Методичка по курсу «Мультиагентні системи» на тему:

**Агентні системи на FlameGPU**

Сергієнко Владислав, СПКм-12

**Мета роботи**

Вивчити основні принципи роботи FlameGPU. Розробити агентну систему симуляції руху людей в приміщенні, використовуючи технологію FlameGPU.

**Короткі теоретичні відомості**

*Агентне моделювання*

Агентне моделювання – це техніка для симуляції складних інтерактивних систем за допомогою специфікації поведінки автономних агентів, що діють одночасно. Такі системи є більш обчислювально вимогливими, але надають природнє і гнучке середовище для вивчення поведінки системи.

*Технологія FlameGPU*

FlameGPU розробляється Річмондом Полом, канцлером центру досліджень CUDA університету Шефілда, який займається симуляцією комплексних систем і паралельними обчисленнями.

FlameGPU допомогає реалізувати агентні симуляції для оцінки і передбачення поведінки групи агентів, базуючись на простих правилах взаємодії між ними.

Переваги FlameGPU – використання потужного графічного модуля для паралельних обчислень за допомогою CUDA, надання можливості візуалізації симуляції, опис комунікації моделей на високому рівні. Технічно, FlameGPU не є симулятором, а – шаблонним середовищем симуляції, здатне проектувати формальний опис агентів безпосередньо у код симуляції.

Недоліки – значна кількість обмежень по типах даних, які можна використовувати при симуляції агентів. Це значно звужує коло задач, які можна вирішувати.

*Агенти у FlameGPU*

У FlameGPU представлення агента засноване на концепції спілкування X-Автоматів , які є розширенням скінченних автоматів (абстракція, яка використовуєтся для опису шляху зміну стану об’єкта), і можуть «спілкуватися» за допомогою повідомлень, додаючи їх у загальнодоступний список, і потім зчитуючи з нього. Функціонал агента представляється як список функцій переходів станів (переміщують агента з одного внутрішнього стану в інший); агенти оновлюють свою внутрішню пам’ять під впливом повідомлень. Функції агентів задаються самостійно розробником додатку, а файли з ними вказуються у описі середовища. Опис моделі симуляції записується у XML-форматі, використовуючи XMML специфікацію (X-Machine Mark-up Language), за відповідними схемами. Типовий XMML модель складається з опису агентів, повідомлень, фукнцій агентів.

*Принцип роботи FlameGPU*

Процес генерації симуляції на FlameGPU зображений на рис. 1. Велику роль грає XSLT-процесор (Extensible Stylesheet Transformation – гнучка функціональна мова, в основі якої лежить XML), який, за допомогою наданих схем може перекладати документи з одного формату у інші. FlameGPU надає свій власний процесор, разом з XSLT-шаблонами симуляції для генерації коду програми.

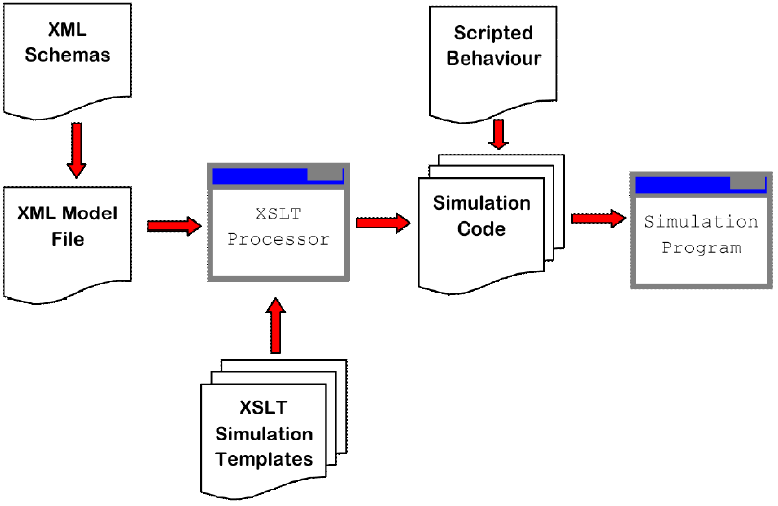
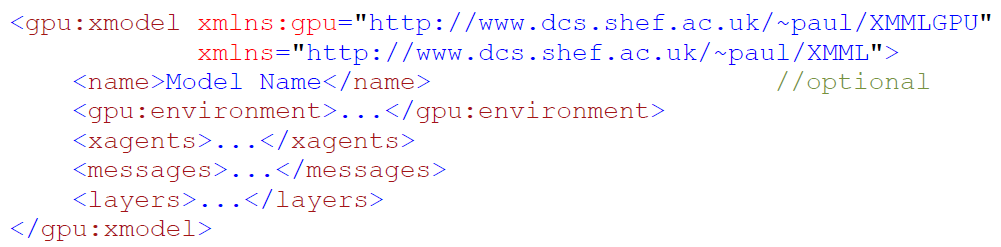


Рис. 1. Процес генерації моделювання FlameGPU - симуляції.

