Bolyai Farkas Elméleti Líceum

Marosvásárhely

SFML Jump 'N' Run Game

"Another Way"

Készítette:

Kéri Csaba & Siko Szabolcs

XII.G Matematika-informatika osztály

Felkészítő tanár: Jakab Irma-Tünde

2021

1. Témaindoklás

Dolgozatunk témája a 2D-s platformjátékok, másnéven Jump 'n' Run vagy platformer játékok világa. Az SFML Jump 'n' Run Game címnek a lényege az érdeklődés felkeltése, és tulajdonképpen magába foglalja a témát és a grafikus könyvtárat is, amelyet használtunk a megírása során. A platformjáték lényege az, hogy a játékos által irányított karakternek különböző elemeken keresztül kell ugrálnia, és/vagy különböző akadályokat kell átugrania.

Mielőtt nekifogtunk a projekt elkészítésének, érdeklődtünk arról, hogy melyikünk mit szeretne csinálni. Mindkettőnk terve az volt, hogy valami izgalmas játékot hozzunk létre. Innen ered az ötletünk, hogy közösen vágjunk bele a munkába.

A fő indok, amiért ezt a témát választottuk az az, hogy mélyebb szinten foglalkozzunk az objektum-orientált programozással, és ez által egy összetett programnak legyünk készítői. Mindig is kíváncsiak voltunk arra, hogy egy játék megírása mögött mennyi nehézség, kitartó munka, belefektetett idő és tudás rejtőzik. Emellett, arra is hangsúlyt szerettünk volna fektetni, hogy egy magasabb szintű kódot írjunk.

2. Rendszer követelmények

2.1. Hardver igény

- A program futtatásához ajánlott 256 MB méretű memória, maga a program maximum 117 MB memóriát igényel egyidőben.

- A játék igényel 31 MB szabad lemezterületet.

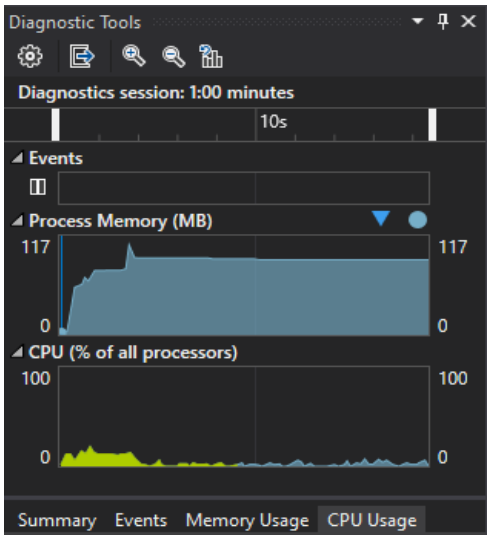
- Bármely 32 vagy 64 bites processzoron futtatható, nincs különösen magas igénye. Tesztelés során "Intel I7-7700K", "Intel I5-7500" valamint "AMD Ryzen 5 4600H" típusú processzorokat használtunk. Ezeken kiválóan működött.

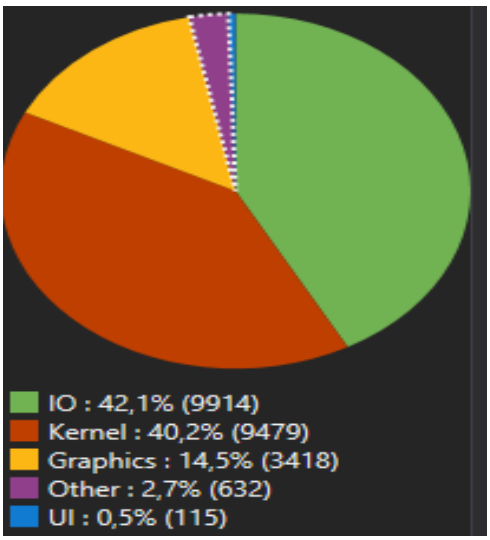
2.2 Szoftver igény

- A szoftver futtatásához Windows operációs rendszer szükséges.

- Ajánlott Windows 10-en futtatni.

A bal oldali diagram a memóriaigényt mutatja, a jobb oldali pedig a processzor használatát különböző területeken a program futtatása közben:





