FLAME GPU Technical Report and User Guide

Document Version 1.0
For FLAME GPU 1.0.4

Dr Paul Richmond (Research Associate)
Department of Automatic Control and Systems Engineering (ACSE)
Member of the Computer Graphics Group (Department of Computer Science)
The University of Sheffield
P.Richmond@Sheffield.ac.uk
www.paulrichmond.staff.shef.ac.uk
www.flamegpu.com

Document Contents

1 Introduction

Agent Based Modelling (ABM) is a technique for the computational simulation of complex interacting systems through the specification of the behaviour of a number of autonomous individuals acting simultaneously. This is a bottom up approach, in contrast with the top down one of modelling the behaviour of the whole system through dynamic mathematical equations. The focus on individuals is considerably more computationally demanding, but provides a natural and flexible environment for studying systems demonstrating emergent behaviour. Despite the obvious parallelism, traditionally frameworks for ABM fail to exploit this and are often based on highly serialised algorithms for manipulating mobile discrete agents. Such an approach has serious implications, placing stringent limitations on both the scale of models and the speed at which they may be simulated. The purpose of the Flexible Large Scale Agent Modelling Environment (FLAME) framework is to address the limitations of previous agent modelling software by targeting the high performance Graphics Processing Unit (GPU) architecture. The FLAME GPU framework is designed with parallelism in mind and as such allows agent models to scale to massive sizes and ensures simulations run within reasonable time constrains. In addition to this visualisation is easily achievable as simulation data is held entirely within GPU memory where it can be rendered directly.

1.1 High Level Overview of FLAME GPU

Technically the FLAME GPU framework is not a simulator, it is instead a template based simulation environment that maps formal descriptions of agents into simulation code. The representation of an agent is based on the concept of a communicating X-Machine (which is an extension to the Finite State Machine which includes memory). Whilst the X-Machine has very formal definition X-Machine agents can be thought of a state machines which are able to communicate via messages which are stored in a globally accessible message lists. Agent functionality is exposed as a set of state transition functions which move agents from one internal state to another. Upon changing state, agents update their internal memory through the influence of messages which may be either used as input (by iterating message lists) or as output (where information may be passed to the message lists for other agents to read). FLAME GPU uses agent function scripting for this purpose where script is defined in a number of *Agent Function Files*. Simulation models are specified using a format called X-Machine Mark-up Language (*XMML*) which is XML syntax with Schemas governing the content. A typically XMML model file consists of a definition of a number of X-Machine agents (including state and memory information as well as a set of agent transition functions), a number of message types (each of which has a globally accessible message list) and a set of simulation layers which define the execution order of agent functions (which constitutes a single simulation iteration). Throughout a simulation, agent data is persistent however message information (and in particular message lists) is persistent only over the lifecycle of a single iteration. This allows a mechanism for agents to iteratively interact in a way which allows emergent global group behaviour.

The process of generating a FLAME GPU simulation is described by the Figure 1. The use of XML schemas forms a large part of the process where polymorphic like extension allows a base schema specification to be extended with a number of GPU specific elements. Given an XMML model definition, template driven code generation is achieved through Extensible Stylesheet Transformations (XSLT). XSLT is a flexible functional language based on XML (validated itself using a W3C specified Schema) and is suitable for the translation of XML documents into other document formats using a number of compliant processors (although the FLAME GPU SDK provides its own). Through the specification of a number of *XSLT Simulation Templates* a *Dynamic Simulation API* is generated which links with the *Agent Function Files* to generate a simulation program.

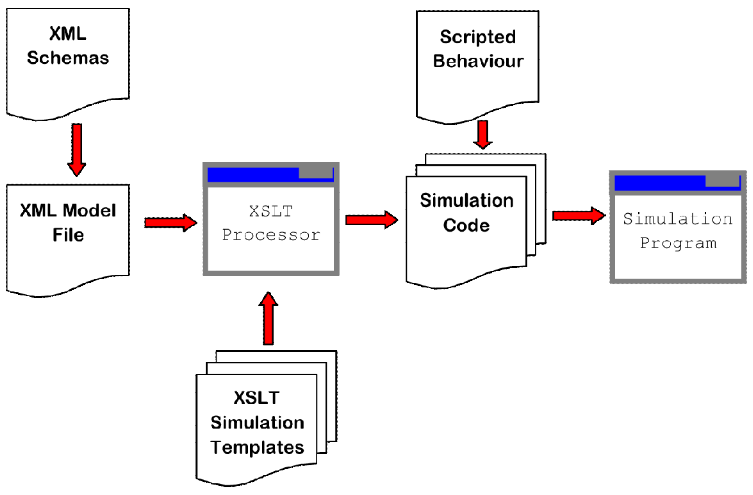


Figure 1 - FLAME GPU Modelling and Simulation Processes

1.2 Purpose of This Document

The purpose of this document is to describe the functional parts which make up a FLAME GPU simulation as well as providing guidance on how to use the FLAME GPU SDK. Chapter 2 describes in detail the syntax and format of the XMML Model file. Chapter 3 describes the syntax of use of agent function scripts and how to use the dynamic simulation API and Chapter 4 describes how to generate simulation code and run simulations from within the Visual Studio IDE. This document does not act as a review of background material relating to GPU agent modelling, nor does it provide details on FLAME GPUs implementation or descriptions of the FLAME GPU examples. For more in depth background material on agent based simulation on the GPU, the reader is directed towards the following document;

Richmond Paul, Walker Dawn, Coakley Simon, Romano Daniela (2010), "High Performance Cellular Level Agent-based Simulation with FLAME for the GPU", Briefings in Bioinformatics, 11(3), pages 334-47.

For details on the implementation including algorithms and techniques the reader is directed towards the following publication;

Richmond Paul (2011), "Template Driven Agent Based Modelling and Simulation with CUDA", GPU Computing Gems Emerald Edition (Wen-mei Hwu Editor), Morgan Kaufmann, March 2011, ISBN: 978-0-12-384988-5

Richmond Paul, Coakley Simon, Romano Daniela (2009), "A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA", Proc. of 8th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 2009), May, 10–15, 2009, Budapest, Hungary

Some examples of FLAME GPU models are described in the following publications;

Richmond Paul, Coakley Simon, Romano Daniela (2009), "Cellular Level Agent Based Modelling on the Graphics Processing Unit", Proc. of HiBi09 - High Performance Computational Systems Biology, 14-16 October 2009,Trento, Italy (additional detail in the BiB paper)

Karmakharm Twin, Richmond Paul, Romano Daniela (2010), " Agent-based Large Scale Simulation of Pedestrians With Adaptive Realistic Navigation Vector Fields", To appear in Proc. of Theory and Practice of Computer Graphics (TPCG) 2010, 6-8th September 2010, Sheffield, UK

