Теплицький О. І., Теплицький І. О.
Криворізький національний університет

Об’єктно-орієнтовані імітаційні моделі: конструювання «Життя»

Змістова частина методичної системи навчання студентів природничих спеціальностей об’єктно-орієнтованого моделювання [1] включає широкий спектр задач із різних предметних галузей і передбачає опанування технології об’єктно-орієнтованого моделювання у середовищах для навчальних досліджень [2].

У першому модулі «Вступ до об’єктно-орієнтованого моделювання» розглядаються базові поняття та уявлення курсу (поняття про моделювання, види моделей, об’єктно-орієнтоване моделювання; об’єктно-орієнтоване програмування та об’єктно-орієнтовані мови; абстракція, інкапсуляція, спадкування, поліморфізм як основа об’єктно-орієнтованої методології; етапи об’єктно-орієнтованого моделювання: об’єктно-орієнтований аналіз, проектування, обчислювальний експеримент та аналіз його результатів) і виконується огляд середовищ об’єктно-орієнтованого моделювання (зокрема, виділяються універсальні середовища моделювання, середовища для конструювання динамічних моделей та середовища для конструювання імітаційних моделей). На підставі аналізу придатності середовищ для моделювання різних класів моделей пропонується у навчанні студентів спеціальностей «Хімія», «Біологія» та «Фізика»:

а) при розгляді динамічних моделей послугуватися середовищами об’єктно-орієнтованого моделювання VPython та Squeak як основними, а Sage, Alice та NetLogo – як додатковими;

б) при розгляді імітаційних моделей скористатися середовищами об’єктно-орієнтованого моделювання Alice та NetLogo як основними, а Sage, VPython та Squeak – як додатковими.

У процес навчання об’єктно-орієнтованого моделювання студентів спеціальностей «Математика», «Інформатика» середовище Sage із додаткового переноситься у основні. Це не вимагає додаткових витрат часу через спільність мови ООП у середовищах VPython та Sage.

Другий модуль «Об’єктно-орієнтовані динамічні моделі» присвячений розгляду динамічних моделей математичної екології (динаміка одновидової популяції, модель «Хижак-жертва», вікові моделі Леслі), класичної механіки (динаміка коливних систем, рух тіл в полі сили тяжіння, моделювання аеродинамічних об’єктів та явищ), молекулярної фізики і фізики твердого тіла (атомна та молекулярна динаміка) та електродинаміки (рух заряду в електричному та магнітному полях).

До третього модуля «Об’єктно-орієнтовані імітаційні моделі» включені моделі кліткових автоматів (модель поширення чуток, модель «Хижак-жертва», модель поширення пожежі, гра «Життя»), стохастичні моделі (модель броунівського руху, модель відмов обладнання, модель росту кристалу), моделі фрактальних об’єктів та процесів (моделі регулярних фракталів, задача перколяції, моделі електролізу, модель утворення берегової лінії).

В третьому модулі курсу розглядається ряд моделей, що мають загальну назву кліткові автомати. Кліткові автомати були вперше розглянуті Дж. фон Нейманом та С. Уламом в 1948 р. як можлива ідеалізація біологічного самовідтворення. Зацікавленість ними обумовлена, головним чином, тим, що більшість з них є прикладами простих динамічних систем (що розглядаються у другому модулі курсу), які дають впорядковані візерунки, що виникають із випадкових початкових умов. Тривимірний характер середовища Alice надає можливість впевнитися в тому, що просторові візерунки багатьох кліткових автоматів нагадують візерунки, які можна спостерігати в природних явищах (поширений приклад – ріст кристалів). Саме тому кліткові автомати розглядають як корисні й цікаві комп’ютерні моделі, які зберігають привабливі можливості для опису складних систем.

Кліткові автомати являють собою моделі фізичних систем, в яких простір і час дискретні, самі ж фізичні величини (у разі потреби) набувають скінченної множини дискретних значень. Для прикладу уявимо регулярну решітку клітин, кожна з яких може знаходитися у скінченному числі можливих станів, наприклад, 0 або 1. Стан системи повністю визначається значеннями змінних в кожній клітині.

Важливими особливостями кліткових автоматів є такі:

1. Стан кожної клітини поновлюється за скінчену кількість кроків (зокрема, на кожному кроці).

2. Ці поновлення значень змінних в кожній клітині відбуваються одночасно (паралельно), виходячи із значень змінних на попередньому кроці.

3. Новий стан клітини залежить лише від локальних значень у сусідніх клітинах.

Для характеристики кліткового автомата зазначимо такі його особливості:

1. Розташування клітин утворює деяку геометричну фігуру (у більшості інших випадків обирають прямокутну решітку, що складається з квадратів).

2. За заданою схемою необхідно визначити те оточення, яке дана клітина «вивчає» при обчисленні свого наступного стану.

3. Кількість станів, яких може набувати клітина, буває різною: Дж. фон Нейман побудував систему, «здатну» до самовідтворення, у якій клітини мали 29 можливих станів, однак більшість автоматів значно простіші.

4. Головне джерело змін у світі кліткових автоматів – це величезна кількість можливих правил для визначення наступного стану клітини, виходячи із станів її сусідів у даний момент.

Всі досліджені правила зміни стану клітини можна поділити на чотири класи:

1) правила, за яких еволюція приводить систему до стійкого та однорідного стану;

2) правила, що ведуть до появи простих структур (стійких або періодичних), які в будь-якому випадку залишаються ізольованими одна від одної;

3) правила, які ведуть до появи хаотичних візерунків;

4) правила, які породжують структури істотної просторової та часової складності.

Мабуть, найвідомішим клітковим автоматом є гра «Життя», запропонована у 1970 р. Дж. Х. Конвеєм. Ситуації, що виникають у процесі гри, нагадують реальні процеси, що відбуваються при зародженні, розвитку та загибелі колонії організмів. Умови народження та загибелі визначаються виключно взаємним розташуванням учасників, а правила гри жорстко визначають, де та коли відбуваються народження та смерть. Гра складається з «циклів життя», або з послідовності дискретних кроків, за допомогою яких імітується зміна поколінь.

