NHF Programozói dokumentáció

Baranyai Ferenc (LITIUP)

Forrásfájlok

* main.c
* Graphics.h/Graphics.c
* GUI.h/GUI.c
* GameElements.h / GameElements.c
* ProcessHandler.h/ProcessHandler.c
* Filemanager.h/Filemanager.c

main.c

A program fő kódja. Itt fut le a fő függvény. Megtalálható benne a játéknak a ciklusa mely a program bezárásáig fut.

Graphics.c

Az SDL könyvtár használatával itt vannak definiálva összetettebb, de a program szempontjából még mindig legalapvetőbb elemek mint például a képek és ezek létrehozásához szükséges függvények.

GUI.c

Nagyban támaszkodik a Graphics.c fájlra mivel ennek segítségével hoz létre interfész elemeket mint például szöveget vagy összetettebb szülő- gyermek relációban lévő képeket amik szükségesek összetett interfész elemekhez.

GameElements.c

Tartalmazza a fontos és alapvető játékelemeket és azok kezeléséhez szükséges függvényeket.

ProcessHandler.c

A játékban futó feldolgozási vagy termelési folyamatokat kezeli.

Filemanager.c

A játék mentéséért felelős kód. Beolvassa vagy éppen kiírja a játék adatait és kezeli az esetleges hibákat.

Grafika

A játék az SDL grafikai könyvtárat használja. A grafikai alapokért felelős fájl a Graphics.c. Ebben vannak definiálva az alap struktúrák és függvények, amik szükségesek a program megjelenítéséért és az átláthatóbb programkódért.

A két legalapvetőbb struktúra a **Vektor2** és a **Color**:

typedef struct Vector2

{

int x,y;

} Vector2;

typedef struct Color

{

int r,g,b;

} Color;

A **Vektor2** egy 2 dimenziós változó mely tartalmaz két számot. Ezzel hatékonyan lehet elraktározni pozíciókat és koordinátákat. A **Color** struktúra segítségével pedig színeket tudunk létrehozni.

A programban megjelenített képek egy struktúrában lettek elraktározva:

typedef struct Image

{

SDL\_Texture\* texture;

SDL\_Rect destination;

} Image;

Az **SDL\_Texture** pointer tartalmazza a már megnyitott texturára mutató memóriacímet és az **SDL\_Rect** változó tárolja el ennek a képnek a méretét és pozícióját. A **CreateImage()** függvényel lehet ilyen változókat létrehozni.

/\*Képek létrehozása\*/

void CreateImage(SDL\_Renderer\* renderer, char \*path, Vector2 positionToPlace, Vector2 size, Image \*img);

A függvény meghívásakor meg kell adni az **SDL\_Renderer** pointert, egy elérési utat a kép forrásához, két darab **Vektor2** változót, amelyek közül az első a kép helyét a második a kép horizontális és vertikális méretét tartalmazza, majd végül egy **Image** pointert melybe a kép elmentésre kerül.

Ehhez a függvényhez szorosan fűződik a **Source** struktúra, ami a texturák forrását tárolja:

typedef struct Source

{

SDL\_Texture\* texture;

char path[100];

} Source;

Ezek a Source elemek egy listában vannak eltárolva és a Image struktúra SDL\_Texture pointere ennek a texture pointerére mutat. Ezzel megoldhatóvá vállik a többször használt képek egyszeri betöltése és másolása. A struktúra másik paramétere egy karaktertömb melyben a texture elérési útja van. Ez alapján lehet megtalálni és lemásolni a texturát. A **CreateImage()** függvény ennek a típusnak a tömbjében keresi meg a forrás képeket majd lemásolja. A Source tömbben való kereséshez van egy **GetSource()** függvény mely egy SDL\_Texture pointerrel tér vissza.

/\*Forrásképek keresése\*/

SDL\_Texture\* GetSource(Source \*sourceList, char \*path, SDL\_Renderer \*renderer);

A kereséshez paraméterként meg kell adni a tömböt, a kép elérési útját illetve egy SDL\_Renderer változót. Az renderer változó megadása azért szükséges, mert ha a tömb nem tartalmazza a keresett kép forrását akkor meghívja autómatikusan a **LoadSource()** függvény mely betölti a képet és a rá mutató pointert tartalmazó Source változót beleteszi a tömbbe.

