# OpenMASK:
# Multi-threaded Animation and Simulation Kernel:
# Theory and Practice

David Margery

January 30, 2002 version

**Abstract**

This document aims to give a general overview of OpenMASK for developpers using it as a target execution platform for their animations, simulations or virtual worlds. Therefore, a short presentation of the motivations for OpenMASK are given explaining the reasons for some design tradeoffs. The core of this document then presents how virtual objects communicate in the OpenMASK framework, how they are activated and the different classes involved in their management. After reading this document, a developper should be able to understand the more detailled documentation that is distributed with the code.

## 1 Introduction

The main goal of OpenMASK, previously named GASP (General Animation and Simulation Plateform), is to provide a basic animation and simulation kernel with the following functionality

1. the kernel should be animation level (descriptive, generative or behavioral) agnostic.
2. the kernel should be animation programming style (reactive, agent oriented, active object ...) agnostic
3. the kernel should be rendering library agnostic
4. the kernel should be capable of multi-threaded execution of the animation or simulation
5. the kernel should be multi-threading style (distributed computing, parallel computing, shared memory architectures) agnostic

Of course the last two points of these main objectives has greatly influenced the conception of the kernel, as we hope to be able to provide performant multi-threading. Furthermore, defining what needs to be simulated or animated does influence the programming style of the object animated or simulated. Finally, because some programming styles are easier to use for some animation levels, the presented kernel isn't quite animation level neutral.

Nevertheless, the different design tradeoffs are rather biased towards multi-threading than any predefined animation programming style or level. Furthermore, the tools provided by OpenMASK aim to enable any programmer to express his animation or simulation with the best suited tool for his work, empowering the kernel programmer to optimise using the semantic information gained by guessing that if one tool has been preferred over another, that choice was probably made with some intention.

In this paper, the terms animation and simulation are used with slightly different meanings. We use simulation when the produced result (which might not be visual) is more important than the time it take to produce, and animation for real-time animation when the time to produce results or to take into account user interaction does matter. We believe that one should be

able to use the same components for both, and that depending on execution context, different optimisations should be performed by the run-time kernel. Furthermore, as this paper is mostly focused on presenting how the kernel should be used to program new components, execution context is unknown and mostly irrelevant. Therefore, animation of simulated objects will be discussed, without presuming that the simulated objects are used in an animation or simulation context.

### 1.1 OpenMASK Overview

In Mask, the basic building block of an application, is the *simulated object*[1]. It is inside a simulated object that all code for the evolution of the object and the communication with other objects is located. Therefore, the two main questions one has to ask are:

1. when is that code executed ? This is the activation problem.
2. how do simulated objects communicate between one other ? This is the communication problem.

The third question one would want to ask is: what is the granularity of a simulated object ? Or what does a simulated object animate ? This is a question we don't want to answer, as the kernel has to be animation level and style agnostic. Experience with OpenMASK shows that simulated objects vary in granularity from complete virtual humans with complex behaviors to a simple inert sphere, and this in the same application.

## 2 Communication between objects

Simulated objects for OpenMASK communicate by many means. The fundamental point to understand, is that communication should only be done with the tools (the classes and functions) provided with the kernel. In particular, as a general rule, no member function invocation should be used. The rational for this, is that multi-threading introduces thread safety problems. The kernel can manage these thread safety problems so that programming a simulated object doesn't require comprehension of the multi-treading paradigm used. But allowing method invocation between objects requires that objects be designed with multi-threading in mind, so that data integrity can be maintained. This is the fundamental design trade-off made during the conception of OpenMASK: to allow for performant multi-threaded run-time kernels, method invocation is forbidden between simulated objects, even if it's possible from a programming point of view, and in specific cases perfectly compatible the general framework presented here. But as these special cases require a good understanding of the general framework, they won't be discussed in this section.

OpenMASK distinguishes several communication styles between simulated objects. The basic communication style is an object reading the attributes of another object in a regular fashion. Attributes are made available for reading by other objects with outputs and control parameters, and are generally read through inputs. The other base communication style is through signals and eventListeners, designed for sporadic communication between objects. From these notions are derived the event for one-to-one communication.

### 2.1 Outputs

An *output*[2] is used when an object has an attribute (it's position for example) that it makes publicly available for reading and whose value is always pertinent: whatever the simulation date, it's value is meaningful. Therefore, interpolation and extrapolation of output values is legitimate. Nevertheless, the type or the semantic of data stored in an output may not be

[1] named `PsSimulatedObject`

[2] named `PsOutput`

appropriate for default polation[3] methods. Therefore, to each output is associated a polator, and output creation is done using the `addOuput` member function, which takes 3 parameters:

1. a name for the created output
2. a type, that inherits from PsType
3. (optional) a polator. If no polator is given, the default polator for the type is used.