Для реалізації гри «Життя» Дж. Х. Конвей спочатку використовував скінченну шахову дошку, у клітинках якої розташовувались шашки одного кольору. Проте на краях дошки правила гри не спрацьовували, тому йому довелося вдатися до концепції нескінченної (замкненої з усіх боків) дошки, що була моделлю планети, повністю вкритої океаном. Острови в океані розташовані на рівних відстанях уздовж меридіанів і паралелей. На кожному острові може мешкати лише одна істота – конвік. Найближчими до кожного острова є завжди 8 сусідніх островів.

Для моделювання простору гри «Життя» оберемо морську сцену із переліку шаблонів, що їх надає Alice. Аналогом острова може бути будь-який кулястий об’єкт, наполовину занурений в море (наприклад бейсбольний м’яч, який є в галереї Alice).

Для моделювання оберемо фрагмент зі 100 островів (сітка розміром 10×10, рис. 1).

Для забезпечення умови необмеженості обраного фрагменту вдамося до «склеювання» його меж, а саме: вважатимемо, що верхня і нижня межі фрагменту межують один з одним (те ж саме відбувається з лівою і правою межами). В результаті з’являється нова властивість: переміщення деякого об’єкта за праву межу поля приводить до появи цього об’єкта біля лівої межі і навпаки. Так само переміщення об’єкта під нижню межу поля приводить до його появи біля верхньої межі і навпаки. У такий спосіб виявляється можливим замість необмеженого поля обійтися полем скінченного і не дуже великого розміру.

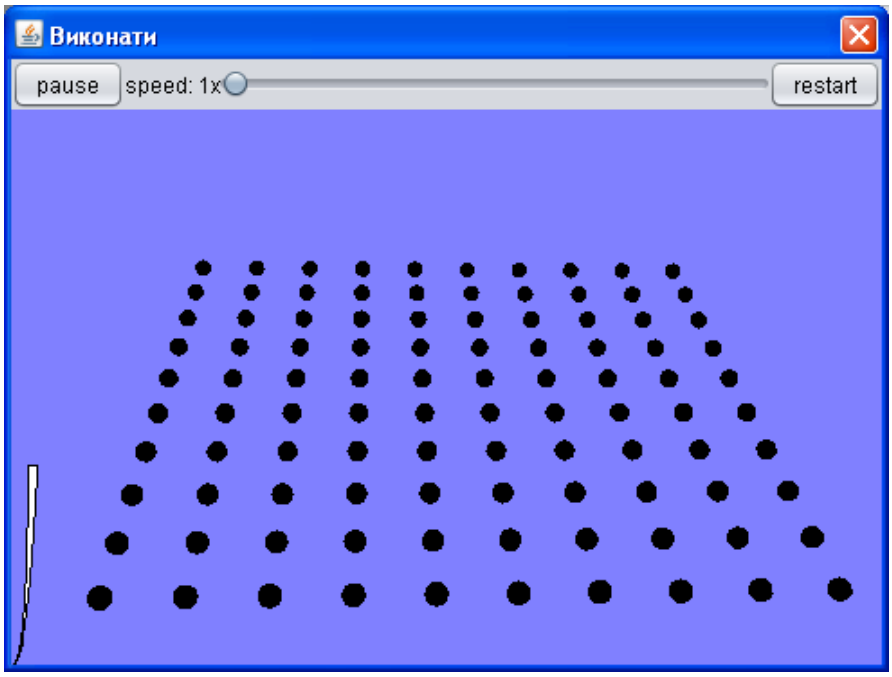

Рис. 1

Спочатку заповнимо морську сцену сотнею однотипних об’єктів. Для цього достатньо створити один об’єкт класу *Baseball*. За замовчуванням ім’я створеного похідного класу буде *MyBaseball*. Назвемо його *convik00*. Змінимо ім’я класу *MyBaseball* на *Convik*, обравши пункт *Rename* (перейменування) у полі «*class:*», після чого оголошуємо створений об’єкт *convik00* вибором пункту «*Declare new instance…*», надаючи кожному новому об’єкту ім’я «*convikXY*», де *XY* – число від 01 до 99. Всі клоновані у зазначений спосіб об’єкти мають спільне розташування та інші властивості, розрізнюючись лише ім’ям.

Клас *Convik* зараз є ідентичним класу *Baseball* у всьому, за виключенням імені. Внесемо до нього наступні зміни:

1. Додамо дві властивості класу (рис. 2):

– *islive* – логічна змінна: значення *true* означає, що острів заселений, *false* – що не заселений;

– *isbelive* – логічна змінна: значення *true* для якої означає, що після застосування правил гри «Життя» острів має бути заселений, а *false* – звільнений.

