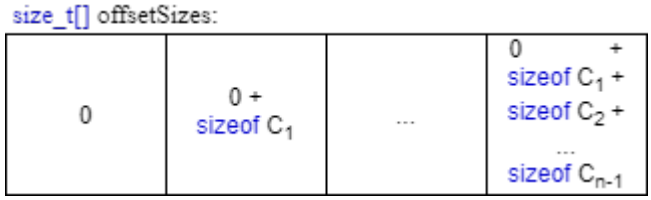
9. ábra: A klaszterek szemléltetése memóriában.

size_t* dataSizes: Egy componentCount méretű tömb, ami eltárolja a különböző komponensek méreteit. Az első elem, az elsőként regisztrált komponens méretét tárolja, a második elem a másodikét, és így tovább.



10. ábra: Méreteket tároló tömb, ahol C1...Cn-ek komponensek.

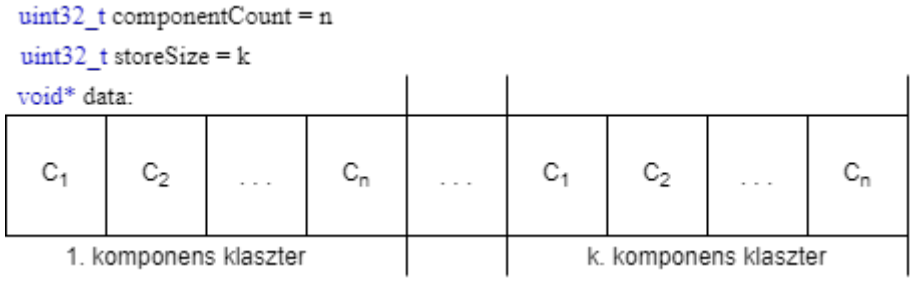
size_t* offsetSizes: Egy componentCount elemű tömb, ami eltárolja, hogy az adott komponens eléréséhez egy komponens-klaszterben mennyit kell arrébb lépni a memóriában. Az első elem, az elsőként regisztrált komponens eltolását tárolja, a második elem a másodikét, és így tovább.



11. ábra: Eltolásokat tároló tömb, ahol C1...Cn-ek komponensek

ComponentSignature* signatures: Egy componentCount elemű tömb, ami eltárolja, hogy az adott komponenst milyen szignatúrával regisztráltak. Az első elem, az elsőként regisztrált komponens szignatúráját tárolja, a második elem a másodikét, és így tovább.
uint32_t storeSize: Eltárolja, hogy mennyi entitás van jelenleg a rendszerben.

uint32_t storeCapacity: Eltárolja, hogy mennyi entitásnak lenne hely a rendszerben. Értéke a program futása közben többször is megnőhet, egészen pontosan duplázódat.
void* data: Memóriában egy clusterSize*storeCapacity méretű tömb. Ez storeSize darab entitáshoz tartozó komponenst tárol el az interfészben elmagyarázott módon.
Új entitás hozzáadása során a tömb mérete és kapacitása nőhet.



12. ábra: A komponenseket tároló void tömb ábrázolása*

ComponentSignature* entitySignatures: Egy storeSize elemű tömb. Eltárolja mindegyik entitáshoz a szignatúráját, azaz, hogy melyik komponenseket használja.

3.2.2.2.2.2 Statikus függvények

bool ExactlyOneBitIsSet (ComponentSignature bits)	Visszaadja, hogy a megadott szignatúrában csak egy bit egyes-e. Többnyire asszertekben használjuk.
bool SignatureExists (ECS_EntityStore* self, ComponentSignature newSignature)	Visszaadja, hogy self ECS_EntityStore signatures tömbjében benne van-e a newSignature. Egy egyszerű pesszimista eldöntési tétel.
uint8_t FindIndexToSignature (ECS_EntityStore* self, ComponentSignature signature):	Visszaadja, hogy self ECS_EntityStore signatures tömbjében hányadik helyen található a signature szignatúrájú elem 0-tól kezdve. Hogyha nincs ilyen asszertál.
void SwapEntityId (EntityId* id1, EntityId* id2)	Megcseréli kettő EntityId értékét. A rendezésnél, pontosabban az ECS_EntityStore_SortQuery-nél lesz hasznos.

uint32_t Partition (ECS_QueryResult* self, uint32_t low, uint32_t high, ECS_EntityStore_Comparator comparator)	A self ECS_QueryResult-ban helyére rak egy elemet úgy, hogy a comparator relációs logika szerint előtte csak nála kisebb, utána pedig csak nála nagyobb elemek szerepelhessenek, majd visszaadja, hogy ez a helyrerakott elem melyik indexen található. ECS_EntityStore_SortQuery miatt lesz rá szükség.
void QuickSort (ECS_QueryResult* self, uint32_t low, uint32_t high, ECS_EntityStore_Comparator comparator)	Rekurzívan rendezi a self ECS_QueryResult-ot a megadott low, high indexek között. ECS_EntityStore_SortQuery miatt lesz rá szükség.

3.2.2.2.2.3 Globálisan látható függvények belső működése

Ebben a részben található függvények elvárt működéséről az interfész részben lehetett olvasni. Most azt írom le, hogy ezek a függvények milyen módon működnek úgy ahogy.

3.2.2.2.2.3.1 ECS_EntityStore_Create:

Paraméterek:

uint32_t capacity	Az ECS kezdő kapacitását adja meg. Ez a program futása során nőhet.
uint8_t componentCount	Meg kell adni, hogy az őt követő paraméterlistában mennyi ECS_ComponentData típusú paramétert regisztrálunk.
...	Ehhez szükséges a legtöbb magyarázat, mert a C programozási nyelv ezen nyelvi eleme nem túl egyértelmű.

	Ez lényegében componentCount darab ECS_ComponentData típusú paraméter egymás után. Ezek kellenek ahhoz, hogy a tetszőleges adatot tároló komponenseket regisztrálni tudjuk az ECS-be.

A függvény elején ellenőrizni kell, hogy a componentCount 32-nél kisebb-e, illetve, hogy a capacity nem 0-e.

Az stdarg.h-ból szerzett funkciókkal végig tudunk iterálni a paraméterlistán. Ekkor számoljuk ki az memóriában történő eltolásokat (offset), egy komponensklaszter méretét, illetve a megadott szignatúrát leellenőrizzük, hogy helyes-e, majd elmentjük.

3.2.2.2.2.3.2 ECS_EntityStore_Destroy:

Felszabadítja a paraméterül megadott ECS_EntityStore self adattagjai, illetve maga által lefoglalt erőforrásokat, ezek:

- dataSizes
- offsetSizes
- signatures
- data
- entitySignatures
- végül saját magát.

3.2.2.2.2.3.3 ECS_EntityStore_CreateEntity:

A paraméterül kapott ECS_EntityStore self-ben létrehoz egy új entitást.

Először növeljük az self méretét, majd amennyiben nem lenne hely az új elemnek, akkor megkétszerezzük a kapacitását. Ellenőrizni kell, hogy van-e elég memória ehhez, ha nincs, akkor leállítjuk a programot. Vitathatóan az jobb lenne, ha ekkor speciális értékkel térnénk vissza, de jelen esetben ezt a megoldást választottam az összes ilyen helyen.