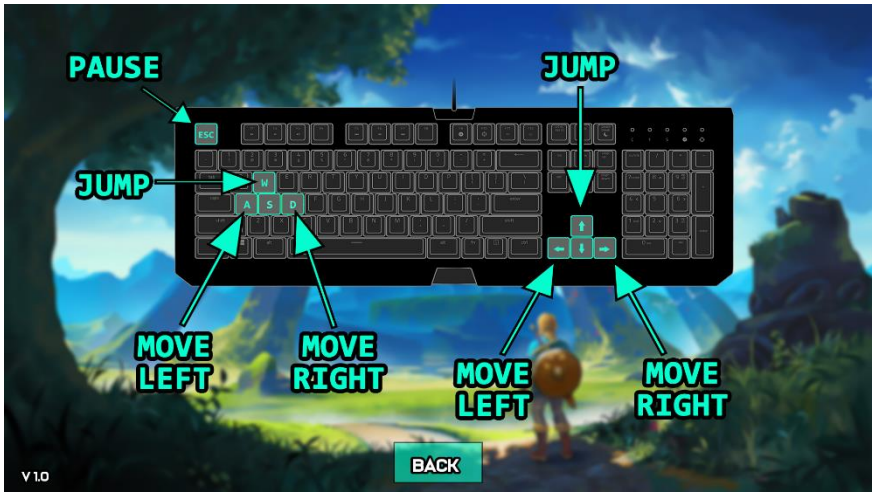
3. Felhasználói kézikönyv

A játékot a mappájában lévő "Another Way.exe" fájllal lehet elindítani. Indításkor a játék főmenüjébe lépünk be, ahol elindul a háttérzenéje, és 4 további lehetőségünk van: "PLAY", "CONTROLS", "CREDITS" és "QUIT".

A "QUIT" gomb megnyomásával kilépünk a játékból. A "CREDITS", "CONTROLS" és "PLAY" gombok pedig egy új oldalra dobnak át. A menü további részében, bármelyik oldalon megjelenő "BACK" gomb szerepe, hogy visszadobjon az azelőtti oldalra. De ezt elérhetjük az "ESC" billentyűvel is.

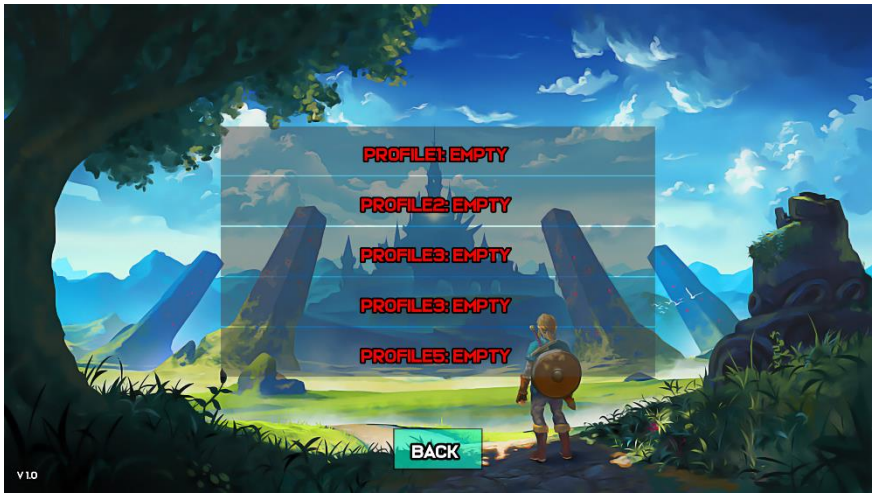
- A "CREDITS" oldalon szerepel a készítők neve, a használt programozói nyelv és könyvtár és a készítés dátumának évszáma.

- A "CONTROLS" oldal arról tájékoztat, hogy melyik billentyűkkel történik a karakter irányítása, illetve melyikkel lehet behozni játékon belül a menüt.



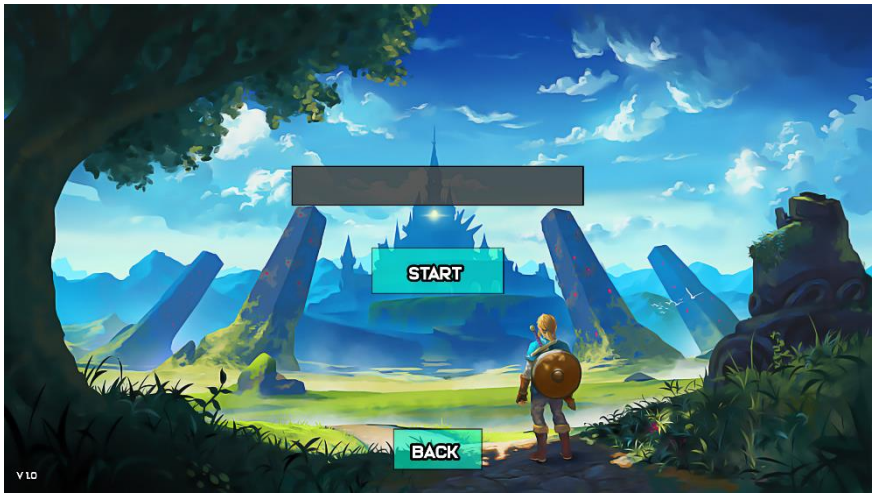
- A "PLAY" oldalon két lehetőség van: "NEW GAME" és "LOAD GAME".

- A "NEW GAME" és "LOAD GAME" oldalon megjelenik 5 profil, amelyekre lehet menteni, és természetesen onnan folytatni a játékot a későbbiekben.