2 FLAMEGPU Model Specification

2.1 Introduction

FLAME GPU models are specified using XML format within an XMML document. The syntax of the model file is governed by two XML Schemas, an abstract base Schema describes the syntax of a basic XMML agent model (compatible with HPC and CPU versions of the FLAME framework) and a concrete GPU Schema extension this to add various bits of additional model information. Within this chapter the XML namespace (`xmlns`) `gpu` is used to qualify the XML elements which extend the basic Schema representation. A high level overview of a an XMML model file is described below with various sections within this chapter describing each part in more detail.

```
<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>
```

2.2 The Environment

The environment element is used to hold global information which relates to the simulation. This information includes, zero or more constant, or global, variables (which are constant for all agents over the period of either the simulation or single simulation iteration), a single non optional function file containing agent function script (see Chapter 3) and an optional number of initialisation functions.

```
<gpu:environment>
    <gpu:constants>...</gpu:constants>              //optional
    <gpu:functionFiles>...</gpu:functionFiles>      //not optional
    <gpu:initFunctions>...</gpu:initFunctions>      //optional
</gpu:environment>
```

2.2.1 Simulation Constants (Global Variables)

Simulation constants are defined as (global) variables and may be of type `int`, `float` or `double` (on GPU hardware with double support i.e. CUDA Compute capability 2.0 or beyond). Constant variables must each have a unique name which is used to reference them within simulation code and can have an optional static array length (of size greater than 0). The `description` element `arrayLength` element and `defaultValue` element are all optional. The below code shows the specification of two constant variables: the first represents a single `int` constant (with a default value of 1), the second indicates an `int` array of length 5. Details on how to set constant variables is described in section 3.9.

```
<gpu:constants>
    <gpu:variable>
        <type>int</type>
        <name>const_variable</name>
        <description>none</description>
        <defaultValue>1</defaultValue>
    </gpu:variable>
    <gpu:variable>
        <type>int</type>
        <name>const_array_variable</name>
        <description>none</description>
        <arrayLength>5</arrayLength>
    </gpu:variable>
</gpu:constants>
```

2.2.2 Function Files

The `functionFiles` element is not optional and must contain a single `file` element which defines the name of a source code file which holds the scripted agent functions. More details on the format of the function file are given in Chapter 3. The example below shows the correct XML format for a function file named `functions.c`.

```
<gpu:functionFiles>
    <file>functions.c</file>
</gpu:functionFiles>
```

2.2.3 Initialisation Functions

Initialisation functions are user defined functions which can be used to set constant global variables. Any initialisation functions defined within the `initFunctions` element are called a single time by the automatically generated simulation code in the order that they appear during the initialisation of the simulation. If an `initFunctions` element is specified there must be at least a single `initFunction` child element with a unique `name`. Section 3.9.1 demonstrates how to specify initialisation functions within a function file.

```
<gpu:initFunctions>
    <gpu:initFunction>
        <gpu:name>initConstants</gpu:name>
    </gpu:initFunction>
</gpu:initFunctions>
```

2.3 Defining an X-Machine Agent

A XMML model file must contain a single `xagents` element which in turn must define at least a single `xagent`. An `xagent` is an agent representation of an X-Machine and consists of a name, optional description, an internal memory set (M in the formal definition), a set of agent functions (or next state partial functions, F, in the formal definition) and a set of states (Q in the formal definition). In addition to this FLAMEGPU requires two additional pieces of information (which are not required in the original XMML specification), a `type` and a `bufferSize`. The `type` element refers to the type of agent with respect to its relation with its spatial environment. An agent `type` can be either `discrete` or `continuous`, discrete agents occupy non mobile 2D discrete spatial partitions (cellular automaton) where as `continuous` agents are assumed to occupy a continuous space environment (although in reality they may in fact be non spatial more abstract agents). As all memory is pre-allocated on the GPU a `bufferSize` is required to represent the largest possible size of the agent population. That is the maximum number of x-machine agent instances of the format described by the XMML model. There is no performance disadvantage to using a large `bufferSize` however it is the user's responsibility to ensure that the GPU contains enough memory to support large populations of agents. It is recommended that the `bufferSize` always be a power of two number (i.e. `1024`, `2048`, `4096`, `16384`, etc) as it will most likely be rounded to one during simulation. For `discrete` agents the `bufferSize` is currently limited to only power of 2 numbers which have squarely divisible dimensions (i.e. the square of the `bufferSize` must be a whole number). If at any point in the simulation exceeds the stated `bufferSize` then the user will be warned at the simulation will exit. Each expandable aspect of an XMML agent representation in the below example is discussed within this section with the exception of agent functions, which due to their dependence of the definition of messages, are discussed later in section 2.5.

```
<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>
```

2.3.1 Agent Memory

Agent memory consists of a number of variables (at least one) which are use to hold information. An agent `variable` must have a unique `name` and may be of `type` `int`, `float` or `double` (CUDA compute capability 1.3 or beyond). Currently agent memory only supports the use of single memory values (i.e. no static or dynamic arrays) and default values are always `0` unless a value is specified within the XML input states file. There are no specified limits on the maximum number of agent variables however the performance tips noted in section 4.7 should be taken into account. Below shows an example of agent memory containing four agent variables representing an agent identifier and three positional values.

```
<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>
```

2.3.2 Agent States

Agent `states` are defined as a list of `state` elements (Q in the X-Machine formal definition) with a unique and non optional `name`. As simulations within FLAMEGPU can continue indefinitely (or for a fixed number of iterations), terminal states (T in the formal definition) are not defined. The initial state (q_0) must however be defined within the `initialState` element and must correspond with an existing and unique state name from the list of states above it.

```
<states>
    <gpu:state>
        <name>state1</name>
    </gpu:state>
    <gpu:state>
        <name>state2</name>
    </gpu:state>
    <initialState>state1</initialState>
</states>
```

2.4 Defining Messages

Messages represent the information which is communicated between agents. An element `messages` contains a list of at least one `message` which defines a non optional `name` an optional `description` of the message, a list of variables, a partitioning type and a `bufferSize`. The `bufferSize` element is used in the same way that a `bufferSize` is used to define an X-Machine agent, i.e. the maximum number of this message type which may exist within the simulation at one time. The partitioning type may be one of three currently defined message partition schemes, non partitioned (`partitioningNone`), discrete 2D space partitioning (`partitioningDiscrete`) or 2D/3D spatially partitioned space (`partitioningSpatial`). Message partition schemes are used to ensure that the most optimal cycling of messages occurs within agent functions. The use of the partitioning techniques is described within this section, as are message variables.

```
<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>
```

2.4.1 Message Variables

The message `variables` element consists of a number of variables (at least one) which are use to hold communication information. A `variable` must have a unique `name` and may be of `type` `int`, `float` or `double` (CUDA Compute capability 2.0 or beyond). As with agent variables message variables only supports the use of single memory values (i.e. no static or dynamic arrays). Likewise there are no specified limits on the maximum number of message variables however increased message size will have a negative effect on performance in all partitioning cases (and in particular when non partitioned messages are used). The format of message variable specification shown below is identical to that of agent memory. The only exception is the requirement of certain variable names which are required by certain partitioning types. Non partitioned messages have no requirement for specific variables. Discrete partitioning requires two `int` `type` variables of `name` `x` and `y`. Spatial partitioning requires three `float` (or `double`) `type` variables named `x`, `y` and `z`. The example below shows an example of message memory containing two message variables named `id` and `message_variable`.

```
<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>
```