/\*Forrásképek betöltése\*/

void LoadSource(Source \*sourceList, char \*path, SDL\_Renderer \*renderer);

Ezzel a kis trükkel elérhetük azt hogy a kép csak akkor töltődik be amikor szükség van rá és onnantól kezdve szabadon másolható. Miután megszereztük a forrás elemet onnantól a **CreateImage()** függvény beírja a textúrára mutató pointert és a **SDL\_Rect** változót a **Image** struktúrába.

A **CreateWindow()** függvény létrehozza a program ablakát, a **SDL\_Window** és **SDL\_Renderer** inicializáálásával.

Maga a játéktérkép rombusz alakú csempékből áll ezzel előállítható az izometrikus nézet. Ezek a csempé a játék elején töltődnek be. Főleg ehez volt szükséges megoldani a kép forrás másolásának lehetőségét mivel egy 50\*50-es térkép már 2500 darab csempét tartalmaz melyeknek nagyrésze ugyan azt a képet kapja.

GUI

A játék grafikus interfésze a GUI.c kód segítségével lett megvalósítva. Az egyik indok amiért ez létre lett hozva hogy tartalmazza a CreateText() függvényt mely egy szöveg képet tud létrehozni a pointerként megkapott memóriacímre. Ebben a függvényben meg lehet szabni a betű méretet, színét és helyét is.

/\*Szöveg kirajzolása\*/

void CreateText(char text[], Color color, int sizeOfText, Vector2 position, SDL\_Renderer\* renderer, TTF\_Font \*font, Image \*img);

A másik fontos része a kódnak a **GUI\_Panel** struktúra.

typedef struct GUI\_Panel

{

Image panelImage;

Image children[50];

int childCount;

bool visible;

} GUI\_Panel;

Ez a struktúra valósítja meg a GUI elemeknél a szülő-gyermek relációt, amivel könnyedén lehet egy panelhez hozzáadni elemeket és ezeket utána egyként kezelni. Ehhez a szerkezethez tartozik **RenderParent()** függvény mellyel könnyedén meg lehet jeleníteni ezeket az összetett képeket.

/\*Összegyűjti a gyermek objektumokat és megjeleníti őket\*/

void RenderParent(SDL\_Renderer\* renderer, GUI\_Panel parent);

A függvénynek paraméterként át kell adni a SDL\_Renderer pointert és a megjelenítendő GUI\_Panel-t. Ha GUI\_Panel visible paramétere hamis, akkor nem jeleníti meg a függvény.

A ShopItem struktúra a megvásárolható elemek tárolását és megjelenítését fogja segíteni, valamint a ShopData a vásárlásgombok frissítéséért felelős.

typedef struct ShopItem

{

char name[50];

int price;

int level;

int time;

unsigned char ID;

} ShopItem;

typedef struct ShopData

{

int actualItem;

ShopItem Items[6];

unsigned short int ChildCount;

unsigned short int ItemCount;

} ShopData;

A ShopItem adatszerkezetben el lehet tárolni az adott tárgy nevét, árát, a minimum szint, amitől meg lehet vásárolni és az idő, ami az elkészítéséhez szükséges. Ezt főként a magok és az épületek paneljén vannak használva. A ShopData szerkezetben pedig az éppen aktuálisan megvett elem indexét, a ShopItem-ek tömbjét és ennek a mérete és a vásárlás gombok gyermek elemeinek száma van elhelyezve.

Az előző két struktúra segítségével képes a RenderShopPanel() függvény mely az ültető és az építő panelen megjelenített elemeket átszínezni aszerint hogy megvásárolható-e vagy sem.

/\*Megjeleníti a vásárló felületeket és megváltoztatja a hozzáférhetés szerint a színeket\*/

void RenderShopPanel(SDL\_Renderer\* renderer, GUI\_Panel \*panel,ShopItem \*items, int childcount ,int money, int level);

Emellett a panelek animációjáért a ShowAnimatedGUI() felelős.

/\*Kiszámolja és legenerálja az eltűnés/előjövetel animációt\*/

void ShowAnimatedGUI(SDL\_Renderer\* renderer, GUI\_Panel \*panel, int windowSizeY, ShopItem \*items, int childcount ,int money, int level);

Ez a függvény ha a paraméterként átadott GUI\_Panel visible változója hamis akkor addig von le a magasságából ameddig el nem ér egy előre meg nem határozott koordinátára. Ha a visible igaz akkor ugyan ez történik csak a magassághoz hozzáad.

