

Programozói dokumentáció

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

A játék alap struktúrája

- [Entity Component System](#)
- [Game: fő struktúra](#)
- [Layout: különböző map-ek és képernyők](#)
- [ComponentLists: komponensek tárolója](#)

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(...)

ECS.h függvényeinek használata

Az #1 ID-jú entity *position* komponensére mutató pointer megszerzése:

```
Position* pos = ECS_getComponent(POSITION, game->currentLayout, 1);
```

Egy layout *position* komponensein végig loop-olás:

```
for(int i = 0; i < ECS_getNumberOfComponents(POSITION, game->currentLayout); i++)  
    dummy(ECS_getNthComponent(POSITION, game->currentLayout, i));
```

Egy entity-hez tartozó összes komponens megszerzése:

```
void** comps = ECS_getEntity(*game->currentLayout, 1);

Position* pos = (Position*)comps[POSITION];
Sprite* sprite = (Sprite*)comps[SPRITE];
...
free(comps) // !!!
```

Game: fő struktúra [^](#)

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

```
typedef struct Game {
    char playerName[255]; // name of the current player
    bool isRunning;

    Layouts* layouts; // list of layouts (more on layouts later)

    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 [^](#)

Az a feladata, hogy adatot tároljon arról, hogy a komponens típusok szerinti komponens listákon belül melyek tartoznak egy layout-ba (egy map-re). A játék menüje is egy layout.

```
typedef struct Layout {
    char LAYOUT_NAME[255];
    Vec2 camera;
    Layer* layers;

    LayoutMap* componentMaps; // list of LayoutMaps
    void** componentLists; // pointer to the game's componentLists
    ...
} Layout;
```

- **LAYOUT_NAME:** A layout neve. Ennek a változónak a layoutok azonosításában van szerepe, például ha layout-ot akarunk váltani, azt a LAYOUT_NAME alapján tehetjük meg.
- **camera:** A vizuális komponensek renderelés előtti transformációjához használt vektor
- **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.

- **componentMaps:** Arról tárol adatot, hogy a game struktúráján belüli komponens listákon belül melyek tartoznak ehhez a layout-hoz.

```
struct LayoutMap {  
    int start;        // index of the first element in it's parent layout  
    int end;          // index of the last element in it's parent layout  
}
```

- **componentLists:** Egy pointer a játék komponens listáihoz. Ennek csak az a létjogosultsága, hogy ha egy függvénynek hozzá kell férnie egy layout-on belüli komponensekhez, akkor elég legyen paraméterként egy layout-ot adni neki.

Magyarázat a komplikáltabb folyamatokhoz

- [Mentés és betöltés](#)
- [Interaction System](#)

Mentés és betöltés[^]

Mentés:

A mentés során a játék komponens listáiból kiíródnak az adatok egy file-ba. Ehhez kellenek a különböző komponens típusok listái 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 {  
    int componentType;        // id of the component type  
    int total_components;     // az ilyen típusú komponensek száma  
    size_t componentSize      // sizeof(ComponentTypeX)  
  
    LayoutMap* layoutMaps;    // layout-ok map-jei (később bővebben)  
} 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;
```

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 jön a következő lépés: az adatok file-ba kiírása. Az hogy az adatok melyik file-ba kerülnek az azon múlik, hogy

melyik játékos állását akarjuk elmenteni, ezért a file neve is: <játékosNév>.data. Ha a játék kezdeti helyzetén változtattunk és ezt szeretnénk elmenteni, akkor az *original.data* file-ba kell írunk.

Mentés után a save file felépítése: (az alábbi kód csak magyarázat, valójában bináris file-ban lesznek tárolva az adatok)

```
"saveFile": {
  "numberOfComponentTypes": int, // same as number of serialisationFragments
  "numberOfLayouts": 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,
    ...
  ]
}
```

betöltés

A betöltés során a program beolvassa a kívánt mentés file-ból az egyes komponensekhez tartozó adatokat és felépíti a layout-ok struktúráját.

Az első lépés beolvasni a file elején lévő két egészet, melyekből kiderül, hogy mentés előtt hány komponens típus volt és hogy hány layout volt a játékban.

Ha tudjuk ezt a két számot akkor be tudjuk olvasni a szintén a file elején lévő *serialisationMapFragment*-eket. Egyesével kell őket olvasni, mert nem konstans a méretük. Az hogy hány fragment-et kell beolvasni, az a beolvasott komponens típus mennyiségéből derül ki, az meg hogy hány *layoutMap* tartozik egy *serialisationMapFragment*-be az meg a layout-ok mennyiségéből.

Ha megvannak a *serialisationMapFragment*-ek akkor ezek alapján be tudjuk olvasni a file maradék részéből a komponenseket és fel tudjuk építeni a layout struktúrát.

Néhány probléma a betöltés kapcsán

- Figyelni kell arra, hogy a beolvasott komponensek mérete megegyezzen a jelenlegi komponens méretekkel. Ez akkor tud hibás lenni, ha a mentés óta megváltozott az egyik komponens struktúrája.
- Néhány komponens adatait felül kell írni mentés után. Ilyenek lehetnek azok a komponensek amikben pointer van egy másik adathalmazra, például egy tilemap-re, vagy azok a komponensek amik időpillanatot tárolnak magukban (pl.: Animation)

Interaction System [^]_▬

Debug

- [Memory leaks](#)

Memory leaks [^]_▬

A memóriaszivárgások debug-olásához a `crtdbg.h` modult használom.