2.4.2 Non partitioned Messages

None partitioned messages do not use any filtering mechanism to reduce the number of messages which will be iterated by agent functions which use the message as input. None partitioned messages therefore require a brute force or *$O(n^2)$* message iteration loop wherever the message list is iterated. As non partitioned messages do not require any message variables with location information the partition type is particularly suitable for communication between non spatial or more abstract agents. Brute force iteration is obviously reasonably computationally expensive, however non partitioned message iteration requires very little overhead (or setup) and as a result for small numbers of messages it can be more efficient than either limited range technique. There is no strict rule governing performance and different GPU hardware will produce different results depending on it capability. It is therefore left to the user to experiment with different message partitioning types within a simulation. The example below shows the format of the `partitioningNone` element tag.

```
<gpu:partitioningNone/>
```

2.4.3 Discrete Partitioned Messages

Discrete partitioned messages are messages which may only originate from non mobile discrete agents (cellular automaton). A discrete partitioning message scheme requires the specification of a `radius` which indicates the range (in in 2D discrete space) which a message iteration will extend to. A `radius` value of `0` indicates that only a single message will be returned from message iteration. A value of greater than `0` indicates that message iteration will loop through ±`radius` in both the `x` and a `y` dimension (e.g. a range of `1` will iterate `3x3=9` messages, a range of `2` will iterate `5x5=25`). In addition to this the agent memory is expected to contain `x` and `y` variables of type `int`. As with discrete agents it is important to ensure that messages using discrete partitioning use only supported buffer sizes (power of 2 and squarely divisible). The width and height of the discrete message space is then defined as the square of the `bufferSize` value.

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

2.4.4 Spatially Partitioned Messages

Spatially partitioned messages are messages which originate from continuous spaced agents in either a 3D environment (i.e. agents with `x` `y` and `z` variables). A spatially partitioned message scheme requires the specification of both a `radius` and a set of environment bounds. The `radius` represents the range in which message iteration will extend to (from its originating point). The environment bounds represent the size of the space which massages may exist within. If a message falls outside of the environment bounds then it will be bound to the nearest possible location within it. The space within the defined bounds is partitioned according to the radius with a total of `P` partitions in each dimension, where for each dimension;

```
P = ceiling((max_bound - min_bound) / radius)
```

The partitions dimensions are then used to construct a partition boundary matrix (an example of use within message iteration is provided in section 3.5.2) which holds the indices of messages within each area of partitioned space. Spatially partitioned message iteration can then iterate a varying number of messages from a fixed number of adjacent partitions in partition space to ensure each message within the specified radius has been considered. The following example defines a spatial partition in three dimensions. For continuously spaced agents in 2D space `P` should be equal to `1` and therefore a `zmin` of `0` would require a `zmax` value equal to `radius` (even in this case a message `variable` with `name` `z` is still required).

```
<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>
```

2.5 Defining an Agent function

An optional list of agent `functions` is described within an X-Machine agent representation and must contain a list of at least a single agent `function` element. In turn, a function must contain a non optional name, and optional description, a current state, next state, an optional single message input, and optional single message output, an optional single agent output, an optional global function condition, an optional function condition, a reallocation flag and a random number generator flag. The current state is defined within the `currentState` element and is used to filter the agent function by only applying it to agents in the specified state. After completing the agent function agents then move into the state specified within the `nextState` element. Both the current and `nextState` values are required to have values which exist as a `state/name` within the state list (`states`) definition. The `reallocate` element is used as an optional flag to indicate the possibility that an agent performing the agent function may die as a result (and hence require removing from the agent population). By default this value is assumed true however if a value of false is specified then the processes for removing dead agents will not be executed even if an agent indicates it has died (see agent function definitions in section 3.3). The `RNG` element represents a flag to indicate the requirement of random number generation within the agent function. If this value is true then an additional parameter (demonstrated in section 3.7) is passed to the agent function which holds a number of seeds used for parallel random number generation.

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

2.5.1 Agent Function Message Inputs

An agent function message `input` indicates that the agent function will iterate the list of messages with a name equal to that specified by the non optional `messageName` element. It is therefore required that the `messageName` element refers to an existing `message/name` defined within the XMML document. In addition to this an agent function cannot iterate a list of messages without specifying that it is an `input` within the XMML model file (message iteration functions are parameterised to prevent this).

```
<inputs>
    <gpu:input>
        <messageName>message_name</messageName>
    </gpu:input>
```

```
</inputs>
```

2.5.2 Agent Function Message Outputs

An agent function message `output` indicates that the agent function will output a message with a name equal to that specified by the non optional `messageName` element. The `messageName` element must therefore refer to an existing `message/name` defined within the XMML document. It is not possible for an agent function script to output a message without specifying that it is an `output` within the XMML model file (message output functions are parameterised to prevent this). In addition to the `messageName` element a message output also requires a `type`. The `type` may be either `single_message` or `optional_message,` where `single_message` indicates that every agent performing the function outputs exactly one message and `optional_message` indicates that agent's performing the function may either output a single message or no message. The type of messages which can be output by discrete agents are not restricted however continuous `type` agents can only output messages which do not use discrete message partitioning (e.g. no partitioning or spatial partitioning). The example below shows a message output using `single_message` `type`. This will assume every agent outputs a message, if the functions script fails to output a message for every agent a message with default values (of `0`) will be created instead.

```
<outputs>
    <gpu:output>
        <messageName>message_name</messageName>
        <gpu:type>single_message</gpu:type>
    </gpu:output>
</outputs>
```

2.5.3 Agent Function X-Agent Outputs

An agent function `xagentOutput` indicates that the agent function will output an agent with a name equal to that specified by the non optional `xagentName` element. This differs slightly from the formal definition of an x-machine which does not explicitly define a technique for the creation of new agents but adds functionality required for dynamically changing population sizes during simulation runtime. The `xagentName` element belonging to an `xagentOutput` element must refer to an existing `agent/name` defined within the XMML document. It is not possible for an agent function script to output a agent without specifying that it is an `xagntOutput` within the XMML model file (agent output functions are parameterised to prevent this). In addition to the `xagentName` element a message output also requires a `state`. The `state` represents the `state` from the list of `state` elements belonging to the specified `agent\name` that the new agent should be in after it has been created. Only `continuous` `type` agents are allowed to output new agents (which must also be of `type` `continuous`). The creation of new `discrete` agents is not permitted under any circumstance. An `xagentOutput` does not require a `type` (as is the case with a message output) and any agent function outputting an agent is assumed to be optional. I.e. each agent performing the function may output either one or zero agents.

```
<xagentOutputs>
    <gpu:xagentOutput>
        <xagentName>agent_name</xagentName>
        <state>state1</state>
    </gpu:xagentOutput>
</xagentOutputs>
```

