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 використовується 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.

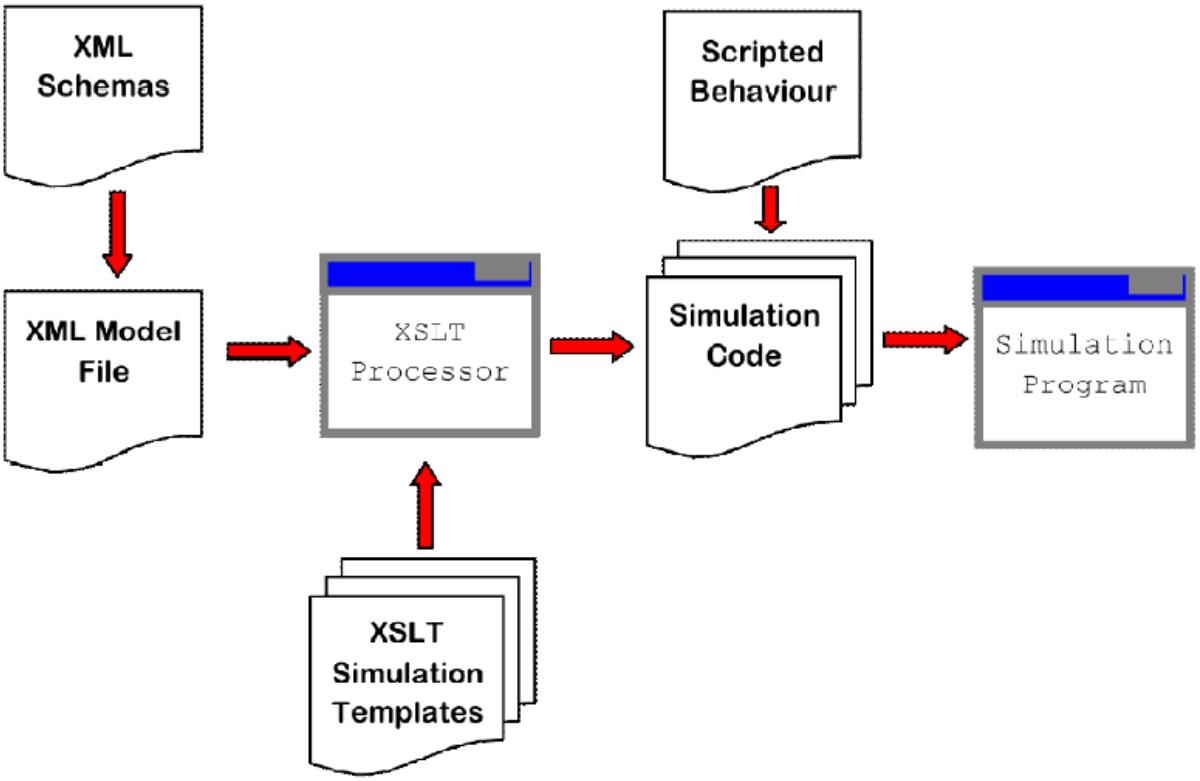

Рис. 1. Процес генерації моделювання FlameGPU - симуляції.

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.

```
<gpu:xmodel xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/XMMLGPU"
            xmlns="http://www.dcs.shef.ac.uk/~paul/XMML">
    <name>Model Name</name>                    //optional
    <gpu:environment>...</gpu:environment>
    <xagents>...</xagents>
    <messages>...</messages>
    <layers>...</layers>
</gpu:xmodel>
```

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.

```
<gpu:environment>
    <gpu:constants>...</gpu:constants>
    <gpu:functionFiles>...</gpu:functionFiles>
    <gpu:initFunctions>...</gpu:initFunctions>
</gpu:environment>
```

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.

```
<gpu:variable>
    <type>int</type>
    <name>const_array_variable</name>
    <description>none</description>
    <arrayLength>5</arrayLength>
</gpu:variable>
```

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 екземплярів агента такого типу, 2^n).

