МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет “Львівська політехніка”

*Кафедра САПР*



Розрахунково-графічнА робота

з дисципліни :

«Методи проектування мультиагентних систем»

Виконав:

ст. групи СПКм-12

Кикуш О.М.

Прийняв:

Романюк А. Б.

Львів 2015

**Мета роботи:** Ознайомитись з основними функціональними можливостями бібліотеки Flame GPU, освоїти сучасні методики та підходи щодо побудови мультиагентних систем за допомогою графічних процесорів (CUDA) та бібліотеки Flame GPU.

**Індивідуальне** **завдання**

Мультиагентні системи - це система, утворена декількома взаємодіючими інтелектуальними агентами. Мультиагентні системи можуть бути використані для розв'язання таких проблем, які складно або неможливо вирішити за допомогою одного агента або монолітної системи. Прикладами таких завдань є онлайн-торгівля, ліквідація надзвичайних ситуацій і моделювання соціальних структур.

Графічний процесор - окремий пристрій персонального комп'ютера або ігрової приставки, виконує графічний рендеринг. Сучасні графічні процесори дуже ефективно обробляють і зображують комп'ютерну графіку, завдяки спеціалізованій конвеєрній архітектурі вони набагато ефективніші в обробці графічної інформації, ніж типовий центральний процесор.

«Гра́ життя́» — клітинний автомат, вигаданий англійським математиком Джоном Конвейем (John Horton Conway) 1970.

Опис цієї гри було опубліковано в жовтневому випуску журналу Scientific American, в рубриці «Математичні ігри» Мартіна Гарднера (Martin Gardner).

Місце дії цієї гри — «всесвіт» — це площина, поділена на клітинки. Кожна клітинка на цій поверхні може знаходитись в двох станах: бути живою або бути мертвою. Клітинка має вісім сусідів. Розподіл живих клітинок на початку гри називається першим поколінням. Кожне наступне покоління утворюється на основі попереднього за наведеними нижче правилами.

**Правила**

* якщо у живої клітини два чи три сусіди – то вона лишається жити;
* якщо у живої клітини один чи немає сусідів – то вона помирає від «самотності»;
* якщо у живої клітини чотири та більше сусідів – вона помирає від «перенаселення»;
* якщо у мертвої клітини рівно три сусіди – то вона оживає.

Дані правила отримали назву генетичних законів Конвея, вони задовольняють три основні умови:

1. не має бути жодної початкової конфігурації, для якої існувало б просте доведення можливості необмеженого росту популяції; мають існувати такі початкові конфігурації, які заздалегідь володіють властивістю безмежно розвиватися;
2. мають існувати прості початкові конфігурації, які протягом значного проміжку часу ростуть, перетерплюють різноманітні зміни та закінчують свою еволюцію одним з трьох наступних способів:
   * 1. повністю зникають;
     2. переходять у стійку конфігурацію та перестають змінюватися взагалі;
     3. виходять у коливальний режим з певним періодом.

Гравець не бере прямої участі у грі, а лише розставляє початкову конфігурацію «живих» клітин, які потім взаємодіють відповідно до правил вже без його участі.

Приклад клітини зображено на рис. 1.



Рис. 1. Клітина block в Game of life.

Найпростий спосіб, як зробити світ нескінченно великим, зробити його циклічним. Це означає, що лівий сусід із лівого стовпця в правий стовпчик і навпаки. Це робить світ нескінченним у плані, що ви можете переміщатися в будь-якому напрямі так далеко, як ви хочете, не потрапивши ні краю. Природно, що загальна кількість лейкоцитів циклічних світ скінченний, а нескінченність не є реальним, але досить хороший для гри життєвих цілей. Приклад клітини і їх околиці в циклічному світі наведена на рис. 2. Фактична реалізація циклічних світі робиться за допомогою модуля оператора.

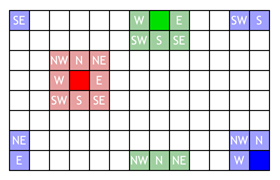


Рис. 2. Клітини і їх околиці в циклічному світі.