2.5.4 Function Conditions

An agent function `condition` indicates that the agent function should only be applied to agents which meet the defined `condition` (and in the correct state specified by `currentState`). Each function condition consists of three parts a left hand side statement (`lhs`), an `operator` and a right hand side statement (`rhs`). Both the `lhs` and `rhs` elements may contain either a `agentVariable` a `value` or a recursive `condition` element. An `agentVariable` element must refer to a agent variable defined within the agents list of variable names (i.e. `gpu:xagent/memory/gpu:variable/name`). A `value` element may refer to any numeric value or constant definition (defined within the agent function scripts). The use of recursive conditions is demonstrated below by embedding a `condition` within the `rhs` element of the top level condition.

```
<condition>
    <lhs>
        <agentVariable>variable_name</agentVariable>
    </lhs>
    <operator>&lt;</operator>
    <rhs>
        <condition>
            <lhs>
                <agentVariable>variable_name2</agentVariable>
            </lhs>
            <operator>+</operator>
            <rhs>
                <value>1</value>
          </rhs>
        </condition>
    </rhs>
</condition>
```

In the above example the function condition generates the following pseudo code function guard;

```
(variable_name) < ((variable_name2)+(1))
```

The `condition` element may refer to any logical operator. Care must be taken when using angled brackets which in standard form will cause the XML syntax to become invalid. Rather than the left hand bracket (less than) the correct xml syntax of `&lt;` should be used. Likewise the right hand bracket (greater than) should be replaced with `&gt;`.

2.5.5 Global Function Conditions

An agent global function condition is similar to an agent function in its syntax however it acts as a global switch to determine if the function should be applied to either all or none of the agents (within the correct state specified by `currentState`). In the case of every agent evaluating the global function condition to `true` (or to the value specified by the `mustEvaluateTo` element) the agent function is applied to all of the agents. In the case that any of the agents evaluate the global function condition to false (or to the logical opposite of the value specified by the `mustEvaluateTo` element) then the agent function will be applied to none of the agents. As with an agent function condition a `globalCondition` consists of a left hand side statement (`lhs`), an `operator` and a right hand side statement (`rhs`). The syntax of the left hand side statement (`lhs`), the `operator` and the right hand side statement (`rhs`) is the same as with an agent function condition and may use recursion to generate a complex conditional statement. The `maxItterations` element is used to limit the number of times a function guarded by the global condition can be avoided (or evaluated as the logical opposite of the value specified by the `mustEvaluateTo` element). For example, the definition at the end of this section, resulting in the following pseudo code condition;

```
(((movement) < (0.25)) == true)
```

May be evaluated as false up to 200 times (i.e. in 200 separate simulation iterations) before the global condition will be ignored and the function is applied to every agent. Following maximum number of iterations being reached the iteration count is reset once the agent function has been applied.

```
<gpu:globalCondition>
    <lhs>
        <agentVariable>movement</agentVariable>
    </lhs>
    <operator>&lt;</operator>
    <rhs>
        <value>0.25</value>
    </rhs>
    <gpu:maxItterations>200</gpu:maxItterations>
    <gpu:mustEvaluateTo>true</gpu:mustEvaluateTo>
</gpu:globalCondition>
```

2.6 Function Layers

Function layers represent the control flow of the simulation processes. The sequence of layers defines the sequential order in which agent functions are executed. Complete execution of every layer of agent functions represents a single simulation itteration which may be repeated any number of times. Syntatically within the model definition a single `layers` element must contain at least one (or more) `layer` element. Each `layer` element may contain at least one (or more) `gpu:layerFunction` element which defines only a `name` which must relate to a function name specified within a correspeonding `xagents/gpu:xagent/functions/gpu:function/name`. Within a given layer, the order of execution of layer functions should not be assumed to be sequential (although in the current version of the software it is, future versions will execute functions within the same layer in parallel). For the same reason functions within the same layer should not have any communication or internal dependencies (for example via message communications or execution order dependency) in which case they should instead be represented within separate layers which guarantee execution order and global synchronisation between the functions. The below example demonstrates the syntax of specifying a simulation consisting of three agent functions. There are no dependencies between `function1` and `function2` which in this case can be thought of as being functions from two different agents definitions with no shared message input or output.

```
<layers>
    <layer>
        <gpu:layerFunction>
            <name>function1</name>
        </gpu:layerFunction>
        <gpu:layerFunction>
            <name>function2</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>function3</name>
        </gpu:layerFunction>
    </layer>
</layers>
```

2.7 Initial XML Agent Data

The initial agent data information is stored in an XML file which is used passed to the simulator as a parameter before running the simulation. Within this initial agent data XML file a single `states` element contains a single iteration number (`itno`) and any number (including 0) of `xagent` elements. The syntax of the `xagent` element depends on the agent definitions contained within the XMML model definition file. A `name` element is always required and must represent an agent name contained within a `xgents/gpu:agent/name` element in the XMML model definition. Following this an element may exist for each of the named agents memory variables (`xagents/gpu:agent/memory/gpu:variable/name`). Each named element is then expected to contain a value of the same `type` as the agent memory variable defined. If the initial agent data XML file neglects to specify the value of a variable defined within an agents memory then the value is assumed to be zero. If an element defines a variable name which does not exist within the XMML model definition then a warning is generated and the value is ignored. The example below represents a single agent corresponding to the agent definition in section 2.3.

```
<states>
    <itno>0</itno>
    <xagent>
        <name>AgentName</name>
        <id>1</id>
        <x>21.088</x>
        <y>12.834</y>
        <z>5.367</z>
    </xagent>
    <xagent>...</xagent>
</states>
```

Care must be taken in ensuring that the set of initial data for the simulation does not exceed any of the defined `bufferSize` (i.e. the maximum number of a given type of agents) for any of the agents. If buffer size is exceeded during initial loading of the initial agent data then the simulation will produce an error.

Another special case to consider is the use of 2D discrete agents where the number of agents within the set of initial agent data must match exactly the `bufferSize` (which must also be a power of 2) defined within the XMML models agent definition. Furthermore the simulation will expect to find initial agents stored within the XML file in row wise ascending order.

3 FLAMEGPU Agent Function Scripts and the Simulation API

3.1 Introduction

Agent function scripts define the behaviour of agents by describing changes to memory and through the iteration and creation of messages and new agents. The behaviour of the agent function is described from the perspective of a single agent however the simulator will apply in parallel the same agent function code to each agent which is in the correct start state (and meets any of the defined function conditions). Agent function scripts are defined using a simple C based syntax with the agent function declarations, and more specifically the function arguments dependant on the XMML function definition. The use of message input and output as well as random number generation will all change the function arguments in a way which is described within this section. Likewise the simulation API functions for message communication are dependent on the definition of the simulation model contained with the XMML model definition. A single C source file is required to hold all agent function declarations and must contain an include directive for the file `"header.h"` which contains model specific agent and message structures. Agent functions are free to use many features of common C syntax with the following important exceptions;

- Globally Defined Variables: i.e. Variables declared outside of any function scope. Are not permitted and should instead be defined as global variables within the XMML model file and used as described in section 2.2.1. Note: The use of pre-processor macro directives for constants is supported and can be freely used without restriction.