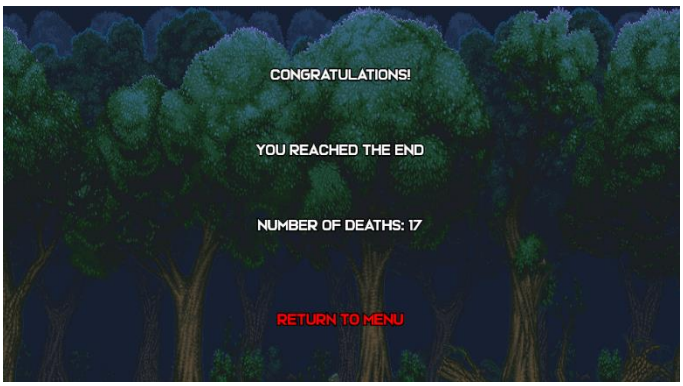


- Ha a "LOAD GAME" oldalon választunk ki egy profilt, akkor elindul a játék, betöltve az azon a profilon levő mentést. Nyilvánvalóan csak azt lehet kiválasztani, amelyiken van mentés.

- Ha a "NEW GAME" oldalon választunk ki egy profilt, akkor átdob egy következő oldalra. Ezen az oldalon először meg kell adni az új mentésnek a nevét(kötelező), majd rákattintva a "START" gombra, vagy az Enter billentyű lenyomásával, elindul a játék. Ha meglévő mentésre indítunk új játékot, akkor felülírja az azelőtti mentést. A játék kezdetekor az adott profilra elmenti a beírt nevet, az aktuális dátumot és időpontot és a karakter kezdési pontját.

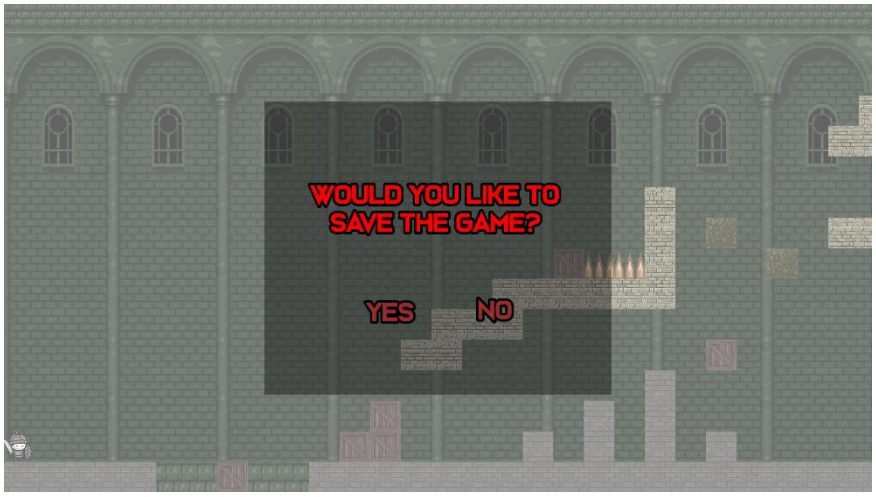


A játékban jelenleg 2 különböző pálya van, amelyeken különböző háttérzene szól. A halálnak és az ugrásnak is eltérő hangja van. A karakterünkkel jobbra kell haladni. Mindkét pályán el vannak helyezve csapdák és ellenségek, amelyekhez hozzáérve a játékos meghal, és visszakerül az addig megszerzett legutolsó "checkpoint"-hoz. Akkor is garantált a halál, ha leesünk az alattunk levő talajról. Az első pályát végigvinni természetesen könnyebb, mint a másodikat. Az első pálya végén átkerülünk a második pályára. Ezt is teljesítve, sikerült végigjátszanunk a játékot, és egy gratuláció fogad minket.



A játékon belüli menüből("ESC" billentyűvel tudjuk behozni) vissza tudunk lépni a főmenübe("BACK TO MENU"), vagy akár ki is tudunk lépni teljesen a játékból("EXIT").

Mindkét lehetőség esetén egy kérdés fogad minket. Rá kell kattintani a "YES" vagy "NO" gombra, attól függően, hogy le szeretnénk-e menteni az akkori álláspontunkat, vagy nem. Arra a profilra történik a mentés, amelyiket kiválasztottuk a játékba lépéskor. A "RESUME" gomb vagy "ESC" billentyű segítségével tudjuk folytatni a játékot.



Ajánlott kihasználni a mentés lehetőségét, mert egy-egy helyen adódhatnak kellemetlen pillanatok, amelyeket jól elhelyezett csapdák vagy akár az ellenségek okozhatják. Sok sikert és jó szórakozást kívánunk!

4. Programozói kézikönyv

Az "Another Way"-nek elnevezett játékot C++ programozási nyelvben készítettük, amely egy általános célú, magas szintű nyelv, amely támogatja az objektum-orientált programozási paradigmát.

A program készítéséhez felhasználtuk a **„Simple and Fast Multimedia Library”** –t, röviden az **SFML**-t, ami egy többplatformos szoftverfejlesztő könyvtár, amelynek célja, hogy egyszerű alkalmazásokhoz programozási felületet biztosítson a számítógépek különféle multimédiás komponenseihez.

