Széchenyi István Egyetem
Gépészmérnöki, Informatikai és Villamosmérnöki Kar
Informatika Tanszék

BEADANDÓ FELADAT
C++ (GKNB_INTM026) C. TÁRGYBÓL

Vas Richárd Csaba
Mérnökinformatikus BSc

2020

Tartalom

1. Feladatleírás

Házifeladatom ként egy Aknakereső játék megvalósítását választottam, ezen keresztül fogom bemutatni a C++11, számos lényeges modulokat, a nyelvben jelenlévő előnyőket és hátrányokat. Továbbá igyekszem bemutatni egy külső grafikus könyvtár (SFML) használatát.

2. Bevezető

Választásom azért egy aknakereső stílusú játékra esett, mert szabályai viszonylag kötetlenek, a koncepciója egyszerű. Az általam megvalósított verzió részben hasonlítani fog a régebbi Windowsos verzióra. Ez is négyzetrácsos pályával rendelkezik, ami 10*10 cella méretű és 10 bombával rendelkezik. A felhasználónak lehetősége lesz feltárni a kívánt cellát, bal egergomb kattintásával, illetve a cellát „megzászlózni”, saját magának való jelölés célljából. A saját verziómban beakartam mutatni a fájlok kezelését, ezért úgy döntöttem, hogy biztosítok a játékosnak lehetőséget mentésre, illetve az elmentett játék-állapot betöltését. Továbba implementáltam egy pontrendszert, amely a feltárt cellák értéke szerint számolja mennyit ért el az adott játékmenetben. (ahány bomba van egy cella körül, akkora számmal rendelkezik és annyi pontot is ér) Nehezítésképpen beraktam egy számlálót mely megmutatja, mennyi ideje van hátra. A játék véget ér, ha bombát fed fel, vagy ha kifut az időből.

Általam felhasznált szoftverek :

- Visual Studio (2017)
- SFML 2.5.1 (x64)
- vld.h

Az SFML egy külső, cross platform, grafikus könyvtár, melynek segítségével, lehetőségem lesz létrehozni egy GUI-t (graphical user interface), amely egy objektíven jobb játékélényt nyújt a felhasználónak, mint egy consolos megoldás. Azért választottam ezt a könyvtárat, mert rendkívül pehelysúlyú, egyszerű a telepítés folyamata, roppant részletes dokumentációval rendelkezik a fejlesztői oldalán (https://www.sfml-dev.org/).

A vld.h: Visual leak detection, egy memóriaszivárgás jelzésére szolgáló könyvtár.

3.Program felépítése

A programot 2 nagy részre lehet osztani, első rész a pálya megalkotásával, bombák elhelyezésével és a cellaértékek megadásával foglalkozik, míg a második az előzőekben létrehozott adatokat a képernyőn való megjelenítésével, kezelésével foglalkozik.

```
auto field_lambda = { ... }
auto gui_lambda= { ... }
int main()
{
    std::thread t1(field_lambda);
    std::thread t2(gui_lambda);
    t1.join();
    t2.join();

    return 0;
}
```

Kapcsolatuk alapján érdemes ezt 2 fajta munkamenetet 1-1 thread formájában implementálni.

Végezze az első thread a pálya és további szükséges elem létrehozását, a 2. thread pedig a grafikus felület kezelését. FONTOS, hogy a 2. thread csak akkor induljon el, ha már az első elvégezte feladatát.

```
auto field_lambda = []() {
    Setup(grid, sgrid);
    Implement_Field(grid);

};
auto gui_lambda= { ... }
int main()
{
    std::thread t1(field_lambda);
    std::thread t2(gui_lambda);
    t1.join();
```

A field_lambda

A lambda 2 függvény meghívását fogja ellátni, egyik a Setup(), másik az Implement_field() lesz. Ez a 2 függvény felelős a pálya elkészítésével, és a bombák, cellák értékadásával. A Setup fv. felvesz kettő fix méretű pályát, az **sgrid** és a **grid**, nevükből adódóan négyzetrácsos elrendezésre utalnak. Az egyes pályák tulajdonképpen mátrixok lesznek. Hatalmas előnye, hogy könnyen kezelhető, módosítható adatforma, melyhez használhatom a vector osztály és minden vele járó előnyt (pl. könnyű iterálás, egyszerű méret lekérdezést...), adatípusa pedig int lesz. Választásom azért az integer-re esett és nem egy objektum oritentált megközelelítésre, mert a későbbiekben elengedhetetlen lesz tudni az egyes cellák egymáshoz való elhelyezkedését, így a mátrix indexeit használhatom majd a grafikus felületen koordináták megadására. (SFML használatakor pixelenként kell számolni a megjelenítani kívánt objektumokat méreteit, helyét, elrendezése egy olyan koordináta rendszer melynek origója a készített ablak bal felső sarka).