A barkács paneleket, szám szerint 3 a CreateCraftPanel() függvénnyel lett elkészítve a program indításakor.

/\*Létrehozza a "barkács" menü elemeket\*/

void CreateCraftPanel(SDL\_Renderer\* renderer,char \*title, Vector2 windowSize, ArrayData \*RecipesData, RecipeType type, Item \*inventory, GUI\_Panel \*panel);

Egy másik fontos függvény az OverUI() mely meghívásakor megadjuk a kurzor pozícióját és a ellenőrizni kívánt kép SDL\_Rect paraméterét mely tartalmazza a méretéit és koordinátáit. A függvény egy logikai értékkel tér vissza attól függően, hogy rajta van-e képen a kurzor.

/\*Ellenőrzi, hogy a megadott koordináta a megadott interfész elem felett van-e\*/

bool OverUI(Vector2 mousePos, SDL\_Rect obj);

A raktárban található lista megjelenítésére a OpenWarehouse() függvény segít.

/\*A raktár listájának létrehozása\*/

void OpenWarehouse(Canvas \*canvas, SDL\_Renderer \*renderer, ArrayData \*InvData,TTF\_Font \*font,int \*ids);

Az összes grafikus interfész elem egy nagy struktúrában lett eltárolva a könnyebb paraméterezés és elérhetőség miatt.

Tartalmazza a vásárlást lehetővé tevő paneleket és ezek megnyitásához szükséges gombok és a szint és pénz megjelenítésénél használt képeket.

typedef struct Canvas

{

GUI\_Panel buildPanel;

GUI\_Panel plantPanel;

GUI\_Panel warehousePanel;

Image buildButton;

Image plantButton;

Image infoBox;

Image moneyText;

Image levelText;

Image levelProgression;

} Canvas;

A RenderCanvas függvény ez előbb említett struktúra megjelenítésére szolgál, ehhez meg kell adni a Canvas pointerét és a megszokott SDL\_Renderer pointert.

/\*Az alap GUI megjelenítése\*/

void RenderCanvas(Canvas \*canvas, SDL\_Renderer \*renderer);

ProcessHandler

A ProcessHandler.c fájl tartalmazza a játékban futó folyamatok (építkezés, növények, barkácsolás) kezeléséhez szükséges függvényeket és eszközöket.

Az egész fájlnak az alapja maga a folyamatot képző struktúra a Process struktúra. Az adatszerkezet magába foglalja a folyamat azonosítóját ami általában a futó folyamatot tartalmazó csempe azonosítója, de barkácsépületeknél már máshogy működik, de arról később. Ezen kívül még megtalálható a típusa, ami egy enum.

typedef struct Process

{

int ProcessID;

ProcessType type;

int TileID;

unsigned long long t;

int misc;

bool done;

} Process;

typedef enum ProcessType

{

Building, Plant,Craft, Loop

} ProcessType;

A típus megmondja, hogy mire jó a folyamat és eszerint lesz majd feldolgozva. Emellett el van raktározva a folyamathoz tartozó csempe azonosítója, illetve azt az időt amikor a folyamat végezni fog, ezzel így egy konstans értéket tudunk eltárolni gyorsítva a feldolgozást. A struktúra egy fő eleme a misc intiger változó. Ebben a változóban van a végtermék azonosítója, építkezés esetén az épület, növény és barkácsolás esetében a végeredményként kapott tárgyat. És végül kapott a szerkezet egy logikai értéket (bool done) mely arra szolgál, hogy ha a folyamat készen van ez igaz értékre változik és akkor kerül csak törlésre, ha a felhasználó begyűjti az elkészült tárgyat.

Ezeket a folyamatokat egy dinamikus tömbbe lett eltárolva a program futása alatt. Ezt a tömböt is egy struktúrában van egy intigerrel társátva ami tartalmazza az aktuális méretét.

Ezt a tömböt az AddProcess() függvény segítségével lehet lefoglalni majd a későbbiekben bővíteni.

typedef struct ProcessList

{

Process \*l;

int n;

} ProcessList;

/\*Folyamat hozzáadása dinamikus tömbhöz\*/

Process\* AddProcess(ProcessList \*p\_list, Process process);

Ez a függvény paraméterként elvárja a ProcessList pointerét és a hozzáadandó folyamatot és visszatér a tömbben lévő folyamat pointerével. Ha egy folyamat fégzett és törölni szeretnénk a listából akkor a RemoveProcess() függvényt tudjuk használni. A függvénynek meg kell adni a dinamikus tömböt, illetve az eltávolítandó folyamat azonosítóját.