Ennek a könyvtárnak néhány objektumát használtuk fel. Ilyen például a **RenderWindow**, ami egy ablakot tud létrehozni, és ehhez van egy **draw**, **clear**, valamint egy **display** nevű függvénye kirajzoláshoz, törléshez és megjelenítéshez.

Még egy fontos adattípus, amelyet az SFML biztosít: a **Textures**. Ebbe a típusba képesek vagyunk beolvasni textúrákat, azonban ezt a típust nem lehet önmagában kirajzolni. Ehhez szükség van egy keretre. A **Sprite** típusnak pont ez a szerepe: ebbe kell a textúrákat betölteni, és akkor ki lehet őket rajzolni.

Azonban figyelni kell az adat tárolására, mivel a **Sprite** típus a textúrának a memóriacímét tárolja, többek között, vagyis ha a textúrának a tárhelyét felszabadítjuk, akkor a **Sprite**-ba betöltött mutató egy üres területre fog mutatni, így a kirajzolás nem fog működni. Vagyis végig szükségünk van a textúrákra, nem elég egyszer betölteni a **Sprite** típusba, majd felszabadítani a memóriát. Tehát külön kell figyelni, hogy a textúra az osztályon belül globális változó legyen, nem pedig a függvényben lokális, mivel az a függvény végeztével elveszíti értékét, és felszabadul a tárhely.

A programot úgy terveztük meg, hogy különböző képernyőfelbontású kijelzőkön is ugyanazt az élményt nyújtsa, vagyis a gombok mérete, elhelyezkedése és a textúrák méretei mind az adott képernyő méretétől függenek, nem statikus érték van megadva. Ezért a textúrák méreteit mindig az adott képernyő szerint növeli, vagy csökkenti a program. Ezt, a **Sprite** típus esetén, a **setScale()** függvényével lehet elérni.

```
float Texture_rate = this->Texture_size / 135.f;

enemy[i]->enemy_sprites.setScale(Texture_rate, Texture_rate);
```

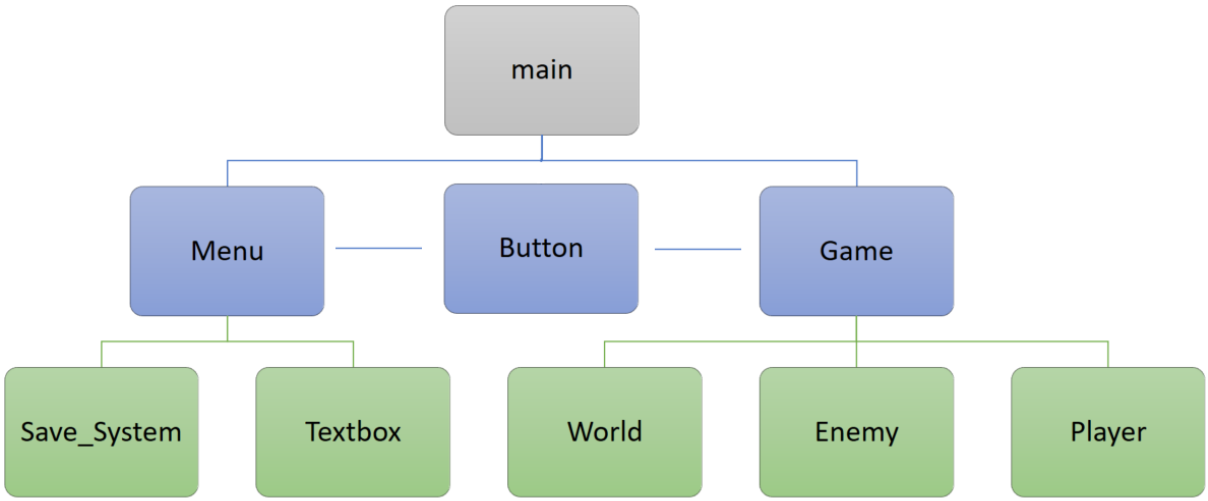
A program készítése során nagy figyelmet szenteltünk arra, hogy objektum-orientált legyen a szoftver, és fontosnak tartottuk a hatékony memóriatárolást, így több helyen dinamikus helyfoglalás történik, és nincsenek felesleges lokális változók, hanem az eredeti változó memóriacímét átadva használjuk azt. A program írásakor figyelembe vettük a láthatósági mezőket, és a függvényeket és változókat aszerint csoportosítottuk.

A program a következőképpen van objektumokra struktúrálva:

main:

- **Menu**
  - **Save_System**
  - **Textbox**
- **Game**
  - **World**
  - **Enemy**
  - **Player**

Valamint a **Menu** és **Game** osztály is használja a **Button** osztályt.



A **main(),** vagyis a fő függvényben hozzuk létre a **Menu** osztályt, és olvassuk be azt a betűtípust, amely végig jelen lesz szöveg-kiíratásoknál. Itt található még egy ciklus, amelyben fut a **Menu,** és itt lesz létrehozva a **Game** osztály is.

