[Bevezetés 3](#_Toc72951441)

[Felhasználói dokumentáció 4](#_Toc72951442)

[Funkciók 4](#_Toc72951443)

[Telepítés 4](#_Toc72951444)

[Fejlesztői dokumentáció 5](#_Toc72951445)

[Fejlesztői környezet megteremtése 5](#_Toc72951446)

[Alkalmazott modulok 5](#_Toc72951447)

[Base 5](#_Toc72951448)

[ECS 5](#_Toc72951449)

[Interfész 6](#_Toc72951450)

[Adatstruktúrák 7](#_Toc72951451)

[Függvények 8](#_Toc72951452)

[Implementálás 9](#_Toc72951453)

[Void mutató 9](#_Toc72951454)

[ECS\_EntityStore 10](#_Toc72951455)

[DataStructures 14](#_Toc72951456)

[DataStructures\_UnorderedArray 14](#_Toc72951457)

[Interfész 14](#_Toc72951458)

[Implementáció 15](#_Toc72951459)

[Camera 16](#_Toc72951460)

[Interfész 16](#_Toc72951461)

[Implementáció 17](#_Toc72951462)

[Collision 18](#_Toc72951463)

[Interfész 18](#_Toc72951464)

[Pathfinding 19](#_Toc72951465)

[A\* algoritmusról 19](#_Toc72951466)

[Az algoritmus működése 20](#_Toc72951467)

[TextureManager 20](#_Toc72951468)

[Demó alkalmazás moduljai 20](#_Toc72951469)

[World 20](#_Toc72951470)

[EntityActions 20](#_Toc72951471)

[World 21](#_Toc72951472)

[View 21](#_Toc72951473)

[Tesztelés 21](#_Toc72951474)

[Irodalomjegyzék 22](#_Toc72951475)

[Mellékletek 23](#_Toc72951476)

# Bevezetés

Szakdolgozatom keretében C programozási nyelvben készítettem el egy játékot, illetve az ehhez szükséges keretrendszert.

Az erőforrásaim túlnyomó részét az utóbbiba fektettem, így ebben az olvasmányban a nagyobb hangsúly az említett keretrendszeren lesz. Szakdolgozatom így két fő részre bontható: a megvalósított modulokra, amikre a játék épül, illetve a játékra.

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. Ezek a modulok többnyire egymásra épülnek.

Ahogy említettem, készült továbbá egy egyszerű, felülnézetes menekülős játék is, ezen keresztül fogom megmutatni a keretrendszerem feature-jeit.

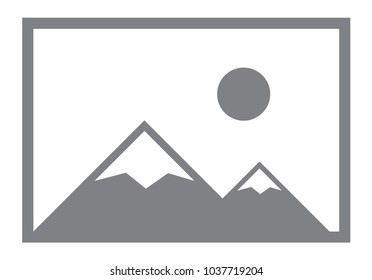
Megemlíteném, hogy sok triviálisnak tűnő feladatom volt, ami nem is olyan egyértelmű. Megemlítenék két példát. Az első, a monitor koordináta rendszere, és a világ logikájának koordinátája között valamilyen kapcsolat megteremtése, konvertálása. A másik pedig az entitások textúráinak helyes sorrendben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

Érdekelt, 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.

# Felhasználói dokumentáció

A játék egyszerű. A játékos egy piros-gúnyás embert irányít, és a célja, hogy ne érje őt utol a zöld szörny. Ha utoléri, egy ablak ugrik fel, és jelzi a játékosnak, hogy a játék véget ért.

A programot azoknak javaslom, akik szeretnek számítógépen játszani, továbbá a keretrendszert pedig azoknak, akik C programozási nyelvben szeretnének játékot fejleszteni. Nekik azonban bizonyára a fejlesztői dokumentáció lesz a hasznosabb.

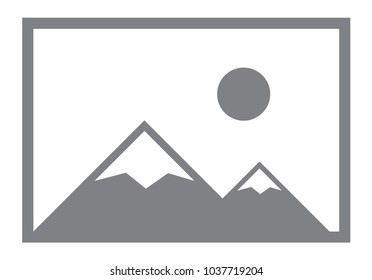


1. ábra: Képernyőkép a főhősről, és a szörnyről

## 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 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.



2. ábra: Játékos megakad egy fán, s a vizen

## Telepítés

TODO

# Fejlesztői dokumentáció

Ebben a fejezetben fejlesztőknek fontos információk találhatóak meg.