Рис. 2

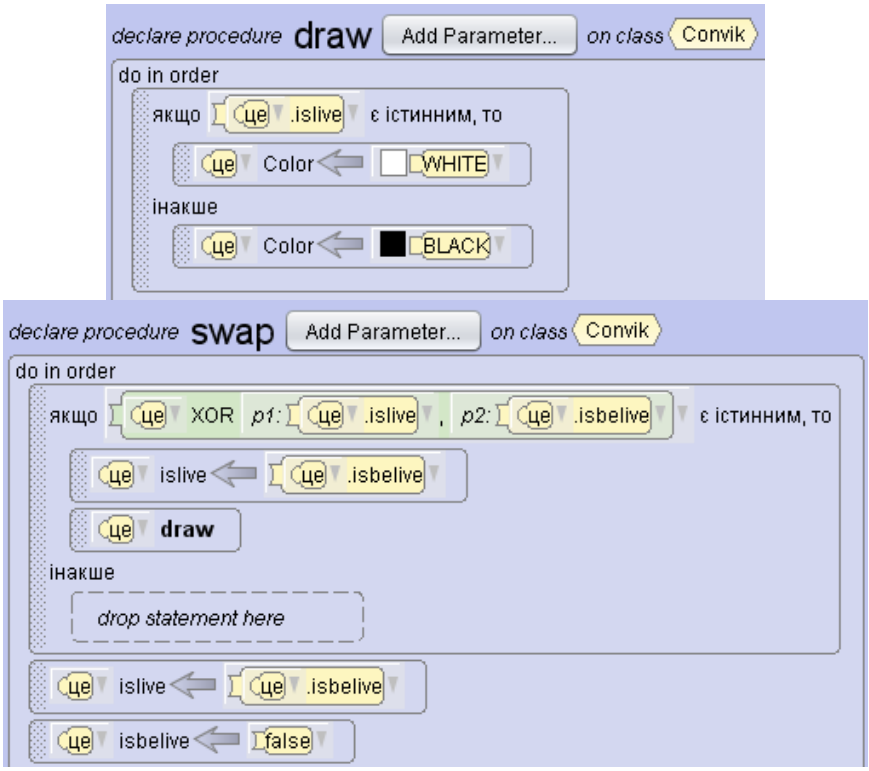

Рис. 3

2. Додамо дві процедури (рис. 3):

– *draw* – обирається колір острова в залежності від того, острів заселений (він білий), чи ні (чорний);

– *swap* – у залежності від того, чи змінюється стан острова (із заселеного на незаселений чи навпаки), змінюватиме його колір.

3. Додамо логічну функцію *XOR*, що повертатиме значення *true* у тому випадку, коли її параметри різняться, і *false* навпаки (рис. 4). Ця функція є допоміжною для процедури *swap*.

Також змінимо конструктор класу *Convik*, задавши за замовчуванням такі значення властивостей: *islive* – *false* (незаселений), *isbelive* – *false* (не претендує на заселення), тому колір острова встановимо чорний. Додатково в конструкторі визначимо «обробник» миші, який при виборі об'єкта мишею змінюватиме стан із заселеного на незаселений (і навпаки) одночасно, відповідно, змінюючи колір об'єкта (рис. 5).

Створений клас *Convik* може бути збережений за допомогою панелі класів (*Save Convik As…*) у вигляді бінарного файлу класу Alice (Convik.a3c), який, за замовчуваенням, зберігається в папці My Classes).

Зв'язування об'єктів класу *Convik* зі сценою виконується автоматично при додаванні об'єктів до сцени: всі додані об'єкти стають властивостями сцени – стандартні об'єкти сцени, такі як *sunLight* (освітлення), *seaSurface* («морська поверхня»), *camera* та додані 100 об'єктів класу *Convik* (рис. 6). Для роботи з однотипними об'єктами доцільно використати масив, який створимо як властивість об'єкта *scene* під ім'ям *convfield* (рис. 7). Це буде масив із 100 створених об'єктів (з індексами від 0 до 99).