**Результат виконання**

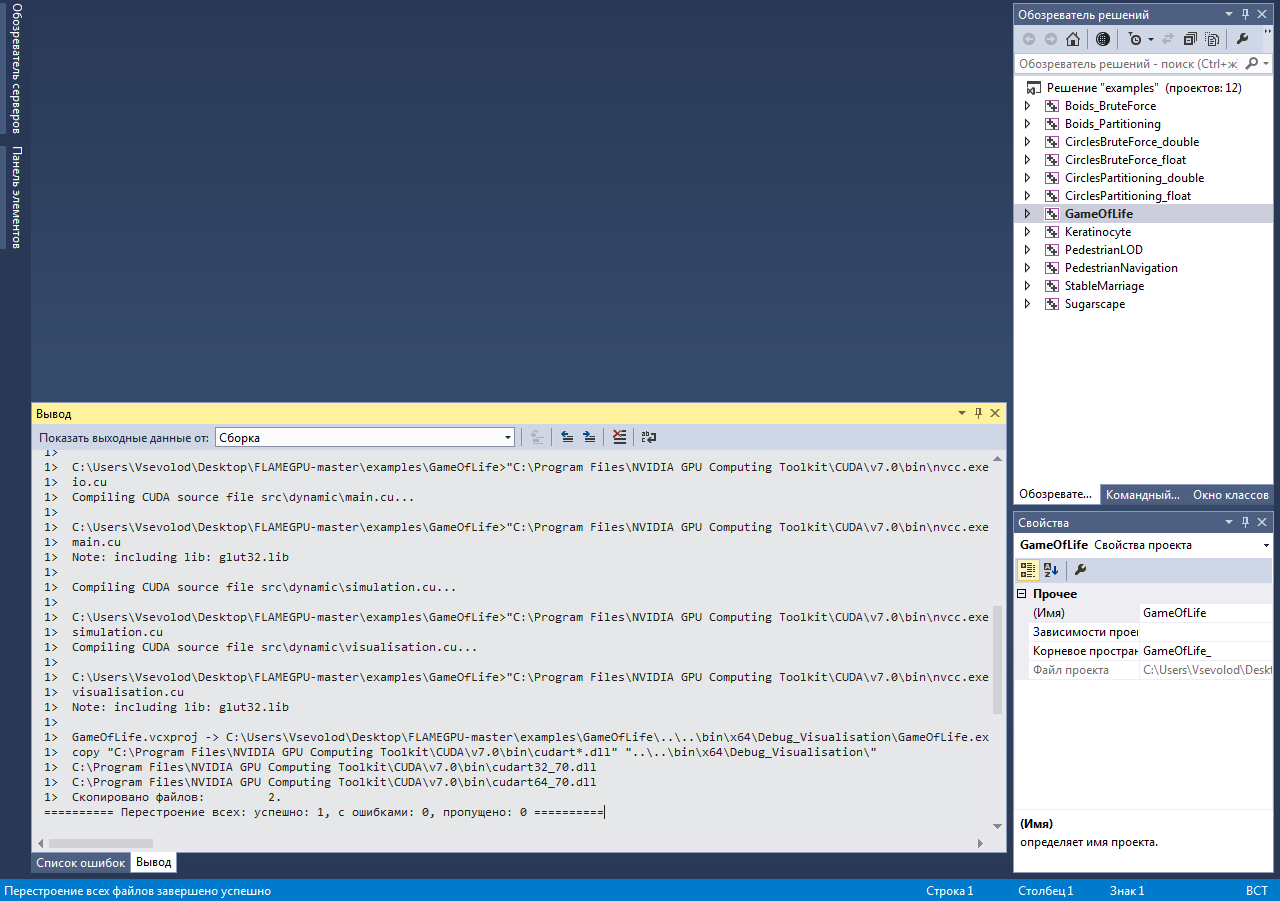


Рис. 3. Зібрання проекту.

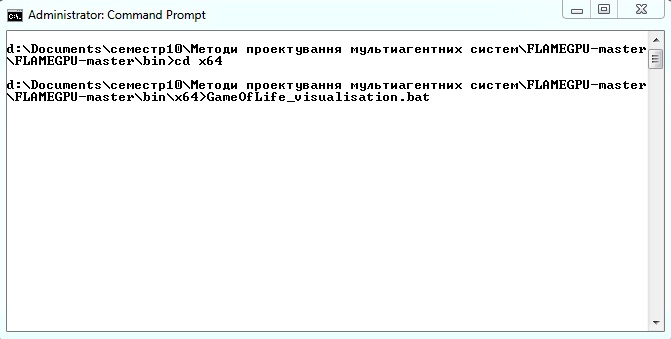


Рис. 4. Запуск скомпільованого файлу через консоль.