*Специфікація моделі симуляції FlameGPU*

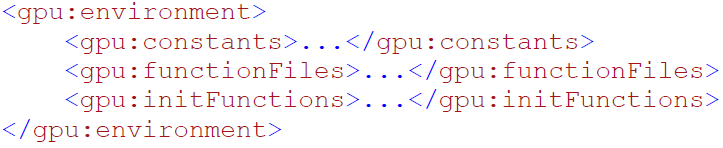
Модель описується у файлі проекта **XMLModelFile.xml**. Головні розділи опису є у лістингу 1.

Лістинг 1. Головні розділи визначення моделі симуляції.



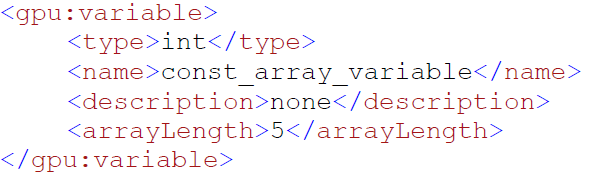
Розділ «Середовище» (визначений у лістингу 2) тримає глобально доступну інформацію для симуляції: константи, функції станів агентів, функції ініціалізації.

Лістинг 2. Головні розділи визначення середовища симуляції.



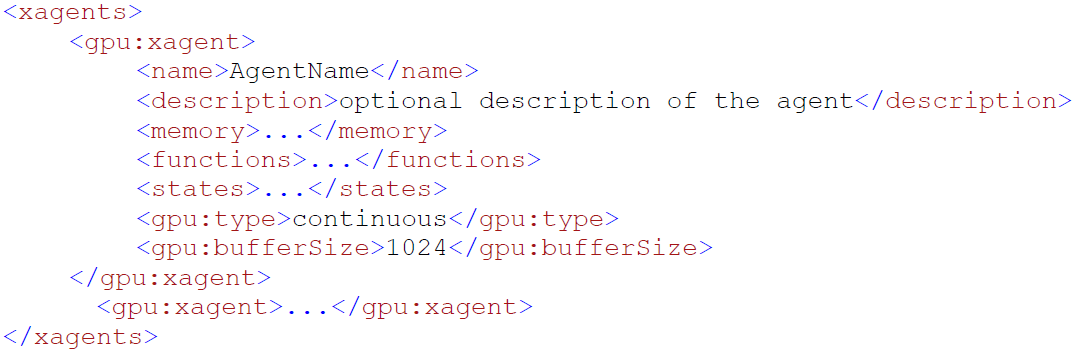
Константи і глобальні змінні (лістинг 3) можуть мати тип **int, float, double**, задану довжину масиву або початкові значення, та повинні мати унікальні назви.

Лістинг 3. Приклад визначення глобальної змінної.



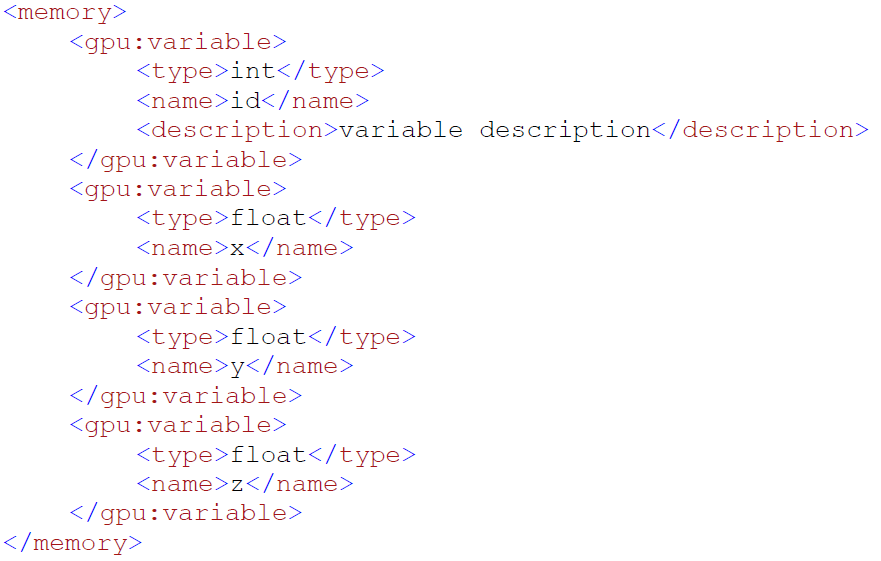
Визначення X-Автомата надано у лістингу 4. Автомат має внутрішню пам’ять (**M** у формальному визначенні), набір функцій агентів (**F** у формальному визначенні) набір станів (**Q** у формальному визначенні). Додатково до оригінальної XMML специфікації, мають бути вказані ще 2 поля: тип (discrete – переміщення по клітинкам, або continuous – можуть займати все середовище) і розмір буфера (1024, 2048, 4096, 16384, … - визначає кількість екзмеплярів агента такого типу, ).