A **Button** osztály, nevéből adódóan, egy gombot hoz létre, és deklarálásakor paramétereket vár el, ezek segítségével inicializálja a konstruktor az objektumot.

```
Button::Button(float x, float y, float width, float height,
    std::string text, int charactersize, sf::Color textcolor, sf::Color texthovercolor, int text_outline_thickness,
    sf::Color text_outline_color, sf::Color fillcolor, sf::Color hovercolor, int keret_outline_thickness,
    sf::Color keret_outline_color, sf::Font* font, sf::RenderWindow* window)
```

Itt adhatjuk meg a gomb méretét, pozícióját, a szövegét, amit majd tartalmazni fog, színét és az ablaknak a memóriacímét, ahova kell majd kirajzolni.

Ennek az osztálynak kétfajta konstruktora van, amelyek különböző paramétereket várnak el. Azért van szükség erre, hogy két típusú gombot lehessen létrehozni. Az egyik fajta konstruktor több paramétert vár el, mint a másik, ezáltal különbözteti meg a rendszer a kettőt egymástól.

Az osztály még tartalmaz egy olyan függvényt, amely azt vizsgálja, hogy rákattintottak-e a gombra, és egy **Render()** függvényt, amely kirajzolja a gombot a meghíváskor.

Ezt az objektumot a **Menu**-ben és a **Game**-ben is használjuk a gombok létrehozásához, a különböző menükben.

A **Menu** objektum létrehozásával együtt jön létre az ablak is, amelynek memóriacímének átadásával a többi osztályból is elérjük majd a kirajzolásoknál.

```
switch (state)
{
case Menu::MainMenu_state:
    MainMenu();
    break;

case Menu::PlayMenu_state:
    PlayMenu();
    break;

case Menu::NewGameMenu_state:
    NewGameMenu();
    break;

case Menu::ProfileUpdate_state:
    Profile_Update();
    break;

case Menu::LoadGameMenu_state:
    LoadGameMenu();
    break;
```

A **Menu** osztálynak a függvényei a különböző almenü részeket töltik be. Ezek között a navigálást egy **enum** típusú változó segítségével oldottuk meg. A **Menu Run()** függvénye ennek a változónak az értékei szerint hívja meg a megfelelő függvényeket.

A **Menu**-ben deklarálva van egy **Save_System** típusú objektum. Ennek az osztálynak az a szerepe, hogy kiolvassa a fájlokból a mentett játékállásokat, és megjelenítse a menüben annak a címét és dátumát. Ha nincs mentés, amit kiolvashat, akkor pedig azt az információt adja át a menünek. Így, a mentett játékot betöltheti a menüben. Az üres mezőt nem lehet kiválasztani. Ha a menüben az új-játék vagy a játék betöltése lehetőséget választjuk, akkor a **main()** ciklusában létrehoz egy **Game** objektumot, amelynek paramétereken keresztül átadjuk az ablak címét, amelyet a **Menu** osztályból tudunk elérni, a betűtípus címét, valamint egy struktúrát, amely a mentési adatokat tárolja.

Ez a struktúra a **Menu**-ben deklarált **Save_System** típusú objektum egyik struktúrája, amely azt jelöli, hogy melyik profilt választotta ki a felhasználó, és ha az a profil nem üres, akkor azt a játék-állást tölti be a program.

```
Save_System::save_file_adatok save;
```

A játékon belül a kurzor nem látható, csak ha a menüt megnyitjuk, amelyet az "ESC" billentyű lenyomásával érünk el.

Ezt a **setMouseCursorVisible()** metódus meghívásával tudjuk elérni.

A **Game** osztály konstruktora létrehoz három másik objektumot az osztályon belül, és ezekre mutatók segítségével hivatkozunk: **World**, **Player**, **Enemy**. Ezen kívül még beállítja a megfelelő játékállást.

```
world = new World(window, save.map_index);

player = new Player(window, world->Texture_size,save);

enemy = new Enemy(window, world->Texture_size, world->IntToString(save.map_index),save.index);
```

A **World** osztály a világ betöltéséért felel. Az objektumnak paraméteren keresztül meg kell adni, hogy melyik pályát kell betöltenie. A Konstruktor meghívja a szükséges függvényeket a világ inicializálására.

```
struct Map_database
{
    int sorok_szama, oszlopok_szama;
    std::vector<std::vector<int>> map;

    int endpoint;

    void initMap();

}Map_database;
```

Kiolvassa egy szöveges állományból a pálya méreteit, végpontját, majd a méretnek megfelelően felépít egy (**std::vector**) mátrixot, amely a térképet tartalmazza.

Mivel a szöveges állományból karakterláncként olvassa ki a térképet, azt még egész számmá kell alakítani.