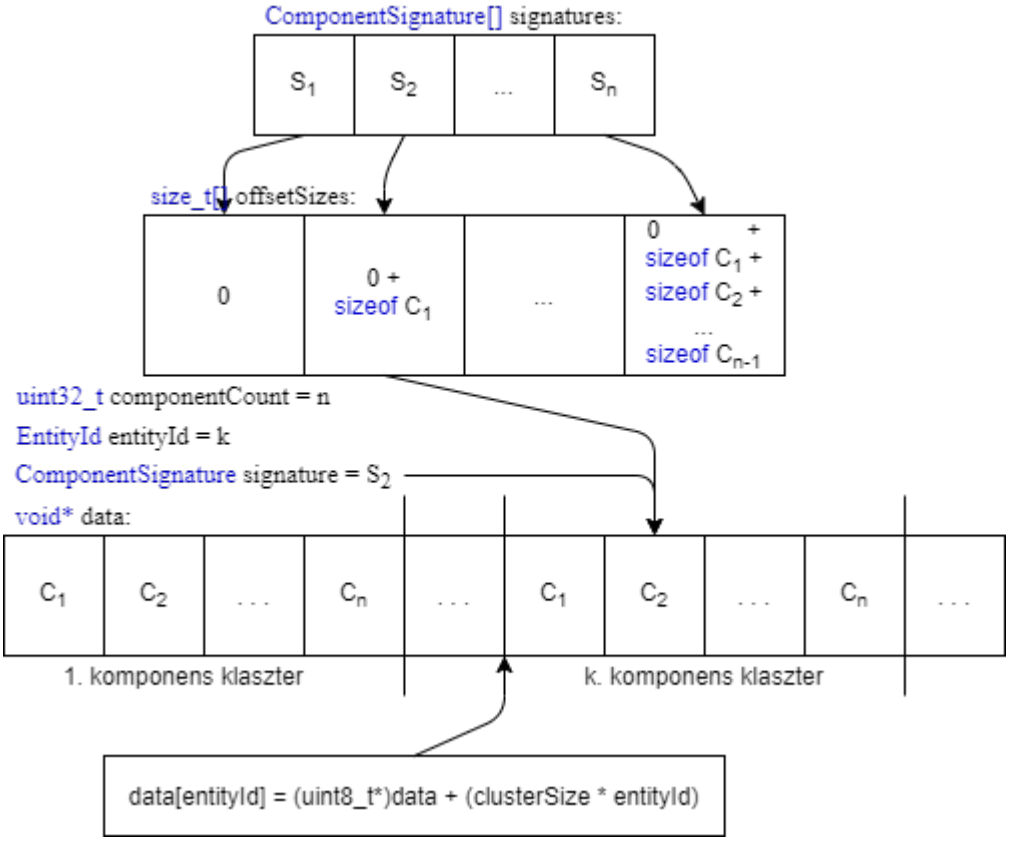
Az új entitás szignatúráját 0-ra állítjuk be, azaz egyik komponenssel sem rendelkezik. Mindezek után visszaadjuk az újonnan létrehozott entitás azonosítóját.

3.2.2.2.2.3.4 ECS_EntityStore_GetComponent:

Mindenekelőtt ellenőrizni kell, hogy a paraméterben megadott ComponentSignature signature szignatúrában csak egy darab egyes szerepel-e, illetve, hogy a kapott EntityId entityId valóban megtalálható a megadott ECS_EntityStore self-ben.

Ezek után pedig megkeressük a void* data tömbben az entityId indexű komponensklasztert, majd ismerve a megadott signature szignatúrájú komponens offset-jét, tudjuk, hogy a komponensklaszterben hol kezdődik a kért komponens.

Ezt többször fogjuk még használni, ezt az alábbi ábra magyarázza:



13. ábra: Hogyan találja meg a lekért komponenst a függvény

Ezen az ábrán jól látszik, hogy a memóriában hogyan történik ez a lekérdezés, illetve honnan tudja a függvény, hogy miután megtalálta a kért entitáshoz tartozó klasztert, utána mennyit kell arrébb lépnie, hogy a kért komponenst elérjük.

3.2.2.2.2.3.5 ECS_EntityStore_AddComponent:

Az előbb látott lekérdező függvényhez hasonlóan ellenőrizni kell a kapott szignatúrát, illetve azonosítót.

Ezt követően ki kell nézni a dataSizes tömbből, hogy a megadott szignatúrájú komponens mennyi helyet foglal, majd az entitáshoz tartozó szignatúrában bekapcsoljuk a hozzáadott komponens bitjét, majd a void* data tömbben belemásoljuk a megadott szignatúrájú komponens helyére a paraméterül kapott adatot.

3.2.2.2.2.3.6 ECS_EntityStore_RemoveComponent:

Az előbb látott lekérdező függvényhez hasonlóan ellenőrizni kell a kapott szignatúrát, illetve azonosítót.
Ez csupán a megadott entitás szignatúráját módosítja olyan módon, hogy a kikapcsolni kívánt szignatúrájú komponens bitjét 0-ra állítjuk.

3.2.2.2.2.3.7 ECS_EntityStore_HasComponents:

Az előbb látott lekérdező függvényhez hasonlóan ellenőrizni kell a kapott szignatúrát.
Megvizsgálja, hogy a megadott szignatúra egyesei szerepelnek-e az entityId azonosítójú entitás szignatúrájában, majd visszatér ezzel az értékkel.

3.2.2.2.2.3.8 ECS_EntityStore_KillEntity:

Az előbb látott lekérdező függvényhez hasonlóan ellenőrizni kell a kapott szignatúrát.
Csökkenti a megadott ECS_EntityStore self méretét, majd a void* data tömb utolsó komponensklaszterját átmásolja a törölni kívánt entitás klaszter helyébe. A szignatúrával is így jár el.

3.2.2.2.2.3.9 ECS_EntityStore_Query:

A megadott signature szignatúra tetszőleges lehet, így nem kell ellenőrizni a helyességét.
A lekérés elején inicializáljuk az ECS_QueryResult-ot, majd a megadott ECS_EntityStore self-ben megkeressük azokat az entitásokat, akiknek a szignatúrája tartalmazza a megadott signature-t, azaz mindenhol, ahol a signature-ben egyes egy bit, ott az entitás szignatúrájának is annak kell lennie.
Miután kigyűjtöttük a szükséges EntityId-ket, visszaadjuk az így előállított ECS_QueryResult-ot.

3.2.2.2.2.3.10 ECS_EntityStore_SortQuery:

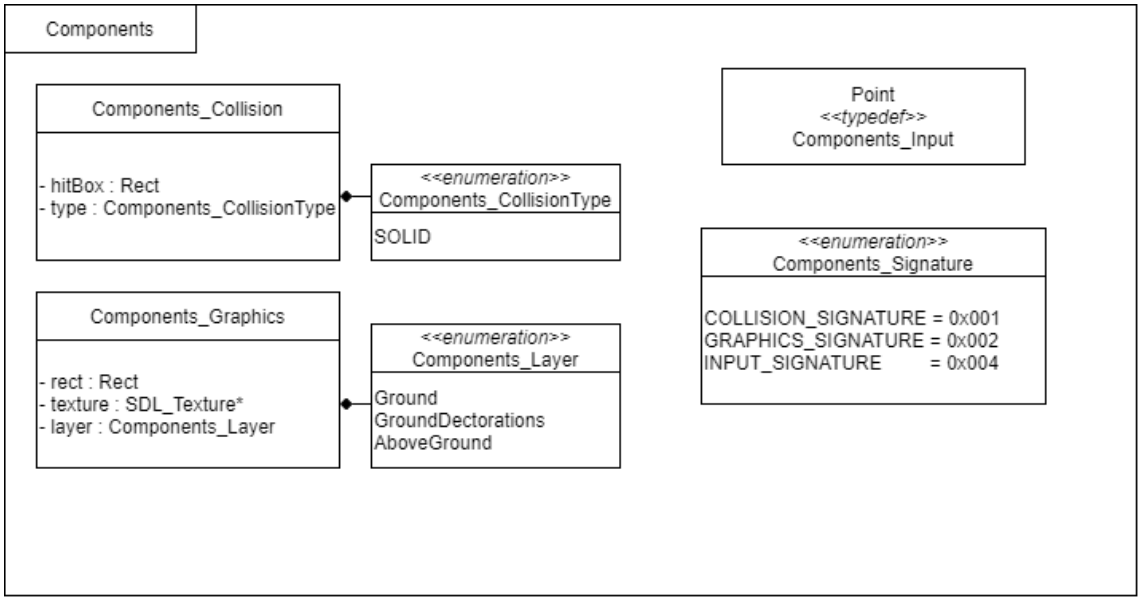
QuickSort-ot alkalmazva rendezi a megadott ECS_QueryResult-ot a kapott comparator logikája szerint.