Лістинг 4. Приклад визначення агента.

```
<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>
```

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.

```
<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>
```

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.

```
<messages>
    <gpu:message>
        <name>message_name</name>
        <description>optional message description</description>
        <variables>...</variables>
        ...<partitioningType/>...   //replace with a partitioning type
        <gpu:bufferSize>1024</gpu:bufferSize>
    </gpu:message>
    <gpu:message>...</gpu:message>
</messages>
```

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

Лістинг 6.2. Приклад визначення змінних повідомлення агента.

```
<variables>
    <gpu:variable>
        <type>int</type>
        <name>id</name>
        <description>variable description</description>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>message_variable</name>
    </gpu:variable>
</variables>
```

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 (3x3), для 2 – 25(5x5).

Лістинг 6.3. Приклад визначення поділу повідомлення агента.

```
<gpu:partitioningDiscrete>
      <gpu:radius>1</gpu:radius>
</gpu:partitioningDiscrete>

<gpu:partitioningSpatial>
    <gpu:radius>1</gpu:radius>
    <gpu:xmin>0</gpu:xmin>
    <gpu:xmax>10</gpu:xmax>
    <gpu:ymin>0</gpu:ymin>
    <gpu:ymax>10</gpu:ymax>
    <gpu:zmin>0</gpu:zmin>
    <gpu:zmax>10</gpu:zmax>
</gpu:partitioningSpatial>
```

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.

```
<functions>
    <gpu:function>
        <name>func_name</name>
        <description>function description</description>
        <currentState>state1</currentState>
        <nextState>state2</nextState>
        <inputs>...</inputs>
        <outputs>...</outputs>
        <xagentOutputs></xagentOutputs>
        <gpu:globalCondition>...</gpu:globalCondition>
        <condition>...</condition>
        <gpu:reallocate>true</gpu:reallocate>
        <gpu:RNG>true</gpu:RNG>
    </gpu:function>
</functions>
```

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

```
__FLAME_GPU_FUNC__ int function1(xmachine_memory_myAgent* xmemory)
{
     xmemory->x = xmemory->x += 0.01f;
     xmemory->no_movements += 1;

     return 0;
}
```

У цьому визначенні фукнції для агента типу **myAgent** інкрементується кількість зміщень, і збільшується позиція по осі **X**. 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:

```
add_location_message(xmachine_message_location_list* location_messages,
                     int id, float x, float y, float z);


__FLAME_GPU_FUNC__ int output_message(xmachine_memory_myAgent* xmemory,
                                      xmachine_message_location_list* location_messages)
{
    int id;
    float x, y, z;

    id = xmemory->id;
    x = xmemory->x;
    y = xmemory->y;
    z = xmemory->z;

    add_location_message(location_messages, id, x, y, z);

    return 0;
}
```

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**:

```
__FLAME_GPU_FUNC__ int input_messages(xmachine_memory_myAgent* xmemory,
                                      xmachine_message_location_list* location_messages)
{
    int count;
    float avg_x, avg_y, agv_z,

    /* Get the first location messages */
    xmachine_message_location* message;
    message = get_first_location_message(location_messages);

    /* Loop through the messages */
    while(message)
    {
        if((message->id != xmemory->id))
        {
            avg_x += message->x;
            avg_y += message->y;
            avg_z += message->z;
            count++;
        }

        /* Move onto next location message */
        message = get_next_location_message(message, location_messages);
    }

    if (count)
    {
        avg_x /= count;
        avg_y /= count;
        avg_z /= count;
    }

    xmemory->x += avg_x*SMALL_NUMBER;
    xmemory->y += avg_y*SMALL_NUMBER;
    xmemory->z += avg_z*SMALL_NUMBER;

    return 0;
}
```

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

```
/* Loop through the messages */
while(message)
{
    if (in_range(message, xmemory))
    {
        if((message->id != xmemory->id))
        {
            avg_x += message->x;
            avg_y += message->y;
            avg_z += message->z;
            count++;
        }
    }

    /* Move onto next location message */
    message = get_next_location_message(message,
                                        location_messages,
                                        partition_matrix);
}
```