The most simple polator is called a naive polator[4], and is pertinent for every type, as it only returns values stored in the history FIFO (of a given length) associated to the output. Other polators use those stored values to calculate any required value.

The list of all the outputs of a simulated object are stored in an output table[5], which can be queried by any object to find the attribute of an object.

The simulated object owner of an output can change its value using the `set` member function, and query the current value of the output using `getLastExactValue`.

### 2.1.1 Advanced topics for outputs

When `set` is called, the parameter of that member function is copied into the history FIFO (which length is calculated by the kernel or read from an environnement variable) maintained by the kernel for each output. When the type of the output is large, or the copy operation costly, one can use a more advanced mechanism that enables the simulated object to use the memory block that would be written by the next set for it's computations. This is done using the `getNextPlaceHolder` member function to get a reference to the seeked memory block, and `setInPlace` to unprotect that memory block so it can be read by the connected inputs. The cost of such a mechanism is paid with distributed OpenMASK kernels: because output values are distributed, any memory allocated during the copy of a value in the output needs to be distributed with the output. Therefore, any dynamic memory allocation done between `getNextPlaceHolder` and `setInPlace` is more costly than basic memory allocation. If the memory allocated doesn't relate to the output value, distribution costs might grow unnecessarily.

From a C++ point of view, nothing prevents one simulated object to call set on the outputs of a second object, because of the way data protection (private or protected) works. Nevertheless, the code has been designed assuming only the reference version (more about that later) of a simulated object will call `set` or `setInPlace`. Data integrity is not guaranteed should someone choose to ignore that principle. Moreover, in debug mode, this will make the application fail. If your design requires that more than one object is able to use `set`, please look at the the section on *control parameters*.

## 2.2 Control Parameters

A *control parameter*[6] is a special type of output for 2 reasons. The first, is that any simulated object can suugest a new value[7] for a control parameter, and the second is that a control parameter isn't stored in the list of outputs of its simulated object, but in the list of control parameters[8]. Nevertheless, it's possible to connect an input to a control parameter.

When `set` is called on a control parameter, a valued event is sent to the owner of the control parameter with the proposed new value. By default, this event is interpreted by an event listener that will change the value of the control parameter according to the last event proposing a new value received by the owner. The only way of changing that behavior, is by redefining the event

[3] polation is defined as one of the following : interpolation, extrapolation or antepolation. It produces a value using the values previously set by the simulated object

[4] named `PsPolator`

[5] named `_outputTable`

[6] named `PsControlParameter`.

[7] by calling `set`

[8] `_controlParameterTable`

listener associated to the control parameter. More about this in the section dedicated to events and event listeners.

## 2.3 Inputs

An *input*[9] is used by simulated objects to read the outputs (in the more general sense : outputs and control parameters) of other simulated objects. There are two sorts of inputs:

1. private inputs: the connection of these inputs to outputs can only be done at the request of the simulated object owner of the input.
2. public inputs: they have the properties of private inputs but also accept connections to other outputs can be done at the request of the owner of the output. When a connection is requested, a connection event is sent, and automatically handled by the default event listener associated to the input.

Public and private inputs are different only by the value of a boolean authorising connection requests originating from other simulated objects.

### 2.3.1 Reading from inputs

The default method for reading the value of an input is by using the `get()` member function of the input. This member function accepts one optional parameter that is interpreted as the delay between the current simulated date and the date of the value that is returned by `get()`. For example, `get(0)` will return a value calculated from the connected output for the current simulated date. `get(20)` will return a value calculated as being the one of the output at a simulated date 20 ms seconds before the current simulated date. Those calculations are done using the polator associated to the output. If a negative value is passsed as a parameter, a estimate for a future value of the connected output will be returned.

In the event that a produced value is required, `getLastExactValue()` should be used.

In the event that the input is used to detect changes in the value of the connected output, a derived class should be used: the *sensitive input*[10]. Inputs of that class can be queried using the `valueChanged()` member function. If a *sensitive notifying input*[11] is used, an event will be sent to the owner of the input each time a new value is given to the connected output.

## 2.4 Signals

A *signal*[12] is a piece of information released to the rest of the simulation by a simulated object. The basic signal has no value attached to it, except an identifier used to distinguish between different signals. Therefore, valued signals are introduced so that signals can be made to carry data.

