Programozói dokumentáció

- változó és függvénynevek:
  - camelCase
  - Ha komponenshez tartoznak (csak függvények): *ComponentName_functionName*

A játék alap struktúrája

Entity Component System ^

A program ECS dizájn minta alapján készül.
Az ECS 3 alapvető eleme:

1. **entity**: az entity-k nem struktúrák, igazából csak elméleti elemek. Egy egész képvisel egy entity-t
2. **component**: a játék adatainak tárolására való. Az, hogy melyik entity-hez tartoznak, az ID-jukból derül ki

   ```
   typedef struct ComponentName{
       // ENTITY_ID: Annak az entity-nek az ID-je amihez a komponens tartozik
       int ENTITY_ID;

       ... // További adatok, melyek ehhez a komponenshez tartoznak
   } ComponentName;
   ```

3. **system**: az egyes komponensekhez tartozó logika. A komponenstől külön modulban van.

Elnevezések:

- **File nevek**: (Amennyiben X a komponens neve)
  - components: X.c, X.h
  - systems: XSystems.c, XSystems.h
- **Függvény nevek**: (Amennyiben X a komponens neve)
  - X_függvényNév(...)

Game: fő struktúra ^

az alábbi kódrészlet nem tartalmazza az egész *Game* struktúrát, csak a fontosabb és összetett elemeit.

```
typedef struct Game {
    Layouts* layouts;

    Tilemap tilemap;
    GameTime time;
} Game;
```

- **layouts**: Egy lista, mely tartalmazza a játékhoz tartozó layout-okat.
- **tilemap**: A játék textúráit tartalmazó tilemap
- **time**: A játék időtől függő logikáiban segít. Ilyen például a PhysicsBodySystems, mert itt a fizikai képletekben szükséges tudni a két update között eltelt időt: Δt

---

Layout: különböző map-ek és képernyők ^

Egy-egy map-hez tartozó komponensek összefogása a feladata. A főmenü is egy Layout

```
typedef struct Layout {
    ComponentLists* components;

    Layer* layers;
    Vec2 camera;
} Layout;
```

- **components**: A layout-hoz tartozó entity-k komponenseinek tárolására való
- **layers**: A renderelés rétegeit határozza meg, valamint hogy az egyes entity-k mennyire legyenek transformálva a kamera által (parallax). Egy layer nem tudja, hogy melyik entity-k tartoznak hozzá, csak az entity tudja, hogy melyik layer-en van.
- **camera**: A vizuális komponensek renderelés előtti transformációjához használt vektor

---

ComponentLists: komponensek tárolója ^

```
typedef struct ComponentLists {
    int total_xComponents;  // ide tartozó x komponensek száma
    X* xComponents;         // x komponensek listája
    int total_yComponents;  // ide tartozó y komponensek száma
    Y* yComponents;         // y komponensek listája
    ...
} ComponentLists;
```

Magyarázat a komplikáltabb folyamatokhoz

- Mentés és betöltés

Mentés és betöltés^

Mentés:

A mentés során a játék ComponentLists-jein belüli összes adat kiíródik a jelenlegi játékoshoz tartozó save mappába. Ehhez kell egy memóriaterület ahol jelen van az összes Layout összes komponense ömlesztve és egy "map" ami leírja, hogy az adat melyik szakasza melyik layout-hoz tartozik és milyen komponensből van. Ez a map később a file elejére kerül.
A map-hez használt struktúra: *SerialisationMapFragment*

```
typedef struct SerialisationMapFragment {
    char componentType[255];     // pl.: Position, Sprite, ...
    int total_components;        // az ilyen típusú komponensek száma
    size_t componentSize         // sizeof(ComponentTypeX)

    LayoutMap layoutMaps[64];    // layout-ok map-jei (később bővebben)
    // azért nem dinamikus tömböt használok, mert a LayoutMap kevés
    // helyet foglal, viszont sokkal bonyolultabb lenne lementeni
} SerialisationMapFragment;
```

A *SerialisationMapFragment* összegyűjti az egyes komponensekhez tartozó adatokat az összes layout-ról. Ez a struktúra még nem használható mentésre, mert itt a komponensek még egymástól függetlenül léteznek.

Ahhoz, hogy betöltéskor meg tudjuk mondani, hogy melyik komponens melyik layout-hoz tartozott, tárolnunk kell, hogy a komponens listán belül mely indexek között, mely layout-hoz tartozó komponensek vannak.
Ezt a feladatot a *LayoutMap* struktúra látja el.

```
typedef struct LayoutMap {
    int start;                   // a layout-hoz tartozó első komponens indexe
    int end;                     // a layout-hoz tartozó utolsó komponens indexe
} LayoutMap;
```

A *SerialisationMapFragment* és a benne tárolt *LayoutMap* struktúrák létrehozása a komponensek feladata lesz. Ez a művelet egyesével van implementálva minden komponens moduljában.

Amikor megvan minden komponens típusnak a hozzá tartozó *SerialisationMapFragment* és egy pointer egy memória területre ahol ömlesztve van az összes létrehozott komponens a saját típusából, akkor a további adatfeldolgozást átveszi az ECS.c modul.

Mentés után a save file felépítése:

```
"saveFile": {
    "numberOfSerialisationFragments": int,
    "serialisationFragments": [
        0: struct SerialisationFragment: {
            "componentType": char[255],
            "total_components": int,
            "componentSize": size_t
            "layoutMaps": [
                0: struct LayoutMap: [
                    "start": int,
                    "end": int,
                ]
                ... (64db)
            ]
        }
        1: struct SerialisationFragment,
        2: struct SerialisationFragment,
        ... (<numberOfSerialisationFragments>db)
    ],
    "components": [
        0: ComponentType0[]: [
            0: layout0: [
```

```
                0: struct ComponentType0,
                1: struct ComponentType0,
                ...
            ],
            1: layout1,
            2: layout2,
            ...
        ],
        1: componentType1,
        2: componentType2,
        ...
    ]
}
```