3.2.2.2.2.3.11 ECS_QueryResult_Destroy

Felszabadítja a paraméterül megadott ECS_QueryResult self-et.

3.2.3 Components modul

A Components modul az előző 3.2.2 Entity Component System fejezetben megismert rendszernek a Component részét fogja alkotni, azaz tetszőleges adatokkal tudunk komponenseket regisztrálni az Entity Component Systembe, és ezen komponensekből fognak a játék entitásai összeállni.



14. ábra: A Components modul struktúrái.

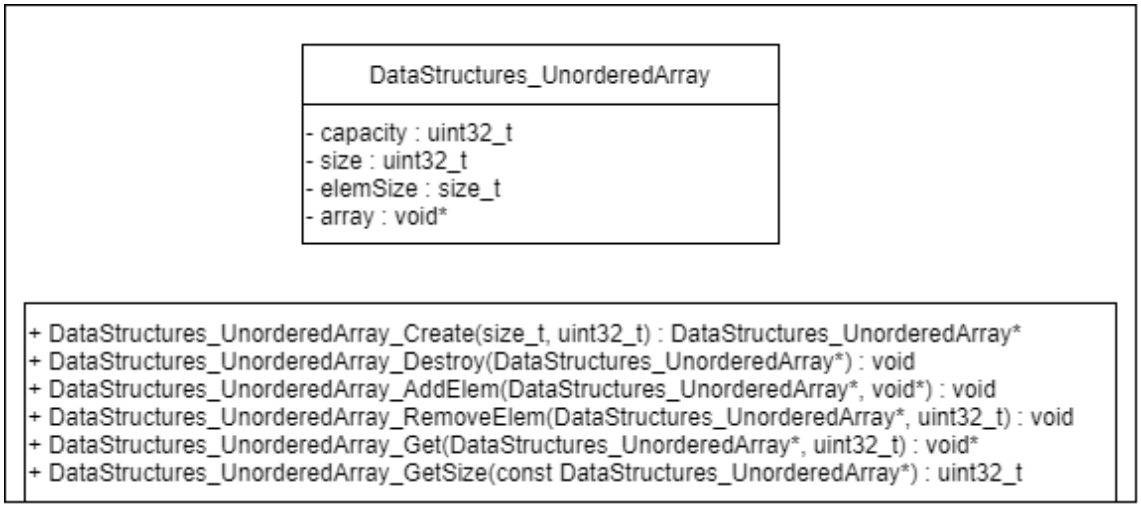
Az fenti ábrán látszik, hogy ebben a modulban nincsenek függvények, ténylegesen csak a komponensek, illetve a szebb kód érdekében rögzítjük a különböző komponensek szignatúráját.

3.2.4 DataStructures modul

Ez a modul hasznos adatstruktúrákat tartalmaz, jelenleg csak egyet, ami egy sorrendet nem tartó tömb. Ezeket rugalmasan lehet használni tetszőleges típussal, így nagyon hasznosak tudnak lenni több helyen is.

3.2.4.1 DataStructures_UnorderedArray

3.2.4.1.1 Interfész



15. ábra: Ábra a nem sorrendtartó tömb belső adatairól, és látható függvényekről.

Az Entity Component Systemhez hasonlóan, ennél az adatstruktúránál is void*-gal érjük el a nagy rugalmasságot, annyi különbséggel, hogy itt minden elem a tömbben elemSize méretű.

DataStructures_UnorderedArray* DataStructures_UnorderedArray_Create (size_t, uint32_t)	Létrehoz egy új tömböt, a megadott kezdő uint32_t kapacitással, amiben az elemek a megadott size_t méretként lesznek kezelve.
void DataStructures_UnorderedArray_Destroy (DataStructures_UnorderedArray*)	Felszabadítja a megkapott DataStructures_UnorderedArray tömböt.
void DataStructures_UnorderedArray_AddElem (DataStructures_UnorderedArray*, void*)	A megadott DataStructures_UnorderedArray tömbbe belemásolja a megadott void* adatot.
void DataStructures_UnorderedArray_RemoveElem (DataStructures_UnorderedArray*, uint32_t)	Törli a kapott DataStructures_UnorderedArray tömbből a megadott uint32_t indexen található elemet.

	Hogyha ez kiindexelne a tömbből, akkor asszertálunk.
void* DataStructures_UnorderedArray_Get (DataStructures_UnorderedArray*, uint32_t)	Visszaadja a kapott DataStructures_UnorderedArray tömbből a megadott uint32_t indexen található elemet. Hogyha ez kiindexelne a tömbből, akkor asszertálunk.
uint32_t DataStructures_UnorderedArray_GetSize (const DataStructures_UnorderedArray*)	Visszaadja a kapott DataStructures_UnorderedArray tömb elemeinek számát.

3.2.4.1.2 Implementáció

3.2.4.1.2.1 DataStructures_UnorderedArray_Create

Amennyiben a függvény kapacitásnak nullát kapna, akkor azt egynek vesszük.
Ezt követően inicializáljuk a méretet 0-ra, elmentjük, hogy a size_t értéket, hogy tudjuk, az elemek mekkorák lesznek, majd lefoglalt tömbbel visszatérünk.

3.2.4.1.2.2 DataStructures_UnorderedArray_Destroy

Felszabadítja a kapott tömb void* data adattagját, illetve magát a tömböt.

3.2.4.1.2.3 DataStructures_UnorderedArray_Get

Visszatérünk a void* array tömb kapott uint32_t indexen található elemével.
Ahogy az interfészben már le volt írva, ellenőrizni kell, hogy a kapott index nem mutat-e ki a tömbből.
Amennyiben még nem olvasta volna a 4.2.2.2.1 Void mutató alfejezetben a void* magyarázatát, és nem érti a működését, akkor erősen javaslom, hogy tekintse meg.

3.2.4.1.2.4 DataStructures_UnorderedArray_AddElem

Amennyiben a beszúrás következtében túlcsordulás lenne, megduplázzuk a kapacitást, amennyiben ez nem sikerülne asszertálunk.

Végül átmásoljuk a kapott void* adatot a tömb végére.

3.2.4.1.2.5 DataStructures_UnorderedArray_RemoveElem

Csökkentjük a tömb méretét, majd a megadott uint32_t indexű elem helyére belemásoljuk a tömb utolsó elemét.
Ahogy az interfészben már le volt írva, ellenőrizni kell, hogy a kapott index nem lóg-e ki a tömbből.

3.2.4.1.2.6 DataStructures_UnorderedArray_GetSize

Lekérdezi a kapott DataStructures_UnorderedArray tömb uint32_t size adattagját, s visszatér vele.