/\*Folyamat eltávolítása\*/

void RemoveProcess(ProcessList \*p\_list, int id);

Ezeken kívül tartalmaz a forrásfájl kisebb függvényeket mint például a GetProcess() amivel a dinamikus tömbben kereshetünk azonosító alapján és visszatér a keresett folyamat pointerével vagy egy NULL pointerrel. A másik függvény csupán tesztelés céljával készült el. A ListProcesses() függvény a futó folyamatokat listázza ki, paraméterként elvárja a kilistázandó ProcessList struktúrát.

/\*A dinamikus tömb kilistázása tesztelés végett\*/

void ListProcesses(ProcessList \*p\_list);

/\*Folyamat keresése azonosító alapján\*/

Process\* GetProcess(ProcessList \*p\_list, int id);

GameElements

A GameElements.c kód tartalmazza a játékelemeket melyek alapvetőek a program futásához. Ha a játékprogramra ránézünk látható, hogy a térkép kisebb rombusz komponensekből épül fel. Ezeket a komponenseket csempének neveztem el. A csempéknek az alapja egy struktúra a Tile. Minden csempe rendelkezik egy azonosítóval, ami alapján meg lehet őket koordináta nélkül találni, emelett rendelkeznek egy névvel, de ez nem egyedi hanem a kép nevét kapják amellyel rendelkeznek. Kettő Vector2 pozíció is el van tárolva. A koordinátái és az kurzorhoz vett relatív pozíció. Az utóbbi szükséges ahoz, hogy egér segítségével körbe lehessen nézni. Rendelkezésre áll még két kép változó, egy amiben a csempe képe van (Image img), amiben egy mellékkép amely szükség esetén a csempe felett jelenik meg, mint például a növények képe a mező fölött és végül van egy ikon kép is ami például lehet nyíl ami a építkezés vagy ültetés közben mutatja a szabad helyeket, vagy lehet egy tárgy képe miután elkészült és be lehet gyűjteni. Ehhez a képhez szorosan tartozik az arrow logikai változó mely tartalmazza, hogy nyíl-e az ikon vagy valami más. Végül a csempe rendelkezik egy pointerrel mely a rajta futó folyamatra mutat. Ezt akkor adjuk hozzá amikor a folyamatot létrehozzuk és hozzáadjuk a futó folyamatok listájához az AddProcess() segítségevel, ami visszatér egy pointerrel amit itt a Tile struktúrában eltároljuk ameddig a folyamat nem törlődik.

typedef struct Tile

{

unsigned short int id;

char name[20];

Vector2 positionToMouse;

Vector2 coordinates;

Image img;

Image additionalImage;

Image icon;

Process \*process;

bool arrow;

} Tile;

Ebből a csempéből nagyon sok van a játék futása alatt, alapbeállítások mellett 50x50 ami 2500 darab. A könnyű hozzáférhetőség miatt ezt egy dinamikus mátrixba lett elmentve. A mátrix a TileMatrix struktúrában van, hogy mellette el lehessen tárolni a magasságát és szélességét. A mátrixnak köszönhetően nem kell kölün függvény a koordináta szerinti keresésnek mivel egyből ezek egyből az index egyik elemére mutatnak

typedef struct TileMatrix

{

Tile\*\* matrix;

int xSize;

int ySize;

} TileMatrix;

A mátrix létrehozásához használható a CreateMatrix() aminek meg kell adni a kívánt mátrix szélességét és magasságát, majd a függvény visszatér a mátrixxal.

/\*Mátrix létrehozása\*/

Tile\*\* CreateMatrix(int xSize, int ySize);

Ennek úgymond a párja a mátrix felszabadítására elkészített függvény a FreeMatrix() ami egy TileMatrix pointert vár paraméterként és ezt utána felszabadítja a memóriából. Ez a függvény csakis a program bezárásakor fut le mert addig szükség van minden csempére.

/\*Mátrix felszabadítása\*/

void FreeMatrix(TileMatrix \*matrix);

Emelett a forrásfájlban helyet kapott még kettő kereső függvény. Az egyik pozíció alapján keres és egy Vector2 értéket vár paraméterként a másik pedig azonosító alapján keres és azonosítót vár paraméterként majd visszatérnek egy pointerrel ami a paraméterként megadott mátrix egyik elemére mutat. Ha nem találnak semmit akkor NULL pointerrel térnek vissza.