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.

## Fejlesztői környezet megteremtése

A szakdolgozatom forráskódja a <https://github.com/vapandris/Archipelago> GitHub oldalról érhető el.

## Alkalmazott modulok

### Base

Ez a modul egyszerű adatstruktúrákat tartalmaz, illetve typedef-eket tetszetősebb kód érdekében. Csak header fileokat tartalmaz.

A képen szöveg, képernyőkép, nyugta látható

Automatikusan generált leírás

### ECS

Az ECS egy programozási minta, amit többnyire játékprogramoknál alkalmaznak, viszont egyéb szoftverekben is használható, ahol az ECS nyújtotta előnyöket ki lehet használni.

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 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.

A képen asztal látható

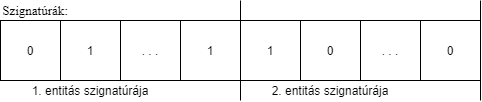
Automatikusan generált leírás

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

Az 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.

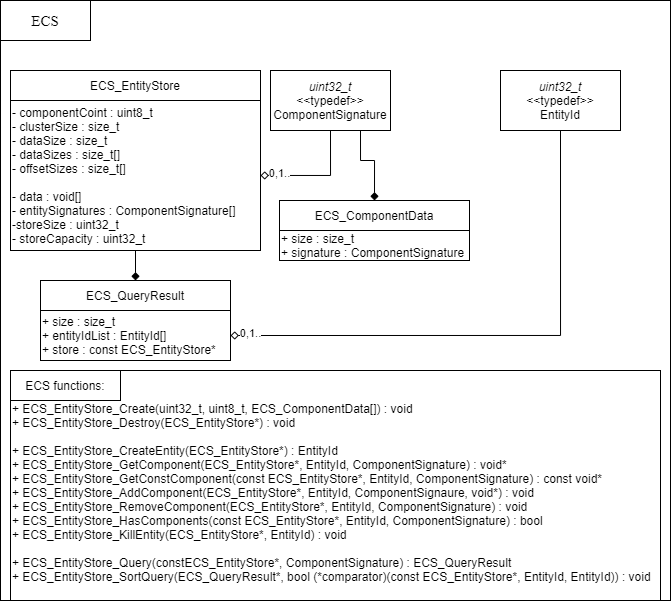


4. ábra: ECS szignatúrák

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.

#### Interfész

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



5. 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.

##### Adatstruktúrák

###### ComponentSignature

A képen asztal látható

Automatikusan generált leírásMinden 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 szignatúráját, ha binárisan össze vagyoljuk, akkor az az érték mind a kettő komponensre hivatkozik.

6. ábra: Szignatúra szemléltetése, össze-vagyolása

###### 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, így nem érdemes ezeket elmenteni hosszú távra.

###### ECS\_EntityStore

Rajta keresztül érhetjük el az Entity-Component-Systemet. Többet róla az implementáció részben.

###### ECS\_QueryResult

Amikor az ECS-en szűrni szeretnék valamilyen szempont szerint, akkor egy ilyen ECS\_QueryResult típusú változót kapunk vissza.

size: Megmondja, 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 pointer, amire a szűrést végrehajtottuk.

###### ECS\_ComponentData

Erre az ECS\_EntityStore\_Create (…) függvényben fogjuk használni. 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.

###### 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 hasonlítsunk össze.

##### Függvények

|  |  |
| --- | --- |
| Definíció | Leírás |
| ECS\_EntityStore\* ECS\_EntityStore\_Create  (uint32\_t, uint8\_t, ECS\_ComponentData…) | Visszaad egy megadott kapacitású ECS-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-t. |
| 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 az ECS mérete 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. |
| 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 módosulhatnak korábban elmentett EntityId-k, így azt nem tanácsos csinálni. |
| 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-el 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 |

#### Implementálás

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 függvényeket is ismertetni fogom.