Лістинг 4. Приклад визначення агента.



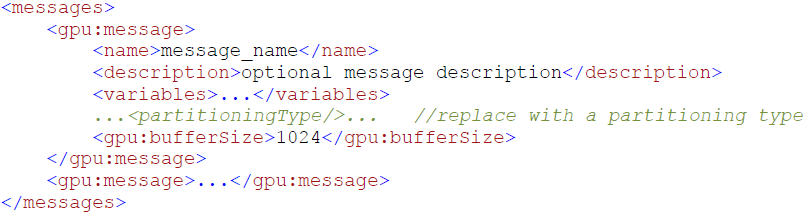
Пам’ять агента (лістинг 5) визначається набором змінних типів **int, float, double** (початкове значення, якщо не задане у XML).

Лістинг 5. Приклад визначення пам’яті агента.



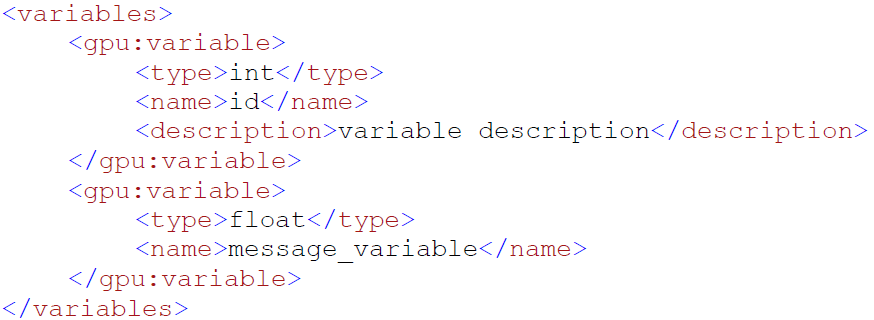
Стани агента задаються лише назвою, додатково вказується початковий стан агента. Повідомлення задається набором змінних, максимальною кількістю повідомлень такого типу (**bufferSize**), а також типом поділу (partitioning): **partitioningDiscrete** (*дискретний* 2Д-поділ), **partitioningSpatial** (2Д або 3Д поділ), **partitioningNone** (немає поділу).

Лістинг 6.1. Приклад визначення повідомлення агента.



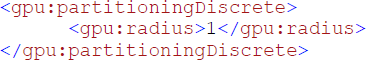
Змінні визначаються подібно визначенню їх для агентів, з такими самими обмеженнями (лістинг 6.2)

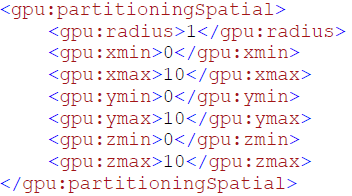
Лістинг 6.2. Приклад визначення змінних повідомлення агента.



Визначення типу поділу можна використовуються як механізм відсіювання повідомлень для агентів, що не відповідають просторовим обмеженням (наприклад, відстань від агента). Для значення 1, буде задіяно 9 клітинок (3х3), для 2 – 25(5х5).

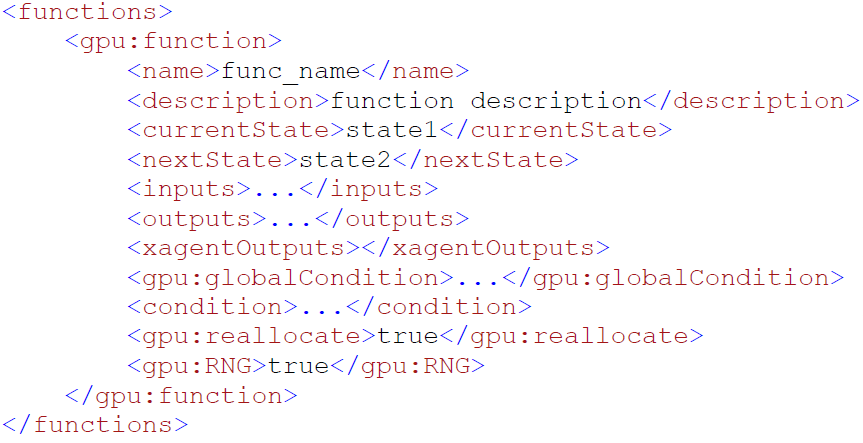
Лістинг 6.3. Приклад визначення поділу повідомлення агента.