3.2.5 Camera modul

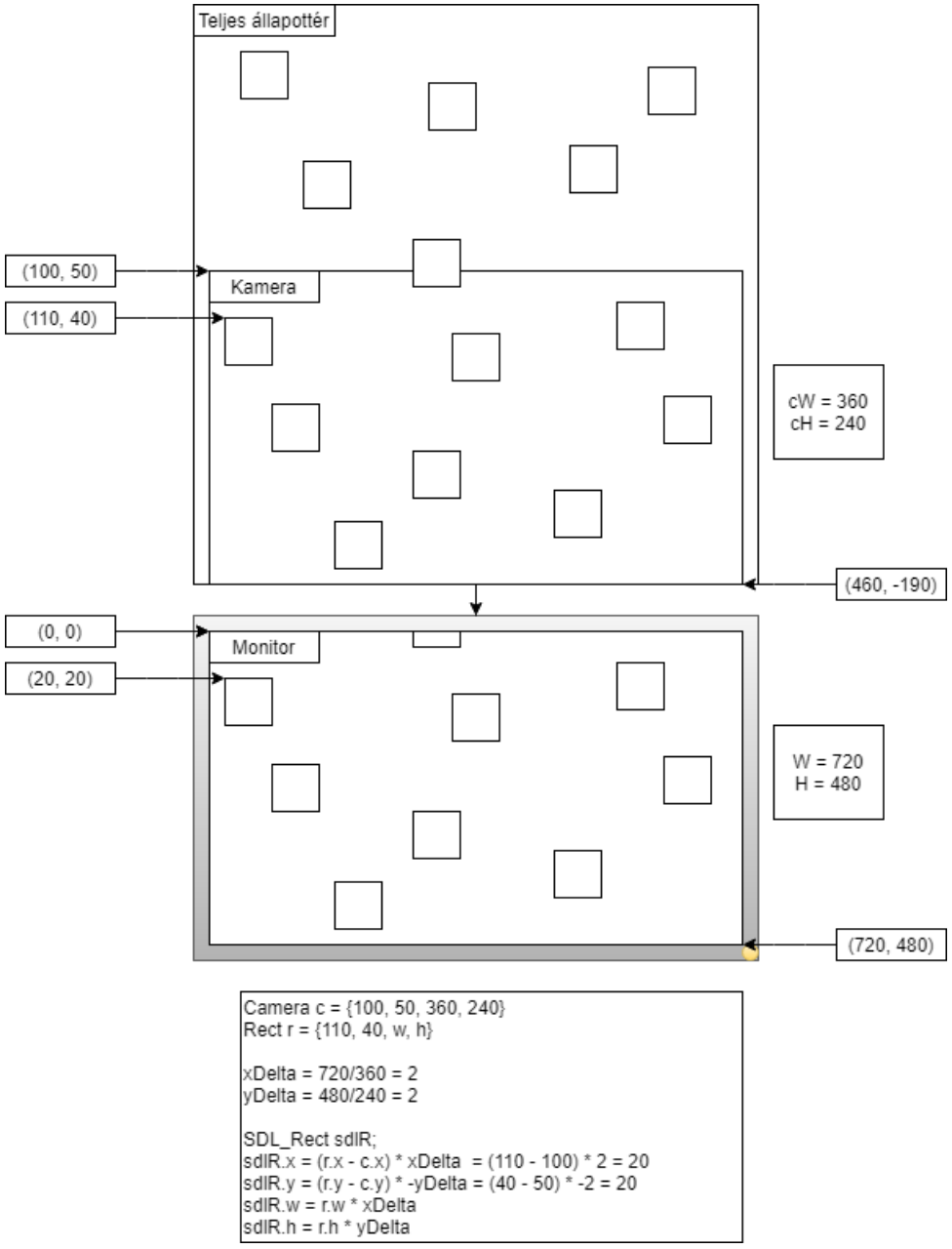
Ez a modul azért felel, hogy a monitor koordinátarendszere, illetve a játék logika koordinátarendszere között kapcsolatot teremtsen, adjon lehetőséget a kettő közötti koordináta átváltásra. Ehhez ismerni kell a kamera játék-logika szerinti helyét, illetve a monitor méreteit pixelben.
Az SDL2-ben a monitor bal felső sarka a (0, 0) koordináta pont, míg a jobb alsó a következő: (*képernyő szélessége pixelben*, *képernyő magassága pixelben*).
Természetesen nem szeretnénk, hogy majd a játékban az entitások elhelyezkedése ehhez legyen hozzá kötve, így ezért szükséges, hogy tudjuk, hogy a kamera jelenleg hol helyezkedik el, illetve, hogy tudjuk a kettő között átváltani.

3.2.5.1 Interfész

Rect Camera_CalculateRectFromSDLRect (const Camera*, UInt16, UInt16, const SDL_Rect*)	A kapott kamera, illetve ablakméretek segítségével kiszámolja a megadott SDL_Rect Világ-beli alakját, mint Rect.
SDL_Rect Camera_CalculateRectFromSDLRect (const Camera*, UInt16, UInt16, const Rect*)	A kapott kamera, illetve ablakméretek segítségével kiszámolja a megadott Rect Világ-beli alakját, mint SDL_Rect.

A modulban található még egy Camera_RenderingData nevű struktúra. Ez azért hasznos, hogy több helyen csökkenteni tudjuk a kért paraméterek számát. Ez eltárolja a kamerát, az ablak méreteit, illetve az SDL_Renderer-t is, amivel rajzolni tudunk a képernyőre.

3.2.5.2 Implementáció



16. ábra: Szemlélteti a kamera modul konvertáló függvények működését.

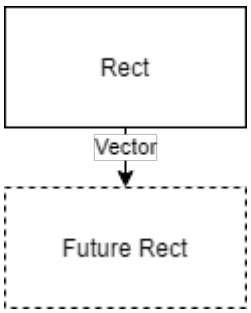
A fenti ábra szemlélteti, hogy milyen módon történik az átváltás az állapottérből a kamerára.

Ami itt látszik, hogy míg a monitor koordinátája lefelé nő, addig az én koordinátarendszerem csökken.

Az egyszerűség kedvéért választottam úgy a méreteket, hogy az xDelta, és yDelta megegyezzen, de ez nem kötelező, azonban akkor torzulás fog fellépni a téglalapokban. Az ábrán nem látszik, viszont a szemléltetett konverzió esetén szükséges kerekíteni, mert a számítógép lebegőpontos ábrázolásából, illetve a lebegőpontos szám egésszé alakítása következtében pontatlanságok léphetnek fel. Szerencsére, ha mindig felfele kerekítjük a számolt értékeket, akkor ebből nem lesz látható probléma.
Az állapottér-monitor rajzon nem látszódik, de a számolásoknál látszik, hogy hogyan lehet átváltani a téglalapok hosszát, magasságát.

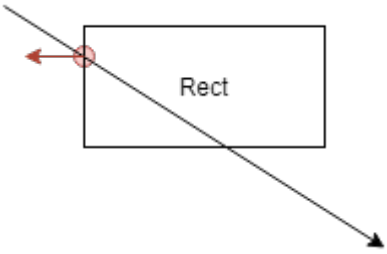
3.2.6 Collision modul

A modul felelősége, hogy eszközt adjon annak detektálására, hogy egy alakzat ütközik-e egy téglalappal. Az ütköző alak lehet pont, másik téglalap, irányvektor, vagy egy mozgó téglalap.
A mozgó téglalapot úgy kell elképzelni, hogy tudjuk a téglalap kiinduló helyzetét, illetve egy vektort, ami elárulja milyen irányba, mennyit haladna a téglalap.



17. ábra: Mozgó téglalap

A mozgó téglalapoknál, illetve vektoroknál fontos megemlíteni, hogy nem elég annyi, hogy lesz-e ütközés, vagy sem, hanem az is érdekel minket, hogy mikor történne az ütközés. Ezt úgy tudjuk megvalósítani, hogy azt mondjuk meg (paraméteren keresztül), hogy a vektor hányad részénél történik az ütközés. Hasonlóan vissza lehet adni, az ütközés pontját, illetve az ütközött felületre merőleges normálvektort.
Az ábrán látszik, a piros pont az ütközés pontja, a piros vektor pedig a normálvektor, az ütközés ideje pedig nagyjából egynegyed.



18. ábra: Ütközés pontja, ideje, normálvektorja

A saját megoldásom során sokat merítettem David Barr implementációjából. (David, 2020)

3.2.6.1 Interfész

bool CollisionDetection_PointRect (const Point*, const Rect*)	Megvizsgálja, hogy a megadott Point ütközik-e a megadott Rect-tel.