##### Void mutató

Mindenekelőtt a void\*-t ismertetem, mert ez nem feltétlenül ismert mindenki számára.

A többi típusos mutatóval ellentétben, void mutatókhoz nem asszociálunk típust. 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 nagyon nehéz lesz használni.

Továbbá, ha érdekel minket egy void\* mutatott értéke, akkor azt először dereferálni kell, hogy tudja a program, a mutatott helyről mennyi memória kell nekünk.

Ha void\*-ot tömbként használjuk, akkor nem tudjuk a szögleteszárójellel lekérdezni a kívánt indexű elemet, hiszen nem tudjuk egyelem méretét, hogy tudjuk, hogy a memóriában mennyit kell arrébb lépni a kezdőponttól, hanem (vitathatóan) a legkényelmesebb megoldás, ha 1 bit-es char\*-ré kasztoljuk, majd a + operátorral annyit lépünk, amennyit szeretnénk.

##### 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.

###### 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.

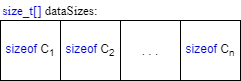
size\_t clusterSize: Mennyi memóriát foglal el egy komponens-klaszter. Ez azonos az összes komponens méretének az összegével.

Ennek segítségével el lehet képzelni, hogy a memóriában egy elemhez egy ilyen clusterSize méretű részt foglalunk le, ami lefoglal az összes lehetséges komponensnek helyet.

A képen szöveg, képernyő, csatlakoztatott látható

Automatikusan generált leírás

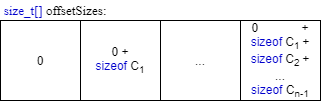
8. ábra: A klaszterek szemléltetése memóriában



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

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.

size\_t\* offsetSizes: Egy componentCount méretű 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.



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

ComponentSignature\* signatures: Egy componentCount méretű 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.

A képen asztal látható

Automatikusan generált leírás

10. ábra: A komponenseket tároló tömb void\* tömb ábrázolása

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

###### 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. |

###### 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.

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.  Ők 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 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.

ECS\_EntityStore\_Destroy:

Felszabadítja a paraméterül megadott ECS\_EntityStore self-et.

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.

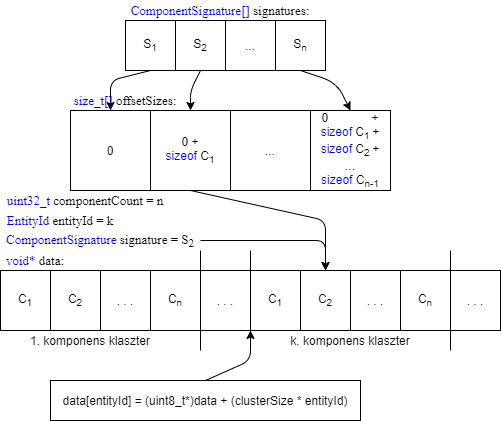
Az új entitás szignatúráját 0-ra állítjuk be, azaz egyik komponenssel sem rendelkezik. Mindezek után visszaadjuk az újonnan létrehozott entitás azonosítóját.

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, így ez az ábra próbálja ezt tovább magyarázni:



11. 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.

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.

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.

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.

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.

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.

ECS\_EntityStore\_SortQuery:

QuickSort-ot alkalmazva rendezi a megadott ECS\_QueryResult-ot a kapott comparator logikája szerint.

ECS\_QueryResult\_Destroy

Felszabadítja a paraméterül megadott ECS\_QueryResult self-et.

### DataStructures

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.

#### DataStructures\_UnorderedArray

##### Interfész

A képen szöveg látható

Automatikusan generált leírás

12. 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. |

##### Implementáció

###### 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.

###### DataStructures\_UnorderedArray\_Destroy

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

###### 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 lóg-e ki a tömbből.

Amennyiben még nem olvasta volna az Entity Component System részben a void mutató magyarázatát, és nem érti a működését, akkor erősen javaslom, hogy tekintse meg.

###### DataStructures\_UnorderedArray\_AddElem

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

Végül átmásoljuk a kapott void\* adatot a tömb végére.

###### 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.

###### DataStructures\_UnorderedArray\_GetSize

Egyszerűen csak lekérdezi a kapott DataStructures\_UnorderedArray tömb uint32\_t size adattagját.