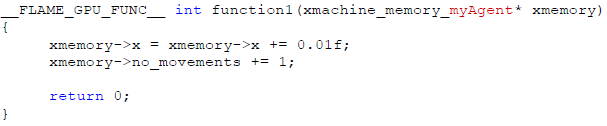


Функція агента визначається за зразком, наведеном у лістингу 7. В ній задаються: поточний і наступний стани, вхідне повідомлення і вихідне повідомлення, вихідні агенти, умова виклику функції для одного агента, умова виклику функції для агентів даного типу, перестворення агента, а також використання генератора випадкових чисел. Для вихідних повідмолень додатково задається поле, що визначає, чи вихідне повідомлення генерується завжди, чи ні.

Лістинг 7. Приклад визначення функції переходу агента в інший стан.

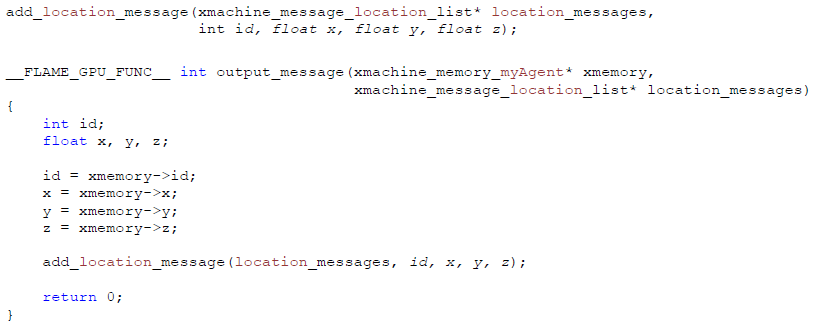


Після того, як код буде згенерований, користувач має згідно згенерованих декларацій функцій дати їм визначення у *\*.с* файлах. Базовий варіант функції виглядає так:



У цьому визначенні фукнції для агента типу **myAgent** інкрементується кількість зміщень, і збільшується позиція по осі **Х**. Функція повертає *не нуль* лише тоді, коли агент «помирає», і більше не бере участі у симуляції.

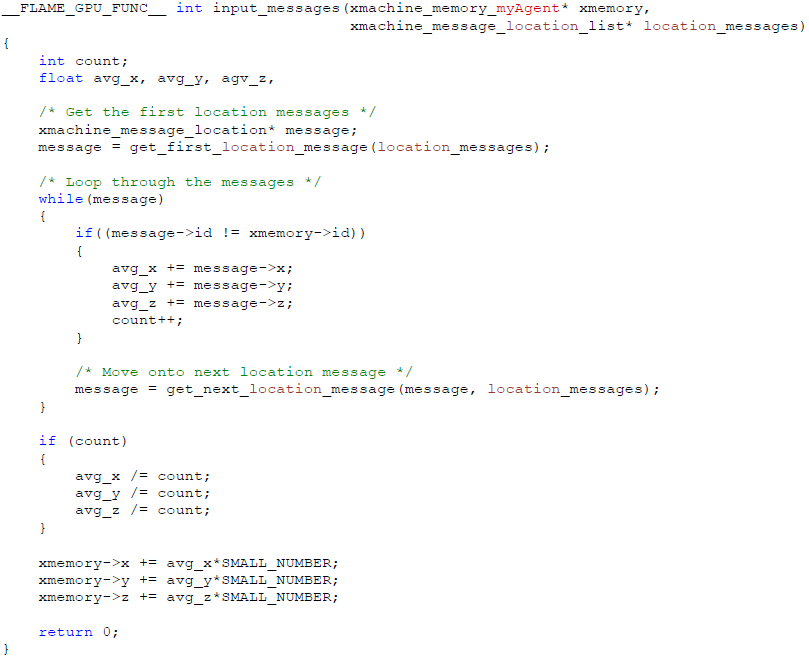
Використання фукнціоналу для додавання повідомлень у чергу наверено нижче:



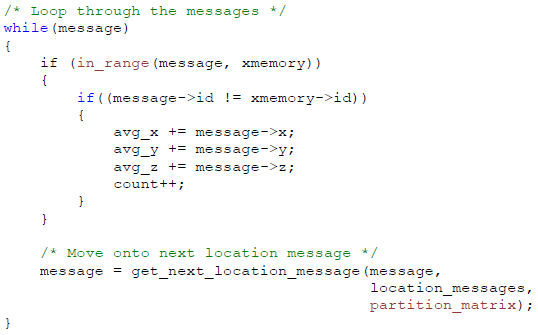
Функція вигляду **add\_*message\_name*\_message(message\_name\_messages, args...)** є згенерованою разом зі **xmachine\_message\_*message\_name*\_list** списком повідомлень цього типу, і користувачем не визначається (будь які зміни будуть втрачені при перезапуску XSLT процесора).

