

Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

SWARM szimulátor

Szoftverarchitektúrák házi feladat

Konzulens

Dr. Blázovics László

BUDAPEST, 2021

Tartalomjegyzék

[1 Az alkalmazás célja, funkciói és környezete 3](#_Toc88322889)

[1.1 Feladatkiírás 3](#_Toc88322890)

[1.2 Az alkalmazás funkciói 3](#_Toc88322891)

[1.3 A szoftver környezete 3](#_Toc88322892)

[2 Megvalósítás 4](#_Toc88322893)

[2.1 Performancia optimalizálás 4](#_Toc88322894)

[2.1.1 Look és Compute fázis 4](#_Toc88322895)

[2.1.2 Move fázis 4](#_Toc88322896)

[2.2 Implementált viselkedési formák 4](#_Toc88322897)

[3 Az alkalmazás telepítése, használata 6](#_Toc88322898)

[3.1 Az alkalmazás funkciói 6](#_Toc88322899)

[3.2 Példa bemenet 6](#_Toc88322900)

[4 Összefoglaló 7](#_Toc88322901)

[Felhasznált eszközök 8](#_Toc88322902)

[Irodalomjegyzék 9](#_Toc88322903)

[Függelék 10](#_Toc88322904)

# Az alkalmazás célja, funkciói és környezete

## Feladatkiírás

A feladat célja egy diszkrét idejű, diszkrét, véges állapotterű szimulációs környezet megvalósítása, amely akár 1000 egyed egyidejű viselkedését képes szimulálni.

## Az alkalmazás funkciói

A szimuláció elindítását a felhasználó parancssorból végezheti el, amely során meg kell adnia egy előre definiált leíró nyelven (továbbiakban: script) készült fájlt, amely tartalmazza az állapottér inicializációját, az egyes egyedekhez tartozó számítási szabályokat, illetve egyéb, a szimulációval kapcsolatos konstansok beállításait.

A szimuláció megállítását a szimuláció futása közben a megfelelő gomb lenyomásával történik meg, amely lehetőséget ad a betekintésre, illetve

A követelmények pontos definíciói elérhetőek a *Követelményspecifikáció* dokumentum *Követelmények* fejezetében.

## A szoftver környezete

Mivel a cél egy olyan alkalmazás fejlesztése volt, amely akár nagy egyedszám mellett is valós idejű szimulációra képes, ezért - az ilyen jellegű problémákhoz legjobban illeszkedő - C++20 nyelvet választottuk.

A nyelv által platformfüggetlen környezetet kapunk, amely által az alkalmazás akár egy beágyazott rendszer részeként is üzemelhet, illetve némi módosítással (pl.: C++/CLI wrapper) beépíthető olyan környezetbe, amelyben a fejlesztés hatékonyabban történhet (pl.: C#, Python).

# Megvalósítás

Az ún. Look-Compute-Move paradigmának megfelelően terveztük meg a rendszert úgy, hogy a valós idejű működés mellett arra is koncentráltunk, hogy a jövőben történő kibővítése a rendszernek (azaz a lehetséges szimulációk terének bővítése) rendkívül egyszerű legyen.

## Architektúra

Az architektúrát három rétegű alkalmazásként készítettük el. Az alkalmazás rétegei az alábbiak:

* Adat beolvasási réteg (Data Acquisition Layer)
* Üzleti logika réteg (Business Logic Layer)
* Felhasználói felület (User Interface Layer)

### Adat beolvasási réteg

**Célja**: Az inicializáló script beolvasása, illetve az üzleti logika számára megfelelő formátumra hozása.

Mi egy saját, egyszerű leíró nyelvet konstruáltunk a beolvasáshoz. Ez igény szerint bővíthető egyéb formátumokkal (pl.: JSON, XML).

**Megjelenés a kódban:**

### Üzleti logika réteg

**Célja:** A szimuláció inicializálása a struktúrált adatok beolvasása után, illetve a szimuláció futtatása.

**Megjelenés a kódban:**

### Felhasználói felület

**Célja:** A szimuláció eredményének ismertetése a felhasználóval.

Esetünkben a felhasználói felület a parancssorból adott eseményeket kezeli le, illetve egy fájlba logolja a szimuláció eredményét.

**Megjelenés a kódban:**

## Performancia optimalizálás

Az alkalmazásunk esetében a performancia kitüntetett szerepű. Emiatt célszerű a lokalitás elvét betartva, minél kevesebb cache-hibát generálva futtatni egy-egy szimulációt. Azt egyből beláthatjuk, hogy az egyedek listáján végig kell iterálnunk, ezáltal két lehetőséget kapunk. Az első lehetőség három ciklusunk van, amelyek megfelelnek a szimuláció fázisainak. A második esetben egy nagy ciklusunk van, amely minden iterációjában egy adott egyeddel teszünk egy fantom lépést, amelyet még nem mentünk el, majd valamilyen stratégia szerint updatelünk miután mindent kiszámoltunk. Előbbi a lokalitás elvét jobban betartja, így gyorsabb futáshoz vezet. A fázisok tervezésének motivációi az alábbiak szerint történtek.

### Look és Compute fázis

A Look, illetve Compute lépések teljesen párhuzamosíthatóak: az előbbi esetben az egyedek percepcióját frissítjük a jelenlegi világnak megfelelően, amely egy olvasást jelent, amely nem okozhatja a kiolvasandó adat bárminemű korruptálódását, utóbbi esetben pedig a számítások csak és kizárólag az egyedek jelenlegi állapotától, illetve saját (már frissített) percepciójuktól függ.

### Move fázis

A Move fázisban döntenünk kell a sorrendiségről. Itt az egyedüli kitételünk a determinisztikus futás volt (pontosabban adott randomszámgenerátor seed mellett), emiatt pedig a Move fázis ilyen jellegű párhuzamosítása nem lehetséges.

## Implementált viselkedési formák

*NotMovingBehaviour*

Az ilyen viselkedés alapján döntést hozó objektumok nem mozognak a pályán, így mint akadályt hoznak létre a szimulációban.

*RandomMoveBehaviour*

Az egyed viselkedése teljesen sztochasztikus, egyenletes eloszlás alapján választ a lehetséges lépések közül.

*SequentialMoveBehaviour*

Az egyed a gráf csúcsainak számozásának megfelelően halad végig, mindig a legnagyob értékű szomszédhoz. Ha olyan csúcsra lép, amely indexénél nincs nagyobb, akkor egyenletes eloszlás szerint véletlenszerűen választ a lehetséges lépések közül.

*MoveFromEntityBehaviour*

Az egyed egy helyben marad, amíg közvetlen környezetébe nem kerül legalább egy másik egyed. Ekkor ellép onnan egyenletes eloszlás szerint véletlenszerűen lép.

# Az alkalmazás telepítése, használata

TODO

## Az alkalmazás funkciói

TODO

## Példa bemenet

TODO

# Összefoglaló

TODO

Felhasznált eszközök

Irodalomjegyzék

Függelék