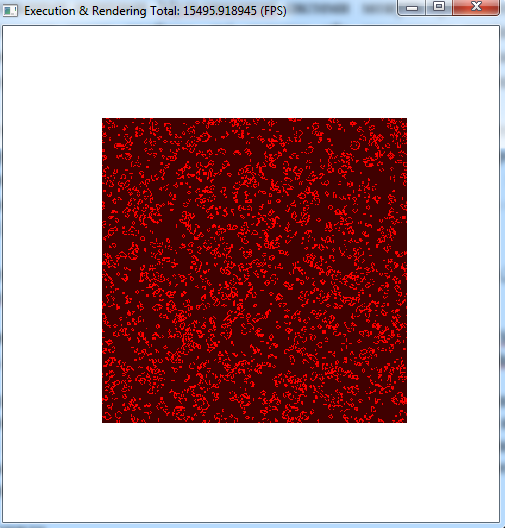


Рис. 5. Результат виконання програми 1.

**Навіщо взагалі використовувати графічний процесор для моделювання чи розробки чи виконання мультиагентної системи**

Агентне Моделювання – це метод обчислювального моделювання складних взаємодіючих систем, через специфікацію поведінки ряду автономних індивідів, що діють одночасно. Цей підхід "знизу вгору", на відміну від "зверху вниз" є моделюванням поведінки всієї системи за допомогою динамічних математичних рівнянь. Орієнтація на індивідах є значно більш вимогливе до обчислювальння, але забезпечує природну і гнучку середовище для вивчення систем, що демонструють виникаючі поведінки. Незважаючи на очевидний паралелізм, традиційно фреймворки для АВМ не використовують це і часто засновані на високопослідовній формі алгоритмів для маніпулювання мобільними дискретними агентами. Такий підхід має серйозні наслідки, ставлячи жорсткі обмеження на масштаб моделі, та швидкість, з якою вони можуть бути змодельовані. Мета FLAME GPU усунути недоліки попередніх агент програмного забезпечення для моделювання, орієнтованих на високу продуктивність GPU архітектури. Система розроблена з урахуванням паралелізму і, таким чином дозволяє моделям агента збільшитися до масивних розмірів і забезпечує моделюванню працювати в розумних обмеженнях часу. В цьому додатку це легко досяжно, оскільки дані моделювання проводяться повністю в GPU пам'яті, де вони передаються напряму.

**Чи доцільно це робити для будь якої системи**

Ні, оскільки симуляції з багатьма агентами в JADE, вона може не виконуватися. Наприклад, ми, хочемо дослідити в реальному часі, комплекс для моделювання натовпу, в такому випадку CUDA, це хороший спосіб вирішити цю проблему. Оскільки ця проблема не може бути змодельована з такою ж кількістю агентів в Jade. CUDA є більш корисним, коли ми маємо багато подібних агентів з меншою кількістю взаємодій між ними, хоча JADE краще використовувати, коли у нас є багато різних типів агентів і немає необхідності моделювати величезної кількості агентів.

**Як автори фреймворка пропонують реалізовувати мультиагентну систему, які принципи вони заклали**

Технічно фреймворк Flame GPU - не симулятор, а шаблон на основі середовища моделювання, яке відображає офіційні описи агентів в коді середовища. Формальне представлення агента базується на концепції комунікації X-машини (яка є розширенням Finite State-машини, яка включає в себе пам'ять). Незважаючи на те, що Х-машина має дуже формальне визначення, агенти Х-машини можуть розглядати стан машин, які здатні спілкуватися за допомогою повідомлень, які зберігаються в загально доступних списках повідомлень. Функціональність агента подається як набір функцій переходів, які переміщають агентів з одного внутрішнього стану до іншого. При зміні стану, агенти оновлюють свою внутрішню пам'ять в залежності від повідомлень, які можуть бути використані або в якості входу (шляхом перебору списків повідомлень), або як вихід (де інформація може бути передана в списках повідомлень для прочитання іншими агентами). З цією метою Flame GPU використовує сценарії функцій агента, де сценарій визначається в ряді файлів функцій агента. Симулятори задаються за допомогою мови розмітки (XMML), яка є синтаксисом XML з відповідними схемами, регулюючими зміст. Як правило, XMML файл моделі складається з визначеної кількості агентів Х-машини, ряду деяких типів повідомлень (кожен з яких має загальнодоступний список повідомлень) і з набору шарів моделювання, які визначають порядок виконання функцій агента для однієї ітерації моделювання. Протягом моделювання дані агента є стійкими, однак інформаційне повідомлення (і, зокрема списки повідомлень) постійні тільки протягом життєвого циклу однієї ітерації. Це дозволяє механізму агентів багаторазово взаємодіяти і формувати групову поведінку.