У прикладі, наведеному вище, тип повідомлення – **location**, пропотипи фукнцій генеруються у файлі header.h. Функція належить агентові типу **myAgent**, яка додає повідомлення з 4 змінних у глобальний список для опрацювання іншими агентами.

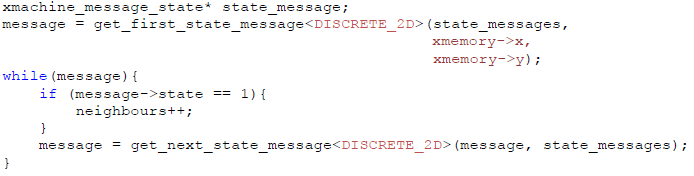
У наступному прикладі наведено функцію для опрацювання повідомлень типу **location** *без поділу* агентом типу **myAgent**, використовуючи згенеровані фукнції **get\_first\_message\_name\_message**, **get\_next\_message\_name\_message**:



Для опрацювання повідомлень з просторовим поділом існує згенерована функція **in\_range**, яка визначає, чи потрібно це повідомлення опрацьовувати:



Для 2Д дискретного поділу, для фукнцій **get\_first\_message\_name\_message**, **get\_next\_message\_name\_message** додатково вказується шиблонний параметр **DISCRETE\_2D**:



*Шаблонні файли FLAMEGPU SDK*

**FLAMEGPU** містить шаблонні файли для генерації динамічного кода для симуляції.

* **header.xslt** –Генерує header.h з даними про агентів і повідомлення, прототипами функцій.
* **main.xslt** – Генерує main.cu; визначає вхідну точку програми, де опрацьовуються вхідні параметри командної стрічки і ініціалізується графічний пристрій.
* **io.xslt** – Опрацювання XML даних.
* **simulation.xslt** – Генерує код для завантаження даних на графічний пристрій, виклики CUDA – функцій для симуляції.

**Хід виконання завдання**

Алгоритм роботи з *FLAMEGPU*, як правило, наступний:

* налаштування проекту
* опис моделі симуляції
* генерація динамічних файлів симуляції
* ініціалізація констант
* заповнення функцій агентів

*Налаштування середовища Visual Studio*

Спочатку необхідно налаштувати дії XSLT-процесора *FLAMEGPU* для того, щоб за шаблонами згенерувати файли симуляції, прототипи функцій. Для цього у налаштування XMLModelFile.xml необхідно вказати шлях до процесора, вхідний файл, і вихідний (рис. 2.) за принципом: **XSLTProcessor.exe** *XMLModelFile.xml* ***functions.xslt functions.c.***

Вхідні дані так само завантажуються з файла, і для того, щоб запускати програму з середовища Visual Studio, потрібно у налаштуваннях проекта вказати шлях до файлу з даними ініціалізації симуляції (рис. 3.).

*Створення опису моделі симуляції*

Для цього потрібно створити у теці проекту файл *XMLModelFile.xml*, у ньому, відповідно до лістингів 1-7:

* Описати константи:
  + **float** TIME\_SCALER – для симуляцій не в реальному часі (можна пришвидшити)
  + **float** STEER\_WEIGHT, AVOID\_WEIGHT, COLLISION\_WEIGHT, GOAL\_WEIGHT – константи для контролю руху агентів
* Описати агента:
  + Описати його пам’ять (М): **float** x, y, z, steer\_x, steer\_y, height, speed; **int** exit\_no (номер виходу для агента)
  + Описати функції Х-Автомата:
    - Функцію виводу повідомлення про позицію агента у список;
    - Функцію для уникнення зіткнень з іншими агентами;
    - Функцію для зміщення агента;
  + Описати стани агента (для даної симуляції задани 1 стан, зробити цього початком)
  + Для функції агент з виводом повідомлення – описати це повідомлення: воно має містити позицію агента; для повідомлення задати оптимальний метод поділу (рекомендується - **partitioningSpatial**)
* Згенерувати динамічний код
  + Визначити функції агента по їх прототипам з header.h