- Include Directives: Are permitted however as agent functions are functions which are run on the GPU during simulation they may not call non GPU code. This implies that including and linking with non CUDA libraries is not supported.
- External Function Calls: As above external function calls may only be made to CUDA `__device__` functions. Many common math functions calls such as *sin*, *cos*, etc. are supported via native GPU implementations and can be used in exactly the same way as standard C code. Likewise additional "helper" functions can be defined and called from agent functions by prefixing the helper function using the `__FLAME_GPU_FUNC__` macro (which signifies it can be run on the GPU device).

The following chapter describes the syntax and use of agent function scripts including any arguments which must be passed to the agent or simulation API functions. As agent functions and simulation API functions are dynamic (and based on the XMML model definition) it is often easier to first define a model and use the technique described within section 4.2 to automatically generate a functions file containing prototype agent function files and API system calls. Alternatively section 3.8 describes fully the expected argument order for agent function arguments.

3.2 Agent and Message Data Structures

Access to agent and message data within the agent function scripts is provided through the use of agent and message data structures which contain variables matching those defined within the XMML definitions. For each agent in the simulation a structure is defined within the dynamically generated `header.h` with the name `xmachine_memory_agent_name`, likewise each message defines a structure with the name `xmachine_message_message_name`. In both cases the structures contain a number of private variables prefixed with an underscore (_) which are used by the API functions and should not be modified directly. In addition to this the simulation API defines structures of arrays to hold agent and message list information. Agent lists are named `xmachine_memory_agent_name_list` and message lists are named `xmachine_message_message_name_list`. These lists are passed as arguments to agent functions and should only be used in conjunction with the simulation API functions for message iteration and the adding of messages and agents. List structures should never be accessed directly as doing so will produce undefined behaviour.

3.3 A Basic Agent Function (Updating and Agents Memory)

The following example shows a simplistic agent function (named `function1`) which has no message input or output and only updates the agents internal memory. All FLAME GPU agent functions are first prefixed with the macro definition `__FLAME_GPU_FUNC__`. In this basic example the agent function has only a single argument, a pointer to an agent structure of type `xmachine_memory_myAgent` called `xmemory`.. In the below example the agent name is `myAgent` and the agent memory contains two variables `x` and `no_movements` of type `float` and `int` respectively. The return type of FLAME GPU functions is always int. A return value of anything other than 0 indicates that the agent has dies and should be removed from the simulation (unless the agent function definition had specifically set the `reallocate` element value to false in which case any agent deaths will be ignored).

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

      return 0;
}
```

3.4 Use of the Message Output Simulation API

Within an agent function script message output is possible by using a message output function. For each message type defined within the XMML model definition the dynamically generated simulation API will create a message output function of the following form;

```
add_message_name_message(message_name_messages, args...);
```

Where `message_name` refers to the value of the messages `name` element within the message specification and *`args`* is a list of named arguments which correspond to the message variables (see section 2.4.1). Agent functions may only call a message output function for the message name defined within the function definitions output (see 2.5.2). This restriction is enforced as message output functions require a pointer to a message list which is passed as an argument to the agent function (`xmachine_message_location_list` in the below example). Agents are only permitted to output a single message per agent function and repeated calls to an add message function will result in previous message information simply being overwritten. The example below demonstrates an agent function (`output_message`) belonging to an agent named `myAgent` which outputs a message with four variables. For clarity the message output function prototype (normally found in `header.h`) is also shown.

```
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;
}
```

3.5 Use of the Message Input Simulation API

As with message outputs, iterating message lists (message input) within agent functions is made possible by the use of dynamically generated message API functions. In general two functions are provided for each named message, a `get_first_message_name_message(args...)` and `get_next_message_name_message(args...)` the second of which can be used within a while loop until it returns a `NULL` value indicating the end of the message list. The arguments of these functions differ slightly depending on the partitioning scheme used by the message. The following subsections describe these in more detail. Regardless of the partitioning type a number of important rules must be observed when using the message functions. Firstly it is essential that message loop complete naturally. I.e. the `get_next_message_name_message` function must be called without breaking from the while loop until the end of the message list is reached. Secondly agent functions must not directly modify messages returned from the get message functions. Changing message data directly will result in undefined behaviour and will most likely crash the simulation

3.5.1 Non Partitioned Message Iteration

For non partitioned messages the dynamically generated message API functions are relatively simple and the arguments which are passed to the API functions are also required by all other message partitioning schemes. The get first message API function (i.e. `get_first_message_name_message`) takes only a single argument which is a pointer to a message list structure (of the form xmachine_message_message_name_list) which is passed as an argument to the agent function. The get next message API function (i.e. get_next_message_name_message) takes two arguments, the previously returned message and the message list. The below example shows a complete agent function (input_messages) demonstrating the iteration of a message list (where the message name location is highlighted within the structure and API names for clarity). The while loop continues until the get next message API function returns a NULL (or false) value. In the below example the location message is used to calculate an average position of all the locations specified in the message list. The agent then updates three of its positional values to move towards the average location (cohesion).

```
__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;
}
```

3.5.2 Spatially Partitioned Message Iteration

For spatially partitioned messages the dynamically generated message API functions rely on the use of a Partition Boundary Matrix (PBM). The PBM holds important information which determines which agents are located within the spatially partitioned areas making up the simulation environment. Wherever a spatially partitioned message is defined as a function input (within the XMML model definition) a PMB argument should directly follow the input message list in the list of agent function arguments. As with non partitioned messages the first argument of the get first message API function is the input message list. The second argument is the PBM and the subsequent 3 arguments represent the position which the agent would like to read messages from (which in almost all cases is the agent position). The get next message API function differs only from the non partitioned example in that the PBM is passed as an additional parameter. The example below shows the same example as in the previous section but using a spatially partitioned message type (rather than the non partitioned type). The differences between the function arguments in the previous section are highlighted in red as is the use of a helper function in_range. The purpose of the in_range function is to check the distance between the agent position and the message. This is important as the messages returned by the get next message function represent any messages within the same or adjacent partitioning cells (to the position specified by the get first message API function). On average roughly 1/3 of these values will be within the actually range specified by the message definitions range value.

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

    /* Get the first location messages */
    xmachine_message_location* location_message;
    message = get_first_location_message(location_messages,
                                         partition_matrix,
                                         xmemory->x,
                                         xmemory->y,
                                         xmemory->z);

    /* 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);
    }

    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;
}
```