/\*Keresés a mátrixban pozíció alapján\*/

Tile\* GetTileFromPosition(TileMatrix \*matrix, Vector2 position);

/\*Keresés a mátrixban azonosító szerint\*/

Tile\* GetTileByID(TileMatrix \*matrix, int id);

Az inventory megvalósításához van az Item struktúra. Ez lehetővé teszi különböző gyűjthető tárgyak leírását és tulajdonságainak tárolását. A struktúrán bellül helyet kapott egy karaktertömb a tárgy nevének tárolásához. Emelett néhány statisztikai adatot is tárol, mint az árát és hogy mennyi van belőle, illetve minden tárgynak van egy azonosítója.

typedef struct Item

{

char Name[100];

unsigned int Price;

unsigned int Amount;

unsigned int id;

} Item;

Ezek a tárgyak későbbiekben a program futása alatt egy tömbben lesznek ami konkrétan már az inventory lesz. Az inventoryban való kereséshez a GetItemByName() függvény használható, mely paraméterként elvár egy nevet az inventory méretét és magát az inventor Item tömböt. Ezután ha megtalálta a keresett elemet visszatér a rámutató pointerrel, ellenkező esetben NULL pointerrel tér vissza.

/\*Tárgy megkeresése az inventoryban\*/

Item\* GetItemByName(char \*name, int n, Item \*itemList);

Mivel a tárgy azonosítója megegyezik a tömbben lévő indexxével ezért arra nincsen kereső függvény.

A játék egyik fontos eleme a barkácsolás. Ehhez receptek vannak ami megmondja, hogy miből mennyi kell különböző dolgok elkészítéséhez. Ezek a receptek a Recipe struktúra segítségével vannak megalkotva. Ennek a szerkezetnek elemei a következőek. Rendelkezik kettő darab integer tömbbel. Az első tömbben el van tárolva az a maximum 2 darab tárgy azonosítója ami szükséges a recept elkészítéséhez, a második pedig tartalmazza hogy ezekből hány darab kell tárgyanként. Ezek után következő két intiger tartalmazza a kész termék azonosítóját és darabszámát. A recept típusa egy nagyon fontos része ennek a struktúrának, mégpedig azért mert ez mondja meg hogy melyik barkácsépületben lehet elkészíteni a 3 közül. Ez egy enum változó mely a RecipeType nevet viseli.

typedef enum RecipeType{Windmill, Bakery, Brewery} RecipeType;

typedef struct Recipe

{

int itemIds[2];

int itemAmounts[2];

int resultId;

int resultAmount;

RecipeType type;

unsigned int t;

unsigned int id;

} Recipe;

Ezek mellett még megtalálható még 2 darab intiger változó. Az elsőben van, hogy mennyi idő vesz igénybe az elkészítés és a másikban pedig a recept azonosítója, ami mint az inventoryban megegyezik a tömbben lévő indexével.

FileManager

Az előző forrásfájlokhoz képest a FileManager.c nem annyira szerves része, mivel ez végzi el a mentési műveleteket ami nélkül a játék játszható marad, de elvész a mentés lehetősége. A mentés három bináris fájlba történik.

Az első és legfontosabb save.bin fájlba történik a pénz, szint, játékidő és a térkép valamint a rajta futó folyamatok mentése. Az első 16 biten van elmentve a játékos egyenlege majd a következő 16 biten az elért szintje és utána lévő 32 biten van a játék első indítása óta eltelt idő. Ezeket az adatokat a SaveStats() függvény menti fájlba. Ahol az előbb felsorolt adatokat meg kell adni paraméterként.

void SaveStats(unsigned int level,unsigned int money, unsigned long time);

Ezeken felül minden egyes csempe kap egy 64 bites részt amire elmentésre kerül 12 biten az azonosítója, 8 biten a típusa és végül van 44 bit amire egyéb tulajdonságok vannak elmentve általában a rajta futó folyamatok. Ezt a részt a SetSave() függvény végzi amely az előbbiekben felsorolt adatokat várja paraméterként.

void SetSave(unsigned short int id, unsigned char type, unsigned long long misc);

