Курсова работа

Имплементация на клетъчен автомат с Python и библиотеката Kivy

1. Въведение
   1. Клетъчни автомати

Клетъчен автомат представлява абстрактен, дискретен модел, състоящ се от съвкупност от *клетки,* разположени в пространство с произволен брой измерения, които за единица време претърпяват определени преобразувания. Всяка клетка има краен брой възможни състояния (общи за всички клетки), които може да приема. Бъдещото състояние на всяка клетка се определя спрямо състоянията на непосредствените й съседи в решетката на автомата. Така един автомат представлява и система за пресмятане и прилагане на функции.

* 1. Играта „Живот“

Конкретната имплементация е на известния автомат, създаден от Джон Конуей, „Живот“ (*Conway's Game of Life*). В него клетките са квадратни и разположени в двуизмерна решетка и приемат две състояния – живи или мъртви, които се представят съответно с черен и бял цвят. За съседи на една клетка се приемат осемте директно заобикалящи я, и преобразуванията във времето се извършват според следните правила:

1. Ако една клетка е жива и има по-малко от две или повече от три живи съседи, умира
2. Ако една жива клетка има две или три живи съседи, не променя състоянието си
3. Ако една мъртва клетка има точно три живи съседи, променя състоянието си на жива

Тези правила се прилагат едновременно върху всички клетки и така се получава следващото състояние на автомата – това е единицата за време – стъпка или „поколение“. Така следващото състояние на автомата е функция на предишното, но до едно състояние може да се стигне от множество различни „начални“ състояния. „Живот“ е доказано аналогичен на универсална машина на Тюринг в публикацията *Winning Ways for Your Mathematical Plays* (Berlekamp, E., Conway J., Guy R., 1982).

1. Спецификации на имплементацията и използваната библиотека
   1. Възможности на приложението

В приложението решетката на автомата е квадратна и има размери 5000 х 5000 клетки. Графичният интерфейс подържа основните функции на играта – определяне на състоянието на отделни клетки, изпълняване на преобразуването на една стъпка или продължително преобразуване на автомата във времето. Освен това позволява увеличаване и намаляване на мащаба на решетката, контрол над скоростта на итерациите на автомата и история, съдържаща последните 99 състояния както и началното състояние.

* 1. Библиотеката Kivy

Kivy е мултиплатформена библиотека за разработка на графични приложения с Python, написана върху OpenGL ES 2. Избрана е за този проект основно заради възможността за лесно и бързо прилагане на трансформации върху графиката, които запазват абсолютни координатите на преобразуваните обекти. Също така предлага и разделение между дизайна на графичния интерфейс и логиката на приложението. Основните характеристики на библиотеката са (конкретните примери за използването им са в следващата секция):

* Възможност за приемане и обработка на мултитъч събития („Event“) за мобилни устройства (не е използвано в текущия проект).
* „Event Bubbling“ - когато едно събитие се приеме от обект в приложението, то по подразбиране се предава на всички останали обекти (в обратен ред на създаването им) освен ако при обработката му този процес не се прекъсне.
* Обектите („Widget“) от библиотеката могат да ползват т.нар. свойства („Property“), които при промяна автоматично създават събитие, което може да се прихване и обработи
* Всеки обект от библиотеката има собствено платно за рисуване (“Canavas”), което може да приема графични инструкции за добавяне на изображения
* Езикът Kv („kv language”) позволява отделянето описанието на графичния интерфейс в .kv файл. В него има описание на обектите (вложени един в друг под формата на дърво) и техните свойства и събитията, свързани с тях (стойностите им могат да бъдат и код на python). Всичко това има еквивалент с чист код на python, но този начин е по-сбит и лесен за разбиране

1. Описание на кода и процеса на работа.

Приложението се състои от четири файла (main.py, gol.kv, gol\_base.py, planetracker.py и scatter\_transforms.py). За най-добро представяне на кода на приложението, ще проследим йерархията в която е изграден, като правим паралели между разглежданите класове/функции, връзката им с .kv файла, действието им при изпълнение на приложението и визуалното им представяне в интерфейса. С това ще опишем и действието на библиотеката като цяло.

* 1. Описание
     1. main.py, App class

Началната точка на приложението е main.py. В него има клас, който наследява класът *App* на Kivy - това е задължително за всяко Kivy приложение - в случая се казва GOL. При изпълнение на програмата библиотеката ще зареди .kv файла със същото име като на този обект (с малки букви; от там - *gol.kv*) преди да изпълни build() метода, които от своя страна трябва да върне обект, съдържащ потребителският интерфейс (най-често *Layou*t). В кода този обект е *baseClas*s, съдържащ се в файла *gol\_base.py*; той ще заеме целият прозорец на апликацията. Методът run() на App стартира основният цикъл на Kivy приложението.