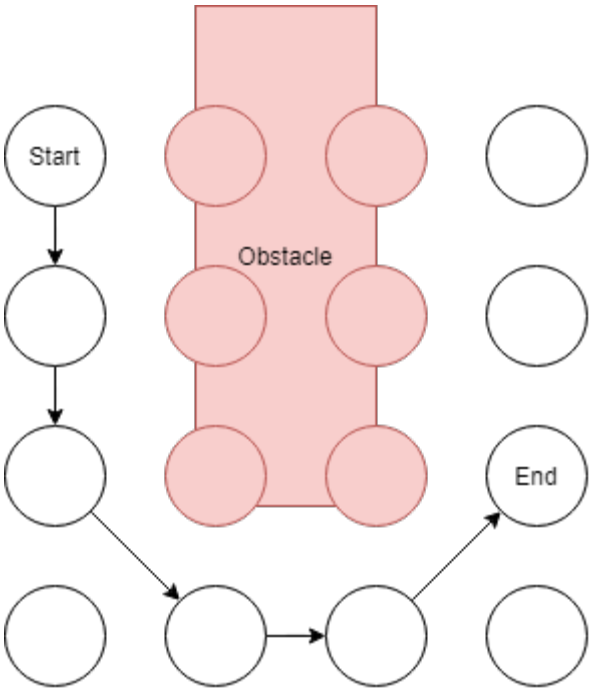
	Ha a Rect szélét érinti, már arra is jelezni kell.
bool CollisionDetection_RectRect (const Rect*, const Rect*)	Megvizsgálja, hogy a két Rect ütközik-e egymással Ha a Rect szélét érinti, már arra is jelezni kell.
bool CollisionDetection_VectorRect (const Vector*, const Rect*, Point* cP, Point* cN, double* cT)	Megvizsgáljuk, hogy a kapott Vector ütközik-e a megadott Rect-tel. Ha igen, akkor az ütközés idejét, helyét, illetve normálvektorját visszaadjuk az utolsó három paraméteren keresztül. • cP (collision-point): ütközés helye • cN (collision-norm): ütközés normálvektorja • cT (collision-time): ütközés ideje
bool CollisionDetection_MovingRectRect (const Rect*, const Point*, const Rect*, Point* cP, Point* cN, double* cT)	Megvizsgáljuk, hogy a kapott Rect, a megadott Point normálvektorral ütközik-e a megadott Rect-tel. Ha igen, akkor az ütközés idejét, helyét, illetve normálvektorját visszaadjuk az utolsó három paraméteren keresztül, az előbb látott módon.

3.2.7 Pathfinding modul

A modul segítségével tudunk két pont között utat találni eléggé rugalmas módon. Ehhez szükséges ismernünk egy ECS_EntityStore-t, hogy meg tudjuk majd ítélni, hogy az útkeresés során lenne-e ütközés, így azokat a pontokat ki kell majd kerülni az utunknak. És természetesen ehhez az előbb látott Collision modul funkcionalitásait használjuk.

3.2.7.1 A* algoritmusról

Az útkereséshez az A* algoritmust fogjuk használni.



19. ábra: Útkeresés tetszőleges méretű rácson

Az útkeresés lényegében egy rácson fog történni, ahol a felhasználó megadhatja, hogy milyen messze legyenek egymástól a rács pontjai, illetve, hogy milyen messze keressen utat. A különböző körök ezek a pontok a rácson, ahol az útkeresés zajlik. Kezdetben csak a kiinduló, és végpontot ismerjük, ahogyan a kiinduló start pontnak a szomszédos pontjait kiterjesztjük, úgy fogunk utat találni.

A piros téglalap egy akadály, amin keresztül nem tudunk utat találni. Ekkor a pirossal jelölt pontokat lényegében nem is generáljuk le.

Az algoritmus használ egy heurisztikát is a (remélhetőleg) gyorsabb úttalálás érdekében. Ez az implementációmban a következő: távolság a starttól + távolság a célig. Az utóbbi természetesen csak egy becslés lehet, viszont, ha folyton szamon tartjuk a starttól vett távolságot, akkor azt biztosan tudni fogjuk.

A rács pontjai a következő adattagokkal rendelkeznek:

Node:

- `Point coords`: A pont koordinátája.
- `double distanceFromStart`: Pontos érték, a start ponttól vett távolság.
- `double distanceToEnd`: Becsült érték, a végpontig vett távolság feltéve, hogy egy vektor mentén jutunk el a célig.
- `Node* parent`: Az őt megelőző pont. Szükséges, hogy vissza tudjuk majd vezetni a talált utat.

Az algoritmus eredményül egy Point-okat tartalmazó DataStructures_UnorderedArray-el fog visszatérni.

3.2.7.1.1 Az algoritmus működése

- Először ellenőrizzük, hogy a célpont nincs-e kinn a keresési körből.
- Ezt követően számon fogjuk tartani a nyitott, illetve a lezárt pontokat. A nyitott pontok halmazában olyanok lesznek, akiket felfedeztünk, de nem vizsgáltunk meg.
- Kezdetben beletesszük a startpontot a nyílt pontok halmazába.
- Ezek után egy ciklus fut addig, amíg nem találtunk utat, vagy ki nem fogytunk a nyílt pontokból.
- Ezek után meg kell keresni a legígéretesebb pontot. Ehhez a korábban említett heurisztikát alkalmazzuk: distanceFromStart + distanceToEnd. Erre a pontra a továbbiakban currentNode-ként fogok hivatkozni.
- A currentNode-ot áthelyezzük a zárt pontok halmazába, hiszen éppen most dolgozzuk fel.
- Hogyha elég közel lenne currentNode a végponthoz, akkor kilépünk a ciklusból, hiszen utat találtunk.
- Ezt követően frissítjük, vagy ha még nem lennének benne a nyílt pontok halmazába, akkor beletesszük őket. Ekkor beállítjuk a pontoknak a távolságát a starttól, illetve a végpontig, illetve a szülőcsúcsot is beállítjuk currentNode-ra. Utóbbiakat csak akkor, ha a currentNode-on keresztül gyorsabban elérnénk az új pontot.
- Végül, ha találtunk utat visszavezetjük a parent-eken keresztül, majd visszaadjuk egy DataStructures_UnorderedArray-ban.

A megoldás során sokat merítettem David Barr implementációjából. (David, OneLoneCoder (javidx9) . Github, 2017)

3.2.8 TextureManager modul

Ez a modul azért felel, hogy a játék során alkalmazott képeket el tudjuk érni, és lekérni, mint SDL_Texture.

A modul egyszerűen működik:

A TextureManager struktúra publikusan nem látható módon tárolja az összes lehetséges textúrát, illetve ezek elérésére biztosít függvényeket.
Jelen esetben ebből csak egy példány van, és azt érjük el a függvényeken keresztül.

3.3 Demó alkalmazás moduljai

3.3.1 World modul

3.3.1.1 EntityActions

Ez az almodul a világhoz tartozó ECS_EntityStore-hoz köt gyakran használt függvényeket, úgy is tekinthetünk rá, hogy az *Entity Component Systemnek* ez a *System* része.
Az alap funkcionalitások, amikre szükségem volt, a következők:

- ECS entitásainak kirajzolása.
- ECS entitásainak inputkezelése. Jelen esetben csak a játékos karaktere reagál inputra.
- ECS entitásainak frissítése minden képkockában. Ebbe beleértjük az ütközések feloldását is.

Ezen túl az áttekinthetőbb kód érdekében létrehoztam a különböző 'típusú' entitásokhoz létrehozó függvényeket is.

3.3.1.1.1 Interfész

void World_EntityActions_DrawEntites (ECS_EntityStore*, Camera_RenderingData*)	A függvény a kapott ECS_EntityStore entitásait kirajzolja, ha rendelkezik grafikus komponenssel. A Camera_RenderingData-ról a *Camera* modulról szóló részben lehet olvasni.
void World_EntityActions_ProcessInput (ECS_EntityStore*, const Uint8*)	A függvény a kapott ECS_EntityStore entitásaira dolgozza fel a billentyűzeten történő inputokat, ha rendelkezik input komponenssel. A második paraméter a billentyűzet állapota, amit az SDL_GetKeyboardState függvénnyel tudunk lekérni,

void World_EntityActions_UpdateEntities (ECS_EntityStore*, Camera*)	A függvény a kapott ECS_EntityStore entitásait frissíti. Ebbe tartozik bele az ütközések feloldása, elkerülése.