3.5.3 Discrete Partitioned Message Iteration

For discretely partitioned messages the dynamically generated message API functions differ from those of non partitioned only in that two additional parameters must be passed to the get first message API function. The two integer arguments represent the position which the agent would like to read messages from within the cellular environment (as with spatially partitioning this is usually the agent position). These values of these arguments must therefore be within the width and height of the message space itself (the square of the messages `bufferSize`). In addition to the additional arguments, the discrete message API functions also make use of template parameterisation to distinguish between the type of agent requesting message information. The template parameters which may be used are either DISCRETE_2D (as in the example below) or CONTINUOUS. This parameterisation is required as underlying implementation of the message API functions differs between the two agent types. The example below shows an agent function (`input_messages`) of a discrete agent (named `cell`) which iterates a message list (of `state` messages) to count the number neighbours with a `state` value of `1`. The differences between the function arguments in the section describing non partitioned message iteration are highlighted in red as is the function parameterisation.

```
__FLAME_GPU_FUNC__ int input_messages(xmachine_memory_cell* xmemory,
                                      xmachine_message_state_list* state_messages)
{
    int neighbours = 0;

    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);
    }

    xmemory->neighbours = neighbours;

    return 0;
}
```

3.6 Use of the Agent Output Simulation API

Within an agent function script agent output is possible by using a message output API function. For each agent type defined within the XMML model definition the dynamically generated simulation code will create an agent output function of the following form;

```
add_agent_name_agent(agent_name_agent, args...);
```

Where `agent_name` refers to the value of the agents `name` element within the agent specification and *`args`* is a list of named arguments which correspond to the agents memory variables (see section 2.5.3). Agent functions may only output a single type of agent and are only permitted to output a single agent per agent function. As with message outputs, repeated calls to an add agent function will result in previous agent information simply being overwritten. The example below demonstrates an agent function (`create_agent`) which outputs a new agent by creating a clone of itself. For clarity the agent output API function prototype (normally found in `header.h`) is also shown.

```
add_myAgent_agent(xmachine_memory_myAgent_list* myAgent_agents,
                  int id, float x, float y, float z);

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

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

    add_myAgent_agent(myAgent_agents, id, x, y, z);

    return 0;
}
```

3.7 Using Random Number Generation

Random number generation is provided via the rnd API function which uses template parameterisation to distinguish between either discrete (where a parameter value of DISCRETE_2D should be used) or continuous (where a parameter value of CONTINUOUS should be used) spaced agents. If a template parameter value is not specified then the simulation will assume a DISCRETE_2D value which will work in either case but is more computationally expensive. The API function has a single argument, a pointer to a RNG_rand48 structure which contains random seeds and is passed to agent functions which specify a true value for the RNG element in the XMML function definition. The example below shows a simple agent function (with no input or outputs) demonstrating the random number generation to determine if the agent should die.

```
#define DEATH_RATE 0.1f

__FLAME_GPU_FUNC__ int kill_agent(xmachine_memory_myAgent* agent,
                                  RNG_rand48* rand48)
{
    float random;
    int die;

    die = 0;         /* agent does not die */
    random = rnd<CONTINUOUS>(rand48);

    if (random < DEATH_RATE)
        die = 1;     /* agent dies */

    return die;
}
```

3.8 Summary of Agent Function Arguments

Agent functions may use any combination of message input, output, agent output and random number generation resulting in a large number of agent function arguments which are expected to be in a specific and pre defined order. The following pseudo code demonstrates the order of a function containing all possible arguments. When specifying an agent function declaration this order must be observed.

```
__FLAME_GPU_FUNC__ int function(xmachine_memory_agent_name* agent,
                                xmachine_memory_ agent_name _list* output_agents,
                                xmachine_message_message_name_list* input_messages,
                                xmachine_message_message_name_PBM* input_message_PBM,
                                xmachine_message_message_name_list* output_messages,
                                RNG_rand48* rand48);
```

3.9 Setting Simulation Constants (Global Variables)

Simulation constants defined within the environment section of the XMML model definition may be directly referenced within an agent function using the name specified within the variable definition (see section 2.2.1). It is not possible to set constant variables within an agent function however the simulation API creates methods for setting simulation constants which may be called either at the start of the simulation (either manually or within an initialisation function) or between simulation iterations (for example as part of an interactive visualisation). The code below demonstrates the function prototype for setting a simulation constant with the name A_CONSTANT.

```
extern "C" void set_A_CONSTANT (float* h_A_CONSTANT);
```

The function is declared using the extern keyword which allows it to be linked to by externally compiled code such as a visualisation or custom simulation loop.

3.9.1 Initialisation Functions

Any initialisation functions defined within the XMML model file (see section 2.2.3) are expected to be declared within an agent function code file and will automatically be called before the first simulation iteration. The initialisation function declaration should be preceded with a `__FLAME_GPU_INIT_FUNC__` macro definition, should have no arguments and should return `void`. The below example demonstrated an initialisation function named `initConstants` which uses the simulation APIs dynamically created constants functions to set a constant named `A_CONSTANT`.

```
__FLAME_GPU_INIT_FUNC__ void initConstants()
{
     float const_value = 8.25f;
     set_A_CONSTANT(&const_value);
}
```

4 FLAME GPU Simulation and Visualisation

4.1 Introduction

The processes of building and running a simulation is made easier described within this chapter as are a number of tools and procedures which simplify the simulation code generation and compilation of simulation executables. In order to use the FLAME GPU SDK it should be placed in a directory which does not contain any spaces (preferably directly within the C: drive or root or root operating system drive). The host machine must also be running windows with a copy of the .NET runtime (used within the XSLT template processor) and must contain NVIDIA GPU hardware with Compute level 1.0.

4.2 Generating a Agent Functions Script

Section 3.8 previously described the exact argument order for agent function declarations however in most cases it is sensible to use the provided XSLT template (`functions.xslt` located in the *FLAMEGPU/templates* folder within the FLAME GPU SDK) to generate a agent function source file with empty agent function declarations automatically using your XMML model file. Once this has been generated the agent function scripts can be implemented within the function declarations rather easily. Care must however be taken in ensuring that if the XMML model file is later modified that the agent function arguments are updated manually where necessary. Likewise be careful not to overwrite any existing function source file when generating a new one using the XSLT template. Generation of blank function source files is not incorporated into the visual studio template project and must be manually accomplished. A .NET based XSLT processor is provided within the FLAME GPU SDK for this purpose (`XSLTProcessor.exe` located in the tool folder) and can be used via the command line as follows (or via the *GenerateFunctionsFileTemplate* batch file located in the *tools* folder of the FLAME GPU SDK);

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

Alternatively any compliant XSLT processor such as Xalan, Unicorn or even Firefox web browser can be used.

4.3 FLAME GPU Template Files

The FLAME GPU SDK contains a number of xslt templates which are used to generate the dynamic simulation code. A brief summary of the functionality and contents of each template file is as follows;

- `header.xslt` – This template file generates a header file which contains any agent and message data structures which are common in many of the other dynamically generated simulation source files. The template also generates function prototypes for simulation functions and functions which are visible externally within custom C or C++ code.