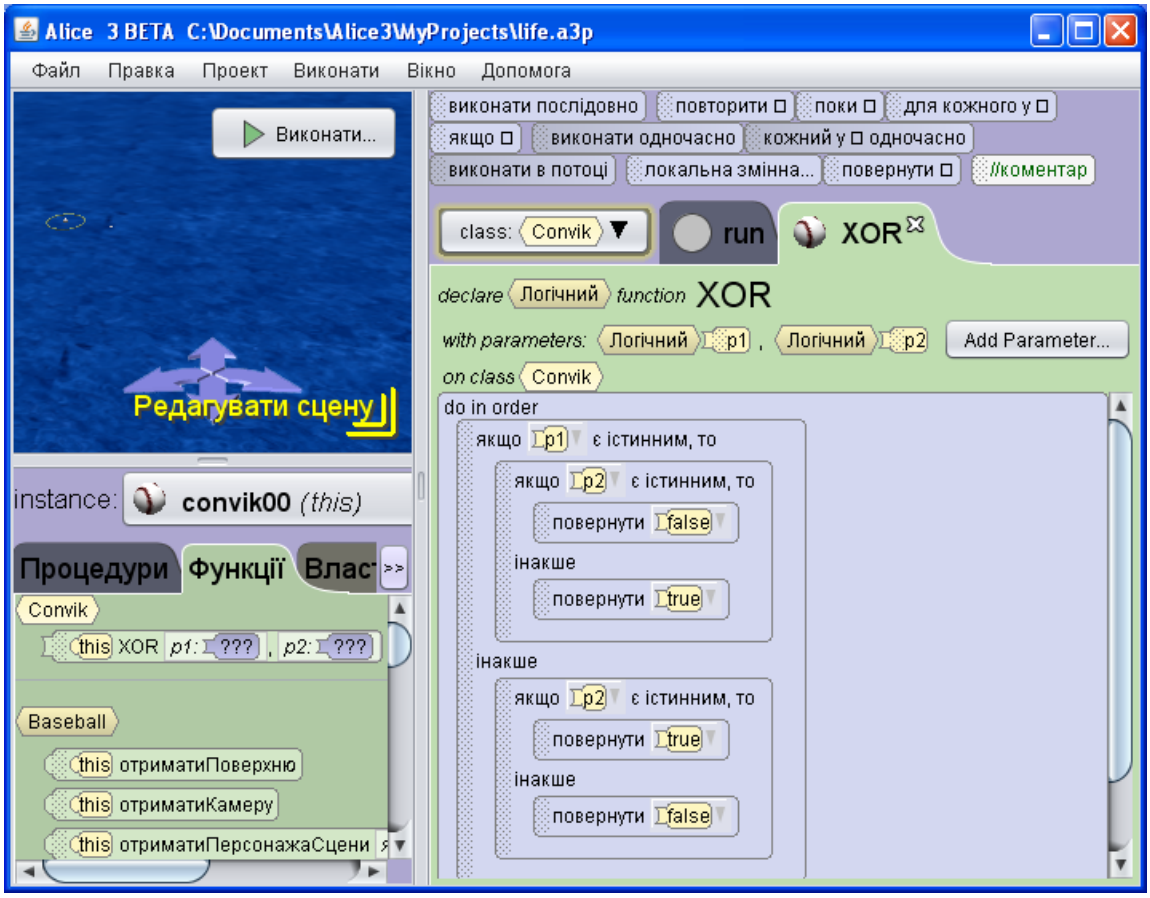

Рис. 4

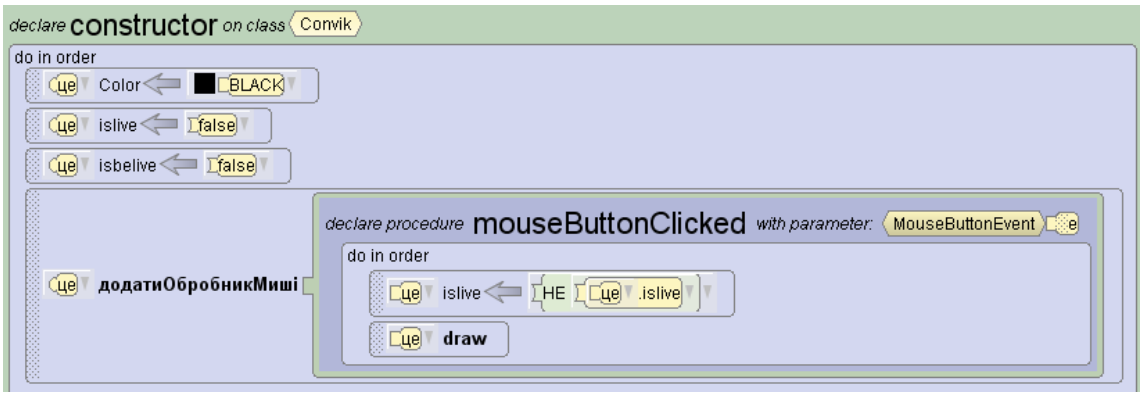

Рис. 5

За допомогою конструктора класу *MyScene* (успадкованого від *Scene*) викликається процедуру *performMySetUp*, яку використаємо для початкового налаштування створених об'єктів (рис. 8):

1. Якщо при створенні конвіків вони «розбіглися» на сцені (через невдале ручне розташування) – перемістимо їх до якогось одного за допомогою стандартної процедури *переміститиДо*.

2. Встановимо прозорість морської поверхні у 0.

3. Зорієнтуємо камеру так, щоб сцену було краще видно (за методами *переміститиУНапрямку* та *повернути*).

Розташуємо острови на паралелях та меридіанах шляхом переміщування вліво та вперед на відстань, що відповідає положенню острова в розглядуваному фрагменті: якщо *step* – це відстань між островами на паралелі (меридіані), то уліво змвстимо на *step*номер рядка острову* у фрагменті архіпелагу, а вперед – на *step*номер стовпця*.

Остання дія у *performMySetUp* – це додавання «обробника» клавіатури: за натисканням на будь-яку клавішу викликатимемо процедуру *onkeypress*, за якою реалізовуватиметься один крок еволюції (одна ітерація) гри «Життя». Враховуючи, що під час виконання кроку еволюції втручання користувача у роботу моделі є недоцільним, виконаємо блокування «обробки» натискань клавіш за допомогою нової властивості класу *MyScene* – логічної змінної *canenter* із початковим значенням *true* (рис. 7). Якщо ця властивість дорівнює *true*, можна виконувати новий крок еволюції, інакше варто повідомити користувача, що модель зайнята роботою. Для цього перед та після виклику процедури виконання кроку еволюції змінимо значення властивості *canenter* на протилежне.