* + 1. gol\_base.py, baseClass (част 1)

Класът *baseClass* е основният клас на приложението. Той съдържа както логиката за прилагането изпълнението на играта, така и всички функции свързани с обектите от потребителския интерфейс. Той наследява оформлението *BoxLayout* с вертикална ориентация, което значи че обектите в него, които имат визуално представяне, ще са разположени един под друг. Този клас е и коренът на дървото описващо потребителския интерфейс в *gol.kv*; в този файл обращенията към него са с името *root*, което е запазено в езикът kivy, за такъв тип класове. Проследявайки gol.kv, където клоните на дървото са означени с табулация, *baseClass* има две директни деца - *ScatterPlaneLayout* (фиг. 1, 1) с *id layoutS* - в него е решетката с клетките на играта - и *GridLayout* (фиг.1, 2)*,* съдържащ менюто под него.

Преди конструкторът на *baseClass* се декларират неговите Kivy свойства (properties), които ще разгледаме по-късно. Самият конструктор приема две променливи - *HALF\_LENGTH* и *SQ\_SIZE* - които определят в пиксели големината на цялата решетка с клетки и размерът на една клетка. Инициализираме също и два основни контейнера. *allCells* е двумерен масив, където ще се пазят референции към графичните инструкции за живите клетки или *None* ако съответната клетка е мъртва. Тази конструкция позволява лесно преобразуване между индексите на масива и координати на клетка от *layoutS. aliveCells* e сет, който ще съдържа кортежи (tuple) с двата индекса от *allCells* на всички живи клетки за бърза проверка дали дадена клетка е жива или не. Останалата част от конструктора се грижи за инициализирането на *layoutS* (който е деклариран само в .kv файла).

* + 1. layoutS, planeTracker, scatter\_transforms.py

Основната функция на *layoutS* е да извършва трансформациите на решетката - като местене и увеличаване/намаляване на мащаба. Неговият клас - *ScatterPlaneLayout* - извършва това автоматично, но в текущото приложение е променено кога тези трансформации се извършват. Файлът *scatter\_transforms.py* съдържа функциите за извикването на тези трансформации както са дефинирани в библиотеката Kivy с промените, че движението може да се извършва само при теглене с дясното копче на мишката (по подразбиране може да е с всяко движение), а промяната на мащаба може да се извършва чрез колелото на мишката (по подразбиране се извършва с мултитъч събитие). При инициализация на *baseClass* функциите от този файл презаписват тези по подразбиране на *layoutS*. В *Canvas* на *layoutS*  се рисуват само линиите на решетката на автомата, а самите живи клетки се нанасят в обект в него - *planeTracker* - който се създава също в конструктора на *baseClass*  и се прикача към този *layout* (така трансформациите на *layoutS* се прилагат и върху него).

Класът *planeTracker*(фиг. 1, 1.1)служи за изобразяване на живите клетки на автомата. Той се инициализира също в *init* на *baseClass* със същите размери като тези на *layoutS*  и по този начин координатите на обектите и събитията предизвикани от потребителя (touch event) съвпадат. Началната точка на тези координати (0,0) е в долният ляв ъгъл на видимата част на *layoutS*  при създаване на прозореца, за това на практика биват и отрицателни (за това за построение на линиите и преобразуване на координати и индекси се използва половината от дължината). В него са предефинирани методите, които съществуват за всеки обеkт в Kivy, *on\_touch\_down* и *on\_touch\_move.* Те по подразбиране се задействат при всяко такова събитие, получавайки профила на събитието, който съдържа информация като кое копче е натиснато, координати на курсора и тн. (аргумента *touch*). Ако това събитие е в границите на *planeTracker* и е породено от натискане на лявото копче на мишката, то се обработва от този обект и на решетката се рисува или трие клетка. В този случай методите връщат *True* и събитието престава да се предава, в противен случай методите връщат съответния метод на класа *Widget*, който наследяват и по този начин събитието се предава за обработка на *layoutS*  и/или впоследствие на останалите обекти - това е т.нар. *event bubbling,* описан в II b.