Az Implement_Field() függvény adja meg a feladat gerincét. Ez fogja ellátni a bombák generálását, azok elhelyezését és a cellák értékének adását. Generálás során véletlenszerűen 15 bombát fog generálni, generálás során fonots megjegyezni, hogy ez 15 külömböző bombát jelent, így ki kell szűrni az azonos bombákat és aszerint újat generálni, ehhez fog segíteni az init() fv. Ellenőrizzük, hogy volt-e már ilyen és ha nem, akkor

elhelyezzük azt a mines_coord vectorba, amely integer párokat tartalmaz. Miután megvannak a koordináták, azok szerint a elhelyezzük a bombákat a grid mátrixban. Bombákat úgy helyezhetünk el, hogy az előbbi koordinátával rendelkező cella értékét 9-re állítjuk. 

Ennek oka az alábbi kép. 12 lehetséges cella érték létezik a játékban, ha a cella értéke 9 akkor az bombával rendelkezik, ha 0 akkor üres stb... A grid fogja tárolni az értékeket még az sgrid a megjelenítendő értékeket fogja tárolni, azokat amiket a felhasználó fog látni, így az sgrid mátrix egyes értékei 10 lesz defaultban, ezért is nagyobbak egy mérettel a mátrixok, hogy legyen egyfajta szegélye, ahol a cellák -1-el rendelkeznek, a mátrix kényelmesebb bejárása érdekében.

Bár ez a két függvény végzi a nehéz emelést, készítettem pár segéd függvényt, amik nyomonkövethetővé teszik ezek munkáját. Ilyen például a traverseM() vagy a printM(), melyekből megtudhatjuk egyes mátrixok jelenlegi állását a console ablakon keresztül.

```
uto gui_lambda= []() {
   /*kezelni kell azt a hibát, hogy a koordináták validak legy
   ne omoljon össze, ha mellé kattintok + időzítő befejezése*/

   srand(time(0));
   sf::RenderWindow app(sf::VideoMode(640, 480), "Aknakereső!"
   bool isgameover = false;

   sf::Texture t;
   t.loadFromFile("tiles.jpg");
   sf::Sprite s(t);

   int jelen_ido = 0;

   //fontokat külsőleg kell hozzáadni, ez a basic
   sf::Font font;
   if (!font.loadFromFile("arial.ttf"))
   {
       // error...
   }

   // CLOCK _ Timer1
   sf::Text t1;
   sf::Clock clock;
   int countdown = 30;
   int timer = clock.getElapsedTime().asSeconds();
   t1.setFont(font);
   t1.setCharacterSize(24);
   t1.setFillColor(sf::Color::Red);
```

A GUI_lambda

Ez a lambda lesz felelős a grafikus felület megteremtéséért, illetve maga a játékmenet lebonyolításáért.

Első dolga, hogy létrehozza az ablakot, amelyet a felhasználó is fog látni, az a sf::RenderWindow app fogja biztosítani. Az SFML könyvtár biztosít előre megírt osztályokat. Azok meghívását követően általános szabály, hogy megkell adnom helyzetétüket, illetve funkciójuktól függően pl. színét, méretét, kitöltését stb. Egy While ciklus lesz a magja a lamdánknak, amely addig, fut míg az alkalmazásunk nyitva van. A kilépő feltételt a cikluson belűl kell megadnunk, ami lehet a játék vége, vagy ha úgy döntünk, hogy bezárjuk a játékot. Felépítés során itt jönnek elő a külsőleg hozzáadott képek és fájlok kezelése. Mivel szükség lesz font használatára szöveg kiírására, ezért a default arial fontott választottam, ami megtalálható az sfml oldalán. Kelleni fog a az előbbiekben bemutatott lehetséges cellák képe. Most játszik fontos szerepet a mátrix alapú adattárolás, mert Sprite osztály a képet hasonlóan kezeli mint, maga a gui felület megjelenítését, itt is pixelenként kell számolni, az alábbi képlet szerint

```
s.setTextureRect(sf::IntRect(sgrid[i][j] * w, 0, w, w));
                    s.setPosition(i*w, j*w);
```

Konrétan leolvassa a megadott területet a képről és azt használja spirte osztály „díszítésére”. felbontás a 32 skalár érték legyen mert a kép mérete (384x32) látható, hogy

ez egy cella mérete 32x32 pixel, így pontosan kijönnek a méretek, és viszonlag könnyen kirajzolható a pálya egy mátrix bejárás segítségével.

A többi megjelenítendő objektummal is így járok el, a LOAD és a SAVE gomb is hasonló logikával van megadva, a jelenlegi pontokat és a számláló puszta szöveg megjelenítésére az előbbiekben felvett font segítségével.

4. Működés bemutatása

A működés vezérlését és a felhasználó közti interakciót a Gui_lambdán keresztül fog történni, itt töltődik be az ablak melyen keresztül a felhasználó az egér használatával inputokat küldhet a programnak. Játékmenet hasonlóan zajlik mint bármely más eddig ismert aknakeresőnél, itt is bizonyos lehetséges események (Event) következnek be, melyeket a program le reagál. Lehetősége van bal klikkel felfedni az adott cellát ahol az egér jelentleg tartozkodik, jobb klikkel azt megjelelölni(zászlózni), illetve ami játékomban egyedi: képes lesz elmenteni a játék jelenlegi állását, vagy az előzőleg elmentett játék állapotot visszatölteni.