Ezután a pálya méretei függvényében létrehozunk egy **sf::Sprite** típusú mátrixot. Ezt az adattípust tudja kirajzolni az SFML. Ennek a mátrixnak az elemeinek beállítjuk a megfelelő textúrákat a térkép szerint, és a képernyő mérete szerint azok nagyságát, valamint elhelyezzük szintén a megfelelő helyekre. Ezután ezt megtesszük a csapdákkal is (**Trap**). Azoknak is beolvassuk a textúráit, majd annak függvényében, hogy melyik pálya, kiolvassuk a megfelelő szöveges dokumentumból, hogy hány darabot hozzunk létre, és annyi ilyen **Trap** típusú struktúrának foglalunk helyet dinamikusan. Erre azért van szükség, hogy mindig csak annyi memóriát igényeljen a program, amennyire az adott pályának szüksége van. Amikor pályaváltás történik, akkor majd ezt a tárhelyet felszabadítja, és az adott pályához tartozó csapdának foglal tárhelyet(nem többet – nem kevesebbet).

```
struct Trap
{
    sf::Sprite sprite;
    sf::Vector2i pos;
    sf::Vector2f position;
    int which_texture;
    bool hidden;
};
```

Ezek után a konstruktor még a pálya szerint beolvassa a háttérképet.

A **World** osztálynak a többi függvénye a világ kirajzolását végzi el csapdákkal, textúrákkal és háttérrel együtt. Egy függvény a játékos a csapdákkal való ütközését ellenőrzi, amihez szükség van egy **Player** típusú mutatóra paraméterként. A **World** destruktorja felszabadítja a tárhelyet a dinamikusan tárolt adatoktól, így nem történik memóriaszivárgás.

A **Player** osztály a játékossal foglalkozik, vagyis ez felel a játékos textúrájának **Sprite** típusba való betöltéséért, illetve a játékos helyzetének és adatainak tárolásáért. Itt található egy **Render()** nevű függvény is, amely kirajzolja a játékos **Sprite**-ját a képernyőre.

Az **Enemy** objektum konstruktora is inicializálja az osztályt, elvégzi az ellenségek textúráinak a beolvasását, majd ezután a megfelelő szöveges dokumentumból kiolvassa az ellenségek számát és adatait.

```
struct Enemies
{
    sf::Sprite enemy_sprites;
    sf::Vector2f position;
    sf::Vector2i pos1, pos2;
    int which_texture;
    bool move_right = true;
    float velocity;
};
```

Minden ellenség adatát letároljuk egy struktúrában, és annyi ilyen típusnak foglalunk tárhelyet, ahány ellenség található az adott pályán. Az adatok alapján elhelyezi ezeket a megfelelő helyekre, és a megfelelő textúrákat állítja be nekik. Ezeknek a struktúráknak is dinamikusan foglalunk tárhelyet. Pályaváltásnál pedig felszabadítjuk a tárhelyet, és csak az adott pálya szerint foglalunk tárhelyet.

Az osztályban található egy **Render()** nevű függvény, amelynek az a szerepe, hogy kirajzolja az ellenségeket. Futási idő hatékonysága szempontjából csak azokat az ellenségeket rajzolja ki, amelyek a játékos látókörén belül van, a távolabbi ellenségeket nem rajzolja ki, ezzel gyorsítva az algoritmust.

Ebben a függvényben meg van hívva egy másik függvény is, amely az ellenségek mozgásáért felel, vagyis, hogy a kezdő- és végpontjuk között mozogjanak. Ezt is csak akkor végzi el, ha egy játékos közelében van. A játékostól nagyon messze hiába mozgatja az ellenségeket, azt úgyse látja a játékos, és nincs kihatással rá, amíg közelebb nem ér. Az ellenségek mozgásának irányától függ, hogy melyik textúrát rajzolja ki. Ha jobbra megy az ellenség akkor a jobbra néző textúrát kell kirajzolnia, ha balra, akkor a balra tekintőt.

Miután a **Game** osztály konstruktora létrehozta ezeket az objektumokat, és azok konstruktorai pedig elvégezték az inicializáló

műveleteiket, még zenét, valamint az újraéledési pontok helyzetét is kiolvassuk az adott pályához.

A **main()** függvényben a **Game** osztály létrehozása után meghívjuk a **Run()** függvényét, amely mindaddig fut, amíg a felhasználó játékban van. Ha visszalép a főmenübe, akkor ez az osztály megsemmisül, és a Destruktor felszabadítja a tárhelyet.

A **game.Run()** függvény tartalmazza a játék fő ciklusát. A ciklus előtt még lekorlátozza a játék képfrissítési-rátáját 60 fps-re (frames per seconds).

A ciklusban három fő művelet van: **Event** (esemény) ellenőrzése, játékos mozgatása, valamint a kirajzolások.

Az **Event**-eket az **SFML_Events()** nevű függvény nézi. Pontosabban azt, hogy a játékos életben van-e; hogy melyik újraéledési pontot haladta meg; vagy hogy lenyomta-e az "ESC" billentyűt, ami szünetelteti a játékot, és behoz egy menüt a játékon belül. Ebből a menüből vissza lehet lépni a főmenübe vagy a Windowsba. A rendszer megkérdi, hogy a felhasználó leszeretné-e menteni a játékállást. Ha igent választ, akkor a megfelelő profil állományába beleírja a program a mentési adatokat. Ellenkező esetben nem.