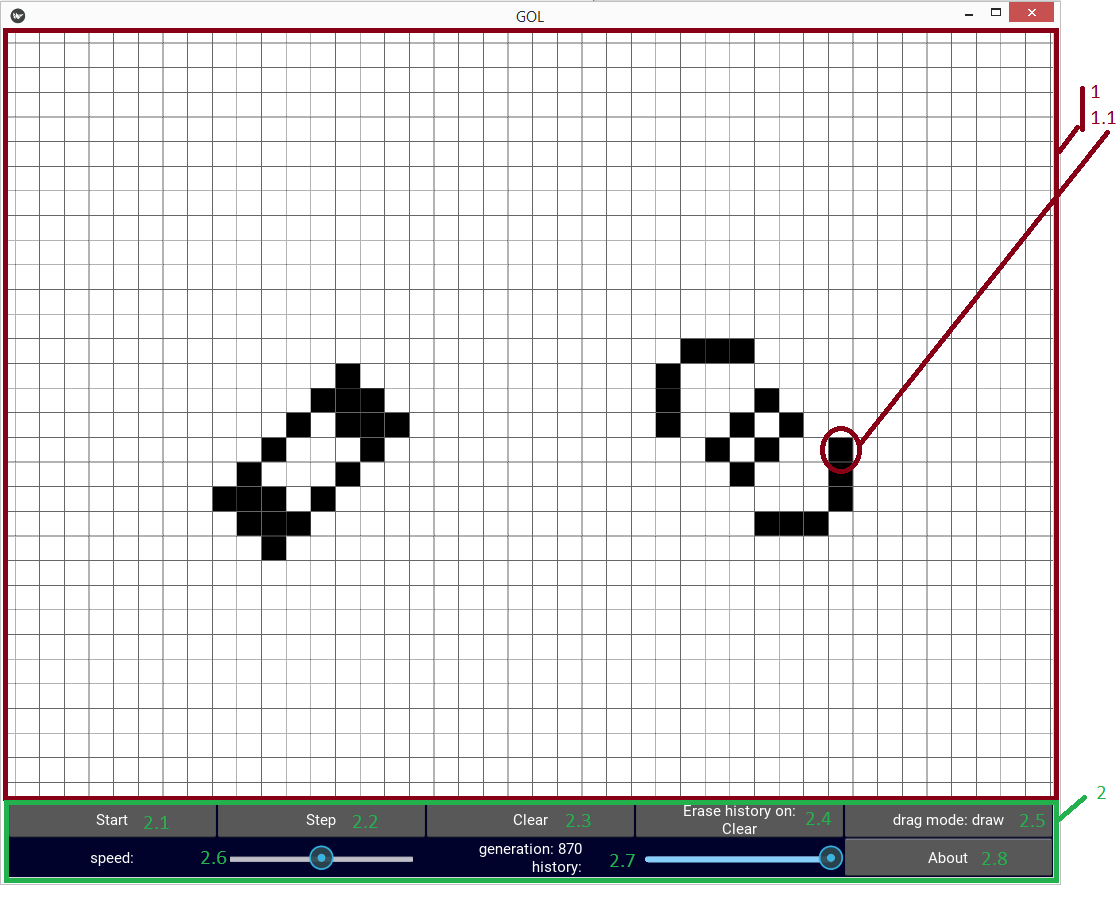
* + 1. gol\_base.py, baseClass (част 2)

С това приключваме инициализацията на *baseClass* и можем да преминем към методите му. За да илюстрираме гореспоменатите свойства (Properties) в Kivy ще започнем с работата на метод свързан с един бутон. Всички обекти от потребителския интерфейс на менюто, описани по-долу, са декларирани само в gol.kv и свързани с методи на *baseClass.*

Методът *runToggle* контролира пускането и спирането на автомата. Той се изпълнява при натискането на първият бутон от менюто (фиг. 1, 2.1) и е „прикачен“ към него в .kv файлът: *on\_press: root.runToggle()*. Методът проверява дали симулацията вече въври чрез свойството на *baseClass running,* което е *BooleanProperty* (тези свойства са типизирани при създаването си) и променя стойността на това свойство на противоположната й. Това от своя страна създава събитие, че тази стойност е променена и характеристиките свързани с нея в .kv файлът се актуализират (автоматично от библиотеката) - например текстът на бутона се сменя от „Start” на „Stop”, защото е свързан с това свойство на *baseClass (text:"Stop" if root.running else "Start")*. Основното действие на този метод е да насрочи (или преустанови) цикличното изпълнение на метода *step*, който съдържа основната логика на автомата, чрез обекта от Kivy Clock: *Clock.schedule\_interval(self.step, self.runDelay)*. Също има и някои действия относно историята, която ще разгледаме накрая. Честотата на този интервал се определя от целочисленото свойство *runDelay,* което се контролира от слайдер (фиг. 1, 2.6). Единственото специфично за този слайдер е, че ако бъде променен докато автоматът „върви“, гореописаният интервал се спира и пуска отново, за да се приложи новата честота.