Ezeken túl, ahogy említettem vannak a következő inicializáló függvények

EntityId World_EntityActions_CreatePlayer (ECS_EntityStore*, Point*)	A megadott ECS_EntityStore-ban létrehoz egy új entitást a következő komponensekkel: • Grafikus komponens • Ütközés komponens • Input komponens A megadott Point koordináta egy referenciapont, ehhez képest helyezi el a komponenseket a világban, például a grafikus komponens koordinátáival megegyezik.
EntityId World_EntityActions_CreateGrassTile (ECS_EntityStore*, Point*);	A megadott ECS_EntityStore-ban létrehoz egy új entitást a következő komponensekkel: • Grafikus komponens A megadott Point koordináta egy referenciapont.
EntityId World_EntityActions_CreateVoidTile (ECS_EntityStore*, Point*);	A megadott ECS_EntityStore-ban létrehoz egy új entitást a következő komponensekkel: • Grafikus komponens • Ütközés komponens A megadott Point koordináta egy referenciapont.

EntityId World_EntityActions_CreateTree (ECS_EntityStore*, Point*);	A megadott ECS_EntityStore-ban létrehoz egy új entitást a következő komponensekkel: • Grafikus komponens • Ütközés komponens A megadott Point koordináta egy referenciapont.
EntityId World_EntityActions_CeateFlowers (ECS_EntityStore*, Point*);	A megadott ECS_EntityStore-ban létrehoz egy új entitást a következő komponensekkel: • Grafikus komponens A megadott Point koordináta egy referenciapont.

3.3.1.1.2 Implementáció

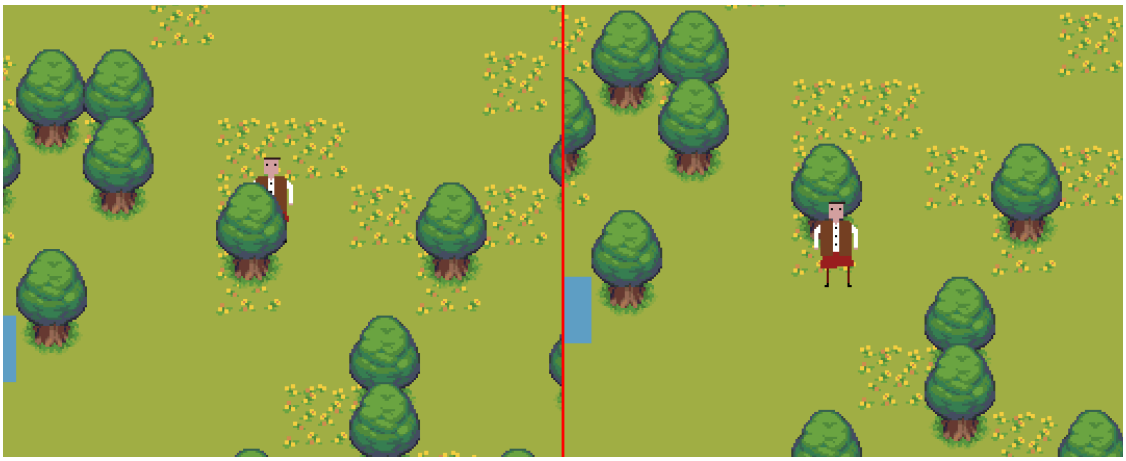
Ebben a részben a 3 főfüggvényt fogom elmagyarázni, illetve az egyik inicializáló függvényt.

3.3.1.1.2.1 World_EntityActions_DrawEntites

Itt nagyon fontos, hogy az entitásokat jó sorrendbe rajzoljuk ki. Ha például az összes füves-talajt a játékos kirajzolása után tennénk, akkor azok lényegében eltakarnák őt, így nem is látszana, hiába van jó helyen a játék logikájában.

Ezért fontos, a Components_Graphics Components_Layer adattagja. Hogyha először a Ground réteg, majd a GroundDecorations, végül az AboveGround entitásokat rajzoljuk ki, akkor ez a probléma megszűnik.

Viszont, ha azt szeretnénk elérni, hogyha egy fa mögött állunk, akkor a fa legyen a játékos előtt, míg, ha a fa elött állunk, akkor a fa takarja a játékost, ahhoz ennél több kell. Szerencsére ehhez csupán az entitások textúrájának alját (y koordináta) kell tudnunk, amit pedig ismerjük.



20. ábra: Helyes kirajzolási sorrend

Szerencsére az ECS modulban már azt megoldottuk, hogy a lekérdezésünket rendezni tudjuk tetszőleges logika szerint, itt már csak azt kell használni, azaz kell egy függvény, ami megmondja, hogy két grafikus komponenssel rendelkező entitás mikor nagyobb, kisebb a rendezés szerint. Ezt a logikát az előző bekezdésben olvasott módon a CompareEntities statikus függvényben írtam le.

Így a World_EntityActions_DrawEntites feladata annyi, hogy lekérdezze a grafikus komponenssel rendelkező entitásokat, rendezze a lekérdezést a CompareEntities logikája szerint, majd minden elemnek a koordinátáit váltsa át a képernyő koordinátarendszerére, majd rajzolja ki az entitáshoz tartozó textúrát.

Ezen a függvényen belül történik az útkeresés is, ami az egér kurzortól fog a játékos hitbox-ig utat keresni, és rajzolni. Az ECS_EntityStore-ban meg kell keresni a játékos komponenseit. Itt kihasználjuk, hogy a játékost elsőként adtuk hozzá az ECS-hez, illetve mivel a program futása közben nem törlünk belőle az esélye sem áll fenn annak, hogy ez az index invalidálódjon.

Az egér helyét a SDL_GetGlobalMouseState függvénnyel kérjük le, majd átváltjuk a kapott pontot a játéklogikába. Ezután a 4.2.7 Pathfinding fejezetben látott függvényt kell használni, majd a kapott pontok mentén egyenest rajzolni a SDL_RenderDrawLine segítségével.

21. ábra: Útkeresés

3.3.1.1.2.2 World_EntityActions_ProcessInput

Ez a függvény a megadott billentyű-állapot szerint módosítja az entitások x-y koordináta szerinti elmozdulását. Ezt az elmozdulást a World_EntityActions_UpdateEntities-ben fogjuk feldolgozni.

3.3.1.1.2.3 World_EntityActions_UpdateEntities

Ez jelen esetben csak a játékost érinti. Amikor az input komponensből látszódik, hogy volt input elmozdulásra valamely irányba, akkor meg kell vizsgálni, hogy lenne-e ütközés egy másik entitással. Ha igen, akkor a visszaadott ütközési idő segítségével tudjuk, hogy mennyit szabad csak arrébb mozogni.
Ezek után a kamerát is mozgatni kell, hogy az kövesse a játékost.

3.3.1.1.2.4 World_EntityActions_CreatePlayer

Létrehozunk az ECS_EntityStore-ban egy új entitást, majd hozzáadjuk a kellő komponenseket, jelen esetben mind a hármat.
Grafikus-komponens:
Kirajzolási sorrendben legyen a felszín, illetve a dekorációs virágok felett. Ezért található az AboveGround rétegen.
A TextureManager-től elkérjük a kellő texturát, és átadjuk a Components_Graphics-nak.

Végül beállítjuk a textúra koordinátáit, hogy megegyezzen a kapott referencia-koordinátákkal.

Components_Collision esetében az ütközést jelző dobozt a játékos lábától szeretnénk, így ott kisebb y-t kell beállítani.