### Camera

Ez a modul azért felel, hogy a monitor koordinátarendszere, illetve a játék logika koordinátarendszere között kapcsolatot teremtsen, adjon lehetőséget a kettő között 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*).

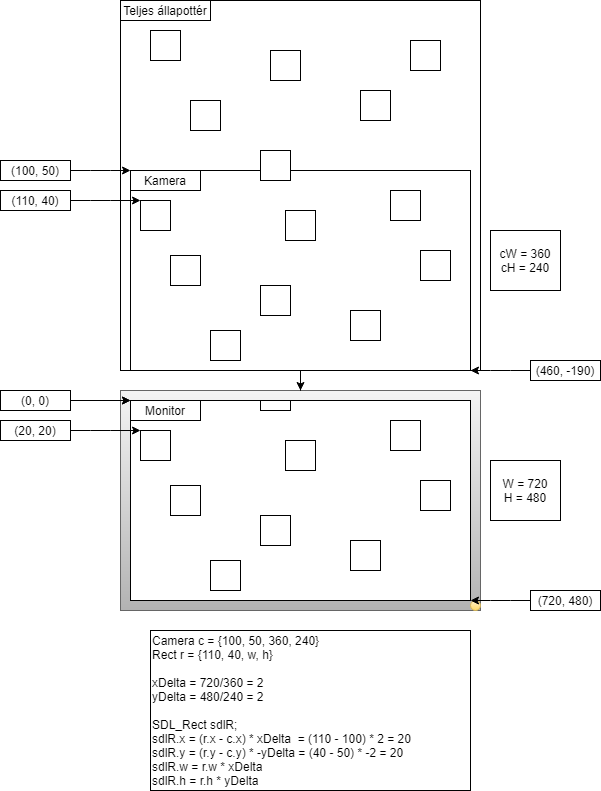
Természetesen nem szeretnénk, hogy majd a játékban az entitások elhelyezkedése ehhez legyen hozzá kötve, így ezért szükséges, hogy tudjuk, hogy a kamera jelenleg hol helyezkedik el, illetve, hogy tudjuk a kettő között átváltani.

#### 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.

#### Implementáció



13. 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.

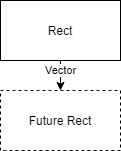
Az egyszerűség kedvéért választottam úgy a méreteket, hogy az xDelta, és yDelta megegyezzen, de ez nem kötelező, azonban akkor torzulás fog fellépni a téglalapokban.

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 kerekíjük 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.

### Collision

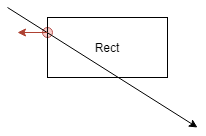
A modul felelősége, hogy eszközt adjon annak detektálására, hogy egy alakzat ütközik-e egy téglalappal. Az ütköző alak lehet pont, másik téglalap, irányvektor, vagy egy mozgó téglalap.



14. ábra: 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.

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.



15. ábra: Ütközés pontja, ideje, normálvektorja

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.

#### Interfész

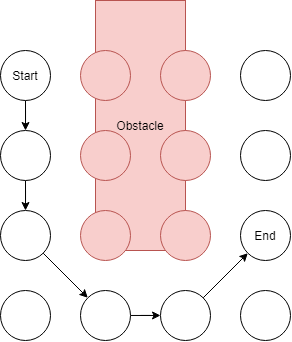
|  |  |
| --- | --- |
| bool  CollisionDetection\_PointRect  (const Point\*, const Rect\*) | Megvizsgálja, hogy a megadott Point ütközik-e a megadott Rect-tel.  Ha a Rect szélét érinti, már arra is jelezni kell. |
| bool  CollisionDetection\_RectRect  (const Rect\*, const Rect\*) | Megvizsgálja, hogy 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. |

### Pathfinding

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.

#### A\* algoritmusról

Az útkereséshez az A\* algoritmust fogjuk használni.



16. 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 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.

### TextureManager

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.

## Demó alkalmazás moduljai

### World

#### 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.

##### Interfész

|  |  |
| --- | --- |
| void  orld\_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. |

#### World

### View

## Tesztelés

# Irodalomjegyzék

# Mellékletek