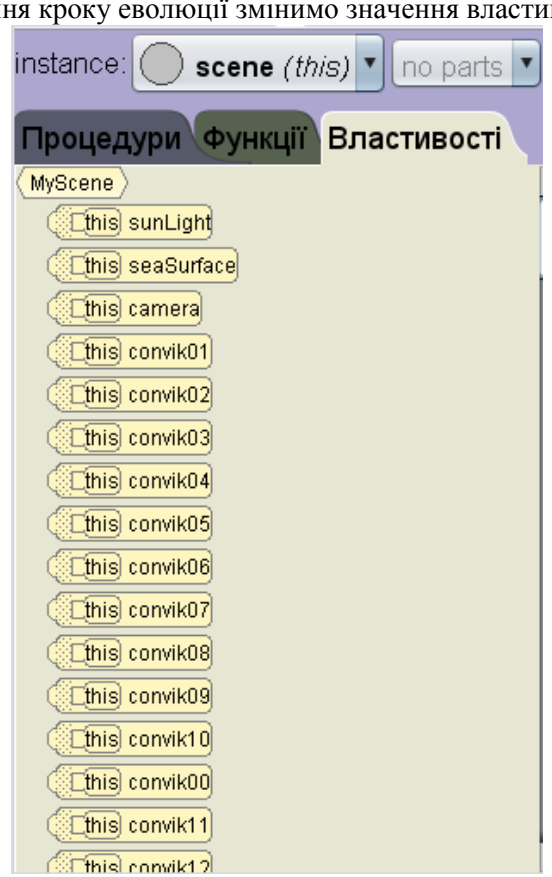

Рис. 6

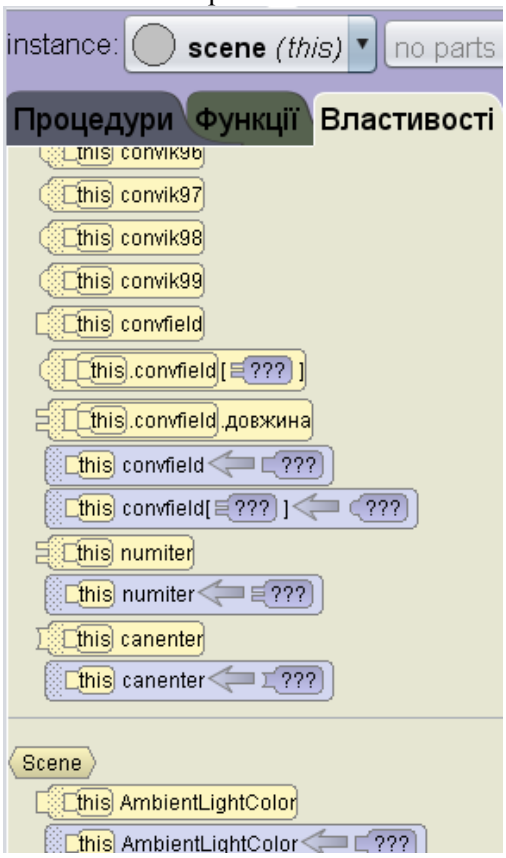

Рис. 7

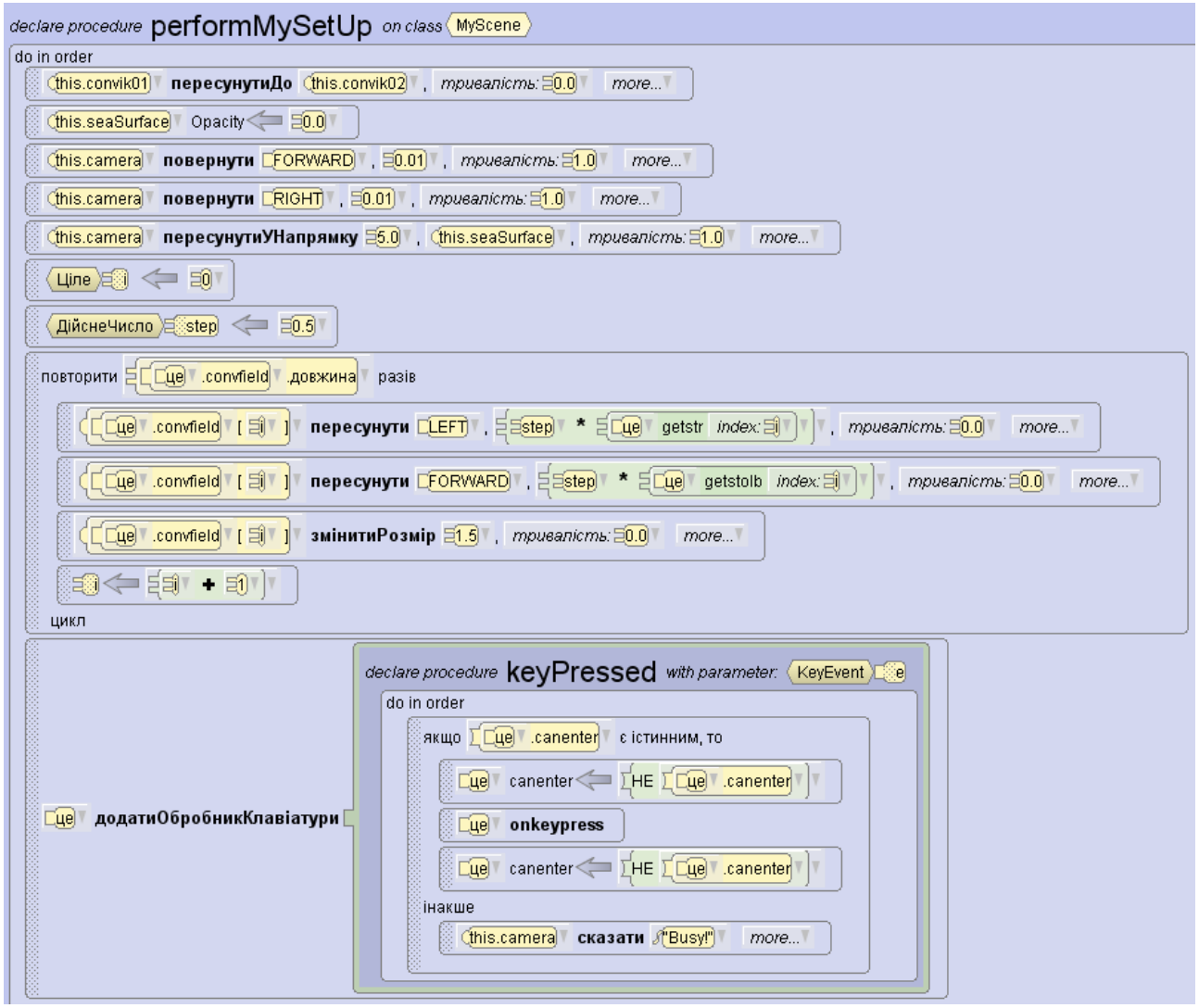
declare procedure performMySetUp on class MyScene
do in order
this.convik01 пересунутиДо this.convik02 , тривалість: 0.0 more...
this.seaSurface Opacity 0.0
this.camera повернути FORWARD , 0.01 , тривалість: 1.0 more...
this.camera повернути RIGHT , 0.01 , тривалість: 1.0 more...
this.camera пересунутиУНапрямку 5.0 , this.seaSurface , тривалість: 1.0 more...
Ціле i 0
ДійсневЧисло step 0.5
повторити це .convfield .довжина разів
це .convfield [i] пересунути LEFT , step * це getstr index: i , тривалість: 0.0 more...
це .convfield [i] пересунути FORWARD , step * це getstolb index: i , тривалість: 0.0 more...
це .convfield [i] змінитиРозмір 1.5 , тривалість: 0.0 more...
i i + 1
цикл
це додатиОбробникКлавіатури
declare procedure keyPressed with parameter: KeyEvent e
do in order
якщо це .canenter є істинним, то
це canenter НЕ це .canenter
це onkeypress
це canenter НЕ це .canenter
інакше
this.camera сказати "Busy!" more...