Kezdetben nem mozog a játékos, ezért a Components_Input-ot inicializálhatjuk 0, 0 koordinátákkal.

3.3.1.2 World

Az almodulnak a következőek a felelősségei:

- Világ reprezentálása adat formában
- Világ generálása
- Entitások kirajzolása
- Entitásokra inputkezelés
- Entitások frissítése

Az utóbbi három funkciót az előbb látott EntityActions almodul már megvalósította, így ezeket csak használni kell.

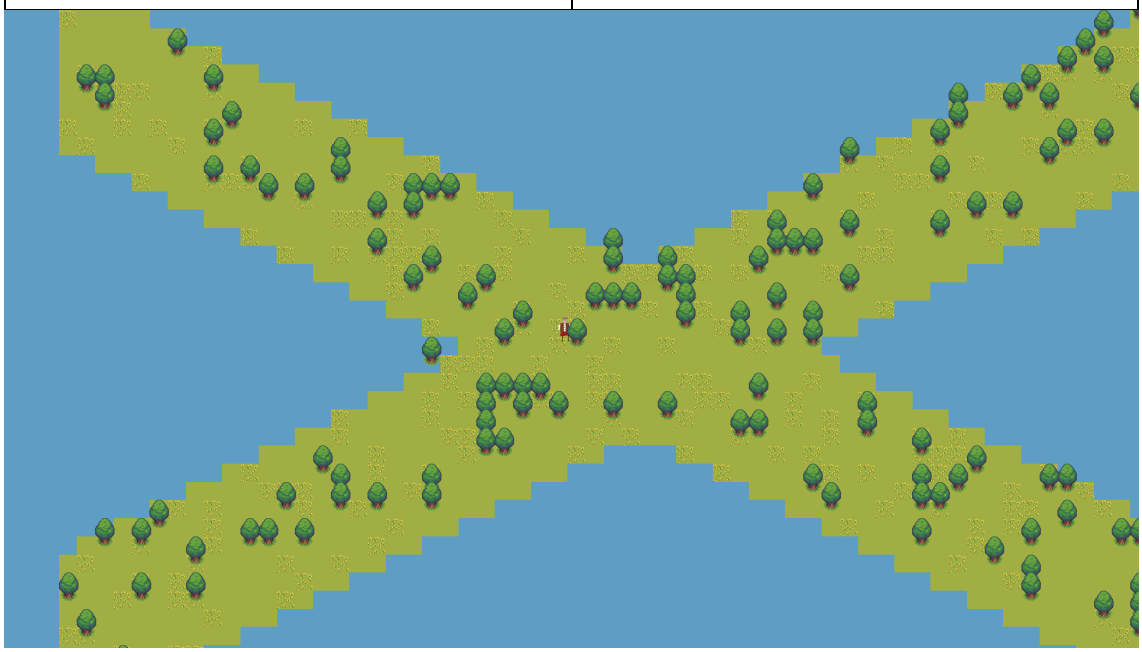
3.3.1.2.1 Interfész

22. ábra: World UML ábrázolása

A World belső szerkezete ismeri, hogy milyen seed-del lett generálva. Tudja, hogy mekkora a teljes világ, ezt a worldRect-ben tároljuk. Ismerni kell a Camera-t is, amit majd a felette elhelyezkedő View modul fogja biztosítani. Végül eltárolja a világban szereplő összes entitást az entities ECS_EntityStore-ban.

World* World_Create (Camera*)	Inicializálja a világot, és visszaad rá egy mutatót.

void World_Destroy (World*)	Felszabadítja a megadott World által lefoglalt erőforrásokat.
void World_Generate (World*, unsigned int)	Legenerálja a megadott World világot a megadott unsigned int seed-del. Az utóbbi egy tetszőleges szám, ami segítségével fog a random generálás zajlani. A szigetet egy X alakban generáljuk le Void csempékkel körülvéve.



23. ábra: X alakú világ

Ezek a függvények meghívják az EntityActions-ben bemutatott függvényeket a kapott paraméterekkel, így az elvárt működésük is megegyezik.

void World_DrawEntities (World*, Camera_RenderingData*)	World_EntityActions_DrawEntites függvényt hívja meg.

void World_ProcessInput (World*, const Uint8*)	World_EntityActions_ProcessInput függvényt hívja meg.
void World_UpdateEntities (World*, Camera*)	World_EntityActions_UpdateEntities függvényt hívja meg.

3.3.1.2.2 Implementáció

3.3.1.2.2.1 World_Create

Ez a rész azért különösen érdekes, mert itt regisztráljuk az Entity Component System-be a különböző komponenseket.

Miután allokáltunk memóriát a World-nek, inicializáljuk az ECS-t. Itt meg kell adni a kezdő kapacitást, a komponensek számát, illetve a komponensek adatait egy ECS_ComponentData formájában, aminek tartalmaznia kell a regisztrálandó komponens méretét, illetve az egyedi egy darab egyest tartalmazó szignatúráját.

A programban jelenleg három komponensből állhatnak össze az entitások, így mind a hármat: Components_Collision, Components_Graphics, Components_Input regisztráljuk.

```
World* result = malloc(sizeof *result);

result->entities = ECS_EntityStore_Create(1, 3,
    (ECS_ComponentData){.size = sizeof(Components_Collision),    .signature = COLLISION_SIGNATURE},
    (ECS_ComponentData){.size = sizeof(Components_Graphics),     .signature = GRAPHICS_SIGNATURE},
    (ECS_ComponentData){.size = sizeof(Components_Input),        .signature = INPUT_SIGNATURE}
);
```

24. ábra: Példa komponensek regisztrálására.

Ezt követően a paraméterül kapott kamera referenciáját eltároljuk.

3.3.1.2.2.2 World_Destroy

Felszabadítjuk az ECS_EntityStore-t, majd a kapott World-et.

3.3.1.2.2.3 World_Generate

- Elmentjük a kapott seed-et, majd véletlen sorozatot generálunk vele.
- Először külön létrehozzuk a játékost az ECS-ben, majd elmentjük a grafikus komponensét, mert erre később szükségünk lesz.

- A kamera x, y koordinátáit beállítjuk úgy, hogy a játékos textúrája legyen a középpontban.
- Fix mérettel létrehozzuk a világot leíró téglalapot. Majd sorról-sorra, oszlopról-oszlopra végig iterálunk a bal felső saroktól kezdve 32 mértékegységenként, ez megegyezik a talaj-elemek méretével.
- A ciklusmagban meghatározzuk, hogy a most vizsgált talaj fű legyen-e, vagy víz. Ezt oly módon teszi, hogyha illeszkedik az y = ½x, vagy az y = -½x függvényre, akkor fű, különben víz. Ez eredményezi az X alakot.
- Ezt követve, ha füvet generáltunk, véletlenszerűen eldöntjük, hogy az adott talajra virág, fa, vagy semmi se kerüljön.

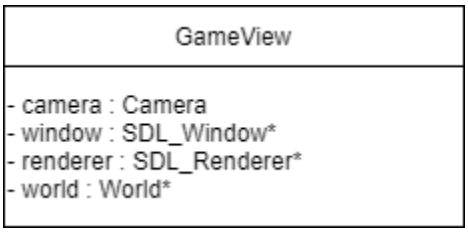
3.3.1.2.2.4 World_DrawEntities, World_ProcessInput, World_UpdateEntities

Ahogyan említettem ezek meghívják az EntityActions megfelelő függvényét, így itt nem fejtem ki ezeket még egyszer.

3.3.2 View modul

Ez a modul felelős a monitorkezelést biztosítani, más moduloknak továbbítani ezeket a függőségeket.

3.3.2.1 Interfész



25. ábra: View UML

A GameView fogja birtokolni az ablakhoz, illetve a rá rajzolást végző rendererhez tartozó referenciát. Továbbá birtokolja a World struktúrát is, és neki adja át a renderer-t, hogy az majd tudjon a képernyőre rajzolni. Ezeken túl a kamerát is ismernie kell, mert ő állítja be annak hosszát és szélességét az ablakméret ismeretében, hogy az arányok megmaradjanak.