A futó folyamatok mentése a 44 biten való mentésénél az első 36 biten a folyamat befejezésének az időpontja van eltárolva. Ez a rész azért olyan nagy mivel ez az időpont az első játéktól kezdődő számláló szerinti idő amikor a folyamat befelyezésre kerül. A maradék 8 biten került elmentésre a folyamat tevékenységének az azonosítója. Ez alatt az érthető hogyha építkezés van akkor az épület azonosítója vagy ültetésnél a növény azonosítója. Ezt az úgymond bejegyzést a GetBinary() függvénnyel lehet megalkotni. A függvény bitshiftelés segítségével egy unsigned long long int-be gyűjti az egy csempéhez tartozó adatokat.

unsigned long long int GetBinary(unsigned short int id, unsigned char type, unsigned long long misc);

A második fájl tartalmazza az inventoryt. Az inventory elemei egyszerűen vannak elmentve. Minden egyes tárgy kap 32 bit helyet. 16 biten van az azonosítójuk és 16 bites helyen van a mennyiségük. Ezt a műveletet a SaveInventory() végzi. Paraméterként meg kell adni az inventory tömbjét és méretét.

void SaveInventory(Item \*inventory, int n);

A harmadik fájlba szorult vissza a program bezárásának időpontja. Ez nagyon fontos hogy ki tudjuk számolni a következő futásig eltelt időt és így letudjuk azt vonni a futó műveletekből, így akkor is tellik a játékban az idő amikor az nem fut. Ehhez a SaveTime() függvényt lehet hasznos.

void SaveTime();

A mentések beolvasásához két függvény áll rendelkezésre az egyik az inventoryt olvassa be már egy Item tömbbe, míg a másik a csempék illetve az egyéb adatokkal tér vissza.

A GetInventory() segítségével történik az inventory mentésének beolvasása. Ehhez paraméterként meg kell adni egy Item tömbböt és annak méretét. A függvény ebbe a tömbbe fogja beletenni a beolvasott és feldolgozott adatokat. Visszatérési értékként egy logikai értéket ad. Ha sikeres volt a beolvasás akkor igazat, ellenkező esetben hamisat.

bool GetInventory(Item \*inventory, int n);

A GetSave() függvény a térkép, pénz, szint és eltelt idő beolvasására alkalmas viszont nem dolgozza fel az adatokat, hanem csak egy unsigned long long int tömbböt ad vissza. Minden elem a tömbben egy-egy csempe adatait tartalmazza, illetve az első a pénz, szint és idő hármast. Ezek az adatok majd a fő programkódban kerülnek feldolgozásra a térkép generálása alatt. A GetInventory() függvénnyel hasonlóan egy logikai értékkel tér viszza és paraméterként meg kell adni a mentés helyét és egy tömbböt amibe a mentést tenni fogja.

bool GetSave(char path[], unsigned long long int \*save);

Main

A main.c a program fő kódja itt fonódik össze az összes előbb említett forrásfájl egy nagy egésszé. Ebben a kódban fut a main függvény melyen belül van a fő ciklus mely a játék ideje alatt végig lefut és frissíti az adatokat. De még előtte a program indításakor definiálunk néhány alap adatot. Először az összes létező tárgy kerül definiálásra majd ezután beolvassuk az inventory mentését ha van ilyen. A inventory az AllItem Item típusú tömbben van tárolva a program futása alatt.

//Inventory

AllItem[0] = (Item){.Name = "Wheat", .Price = 3, .Amount = 0, .id = 0};

AllItem[1] = (Item){.Name = "Potato", .Price = 5, .Amount = 0, .id = 1};

AllItem[2] = (Item){.Name = "Corn", .Price = 6, .Amount = 0, .id = 2};

AllItem[3] = (Item){.Name = "Tomato", .Price = 6, .Amount = 0, .id = 3};

...

Ha ez megvan folytatjuk az épületek és növények defniálásával. Az épületek és a növények is a már említett ShopData struktúrában vannak elhelyezve.

//Épületek

BuildingData.Items[0] = (ShopItem){"Field",30,1,10,21};

BuildingData.Items[1] = (ShopItem){"Beehive",500,5,60,22};

BuildingData.Items[2] = (ShopItem){"Greenhouse",2000,10,300,23};

BuildingData.Items[3] = (ShopItem){"Ground",10,1,60,10};

BuildingData.Items[4] = (ShopItem){"ConstructionSite",0,1,60,1};