Рис. 8

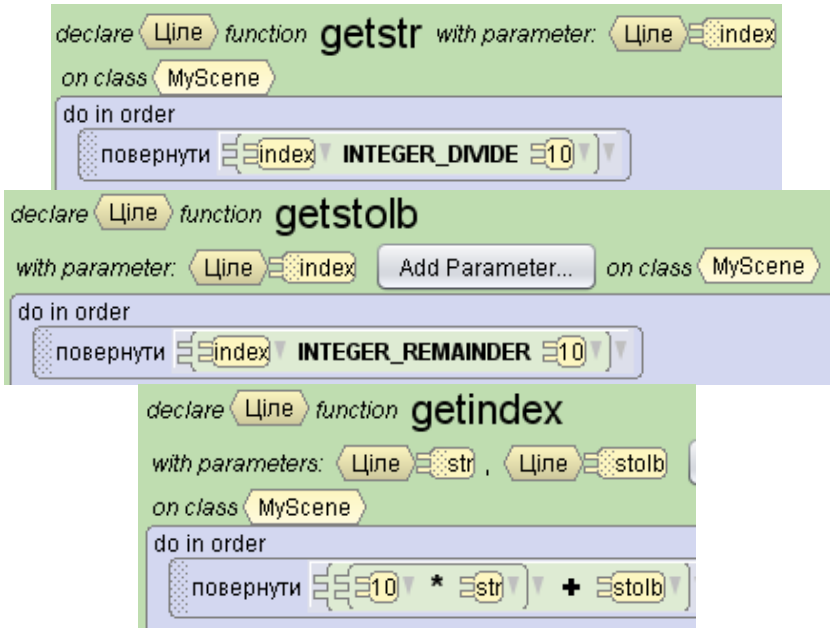
declare Ціле function getstr with parameter: Ціле index
on class MyScene
do in order
повернути index INTEGER_DIVIDE 10
declare Ціле function getstolb
with parameter: Ціле index Add Parameter... on class MyScene
do in order
повернути index INTEGER_REMAINDER 10
declare Ціле function getindex
with parameters: Ціле str , Ціле stolb
on class MyScene
do in order
повернути 10 * str + stolb

Рис. 9

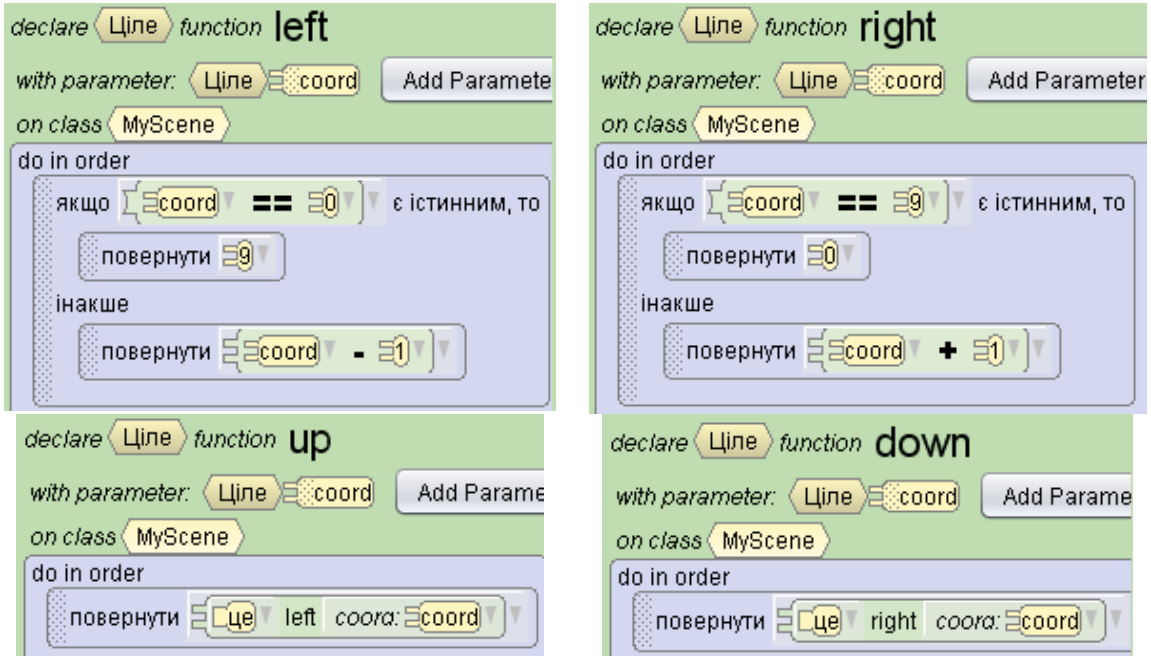

Рис. 10

Для того, щоб сигналізувати про завершення кроку еволюції, створимо ще одну властивість: *numiter* – цілочисельну змінну із початковим значенням 1.

Для роботи процедури *onkeypress* створимо наступні додаткові функції (рис. 9):

– *getstr* та *getstolb* – повертають номер рядка та стовпця відповідно за одновимірним поданням індекса елемента;

– *getindex* – повертає одновимірний індекс за номером рядка та стовпця.

Необхідність у цих функціях зумовлена відмінністю між внутрішнім поданням об’єктів масиву (одновимірним від 0 до 99) та зовнішнім їх поданням на екрані (двовимірна сітка 10×10 з індексами рядків та стовпців від 0 до 9).

– *left*, *right*, *up*, *down* – набір функцій для реалізації склеювання меж фрагменту (рис. 10);

– *getneigbcount* – повертає кількість заселених островів, що межують із даним: на сітці це такі острови, що розташовані зліва та вгорі, вгорі, справа та вгорі, зліва, справа, зліва та знизу, знизу, справа та знизу (рис. 11).

При натисканні будь-якої клавіші за процедурою *onkeypress* виконуються наступні дії (рис. 12):

1) встановлюється номер кроку еволюції (номер ітерації) у 0;

2) для кожного острова:

а) підраховується кількість сусідніх заселених островів;

б) визначається значення властивості *isbelive* у такий спосіб: якщо острів незаселений і кількість сусідніх заселених островів дорівнює трьом, то острів стає кандидатом на заселення; інакше, якщо острів заселений та кількість сусідніх заселених островів – 2 чи 3, то острів зберігає свій статус заселеного; інакше – наступний статус острова буде «незаселений».

3) для кожного острова викличемо процедуру *swap*.

4) виведемо номер кроку еволюції (номер ітерації) та збільшимо його на одиницю.