Для 2Д дискретного поділу, для фукнцій **get_first_message_name_message**, **get_next_message_name_message** додатково вказується шиблонний параметр **DISCRETE_2D**:

```
xmachine_message_state* state_message;
message = get_first_state_message<DISCRETE_2D>(state_messages,
                                               xmemory->x,
                                               xmemory->y);
while(message){
    if (message->state == 1){
        neighbours++;
    }
    message = get_next_state_message<DISCRETE_2D>(message, state_messages);
}
```

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 (M): **float** x, y, z, steer_x, steer_y, height, speed; **int** exit_no (номер виходу для агента)
    - Описати функції X-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, генерується 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;
}
```

2) 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;
```

```
}
```

3) 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.

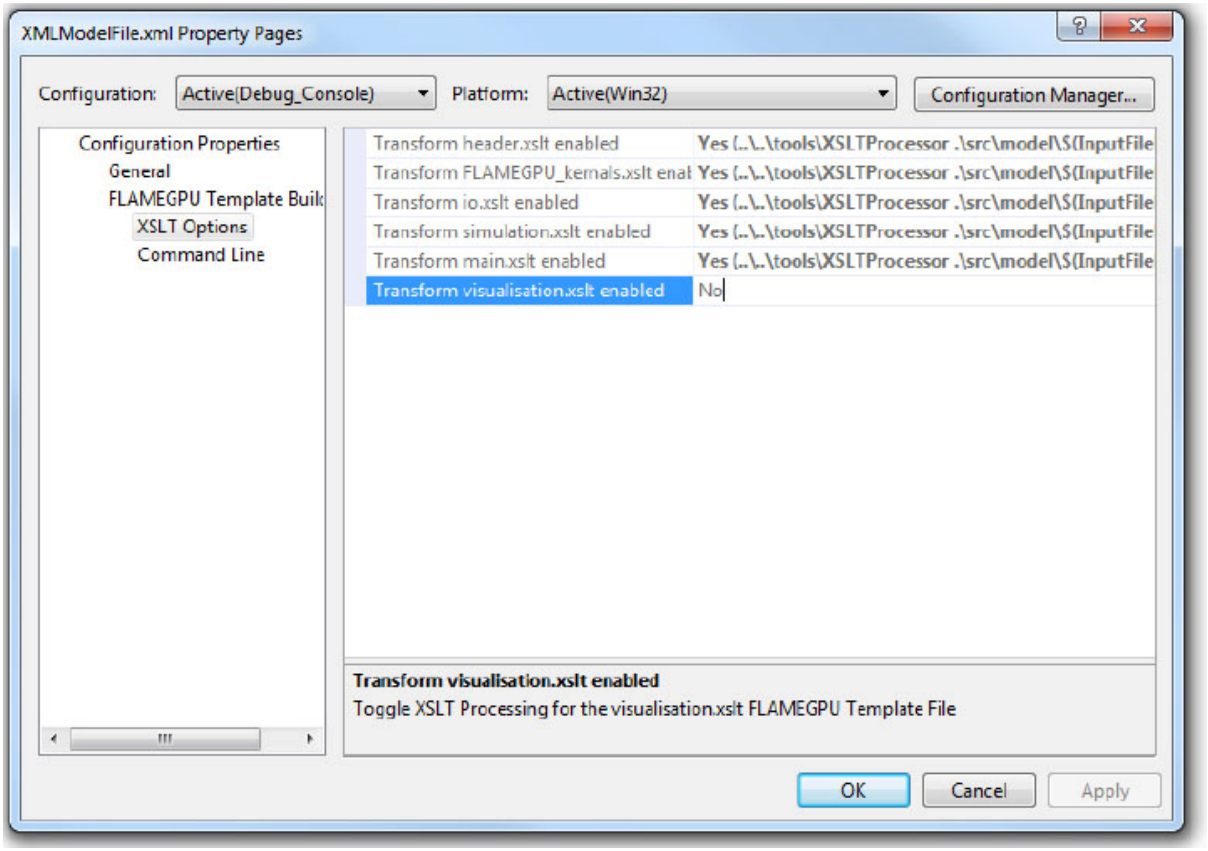

Рис. 2. Налаштування XSLT процесора для генерації динамічного коду симуляції.