Az előbbiekben leírtak a Gui_lambdában megtalálható while ciklusban kezelt Eventek.

```
sf::Vector2i pos = sf::Mouse::getPosition(app);
int x = pos.x / w;
int y = pos.y / w;
```

```
sf::Event e;
while (app.pollEvent(e))
{
    if (e.type == sf::Event::Closed)
        app.close();
    if (e.type == sf::Event::MouseButtonPressed) {

        /*le kell kezelni, ha pályán kívül kattintasz*/
        if (is_field_koord(x, y)) {
            if (e.key.code == sf::Mouse::Left) sgrid[x][y] = grid[x][y];
            else if (e.key.code == sf::Mouse::Right) sgrid[x][y] = 11;
            if (sgrid[x][y] == 9)
            {
                reveal(grid, sgrid);
```

A fenti képen látható, hogy a program lekéri az egér jelenlegi koordinátáit, majd azokat a kívánt „felbontáshoz” igazítom. Bár ez az érték nem mindig lenne int, a fordító azzá tördeli. Fontos figyelmenbelűl tartani, hogy ezek a koordináták lehetnek és lesznek is mátrix koordináták, így MINDIG ellenőrzini kell hogy az aktuális kattintás nem- e esik kívűl a mátrix tartományán. Az ilyen felületeket külön kell ellenőrzni, ezért van 3 segéd fv.

```
bool is_field_koord(int ix, int iy) { ... }
/*a gomboknak meg kell számolnom, hol kezdődnek a pixelek száma és hol végződnek*/
bool is_load_koord(int ix, int iy) { ... }
bool is_save_koord(int ix, int iy) { ... }
/*fajlkezelo fvek implementalasa, majd azok meghívas a while ciklusban*/
```

A képen látható 3 igazság fv. fogja ellenőrizni, hogy a kapott koordináták validak- e, majd a többi függvény csak a csekkolás után hívható meg. Többek között ezért is volt fontos, hogy koordináták szerint iteráljuk a mátrixot.

Ha már túljutottunk az inputok ellenőrzésén, akkor látszódik a valódi játékmenet. Ahogy már említettem, Bal klikk hatására a lehet cellákat felfedni, vagy a játékot elmenteni és/vagy betölteni, ilyenkor csak azt kell figyelembe venni, hogy a valid input az éppen egy bomba-e vagy üres mező. Hiszen ha bomba akkor véget ér a játék, azomban mi van ha üres cellát találtunk. Ha üres cellát találtunk, akkor a megszokás szerint felkell fedetni az ÖSSZES egytérben levő cellát. Ez pontosan azt jelenti, hogy szükség van egy függvényre, ami az összes nem üres cellát által közrezárt üres cellát felfedi. Ehez a hírás Elárasztásos kitöltés (Flood fill) algortimust választottam. A logikai az, hogy átadunk egy koordinátát, majd onnan elindulunk egy irányba, addig megyünk, amíg nem találunk egy nem üres cellát, ha találunk akkor egy másik irányba, addig csináljunk, amíg nem leszünk képesek tovább haladni, természetesen figyelemben kell tartani az előzőleg bejárt cellákat is. A flood.h ban megtalálható egy struktúra, amit a koordináták tárolására hoztam létre, tovább egy bennevan-e fv. ami ellenőrzi

```
/*ide kell kezelni, ha palyan kivul kattintasz*/
if (is_field_koord(x, y)) {
    if (e.key.code == sf::Mouse::Left) sgrid[x][y] = grid[x][y];
    else if (e.key.code == sf::Mouse::Right) sgrid[x][y] = 11;
    if (sgrid[x][y] == 9)
    {
        reveal(grid, sgrid);
        isgameover = true;
    }
    if (sgrid[x][y] == 0)
    {
        floodFill(grid, x, y);
    }
}//sztem else ifekkel kéne elválasztani a kattintások helyét
```

```
void floodFill(std::vector<std::vector<int>>& grid, int x, int y)
{
    if (grid[x][y] == 0 && !bennevane(vec, x, y))
    {

        /*grid[x][y] = 1;*/
        sgrid[x][y] = 0;

        vec.push_back(flood(x, y));
        floodFill(grid, x + 1, y);
        floodFill(grid, x - 1, y);
        floodFill(grid, x, y - 1);
        floodFill(grid, x, y + 1);
    }
    else
    {
        return;
    }

}
```

hogy a megadott koordináta benne van- e a vektorban. A képen látható a megvalósítás. Egy rekúrzív függvény, ami bárjára a gridet, egyes irányok szerint és az sgrid értékét „felfedi” azaz 10esről 0-ra állítja.

Lehetőség van továbbá a játék jelenlegi állapotát elmentei, idővel együtt egy „game_state.txt” nevű fájlban, természetesen ha nincs akkor csinál egyet a savetofile() fv. a loadfromfile() pedig betölti.

Végeredmény összegzése

Feladatom megvalósítása során törekedtem arra, hogy felhasználjam néhány c++11, által nyújtott új lehetőségeket, megismerkedjek c++ grafikus könyvtárak használatával és még pár nagyon hasznos algoritmussal.