- `main.xslt` – This template file generates a source file which defines the main execution entry point function which is responsible for handling command line options and initialising the GPU device.
- `io.xslt` – This template file generates a source file which contains functions for loading initial agent XML data files (see section 2.7) into the simulation and saving the simulation state back into XML format.
- `simulation.xslt` – This template file generates a source file containing the host side simulation code which includes loading data to and from the GPU device and making a number of CUDA kernel calls which perform the simulation process.
- `FLAMEGPU_kernels.xslt` – This template file generates a CUDA header file which contains the CUDA kernels and device functions which make up the simulation.
- `visualisation.xslt` – This template file generates a source file which will allow basic visualisation of the simulation using sphere based representation of agents in 3D space. The source file is responsible for CUDA OpenGL interoperability and rending using OpenGL. The source file includes a `visualisation.h` file containing a number of definitions and variables which is not generated by any templates and should be specified manually.

4.4 Compilation Using Visual Studio

The FLAME GPU SDK and examples are targeted at CUDA 3.1 for compilation under windows using Visual Studio 2008. The decision to target windows is influenced entirely by visual studios excellent IDE tools and debugging facilities for CUDA programming and not as a result of any windows specific functionality. Porting to a Linux environment should be relatively simple. In addition to this Visual Studios XML editor includes validation support and XML tag auto completion which makes defining an XMML model incredibly easy. The following subsections describe the various aspects of a FLAME GPU project file and describe the build processes.

4.4.1 Visual Studio Project Build Configurations

The FLAME GPU examples and template project file contain build configurations for both 32 bit windows (Win32) and 64 bit Windows (x64) environments which can be changed using the "Solutions Platforms" drop down toolbar item. For each platform the project also contains four configurations for debugging (Debug) and release versions (Release) of both console based simulation and visualisation simulation. The two debug options disable all compiler optimisations and generate debug information for debugging host (non GPU) code and enables CUDA device emulation for GPU (device) debugging. The visualisation configurations enable building of visualisation code and specify a pre processor macro (`VISUALISATION`) which is used by a number of pre processor conditionals to change the simulations expected arguments (see section 4.5).

4.4.2 Visual Studio Project Virtual File Structure

Within the FLAME GPU examples and template projects code is organised into the following virtual folders;

- FLAME GPU – Consisting of a folder containing the FLAME GPU XML schemas and Code generating templates. These files are shared amongst all examples so editing them will change simulation code generated for other projects.
- FLAMEModel – Contains the XMML model file and the agent functions file (usually called `functions.c`). Note that the `functions.c` file is actually excluded from the build processes as it is built by the dynamically generated simulation.cu source file which includes it.

- Dynamic Code – Contains the dynamically generated FLAME GPU simulation code. This code will be overwritten each time the project is built so any changes to this files will be lost unless template transformation is turned off using the FLAME GPU build rule (see section 4.4.4).
- Additional Source Code – This folder should contain any hard coded simulation specific source or header files. By default the FLAME GPU project template defines a single `visualisation.h` file in this folder which may be modified to set a number of variables such as viewing distance and clipping. Within the FLAME GPU examples this folder is typically used to sore any model specific visualisation code which replaces the dynamically generated visualisation source file.

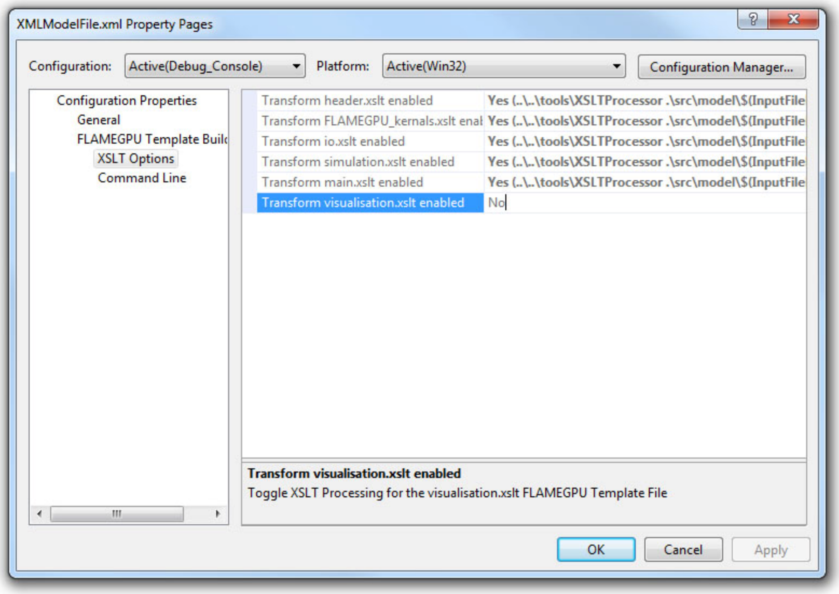
The physical folders of the SDK structure a self explanatory however it is worth noting that executable files generated by the Visual Studio build processes are output in the SDKs "bin" folder which also contains the CUDA run time *dlls*.

4.4.3 Build Processes

The Visual Studio build process consists of a number of stages which call various tools, compilers and linkers. The first of these is the FLAME GPU build tool (described in more detail in the following section) which generates the dynamic simulation code from the FLAME GPU templates and mode file. Following this the simulation code (within the Dynamic Code folder) is built using the CUDA build rule which compiles the source files using NVIDIAs `nvcc` CUDA compiler. Finally any C or C++ source files are compiled using MSVC compiler and are then linked with the CUDA object files to produce the executable. To start the build processes select the "Build" menu followed by "Build Solution" or use the F7 hotkey. If the first build step in the Visual Studio skips the FLAME GPU build tool a complete rebuilt can be forced by selecting the "Build" menu followed by "Rebuild Solution" (or Ctrl + Alt + F7).

4.4.4 FLAME GPU Build Rule Options

The FLAME GPU build rule is configured by selecting the XMML model file properties. Within the Build rue the XSLT options tab (Figure 2) allows individual template file transformations to be toggled on or off. These options are configuration specific and therefore console configurations by default do not processes the visualisation template.



Configuration: Active(Debug_Console)   Platform: Active(Win32)

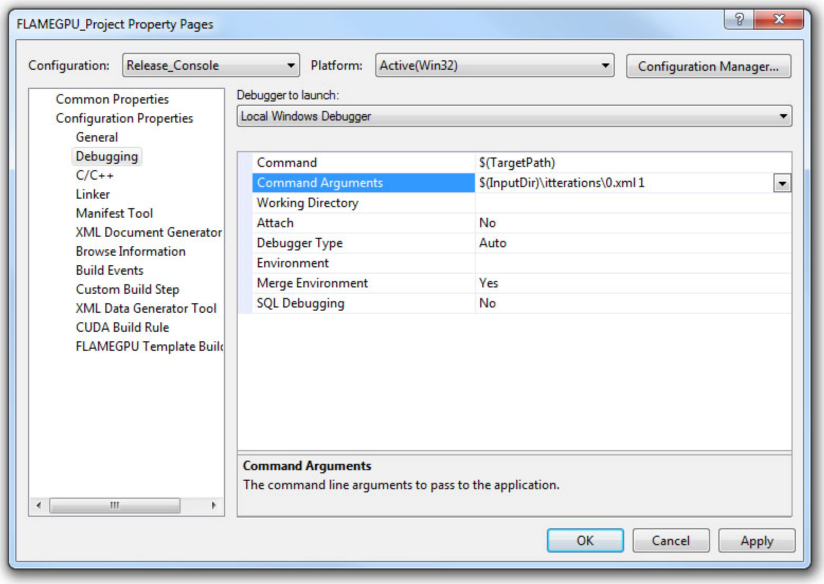
Transform header.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform FLAMEGPU_kernals.xslt enal	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform io.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform simulation.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform main.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform visualisation.xslt enabled	No