View_GameView* View_GameView_Create ()	Létrehoz egy új nézetet, és visszaad hozzá egy referenciát.

	A nézet ekkor hozza létre az ablakot, a renderer-t, illetve a kamerát és a világot is.
void View_GameView_Destroy (View_GameView*);	Felszabadítja a megadott View_GameView* által lefoglalt erőforrásokat.
void View_GameView_Loop (View_GameView*);	Ez az egész játéknak a szíve. Ekkor generáljuk le a világot. Feladata, hogy meghatározott időközönként frissítse a világ entitásait, és rajzolja ki őket, illetve vizsgálja, hogy volt-e input, és továbbítsa azt a World-nek.

3.3.2.2 Implementáció

3.3.2.2.1.1 View_GameView_Create

Allokáció elvégzése után létrehozzuk az ablakot az SDL_CreateWindow függvénnyel, és ekkor hozzuk létre a renderer-t is az SDL_CreateRenderer segítségével.

A TextureManager-t inicializáljuk, majd beállítjuk a kamera adatait úgy, hogy az arányok megmaradjanak.

A világot itt hozzuk létre, viszont a legenerálása nem ennek a függvénynek a felelősége.

3.3.2.2.1.2 View_GameView_Destroy

Fel kell szabadítanunk a nézet által lefoglalt erőforrásokat:

- ablakot
- renderer-t
- világot
- TextureManager-t

Végül a nézetet is felszabadítjuk.

3.3.2.2.1.3 View_GameView_Loop

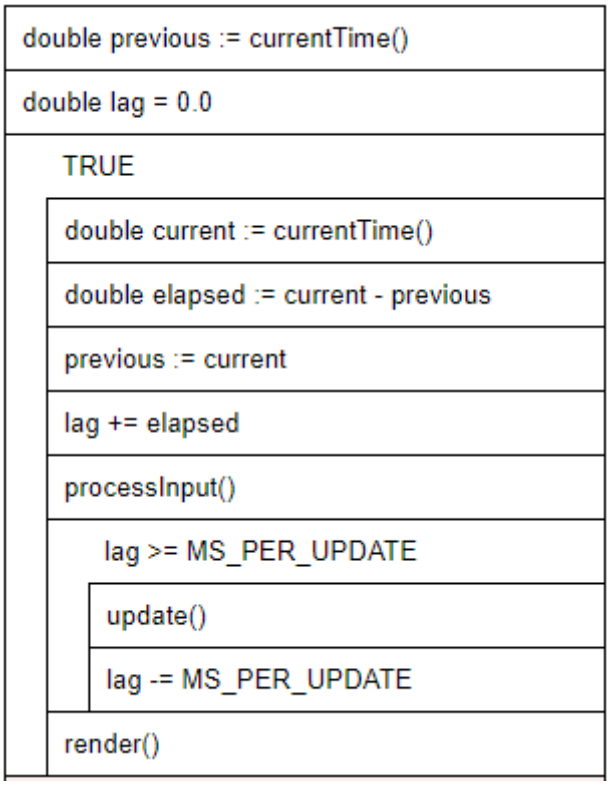
Legeneráljuk a világot egy véletlenszerű seed-del, majd előállítjuk a Camera_RenderingData-t, ami az entitások kirajzolásához szükséges információt tartalmazza.

3.3.2.2.1.3.1 A játék ciklusa

Arra több megoldás van, hogy ezt a ciklust milyen módon kezeljük, de az összes egy hasonló koncepcióra épül:

1. Input feldolgozása
2. frissítés
3. rajzolás

És ezt a három feladatot ismételjük, amig ki nem lép a játékból a felhasználó. Ezzel a legnagyobb probléma az, hogy különböző hardware-rel rendelkező gépeken más lesz a játékélmény. Ha valakinek nagyon lassú a gépe, akkor lassítottfelvételben fogja játszani a játékot, ha valakinek túl gyors, akkor meg a játék is sokkal gyorsabban fog zajlani.
Erre több megoldás is van, amit én választottam a következő:



26. ábra: Stuktogram a játék ciklusáról

Ez úgy működik, hogy ismerjük mennyi idő telt el az előző iteráció óta, és azt is, hogy milyen gyakran szeretnénk frissíteni a világ entitásait. Így tudjuk, hogy mikor egy új iteráció kezdődik, hogy most kell frissíteni, vagy sem.
Ha több kör frissítés kimaradt azt is meg tudjuk oldani, hiszen a lag-ból tudjuk, hogy mennyivel vagyunk elcsúszva a valóságos időtől.

A megoldás Robert Nystrom megoldásán alapszik, amit az általa írt Game Programming Patterns Game Loop fejezetében olvashatnak. (Robert, Game Programming Patterns, 2014)

3.4 Tesztelés

3.4.1 Egységtesztek

A Unit teszteket c++-ban mert kényelmes Framework, a doctest (Viktor, dátum nélk.) keretrendszer segítségével készítettem el, a következő modulokhoz: Camera, Collision, DataStructures, ECS, Pathfinding.

Camera tesztjeiben vannak előre definiálva különböző kamerák, amik segítségével történnek különböző konverziók, illetve egy fix képernyő szélességgel, illetve magassággal dolgozik.

Collision tesztjeiben a különböző geometriai alakokkal történő ütközések, detektálását, vagy ha nincs ütközés, akkor annak hiányát ellenőrizzük. Itt a CollisionDetection függvényeinek definiáljuk a helyes működést.

A DataStructures tesztjében a jelenleg egyetlen adatstruktúránkra írtam teszteket, a rendezetlen tömbre. Szükséges tesztelni a helyes inicializálást, elem hozzáadást, elem levételt, lekérdezést.

Az ECS-re is készültek egységtesztek. Tesztelésre került az inicializálás, egyszerű komponensekkel, új entitás hozzáadása. Entitásokhoz tartozó komponens lekérése, hozzáadása, törlése, létezésének lekérése. Ezentúl a lekérdezések (Query) helyessége is.

A Pathfinding tesztjeiben szükséges egy mock ECS olyan entitásokkal, amikre a CollisionDetection alkalmazható. Ezek után lehet tesztelni az A* algoritmus helyességét.

3.4.2 További tesztelések

Ezeken túl kézzel is sokat teszteltem, illetve Linux-on le lett tesztelve, hogy van-e benne memóriaszivárgás, és a használt address sanatizer (asan) [https://en.wikipedia.org/wiki/AddressSanitizer] nem talált.

4 Irodalomjegyzék

Austin, M. (2019. június 25). *A Simple Entity Component System (ECS) [C++] Austin Morlan*. Forrás: https://austinmorlan.com/posts/entity_component_system/#the-entity

David, B. (2017. október 9). *OneLoneCoder (javidx9) . Github*. Forrás: https://github.com/OneLoneCoder/videos/blob/master/OneLoneCoder_PathFinding_AStar.cpp

David, B. (2020. július 13). *OneLoneCoder (javidx9) . Github*. Forrás: https://github.com/OneLoneCoder/olcPixelGameEngine/blob/master/Videos/OneLoneCoder_PGE_Rectangles.cpp

Dylan, F. (2020. október 1). *Dylan Falconer / ember-ecs*. Forrás: https://gitlab.com/Falconerd/ember-ecs

Robert, N. (2014). *Game Programming Patterns.* Genever Benning.

Robert, N. (2014). Game Programming Patterns. In N. Robert, *Game Programming Patterns* (old.: Game Loop). Genever Benning.

Sam, L., & SDL Community. (1998). *Simple DirectMedi Layer - Homepage*. Forrás: http://libsdl.org/

Viktor, K. (dátum nélk.). *onqtam/doctest: The fastest feature ritch C++ 11/14/17/20 signle header library*. Forrás: https://github.com/onqtam/doctest