For an object to be aware of signals released in the environment by other objects, it must register. If one object is only interested in signals originating from a particular object, registration should be done through that object. Otherwise, it is done through the *controller* (see 18). When registering for a signal, an object can decide what information it will receive when the signal is released. That information is either an event prototype or the identifier of the signal if no event prototype is specified.

Indeed, when a signal is released in the environment, all registered objects receive the corresponding event, depending on the optional event prototype registered for that signal.

[9] named `PsInput`.
[10] named `PsSensitiveInput`.
[11] named `PsSensitiveNotifyingInput`.
[12] named `class PsSignal`.

### 2.4.1 System Signals

During the simulation, the controller fires system signal to notify interested objects of particular events in the simulation. These events are the creation or the destruction of an object, the recomputing of the scheduling data structures, etc.

## 2.5 Events and valued events

An *event*[13] is composed of four fields. Its sender, its receiver, the date of sending and the event identifier, which should be viewed as a string identifying the event. A valued event is an event to wich a valu field is associated. That value can be of any OpenMASK conforming type[14]. Therefore, when receiving an event wich is known to be valued, a cast in the exact type of the valued event should be used, because of the use of a generic (template) class to implement valued events. For implementation reasons, one should use the cast function provided (`event_cast`) rather than dynamic_cast.

## 2.6 Event processing and event listeners

Event processing is done at the object handle level. All events received by a simulated objects are ordered as they arrive at the object based on

1. their associated date
2. their arrival order

Before any event processing takes place, the `prepareEventProcessing` member function of the simulated object is called. Then each event is processed, first by looking for an event listener registered for the corresponding event identifier.

1. If one is found, it receives the event for processing. Depending on the result of that processing, the event is then propagated along the event listeners list.
2. If such an event listener doesn't exist, or if its treatment of the event is incomplete[15] returns false), the `processEvent` member function of the simulated object is called. After all events were processed, `eventsProcessed` is called.

### 2.6.1 Event listeners

*Event listeners*[16] are used to associate the reception of an event to the corresponding calculations. When two event listeners are registered for the same event, the last one registered is called first, to enable redefinition of the reaction to an event by a derived simulated object.

# 3 Activation Policy

Each simulated object has two activation methods. The first, the `compute` member function, is called at a certain frequency for objects with a frequency in the active state (see the state diagram). The second is related to event processing. For suspended objects, event processing takes place at the rhythm events are received, but not faster than the controller's frequency. For active objects, event processing takes place at the frequency of the simulated object if the `processEventsASAP` returns false (default policy), as soon as possible otherwise.

[13] named `PsEvent`.
[14] ie inherits from `PsType`
[15] in which case `processEvent`
[16] named `PsEventListener`.

### 3.1 Simulated Object State

As implied by the previous paragraph, a simulated object can be in different computing states, as described by the following diagram (fig 1).

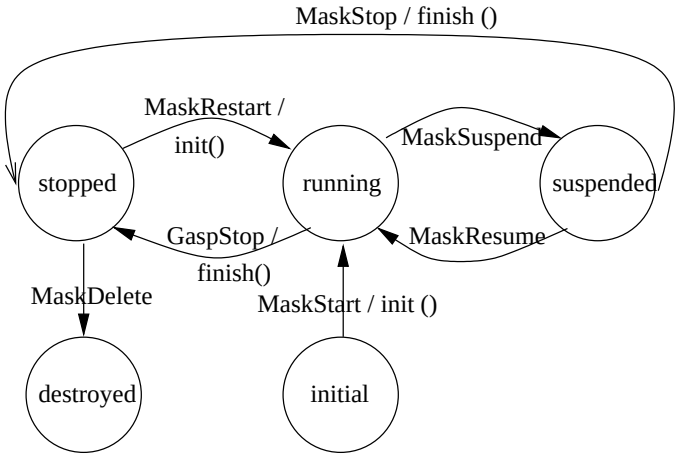


Figure 1: State diagram for a simulated object

### 3.2 Scheduling Policy

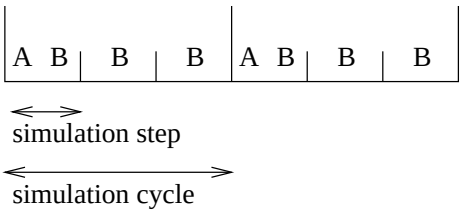


Figure 2: Activation cycles with OpenMASK

The base scheduling block for OpenMASK is the simulation step (see fig 2, where A' frequency is 10hz and B's frequency is 30Hz). All scheduling takes place within the borders of a simulation step. For the scheduling perspective, OpenMASK is a synchronous system. One object cannot be scheduled twice while other objects are in the course of one single computing. This last point is often misunderstood, as it's possible for different objects to have different frequencies.