Опишемо константи:

<gpu:constants>

<gpu:variable>

<type>float</type>

<name>TIME\_SCALER</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>STEER\_WEIGHT</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>AVOID\_WEIGHT</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>COLLISION\_WEIGHT</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>GOAL\_WEIGHT</name>

</gpu:variable>

</gpu:constants>

У header.h згенеруються змінні для констант, а також функції для їхньої ініціалізації:

\_\_constant\_\_ float TIME\_SCALER;

\_\_constant\_\_ float STEER\_WEIGHT;

\_\_constant\_\_ float AVOID\_WEIGHT;

\_\_constant\_\_ float COLLISION\_WEIGHT;

\_\_constant\_\_ float GOAL\_WEIGHT;

extern "C" void set\_TIME\_SCALER(float\* h\_TIME\_SCALER);

extern "C" void set\_STEER\_WEIGHT(float\* h\_STEER\_WEIGHT);

extern "C" void set\_AVOID\_WEIGHT(float\* h\_AVOID\_WEIGHT);

extern "C" void set\_COLLISION\_WEIGHT(float\* h\_COLLISION\_WEIGHT);

extern "C" void set\_GOAL\_WEIGHT(float\* h\_GOAL\_WEIGHT);

Опишемо пам’ять агента:

<memory>

<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>velx</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>vely</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>steer\_x</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>steer\_y</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>height</name>

</gpu:variable>

<gpu:variable>

<type>int</type>

<name>exit\_no</name>

</gpu:variable>

<gpu:variable>

<type>float</type>

<name>speed</name>

</gpu:variable>

</memory>

Згенерований код матиме вигляд:

struct xmachine\_memory\_agent\_list

{

int \_position [xmachine\_memory\_agent\_MAX];

int \_scan\_input [xmachine\_memory\_agent\_MAX];

float x [xmachine\_memory\_agent\_MAX];

float y [xmachine\_memory\_agent\_MAX];

float velx [xmachine\_memory\_agent\_MAX];

float vely [xmachine\_memory\_agent\_MAX];

float steer\_x [xmachine\_memory\_agent\_MAX];

float steer\_y [xmachine\_memory\_agent\_MAX];

float height [xmachine\_memory\_agent\_MAX];

int exit\_no [xmachine\_memory\_agent\_MAX];

float speed [xmachine\_memory\_agent\_MAX];

};

Опишемо функції агента. Має бути 3 функції – вивід позиції (*вказується вихідне повідомлення*), уникнення зіткнень (*вказується вхідне повідомлення з 1 фукнції, генeрується випадковий напрямок*), переміщення агента:

<functions>

<gpu:function>

<name>output\_pedestrian\_location</name>

<currentState>default</currentState>

<nextState>default</nextState>

<outputs>

<gpu:output>

<messageName>pedestrian\_location</messageName>

<gpu:type>single\_message</gpu:type>

</gpu:output>

</outputs>

<gpu:reallocate>false</gpu:reallocate>

<gpu:RNG>false</gpu:RNG>

</gpu:function>

<gpu:function>

<name>avoid\_pedestrians</name>

<currentState>default</currentState>

<nextState>default</nextState>

<inputs>

<gpu:input>

<messageName>pedestrian\_location</messageName>

</gpu:input>

</inputs>

<gpu:reallocate>false</gpu:reallocate>

<gpu:RNG>true</gpu:RNG>

</gpu:function>

<gpu:function>

<name>move</name>

<currentState>default</currentState>

<nextState>default</nextState>

<gpu:reallocate>false</gpu:reallocate>

<gpu:RNG>false</gpu:RNG>

</gpu:function>

</functions>

Тип вихідного повідомлення про позицію агента:

<messages>

<gpu:message>

<name>pedestrian\_location</name>

<variables>

<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>

</variables>

<gpu:partitioningSpatial>

<gpu:radius>0.025</gpu:radius>

<gpu:xmin>-2.0</gpu:xmin>

<gpu:xmax>2.0</gpu:xmax>

<gpu:ymin>-2.0</gpu:ymin>

<gpu:ymax>2.0</gpu:ymax>

<gpu:zmin>0.0</gpu:zmin>

<gpu:zmax>0.025</gpu:zmax>

</gpu:partitioningSpatial>

<gpu:bufferSize>16384</gpu:bufferSize>

</gpu:message>

Згенеровані прототипи:

\_\_FLAME\_GPU\_FUNC\_\_ int output\_pedestrian\_location(xmachine\_memory\_agent\* agent, xmachine\_message\_pedestrian\_location\_list\* pedestrian\_location\_messages);