GPU надає деякі переваги в продуктивності для моделювання агента ніж більш традиційні альтернативи, засновані на процесорі. Маючи це на увазі, можна написати дуже суб-оптимальний код, який матиме меншу продуктивість. Нижче подано список порад для підвищення продуктивності при створенні файлів моделі Flame GPU;

FLAME GPU є оптимальним для використання, коли в ньому є дуже велика кількість відносно простих агентів, які можуть бути розпаралелені.

Популяції агентів з дуже низьким числом будуть погано працювати (в крайніх випадках повільніше, ніж якби вони були змодельовані з використанням процесора). Якщо вам потрібна популяція агента з дуже малою кількістю агентів, то задайте власний код для моделювання процесором та передавайте будь-яку важливу інформацію в константах моделювання для зчитування більшими популяціями агентів під час моделювання FLAME GPU.

Виведення інформації на диск (XML файлів) є вкрай повільним в порівнянні зі швидкостями моделювання, тому краще виводьте інформацію візуально або лише після великої кількості ітерацій моделювання.

*Поради для підвищення продуктивності для моделі Специфікація*

Мінімізуйте кількість змінних з агентами і даними повідомлень, всюди де це можливо. Спробуйте концептуалізувати і повністю визначити модель до завершення сценарію функції агента, щоб уникнути помилок з її аргументами. Намагайтеся думати в термінах X-Machines агентів!

*Поради по написанню функцій агентів*

Невеликі ресурсномісткі обчислювальні функцій агента більш ефективні, ніж функції, які тільки перебирають повідомлення. Спробуйте звести до мінімуму кількість списків повідомлень, що повторюються.

Тримайте функції агента малими і не визначайте більше локальних змінних, ніж це строго необхідно. Використовуйте повторно локальні змінні там, де це можливо , якщо вони більше не потрібні і перш ніж вони вийдуть з області видимості

*Поради по роботі з ітераціями повідомлень*

Для малих популяцій агентів (як правило, менше, ніж 2000, але залежить від апаратного забезпечення і моделі) несекціонірованні повідомлення потребують менше накладних витрат, це схоже з порівнянням із просторовим розділенням.

**Що потрібно зробити щоб запустити чи реалізувати мультиагентну систему (кроки)**

1. *Для початку*

Процес побудови та запуску моделювання не представляє труднощів, оскільки є цілий ряд інструментів і процедур, які спрощують моделювання генерації коду і компіляції моделювання виконуваних файлів. Для того, щоб використовувати FLAME GPU SDK, повинен бути поміщений в каталог, який не містить пробілів (бажано прямо на диск c: або кореня, або кореня диска операційної системи). Хост-машина повинна бути запущена Windows з копії .Net runtime (використовується усередині шаблона XSLT процесор) і повинен містити Графічний процесор NVIDIA з обчислювальним рівнем 1.1.

*2. Генерування сценарію функції агента*

Для оголошення функцій агента доцільно використовувати спеціальний шаблон XSLT ( functions.xslt розташований у папці FLAME GPU/templates) для автоматичного генерування вихідного файлу функції з пустим оголошенням використовуючи XML модель. Після цього сформований скрипт функцій агента може бути легко оголошений. Однак необхідно памятати, що якщо файл моделі XMML пізніше змінений, то аргументи функцій агента оновлюються вручну, коли це є необхідно. Точно так само будьте обережні, щоб не перезаписати всі існуючі вихідні файли функції при створенні нового, використовуючи шаблон XSLT. Генерація порожніх вихідних файлів функцій не включена у візуальному шаблоні проекту студії і повинні бути виконані вручну. Для цієї мети надається XSLT процесор на базі .NET в FLAME GPU SDK, що може бути використаний з допомогою командної стрічки як вказано нище (або за допомогою batch файлу GenerateFunctionsFileTemplate, який знаходиться в папці FLAME GPU SDK )

XSLTProcessor.exe XMLModelFile.xml functions.xslt functions.c

*3. FLAME GPU шаблонні файли*

FLAME GPU SDK містить ряд XSLT-шаблонів, які використовуються для створення динамічного моделювання коду. Короткий опис функціональності і змісту кожного файлу шаблону виглядає наступним чином;

header.xslt – цей файл шаблону генерує файл заголовка, який містить якогось агента і повідомлень структури даних, які є спільними в багатьох інших вихідних файлах моделювання, що динамічно генеруються. Шаблон також генерує прототипи функцій для моделювання функцій, які видно зовні в межах користувацького коду C або C++.