A másik fontos művelet, amelyet minden futtatásnál elvégez a rendszer, az a játékos mozgása, amelyért a **Player_movement()** nevű függvény felel. Ebben a függvényben mindig megnézi a rendszer, hogy történt-e billentyűlenyomás. Annak függvényében állítja be az X koordináta szerinti sebességet pozitív vagy negatív értékre. Jobbra pozitív, balra negatív.

Ezután a **Player** objektum pozíciójához hozzáadjuk a sebességet, majd megnézzük, hogy ütközött-e. Az ütközés észleléséhez a **World** osztály **Map_database** struktúrájának **map** nevű mátrixát használjuk, amely tartalmazza a térképet. Ha a mátrix adott eleme nem 0, akkor áthatolhatatlan a "blokk". Ha a **Player** új pozíciója belelóg egy ilyenbe, akkor visszadobja a játékost a "blokk" széléhez úgy, hogy ne lógjon bele. A **Player** objektumnak van egy **player_mapindex** nevű, **sf::Vector2i** típusú változója, amely azt jelöli, hogy a játékos hol található a térképet jelző mátrixban. Ezek segítségével történik az ellenőrzés.

```
if (world->Map_database.map[i][j] > 0 and
    world->Map_Sprites[i][j].getGlobalBounds().intersects(player->Player_sprite.getGlobalBounds()))
```

Y koordináta szerint is hasonlóan történik az ellenőrzés, csak ahhoz még a függvény elején hozzá kell adni egy másik sebességet, amely a gravitációt jelöli. Ha a karakter a földön áll, akkor a gravitáció nem hat rá.

A térképen való mozgás szempontjából fontos az **sf::View** típusú változó mozgatása. Ez csak akkor fog mozdulni, ha a játékos elérte a képernyő közepét. Ezután pedig mozgás esetén a világ és a háttér is mozdul. A játékos végig a képernyőnek a közepén lesz.

```
if (player->Player_mapindex.x + 1 > world->MaxView.x / 2)
{
        view.move(player->Player_position.x - pos_x, 0);
        world->Move_background(player->Player_position.x - pos_x);
}
```

```
window->clear();

window->setView(view);

world->Render_background();

world->Render_map(player->Player_mapindex.x);

world->Render_trap(player);

enemy->Render(player, world->MaxView.x);

player->Render();

window->setView(window->getDefaultView());

window->draw(death_count);

window->display();
```

A harmadik fontos művelet, amelyet mindig elvégzünk, az a kirajzolások.

Le kell törölni a képernyő tartalmát, majd kirajzolni a hátteret, világot (térképet), csapdákat és a játékost. Ezeket a műveleteket a **World**, **Enemy** és **Player** osztályok **Render()** nevű függvényei végzik el. Itt csak meg kell hívni őket, majd ezek után egy **display()** függvényt, amely megjeleníti a kirajzolt elemeket.

Az **Enemy** és **World** osztályok textúráit ciklus segítségével olvassuk be. Deklarálunk egy karakterláncot, ami az elérési utat tartalmazza, és ehhez mindig hozzáfűzzük a ciklusvezérlő akkori értékét, úgy, hogy a ciklusvezérlőt átalakítjuk karakterlánccá, majd hozzáfűzzük a karakterlánc végéhez. Ezután még hozzáfűzzük a kiterjesztését a fájlnak, így be tudjuk sorba olvasni az összes textúrát, majd eltárolni egy, ilyen adat tárolására képes, tömbben.

```
for (int i = 0; i < number_of_traps; i++)
{
    std::string texture_names = "Traps/";
    texture_names += IntToString(i);
    texture_names += ".png";
    Traps_texture[i].loadFromFile(texture_names);
}
```

Ehhez még az kell, hogy a textúrákat szám szerint nevezzük el. Így a programban is hivatkozhatunk rá úgy, mint a nullás vagy hármas, mivel ezek így a beolvasás után a textúratömb index-éül fognak szolgálni.

Ha a játékos meghal, ami csapdával vagy ellenséggel való ütközés esetén történhet meg, akkor a legközelebbi újraéledési pontra állítjuk a helyzetét. Ha a játékos elér a pálya végére, akkor meghívjuk a **Switch_Map()** nevű függvényt, aminek a pályaváltás a feladata. Ez meghívja a többi objektum pályaváltó függvényeit. Ezek, az új pálya adatai szerint betöltik a megfelelő fájlokból az adatokat, hasonlóan, mint amikor az objektum létrejön.