На прикладі цієї моделі можна проілюструвати основні принципи об’єктно-орієнтованого програмування:

1) *наслідування* – існуючих характеристик класів недостатньо для побудови моделі, тому довелося не просто створити класи-нащадки, й змінити їх функціональність у порівнянні з класами-батьками: додати нові дані (властивості) та методи (процедури та функції);

2) *інкапсуляція* – дії, що є специфічними для класу, реалізуються у його властивостях, доступ до яких з інших класів можливий лише через створення та виклик їх методів;

3) *абстрагування* – реалізується функціонально-процедурною декомпозицією за методами класів у такий спосіб, щоб кожен метод був осяжним та зрозумілим, та створенням спеціалізованих класів (*Convik* та *MyScene*);

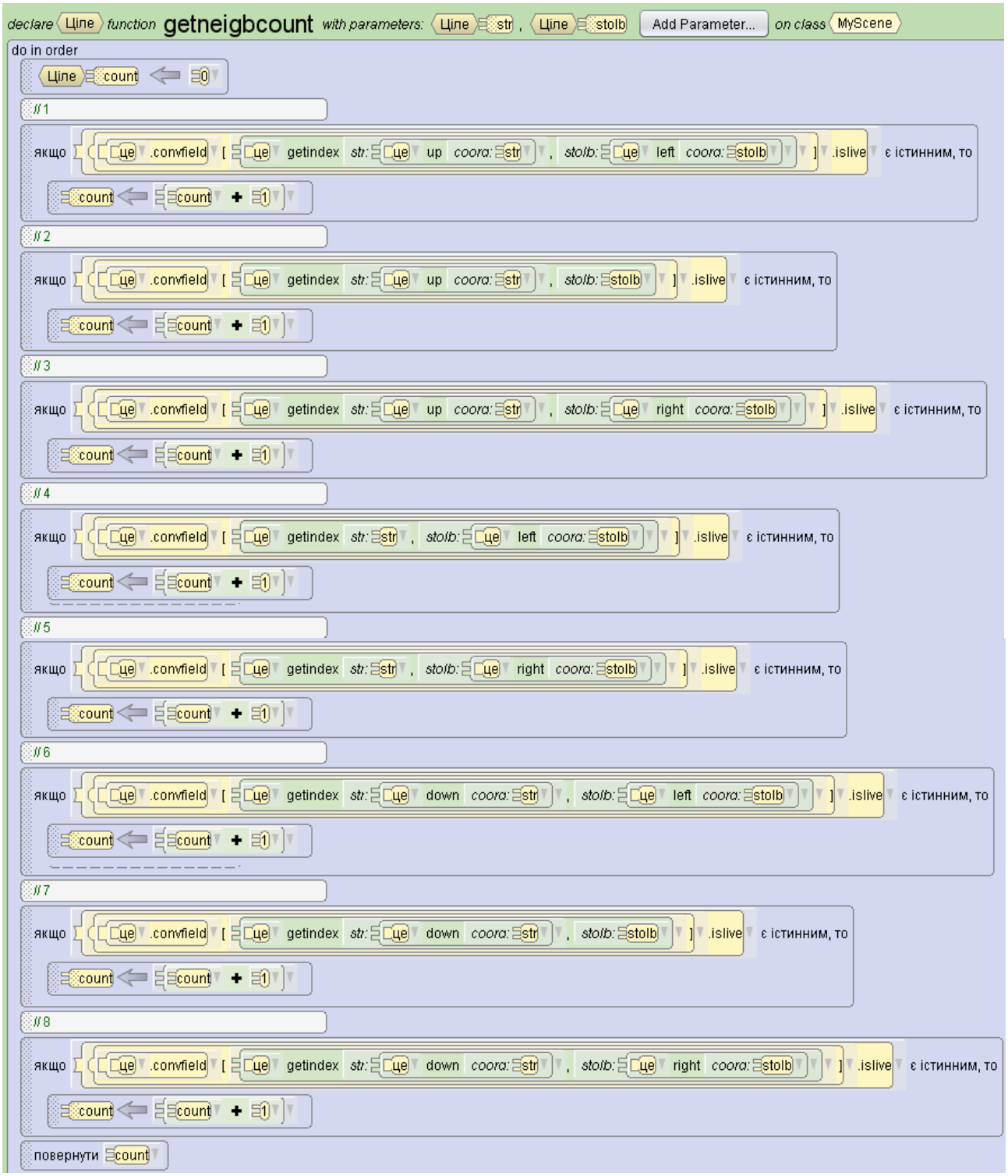
declare Ціле function getneigbcount with parameters: Ціле str , Ціле stolb Add Parameter... on class MyScene
do in order
Ціле count 0
// 1
якщо це .convfield [це getindex str: це up coora: str , stolb: це left coora: stolb] .islive є істинним, то
count count + 1
// 2
якщо це .convfield [це getindex str: це up coora: str , stolb: stolb] .islive є істинним, то
count count + 1
// 3
якщо це .convfield [це getindex str: це up coora: str , stolb: це right coora: stolb] .islive є істинним, то
count count + 1
// 4
якщо це .convfield [це getindex str: str , stolb: це left coora: stolb] .islive є істинним, то
count count + 1
// 5
якщо це .convfield [це getindex str: str , stolb: це right coora: stolb] .islive є істинним, то
count count + 1
// 6
якщо це .convfield [це getindex str: це down coora: str , stolb: це left coora: stolb] .islive є істинним, то
count count + 1
// 7
якщо це .convfield [це getindex str: це down coora: str , stolb: stolb] .islive є істинним, то
count count + 1
// 8
якщо це .convfield [це getindex str: це down coora: str , stolb: це right coora: stolb] .islive є істинним, то
count count + 1
повернути count