Transform visualisation.xslt enabled
Toggle XSLT Processing for the visualisation.xslt FLAMEGPU Template File

Figure 2 - FLAME GPU Build Rule XSLT Options Tab

4.4.5 Visual Studio Launch Configuration Command Arguments

In order to set the execution arguments (described in the next section) for simulation executable in any one of one of the four launch configurations, the "Command Arguments" property can be set form the Project Properties Page (Select "Project" Menu followed by "FLAMEGPU_Project Properties"). The "Command Arguments" property is located under "Configuration Properties -> Debug" (see Figure 3). Each configuration has its own set of "Command Arguments" so when moving between configurations these will need to be set. Likewise the "Configuration Properties" are computer and user specific so these cannot be preset and must be specified the first time each example is compiled and run. The Visual Studio macro `$InputDir` can be used to specify the working directory of the project file which makes locating initial agent data XML files for many of the examples much easier (these are normally located in the iterations folders of each example). Once a The Command Arguments have been set the simulation executable can be launched by selecting "Start Debugging" from the "Debug" menu or using the F5 hotkey (this is the same in both release and debug launch configurations).



Configuration: Release_Console
Platform: Active(Win32)

Debugger to launch:
Local Windows Debugger

Command	$(TargetPath)
Command Arguments	$(InputDir)\itterations\0.xml 1
Working Directory	
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No

Command Arguments
The command line arguments to pass to the application.

Figure 3 – FLAME GPU Project Properties Page

4.5 Simulation Execution Modes and Options

FLAME GPU simulations require a number of arguments depending on either console or visualisation mode. Both are described in the following subsections.

4.5.1 Console Mode

Simulation executables built for console execution require two arguments (usage shown below). The first of which is a file location for an initial agent XML file containing the initial agent data. The second argument is the number of simulation iterations which should be processed. A number of optional CUDA arguments may also be passed (i.e. `device=1` to specify the second CUDA enabled GPU device within the host machine should be used for simulation) if required.

```
FLAMEGPU_simulation.exe [XML model data] [Iterations] [Optional CUDA arguments]
```

The result of running the simulation will be a number of output XML files which will be numbered from `1` to `n`, where `n` is the number of simulations specified by the iterations argument. It is possible to turn XML output on or off by changing the definition of the `OUTPUT_TO_XML` macro located within the `main.xslt` template to true (`1`) false (`0`).

4.5.2 Visualisation Mode

Simulation executables built for visualisation require only a single argument (usage shown below) which is the same as the first argument for with console execution (an initial agent XML file). The number of simulations iterations is not required as the simulation will run indefinitely until the visualisation is closed. As with console execution it is possible to specify optional CUDA arguments.

```
Usage: main [XML model data] [Optional CUDA arguments]
```

Many of the options for the default visualisation are contained within the `visualisation.h` header file and include the following;

- SIMULATION_DELAY – Many simulations are executed extremely quickly making visualisation a blur. This definition allows an artificial delay by executing this number of visualisation draw loops before each simulation iteration is processed.
- WINDOW_WIDTH and WINDOW HEIGHT – Specifies the size of the visualisation window
- NEAR_CLIP and FAR_CLIP – Specifies the near an far clipping plane used for OpenGL rendering.
- SPHERE_SLICES – The number of slices used to create the sphere geometry representing a single agent in the visualisation.
- SPHERE_STACKS - The number of stacks used to create the sphere geometry representing a single agent in the visualisation.
- SPHERE_RADIUS – The physical size of the sphere geometry representing a single agent in the visualisation. This will need to be a sensible value which corresponds with the environment size and agent locations within your model/simulation.
- VIEW_DISTANCE – The camera viewing distance. Again this will need to be a sensible value which corresponds with the environment size and agent locations within your model/simulation.
- LIGHT_POSITION – The visualisation will contain a single light source which will be located at this position.

4.6 Creating a Custom Visualisation

Customised visualisation can easily be integrated to a FLAME GPU project by extending the automatically generated visualisation file (the output of processing `visualisation.xslt`)[1]. Many of the FLAME GPU SDK examples use customised visualisations in this way. As with the default visualisations any custom visualisation must define the following function prototypes defined in the automatically generated simulation header.

```
extern "C" void initVisualisation();

extern "C" void runVisualisation();
```

The first of these can be used to initialisate any OpenGL memory and CUDA OpengGL bindings as well as displaying the user interface. The second of these functions must take control of the simulation by repeatedly calling the draw and `singleIteration` (which advances the simulation by a single iteration step) functions in a recursive loop. A more detailed description of the default rendering technique is provided within other FLAME GPU documentation (listed in Section 1.2).

4.7 Performance Tips

The GPU offers some enormous performance advantages for agent simulation over more traditional CPU based alternatives. With this in mind it is possible to write extremely sub optimal code which

[1] When doing this within Visual Studio it is important to turn off the template processing of the `visualisation.xslt` file in each of the launch configurations as processing them will overwrite any custom code!

will reduce performance. The following is a list of performance tips for creating FLAME GPU model files;

General Usage of FLAME GPU

- FLAME GPU is optimal where there are very large numbers of relatively simple agents which can be parallelised.
- Populations of agents with very low numbers will perform poorly (in extreme cases slower than if they were simulated using the CPU). If you require an agent population with very few agents consider writing some custom CPU simulation code and transferring any important information into simulation constants to be read by larger agent populations during the FLAME GPU simulation step.
- Outputting information to disk (XML files) is painfully slow in comparison with simulation speeds so consider outputting information visually or only after larger numbers of simulation iterations.

Model Specification

- Minimise the number of variables with agents and message data where possible.
- Try to conceptualise and fully specify the model before completing the agent functions script to avoid making mistakes with agent function arguments. Try to think in terms of X-Machines agents!

Agent Function Scripting

- Small compute intensive agent functions are more efficient than functions which only iterate messages. Try to minimise the number of times message lists are iterated.
- Keep agent functions small and do not define more local variables than is strictly required. Reuse local variables where possible if they are no longer needed and before they go out of scope.

Message Iteration

- For small populations of agents (generally less than 2000 but dependant on hardware and the model) non partitioned messaging has less overhead and is similarly comparable to spatial partitioning.
- For large populations of distributed agents with limited communication spatially partitioned message communication will be much faster.