A játék működésében fontos szerepe van a szöveges állományoknak, amelyek információkat tartalmaznak, melyek nélkül nem működhet a program. Példaként az ellenségeket tartalmazó szöveges állomány szerkezete:

- ellenségek száma, majd minden ellenséghez a következő adatok:
  - Kezdő pozíció X koordináta – mátrix szerint – intiger
  - Kezdő pozíció Y koordináta – mátrix szerint – intiger
  - Végső pozíció X koordináta – mátrix szerint – intiger
  - Végső pozíció Y koordináta – mátrix szerint – intiger
  - Textúrák száma – intiger
  - Mozgási sebesség – float típusú

Vagy a csapdákat tartalmazó szöveges állomány szerkezete:

- csapdák száma, majd minden csapdához a következő adatok:
  - Csapda X koordinátája – mátrix szerint – intiger típusú
  - Csapda Y koordinátája – mátrix szerint – intiger típusú
  - Textúra száma – intiger típusú
  - Láthatatlan vagy sem – boolean típusú

A játék fontos része a játékos halála: ehhez szükséges érzékelni a játékos **Sprite** típusának az ütközését csapda vagy ellenség **Sprite** típusával. Ezt a műveletet, egy beépített függvény segítségével néztük meg, ami úgy működik, hogy a **Sprite**-t téglalap alapon körbedobozolja, bekeretezi, és ha ez a két doboz bármely koordinátája fedi egymást, akkor történt ütközés.

A **main()** függvény ciklusa akkor áll meg, ha mind a **Menu** és **Game** osztály boolean típusú változója (amely a működést jelöli), hamis lesz, és akkor véget ér a program.

```
bool World::trap_collision(Player* player, Trap* trap)
{
    if (trap->sprite.getGlobalBounds().intersects(player->Player_sprite.getGlobalBounds()))return true;

    return false;
}
```

5. Fejlesztési lehetőségek

A programnak több fejlesztési lehetőséget is kigondoltunk, amelyekkel a későbbiekben kibővíthető a játék.

Több lehetőséget lehetne beépíteni a rendszerbe az ellenség mozgására. Mint például, hogy tudja követni a játékost, vagyis ne csak a megadott területen belül mozogjon. Ezt egy útkereső műveletsorral lehetne megvalósítani, példának okáért egy módosított **Dijkstra** algoritmussal. A "térkép" tárolása lehetővé tenne egy ilyen módosítást.

Ami még egy érdekes bővítés lenne, az a láthatatlan "blokkok" bevezetése adott pályarészekre, amelyek váratlan nehézségeket okozhatnának a játékos számára. Valamint gondoltunk még arra is, hogy úgynevezett "ál-blokkot" használjunk, aminek a lényege az lenne, hogy a játékos egy teljesen közönséges talajnak higgye, azonban abban a pillanatban, mikor ráugrik, keresztül esik rajta. Ez a fejlesztés nehezebbé, de ugyanakkor izgalmasabbá tenné a játékot. Emellett, meg lehetne valósítani azt is, hogy egyes szakaszokon mozogjanak a "blokkok", ami által változatossá válik ugyanaz a rész, akárhányszor is térünk oda vissza.

További lehetőség a pályák számának bővítése, a jelenlegi kettő mellé. Ezt egy "örök fejlesztésnek" neveztük el, ugyanis a program úgy lett kialakítva, hogy mindenféle kódírás nélkül is lehessen újabb pályákat hozzáadni. Egyetlen dolgot kell tenni, mégpedig meg kell változtatni(növelni) a bemeneti állományban levő pályák számát. Ezután új szöveges állományokban annyi pályát szerkeszthetünk, amennyit csak szeretnénk. Ezeknek a "térképeknek" a szerkezete egyszerű szöveges állományban(.txt) vannak tárolva, így akárhány pályával bővíthető a játék. A "térképek" között még a sorrendet is meg lehet változtatni a szöveges állomány végén szereplő számok felcserélésével.

Egy másik opció, hogy amikor létrehoz a felhasználó egy új játékmenetet, eldönthesse azt, hogy milyen nehézségi fokon szeretne játszani. Ettől függően lenne több/kevesebb ellenség és/vagy csapda, de akár befolyásolná azt is, hogy az ellenségek mennyire gyorsan mozogjanak. Nehéz fokozaton el tudnánk képzelni azt is, hogy lenne egy bizonyos korlátozás a halálok számán. Például, ha elérné ezt a korlátot a játékos, akkor az adott pályát az elejéről kellene kezdje.

Egy kisebb változtatás akár az is lehetne, hogy amikor a játékos elmenti az aktuális játékmenetét és kilép, akkor a rendszer a legutoljára elért "checkpoint"-ot mentené le. Ez is egy nehezítési opció, mert jelenleg ha minden apró haladás után a felhasználó ment egyet, akkor sokkal hamarabb fejezi be a játékot, mivel lerövidíti a pályát azzal, hogy nem az újraéledési ponttól kezdi újra a halál után (ha persze minden halál után kilép mentés nélkül és visszalép a korábbi mentésre).

6. Szakirodalom

https://www.sfml-dev.org/learn.php

https://en.sfml-dev.org/forums

https://www.w3schools.com/cpp/cpp_oop.asp

https://www.geeksforgeeks.org/c-plus-plus/

7. Tartalomjegyzék