Рис. 11

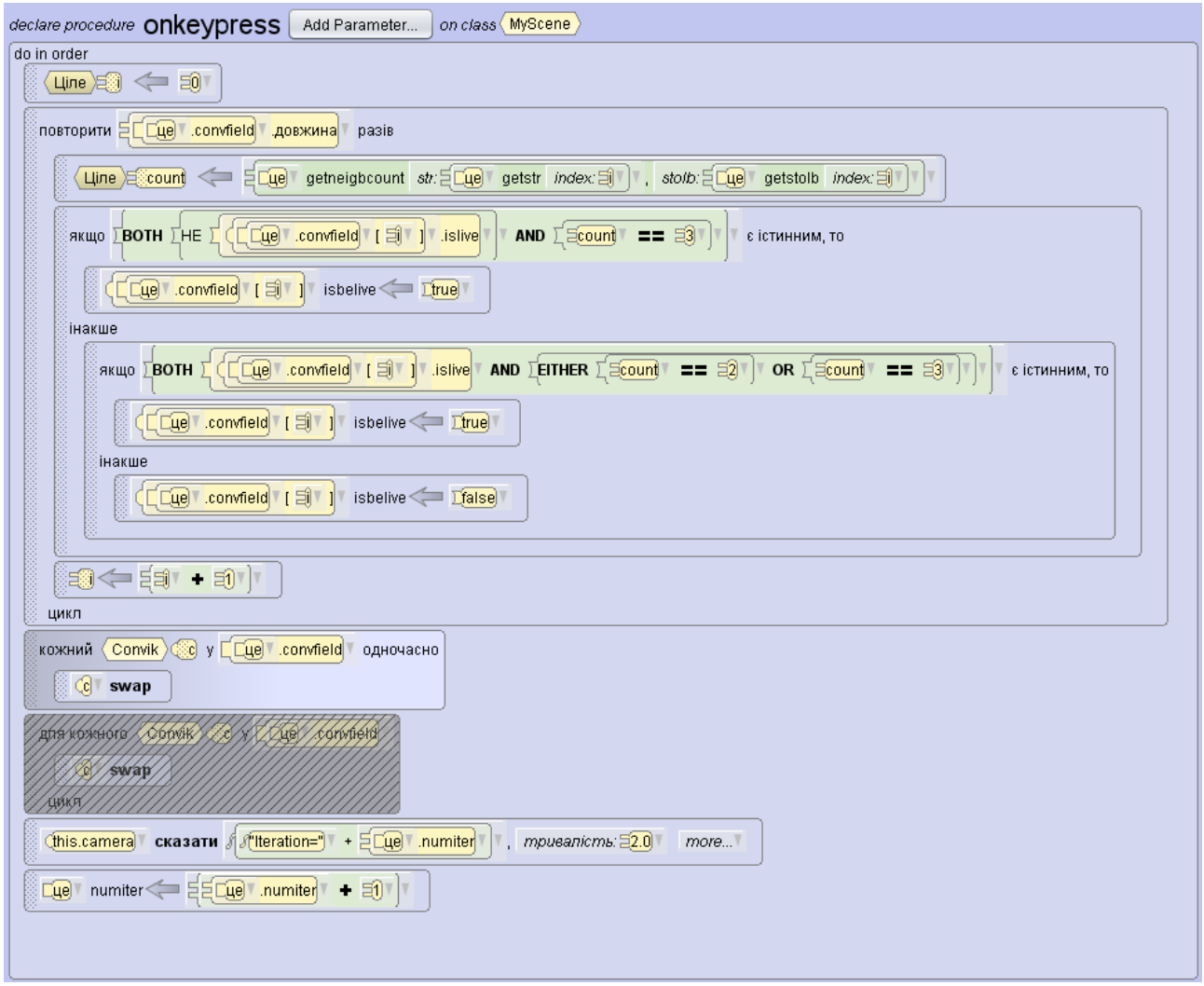

Рис. 12

4) *модульність* – реалізується через можливість зберігання та повторного використання розроблених класів у нових проектах;

5) *ієрархія* – реалізується у системі класів Alice (як вбудованих, так і створених користувачем), що походять від єдиного абстрактного класу *Object*;

6) *типізація* – реалізується будовою мови Alice та Java. Прикладом узгодження типів є можливість поєднання різних об’єктів процедурами *сказати* та *подумати*;

7) *паралелізм* – може бути реалізований у процедурі *onkeypress*: визначення того, який буде наступний стан острова (заселений, чи ні), не залежить від порядку опрацювання елементів;

8) *стійкість* – властивість об’єктів існувати в часі та у просторі забезпечується послугами середовища Alice (збереженням об’єктів у складі проекту) та його розміщенням у Internet.

В даній моделі відсутні приклади *поліморфізму* – можливості визначення об’єкту у процесі звернення до нього, проте враховуючи, що мова Alice є Java-подібною, а у Java всі методи є віртуальними, то й у Alice всі процедури та функції є поліморфними.

Ознакою «чистоти» застосування концепцій об’єктно-орієнтованого програмування при розробці даної моделі є також те, що не довелося створити жодного рядка у процедурі *run* класу *MyScene*.

Чи не найкращою ілюстрацією корисності концепції паралелізму є багатократне (до 100 разів) прискорення швидкості зміни стану острова простою заміною циклу «*для кожного об’єкту у масиві*» (показаний на рис. 12 як закоментований) на «*кожний об’єкт у масиві одночасно*» у функції *onkeypress*, адже при виклику процедури *swap* тривалість одночасного стану оновлення островів практично не відрізняється від тривалості оновлення стану одного острова.

Література

1. Теплицький О. І. Засоби навчання об’єктно-орієнтованого моделювання студентів природничих спеціальностей педагогічних університетів / О. І. Теплицький // Збірник наукових праць Кам’янець-Подільського національного університету. Серія педагогічна. – Кам’янець-Подільський : Кам’янець-Подільський нац. ун-т імені Івана Огієнка, 2011. – Вип. 17 : Інноваційні технології

управління компетентнісно-світоглядним становленням учителя: фізика, технології, астрономія. – С. 246-248.

2. Теплицький О. І. Об’єктно-орієнтоване моделювання в Alice. Частина 1 / О. І. Теплицький; за науковою редакцією академіка НАПН України М. І. Жалдака. – К. : НПУ імені М. П. Драгоманова, 2011. – 56 с.