\_\_FLAME\_GPU\_FUNC\_\_ int avoid\_pedestrians(xmachine\_memory\_agent\* agent, xmachine\_message\_pedestrian\_location\_list\* pedestrian\_location\_messages, xmachine\_message\_pedestrian\_location\_PBM\* partition\_matrix, RNG\_rand48\* rand48);

\_\_FLAME\_GPU\_FUNC\_\_ int move(xmachine\_memory\_agent\* agent);

Згенерований список повідомлень:

struct xmachine\_message\_pedestrian\_location\_list

{

int \_position [xmachine\_message\_pedestrian\_location\_MAX];

int \_scan\_input [xmachine\_message\_pedestrian\_location\_MAX];

float x [xmachine\_message\_pedestrian\_location\_MAX];

float y [xmachine\_message\_pedestrian\_location\_MAX];

float z [xmachine\_message\_pedestrian\_location\_MAX];

};

Напишемо реалізацію для функцій агента по прототипам (файл functions.c, потрібно вказати в описі моделі):

1. Функція виводу позиції агента, для опрацювання агентами, що розташовані поруч:

\_\_FLAME\_GPU\_FUNC\_\_ int output\_pedestrian\_location(

xmachine\_memory\_agent\* agent,

xmachine\_message\_pedestrian\_location\_list\* pedestrian\_location\_messages

)

{

add\_pedestrian\_location\_message(pedestrian\_location\_messages, agent->x, agent->y, 0.0);

return 0;

}

1. Функція для оминання перешкод і корегування напрямку агента до його цілі (у прикладі - оминання інших агентів, самостійно – додати оминання стін приміщення)

\_\_FLAME\_GPU\_FUNC\_\_ int avoid\_pedestrians(

xmachine\_memory\_agent\* agent,

xmachine\_message\_pedestrian\_location\_list\* pedestrian\_location\_messages,

xmachine\_message\_pedestrian\_location\_PBM\* partition\_matrix,

RNG\_rand48\* rand48

)

{

float2 agent\_pos = make\_float2(agent->x, agent->y);

float2 agent\_vel = make\_float2(agent->velx, agent->vely);

float2 navigate\_velocity = make\_float2(0.0f, 0.0f);

float2 avoid\_velocity = make\_float2(0.0f, 0.0f);

xmachine\_message\_pedestrian\_location\* current\_message =

get\_first\_pedestrian\_location\_message(

pedestrian\_location\_messages,

partition\_matrix,

agent->x,

agent->y,

0.0

);

while (current\_message)

{

float2 message\_pos = make\_float2(current\_message->x, current\_message->y);

float separation = length(agent\_pos - message\_pos);

if ((separation < MESSAGE\_RADIUS)&&(separation>MIN\_DISTANCE)){

float2 to\_agent = normalize(agent\_pos-message\_pos);

float ang = acosf(dot(agent\_vel, to\_agent));

float perception = 45.0f;

//STEER

if ((ang < RADIANS(perception)) || (ang > 3.14159265f-RADIANS(perception))){

float2 s\_velocity = to\_agent;

s\_velocity \*= powf(I\_SCALER/separation, 1.25f)\*STEER\_WEIGHT;

navigate\_velocity += s\_velocity;

}

//AVOID

float2 a\_velocity = to\_agent;

a\_velocity \*= powf(I\_SCALER/separation, 2.00f)\*AVOID\_WEIGHT;

avoid\_velocity += a\_velocity;

}

current\_message = get\_next\_pedestrian\_location\_message(

current\_message,

pedestrian\_location\_messages,

partition\_matrix

);

}

//random walk goal

float2 goal\_velocity = make\_float2(0.0f, 0.0f);;

goal\_velocity.x += agent->velx \* GOAL\_WEIGHT;

goal\_velocity.y += agent->vely \* GOAL\_WEIGHT;

//maximum velocity rule

float2 steer\_velocity = navigate\_velocity + avoid\_velocity + goal\_velocity;

agent->steer\_x = steer\_velocity.x;

agent->steer\_y = steer\_velocity.y;

return 0;

}

1. Функція для зміщення агента у просторі:

\_\_FLAME\_GPU\_FUNC\_\_ int move(xmachine\_memory\_agent\* agent){

float2 agent\_pos = make\_float2(agent->x, agent->y);

float2 agent\_vel = make\_float2(agent->velx, agent->vely);

float2 agent\_steer = make\_float2(agent->steer\_x, agent->steer\_y);

float current\_speed = length(agent\_vel)+0.025f;//(powf(length(agent\_vel), 1.75f)\*0.01f)+0.025f;