Методът *step* прилага правилата на автомата, за да получи следващото състояние на решетката от клетки и може да се изпълнява продължително (както е описано по-горе) или еднократно при натискане на бутона Step (фиг. 1, 2.2), който е изключен ако свойството *base.running* е *True*. При извикването се итерира през всички живи клетки и в речник с ключ индексите, които съседите им имат в *allCells* (което съвпада и с формата на елементите от сета *aliveCells* както беше описано по-горе), се добавя единица. Използвана е структурата *defaultdict* със стойност по подразбиране 0 - така се избягва проверка дали дадената клета вече е била добавена (т.е. дали съществува този ключ). След като са изброени всички съседи по този начин, се итерира през всички клетки, които могат да се променят (т.е. живите клетки и мъртвите клетки които имат живи съседи) и се прилагат правилата на играта, като промените се нанасят в *Canvas* на инстанцията на *planeTracker,* която се казва *drawPlane* в този клас. Единствената проверка спрямо размерите на *drawPlane*, която трябва да се извърши е при добавянето на нова жива клетка, дали тя няма се окаже извън границите му.

Историята на предишните състояния на автомата представлява масив, който пази копия от 99-те последни стойности на *aliveCells* и началното състояние на автомата, защото както беше споменато в описанието на играта, не е възможно от текущото състояние еднозначно да се определи какво е било миналото. Този масив се актуализира в метода *step*. Паралелно с него има и масив, който пази номера на поколението на съответното състояние (всяко следващо състояние на автомата е ново поколение). Движението в историята се извършва със слайдер (фиг. 1, 2.7), чиято максимална стойност се променя според дължината на масива - заради това масивът е свойство - *ListProperty* - на *baseClass*. Историята се трие или при повторно пускане на автомата или при изтриване на всички клетки от метода clear (фиг. 1, 2.3), като това се определя от *BooleanProperty,* контролирано от бутон (фиг 1, 2.4). С този слайдер можем да илюстрираме и още едно свойство на *event bubbling* - тъй като той получава събитието *on\_touch\_move* по-рано от инстанцията на *planeTracker,* например, е нужно да проверим дали събитието е започнало върху него преди да го обработим в неговия *on\_touch\_move* метод. Това става като на метода прикачен към него (root.*browseHistory* в .kv) подадем инстанцията на този слайдер и профила на събитието и проверим координати им се припокриват с метода *collide\_point* (който е на наследения клас *Widget*).

Последните два бутона от менюто са сравнително прости. Единият (фиг. 1, 2.5) контролира чрез *BooleanProperty* дали при движение с ляво копче на мишката в *drawPlane* се трият или рисуват нови клетки. Вторият (фиг. 1, 2.8) създава при натискане прост прозорец с малко информация за приложението. Създаването и унищожаването на прозореца се контролира автоматично от библиотеката; може да се отбележи само, че текста, който използва е стилизиран с специфични за Kivy *inline* стилове като *[size=14]*, който могат да се интерпретират от обектите тип *Label* когато в конструктора им се добави флагът *markup=True*.

Фифура 1 - Графичен Интерфейс

1. Производителност

Както може да се съди по метода *baseClass*.*step*. С увеличаването на броя на живи клетки, скоростта на изпълнението на цикъла за преобразуване на автомата спада. Без да се обвързваме с конкретна оценка са сложността на този метод, от практически измервания си личи, че е по-лоша от линейна. Резултатите са представени в фигура 2, измерванията за всяка бройка живи клетки са направени между 1 и 8 пъти (средно 5) и е взет средния резултат. Използван е метода *default\_timer* oт модула *timeit*. Под 2000 живи клетки не може да се уести забавяне, защото максималната честота на интервала е 0.1 секунди. Около 4-5000 живи клетки се наблюдава и лаг при трансформация на *layoutS*. Тези резултати не са особено добри, но самите свойства на автомата са такива, че неограничен растеж на живите клетки се среща рядко (а и в тези частни случаи растежът е най-често логаритмичен или линеен) - най често се стига до осцилиращи или непроменяеми състояния.

Фигура 2 - живи клетки/време на step в секунди