Проект по функционално програмиране
„Conway's Game of Life“
Василен М. Чижов, 3 курс, Информатика, ФМИ

1. Анализ на задачата

Задачата реално се свежда до получаване на конфигурация от дадена предишна по вече дефинирани правила. В случая началната конфигурация се задава от потребителя и идеята е да се опише алгоритъм, по който от началната конфигурация да се получи следващата. За представянето на конфигурациите има множество варианти, като често бива използван двумерен масив (или матрица) в който дадена клетка може да е или жива или мъртва. На всеки ход за всяка жива клетка се преброяват живите и съседи (8-те клетки около нея), ако те са 2 или 3, в следващата конфигурация тази клетка остава жива, иначе се промененя на мъртва; от друга страна за всяка мъртва келтка, която е съсед на жива клетка, се преброява колко живи съседа има – ако те са точно 3 тя се променя на жива, иначе остава мъртва. Алгоритъма изглежда сравнително прост, като има множество възможности които могат да го усложнят сравнително. Аз съм избрал по-различен подход, от пазенето на матрица в паметта и обработването и, по няколко причини – матрицата обикновено се налага да е със сравнително ограничени размери поради паметта – т.е. конфигурациите не могат да нарастват след един момент, другият минус е че е заделено огромно количество памет, което не е наистина нужно, разбира се това има своите предимства – простотата на представянето, както и бързото достъпване до дадена клетка/съседите и. В моята реализация, аз пазя списък от живите клетки в паметта, всяка представена чрез списък от координатите и. Нека примерно имаме следните живи клетки (x0,y0), … , (xn, yn), тогава списъка ще е от вида ((xi0 yi0) … (xin yin)), като клетките са подредени първо по нарастване на x координатата им и второ по нарастване на y координатата им. Разбира се има и по-ефективни методи на подреждане, някои от които ще бъдат обсъдени в 3, но за простота и поне някаква оптимизация съм избрал този. Алгоритъма за стъпката е следният: нека Ak е конфигурацията (списъка) на стъпка k, от Ak получаваме множество R от клетките които са от Ak и за които бр. съседи == 2 или 3 (броят съседи на дадена клетка лесно може да бъде намерен чрез обхождане на списъка Ak), и получаваме дурго множество Z от всички мъртви клетки, които се явяват съседи на жива клетка и имат броят на живите им съседи е точно 3. Обединявайки

двете множества получаваме конфигурацията Ak+1.

2.Описание на основните функции и структури от данни изпозвани при решаването на зададения проблем

Основните структури използвани са множества от двуизмерни и триизмерни вектори подредени първо по първата координата, и второ по втората, във възходящ ред. Векторите са списъци с два или три елемента.

Базовите функции за работа с клетките са: (make-live x y G) което добавя клетка с коорд. x y, в множеството на живите клетки G, ако няма вече такава; (make-dead x y G) премахва клетка с коорд. x y от G, ако има такава клетка; (live? X y G) проверява дали има клетка с коорд. x y в G; (neighbours x y G) дава списък от всички живи съседи на G; (generate-neightbours x y) дава списък от всички съседи на G; (neighbours-diff A B) намира разликата между множество A, което се очаква да е генерирано от generate-neighbours и множеството B, което се очаква да е генерирано от neighbours.

Основни функции за задачата са следните: (merge-vsets A B) действа като обединение на A и B, като изкарва списък който е сортиран първо по x и второ по y; (insert-semi-step x y Z) имайки списък от наредени триизмерни вектори Z, ако в Z няма вектор с координати x y добавя такъв вектор с 3-та координата 1, ако вече има такъв вектор – увеличава 3-тата му координата с 1 – т.е. брои колко пъти вектора (x y) е бил слаган в списъка; (filter-semi-step Z) филтрира всички вектори от Z, на които 3-тата им координата е равна на 3 и връща списък от тях без 3-тата им координата.

Най-главната част – т.е. стъпката се извършва чрез (make-step G). Тя извиква спомагателна функция, която обхожда G последователно, като генерира списите R и Z от 1 и в крайна сметка ги обединява. Обхождайки G, за всяка клетка (x y) се намират съседите и броят им чрез съответно (neighbours x y G) и length. Ако клетка има 2 или 3 живи съседа се добавя към R чрез push-back, иначе R остава същото. Едновременно с това всички съседи, които не са живи се добавят във Z чрез insert-semi-step (съседите които не са живи са просто всички съседи (generate-neioghbours x y G) без (neighbours-diff) живите

(neighbours x y G). След краен брой стъпки ще сме обходили цялото G. Тогава просто филтрираме клетките които ще станат живи на следващия ход от Z чрез (filter-semi-step Z) и ги обединяваме с тези, които ще останат живи R, чрез (merge-vsets R Z). Така получаваме конфигурацията на следващата стъпка.

Други функции, които могат да се споменат са: (make-mat-from-cells x0 y0 x1 y1 G) генерира матрица от конфигурацията G в правоъгълника от x0,y0 до x1,y1 (x0<x1, y0<x1); (make-cells-from-mat xo yo M) генерира конфигурация от матрицата M с отместване xo,yo; (bounding-box-cells G) дава координатите описващи минималният правоъгълник, такъв че съсдържа всички клетки от G; (offset-cells ox oy G) транслира координатите на всяка клетка от G с (ox,oy); (rotate-cells G) разменя x и y координатите в G.

Функции за изобразяване на екрана: (drawGrid vp x0 y0 dx dy x1 y1) рисува решетка във viewport-a vp с отместване dx,dy и нач. координати x0 y0; (drawCells vp x0 y0 x1 y1 l) рисува всички клетки от l, които са във видимата зона, във vp.

Запис на конфигурация на файл става чрез (save-cells-to-file filename G), зареждане на конфигурация от файл става чрез (load-cells-from-file filename).

C (2dGraphics screenX screenY spX spY zoomX zoomY G) става започването на програмата в графичен режим (screenX, screenY) е резолюцията, с която желаем да стартираме програмата, (spX, spY) е позицията от която искаме нашата „камера“ да започне, (zoomx, zoomY) е мащаба с който искаме да виждаме „дъската за игра“, G е произволна конфигурация.

3. Идеи за бъдещи подобрения

- Повечето операции свързани с списъците от вектори могат да използват бинарно търсене.
- Ефикасността (процесорно време/памет) може да се увеличи чрез ползването на мутиращи операции.
- Може клетките не просто да се сортират а, пространството да се разделя на региони в зависимост от концентрацията на клетки в дадена част от пространството.
- hashlife