//apply more steer if speed is greater

agent\_vel += current\_speed\*agent\_steer;

float speed = length(agent\_vel);

//limit speed

if (speed >= agent->speed){

agent\_vel = normalize(agent\_vel)\*agent->speed;

speed = agent->speed;

}

//update position

agent\_pos += agent\_vel\*TIME\_SCALER;

//update

agent->x = agent\_pos.x;

agent->y = agent\_pos.y;

agent->velx = agent\_vel.x;

agent->vely = agent\_vel.y;

//bound by wrapping

if (agent->x <= d\_message\_pedestrian\_location\_min\_bounds.x)

agent->x=d\_message\_pedestrian\_location\_max\_bounds.x;

if (agent->x > d\_message\_pedestrian\_location\_max\_bounds.x)

agent->x=d\_message\_pedestrian\_location\_min\_bounds.x;

if (agent->y <= d\_message\_pedestrian\_location\_min\_bounds.y)

agent->y=d\_message\_pedestrian\_location\_max\_bounds.y;

if (agent->y > d\_message\_pedestrian\_location\_max\_bounds.y)

agent->y=d\_message\_pedestrian\_location\_min\_bounds.y;

return 0;

}

Інший функціонал генерується за допомогою шаблонів. Зкомпілювати програми симуляції, згенерувати файл з ініціалізацією агентів (**0.xml**). Запустити. Спробувати збільшити кількість агентів (і, відповідно, повідомлень), і перевірити вплив на швидкодію симуляції.

Самостійно додати візуалізацію для симуляції по прикладам з репозиторію FlameGPU.

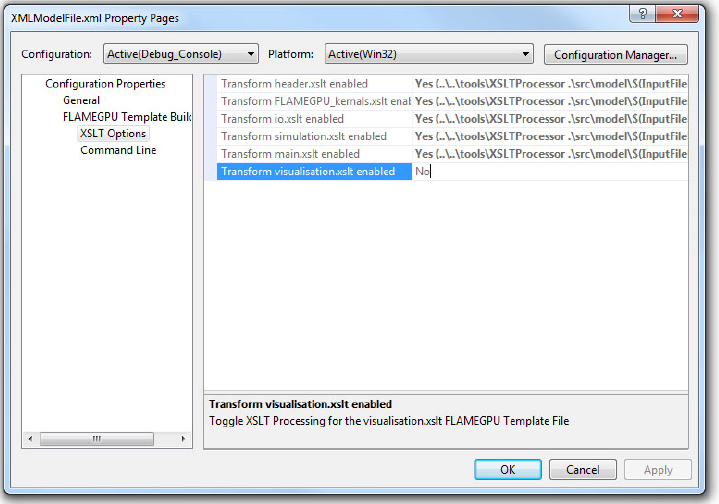


Рис. 2. Налаштування XSLT процесора для генерації динамічного коду симуляції.

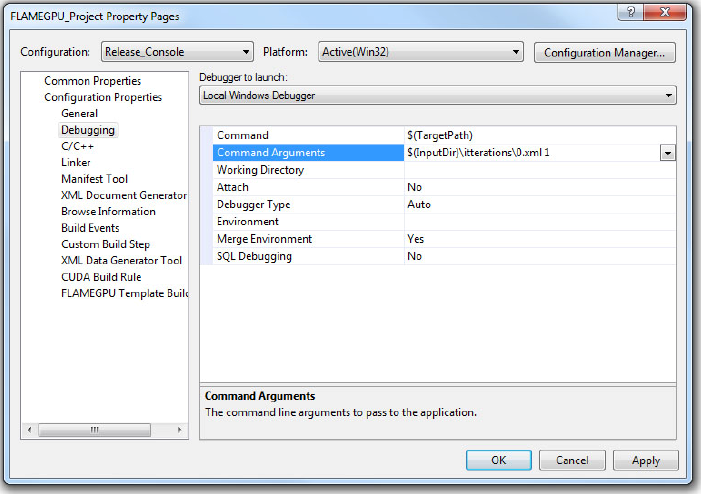


Рис. 3. Налаштування шляху до файла з початковими даними симуляції.

**Контрольні запитання**

1. Що таке агентне моделювання (ABM – Agent-Based Modelling)?
2. Властивості агентної системи?
3. За допомоги чого спілкуються агенти типу Х-Автоматів?
4. Які типи змінних підтримуються системою?
5. Які типи поділу повідомлень доступні для використання?
6. Особливості типу **partitioningSpatial**?
7. Як додати на вхід функції агента генератор випадкових чисел?
8. Як вказати вихідне повідомлення для функції агента?
9. Що таке XSLT процесор?

**Список літератури**

1. Офіційна документація