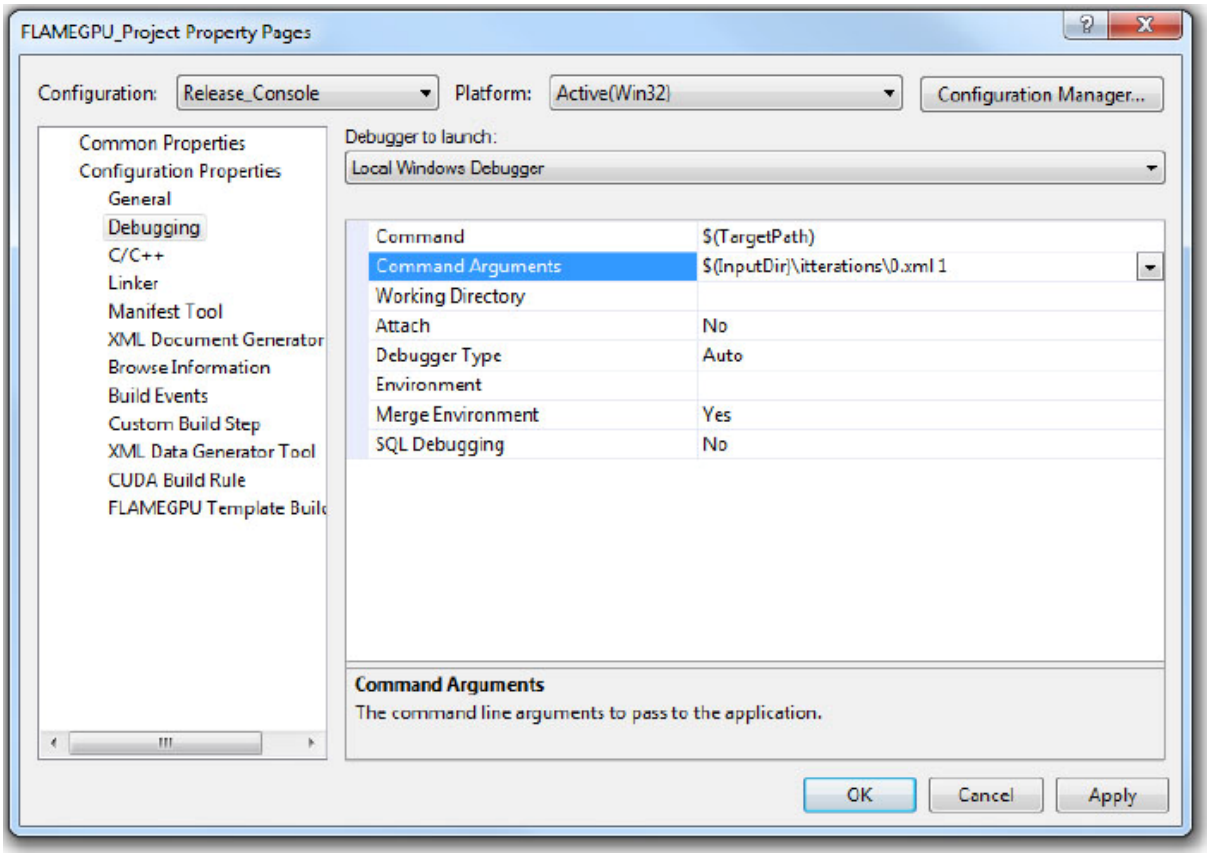

Рис. 3. Налаштування шляху до файла з початковими даними симуляції.

Контрольні запитання

1) Що таке агентне моделювання (ABM – Agent-Based Modelling)?
2) Властивості агентної системи?
3) За допомоги чого спілкуються агенти типу X-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