Programozói dokumentáció
Életjáték

A program felépítése

A program 5 db modulból épül fel.

main.c – a főprogram, kezeli a felhasználó által generált eseményeket.

lepteto.c – a szimuláció „lelke”, az adott állásból a következő állást generálja.

fileio.c – kapcsolatot teremt a külvilággal, végzi a betöltést fájlból, a mentést fájlba, és a konfigurációs fájl beolvasását induláskor.

rajzol.c – grafikus megjelenítésért felel, a képernyőre rajzolja ki a programelemeket.

types.c – a globális változókat tárolja, illetve a hozzátartozó header fájlban vannak a típusdefiníciók.

A modulok belső működése

types.c

A következő típusokat definiálom:

typeconfig – mindenféle adatot tartalmaz a megjelentetett táblával kapcsolatban, úgy, mint például a kép mérete, a cellák mérete, a háttér színe, a szegélyek láthatósága stb.

Tabla – Az életteret tárolja, pontosabban, hogy hány sor és oszlopból áll a négyzetháló, és egy pointert (char **T), ami az értékekre mutat. Így a Tabla egyes celláit kettős indexeléssel érhetjük el. A memóriafoglalásnál sorfolytonos foglalást használok, így két malloc és free elég. Először lefoglalok egy tömböt pointereknek, majd egy tömböt a valódi adatoknak, és a pointerek tömbjét a megfelelő adatokra állítom. A char típust azért választottam, mert az int hely pazarlóan sok memóriát foglalt volna, hiszen 0-7-ig írok csak számokat a táblába.

typecell – egy adott sejttípus terjedési képességeit tárolom el ebben.

A programban összesen 5 globális változót használok, bár ezek összetett adattípusok.

Ezek közül a fontosabbak maga a Tabla, a Config, és az SDL függvényeinek átadott screen. Ezek közül legalább kettőt szinte minden programbéli függvény használ. Először paraméteres átadással oldottam meg mindent, de a kód nagyon bonyolult és áttekinthetetlen lett, ezért hoztam létre globális változókat.

fileio.c

Erről a modulról a mentés egyedisége miatt kell beszélni.

Mivel minden cellában 0-tól 7-ig kerülnek számok, ezek három biten elférnek. Így 3 bájtba el tudok menteni akár 8 db cellát is. Például: 0 1 2 3 4 5 6 7 → 00000101_00111001_01110111

Így eléggé minimalizálni tudtuk a helyfoglalást (legalábbis kellően nagy táblák esetén), viszont cserébe biteltoló műveletekkel kell varázsolni, és bináris fájlba menteni az adatokat, ami

bonyolultabbá, és hosszabbá teszi a műveleteket. Azonban az álláspontom az, hogy adatok mentésénél és beolvasásánál a memória szerint kell elsődlegesen optimalizálni és ezért hajlandóak vagyunk engedni egy kicsit a futásidő gyorsaságából.

lepteto.c

Ebben a modulban két függvény van.

A clear egyszerűen csak kinullázza az összes cellaértéket.

A leptet az, ami a szimulációt valóban végzi. Az életjáték tulajdonságaiból következik, hogy a szimuláció generálható úgy, hogy egy teljesen új pályára tesszük le a következő generációban élő sejteket, majd a végén a régi táblát eldobjuk, és az új értékeit másoljuk bele. Ezt fogjuk kihasználni, ezért a függvény elején létrehozunk az eredetivel azonos méretű pályát.

Fontos megjegyezni, hogy az életteret eltároló Tabla.T-ben két oszloppal és sorral több van, mint szükséges. Ezeket mint strázsákat használjuk fel, első előtti és utolsó utáni sornak illetve oszlopnak.
Erre azért van szükség, hogy a szimuláló algoritmus könnyebben kezelhető legyen, hiszen így nem kell külön dolgozni az első és az utolsó sorokkal/oszlopokkal.
Ha a Continuous tulajdonság aktív, akkor ezeket a strázsákat fel is tölthetjük a megfelelő másolatokkal, ellenkező esetben nullákkal.

Miután ennyi előkészülettel megvagyunk, azt, hogy egy adott sejt él-e a következő generációban, azt a következő algoritmus dönti el.
Megszámoljuk, hogy az adott sejt közvetlen környezetében (beleértve saját magát is) mennyi sejt található az egyes fajtákból. (Úgymond statisztikát készítünk róluk.)
Itt is alkalmaztam egy minimális optimalizációt, ugyanis a statisztikát nem nullázom ki minden sejt alkalmával, hanem kivonom belőle azokat a sejteket, akik már nem számítanak a következő sejt szempontjából, és beleveszem azokat, akik már számítani fognak. Ha mindig elölről kezdeném a számolást, az 18 lépés lenne, de mivel van 6 átfedés, így csak 3-at kell kivonnom, és hármat hozzáadnom, ami csak hat lépés. Így a hatékonyság majdnem háromszoros.

Majd ezután esetekre bontunk, aszerint, hogy az adott hely üres-e, vagy ha van ott sejt, akkor az baráti-e vagy ellenséges. Mindegyik esetben képzünk egy releváns összeget, ami úgy áll elő, hogy a saját sejtek kétszereséhez hozzáadjuk az üres környező helyeket és kivonunk belőle kilencet. Így egy olyan számot kapunk, ami egyféle sejt esetén a sejtek számát adja meg, viszont ellenséges környező sejtek esetén ennél kevesebbet.
Megvizsgáljuk, hogy ez a releváns szám, hogy viszonyul a sejt terjedési attribútumához az adott esetben, és ez alapján eldöntjük, hogy éljen-e a sejt a következő generációban.

Miután az összes sejtre ezt elvégeztük, az eredeti tábla tartalmát felülírjuk az újjal.

rajzol.c

Ebben a modulban lévő függvények sajátosságai, hogy semmit nem változtatnak meg a programban, csak kirajzolják a megfelelő programrészeket. A rajzol_menu értelemszerűen a menüt rajzolja ki, a rajzol_tabla a tábla részeit, a rajzol_egesz a sejteket, a rajzol_egyet egy darab sejtet, a rajzol_urestable pedig egy még nem létrehozott, csak létrehozandó táblát rajzol meg a képernyőn.

Itt van egy belső függvény is, ami egy rgb színből saját szabályok alapján szürkeárnyalatos színt kreál.

main.c

A főprogram kezeli az eseményeket, ehhez szükséges függvényeket tartalmaz. Ezen kívül van benne egy tablaletrehoz és egy tablatorol függvény, melyek fő funkciójuk, hogy memóriát foglalnak és szabadítanak fel az élettérnek. Minden memóriafoglalással kapcsolatos művelet ebben a két függvényben kell, hogy történjen, egyetlen kivétel még a játék betöltése, hiszen elképzelhető, hogy más méretű táblát akarunk betölteni, mint ami már létre van hozva, ha egyáltalán van létrehozva pálya.