What the controller does, is that it computes the greatest common divisor (gcd) and the lowest common multiplier (lcm) of all the non zero frequencies. The gcd then becomes the cycle frequency of the scheduler and the lcm the step frequency. Therefore, for the scheduler, there are a certain number of steps in an activation cycle. For an object having a frequency equal to the cycle frequency, it will be scheduled on the first[17] simulation step of each cycle(A's case in fig 2). Nevertheless, the next simulation step will only start when all objects scheduled in the current simulation step are finished.

[17]This is an implementors choice: further versions would need to be able to specify that parameter for each object

This policy implies that the time it takes to compute each simulation step varies from one step to another. Therefore, even if simulated time (time as seen by simulated objects) flows regularly, from an observer's perspective, it will probably seem jerky. One should therefore be very cautious when planning multifrequence applications, for the results could be different of those expected.

Technically, it wouldn't be too difficult to overcome that problem, but further research is needed before it is implemented, because the synchronous activation mode is deeply rooted in quite a few concepts underlying the OpenMASK framework.

# 4 Object Management

## 4.1 Base classes for Object Management

The management of simulated objects is done throught a few classes. Each object is known to other objects throught it's *object description*[18], which is static and stored in the simulation tree. An object description is composed of its name, its class, its frequency, its process and its configuration parameters.

The *controller*[19] is responsible for object activation and other functions that cannot be performed at the simulated object level. Therefore the controller performs object creation and destruction, signal handling, and maintains the simulation date.

The controller handles object through an *object handle*[20], which is the data structure the controller uses to store any run-time information it needs for object activation and communications. In distributed version, a *reference object handle* is used for the reference version (the version activated) of the simulated object, and a *miror object handle* is used by controllers handling copies of the simulated objects if they are needed for inter object communication.

## 4.2 The Simulation Tree

For a given application, simulation object are organised in a *simulation tree* [21]. The root object of that tree is the controller. Semantic of the simulation tree is left to the application designer.

The two main OpenMASK functionalities are affected by the way object are structured in the simulation tree.

1. object creation, as interpretation of the string representing the created object's class is done, at first at the simulated object's father (in the simulation tree) level, and propagated down to the controller if necessary.
2. search functions. When looking for objects of a precise type, the search can be global, of limited to the descendants, the sons, or the brothers of an object in the simulation tree.

It is important to understand that the simulation tree is different from the geometric graph and from the inheritance tree.

## 4.3 Object Creation

### 4.3.1 When ?

Simulated objects in OpenMASK can be created either

1. Statically: their description is specified in the simulation tree given to the controller at construction.

[18] named `PsObjectDescription`.
[19] named `PsController`.
[20] named `PsnObjectHandle`.
[21] named `PsSimulationTree`.

2. Dynamically, at the programmer's initiative[22].
3. Dynamically, as needed by other objects. This is true of object created during distributed execution.

In all those cases, the object is sent a MaskStart system event which triggers, at the next simulation step, the successive call of init and compute. For static objects, MaskStart is sent at a date preceding the initial simulation date. For the other methods, MaskStart is sent at the date of the request triggering the dynamic creation.

### 4.3.2 How ?

In C++, there is no way of creating an object using the string describing its class. Therefore, a specific mechanism is used. All objects, including the controller, can create objects of a certain class, thus creating what we call sub-objects. This is done using an *object creator*[23] which creates new objects when one of its specific member functions[24] is called. A generic specialisation of the object creator[25] creates an object of the type of its template parameter. Therefore, by associating a string and an object creator in a map, specific to each simulated object, one is able to create objects using a string describing its class. This creation is performed by the father (in the simulation tree) of the object which will be created, or by the nearest ancestor that is able to create the correct object.

### 4.3.3 Integration in the activation mechanism

For simulated objects with a simulation frequency, two options are available at creation.

1. The simulation adapts all its data structures so as to respect the given frequency (default behavior). Beware: the fact that the controller's frequency can change may affect a simulated object's behavior. Therefore, the controller fires a signal whenever this happens: `MaskControllerFrequencyChange`.
2. The frequency of the object descriptor is adapted before effective creation. This can be done using a controller's member function[26], which will compute the nearest available frequency to the frequency specified.

[22] using the controller's `createObject` member function
[23] class `PsSimulatedObjectCreator`
[24] named `createSimulatedObject`
[25] named `PsSimpleSimulatedObjectCreator`
[26] named `computeAdequateFrequency`