main.xslt - цей файл шаблону генерує вихідний файл, який визначає основну функцію виконання точки входу, відповідає за обробку опцій командного рядка і ініціалізацію графічного процесора пристрою.

io.xslt – цей файл шаблону генерує вихідний файл, що містить функції для завантаження початкових файлів даних агента XML в моделюванні та збереження стану моделювання назад у форматі XML.

simulation.xslt – цей файл шаблону генерує вихідний файл, що містить код моделювання приймаючої сторони, яка включає завантаження даних з і в GPU пристрій і робить ряд викликів ядра CUDA, які виконують процес моделювання.

FLAMEGPU\_kernels.xslt – цей файл шаблону генерує вихідний файл CUDA, яка містить ядра CUDA і функції пристроїв, які складають моделювання.

visualisation.xslt – цей файл шаблону генерує вихідний файл, який дозволить основну візуалізацію моделювання з використанням сфери на основі подання агентів в 3D-просторі. Вихідний файл відповідає за CUDA OpenGL сумісність та поділ за допомогою OpenGL. Вихідний файл включає файл ''visualisation.h'', що містить ряд визначень і змінних, що не створені жодним шаблоном і повинні бути вказані вручну.

*4. Компіляція за допомогою Visual Studio*

FLAME GPU SDK і приклади орієнтовані на CUDA 4.1 для компіляції за допомогою Visual Studio 2008. Рішення знаходиться за допомогою інструментів Visual Studio IDE і засобів налагодження для CUDA. Імпортування в середовище Linux повинно бути відносно простим. На додаток до цього візуальний редактор студії XML включає підтримку перевірки XML-тегів і автозавершення, який робить визначені моделі XMML неймовірно легкими.

**Що таке X-Machines**

*Визначення X-Machine Agent*

У файлі XMML повинен бути принаймні один елемент xagent, який повинен визначити принаймні одного xagent. Xagent є агентом представлення X-Machines і складається з імені, опису , набору функцій, внутрішньої пам'яті і безлічі станів. Для FLAME GPU потрібно дві додаткові інформації (які не потрібні в оригінальній специфікації XMML), тип і розмір буфера. Тип елемента відноситься до типу агента по відношенню до його зв'язку з його просторовим середовищем. Тип агента може бути дискретним або безперервним. Вся пам'ять попередньо виділяється на GPU, розмір буфера потрібно представляти максимально можливий розмір населення агента. Тобто максимальне число випадків X-Machines агента у форматі, описаному в моделі XMML. Це не є недоліком продуктивності використання великого розміру буфера, це відповідальність користувача, що GPU містить достатньо пам'яті для підтримки великих популяцій агентів. Рекомендується розмір буфера завжди була в числі степеня 2 (тобто 1024, 2048, 4096, 16384 і т.д.), швидше за все, округлиться до одного під час моделювання. Для дискретних агентів розмір буферу в даний час обмежується ступенем числа 2. Якщо в будь-який момент у процесі моделювання перевищить розмір буферу то користувач буде про це попереджений.

<xagents>

<gpu:xagent>

<name>AgentName</name>

<description>optional description of the agent</description>

<memory>...</memory>

<functions>...</functions>

<states>...</states>

<gpu:type>continuous</gpu:type>

<gpu:bufferSize>1024</gpu:bufferSize>

</gpu:xagent>

<gpu:xagent>...</gpu:xagent>

</xagents>

*Agent Memory*

Agent Memory складається з ряду змінних (принаймні, однієї), яка використовуватиметься для зберігання інформації. Змінна агента повиненна мати унікальне ім'я і може бути типу int, float or double. В даний час Agent Memory підтримує тільки використання окремих областей пам'яті (тобто не статичні або динамічні масиви) і значення за замовчуванням завжди 0, якщо значення не вказано в файлі станів XML. Там немає заданих максимальних меж. Нище наведено приклад агента.

<memory>

<gpu:variable>

<type>int</type>

<name>id</name>

<description>variable description</description>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>x</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>y</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>z</name>

</gpu:variable>

</memory>

**Висновок**

На цій розрахунково-графічній роботі, я ознайомився з основними функціональними можливостями бібліотеки Flame GPU. Також провів аналіз існуючих мультиагентних систем реалізованих за допомогою цієї бібліотеки.