Design document for a node system for game logic

Sjoerd de Vries, December 2009
sjdv1982@gmail.com

Terminology:

Nodes are classes of linkable components. Nodes must be declared within the node system. A node graph consists of node instances, linked by channels. A channel is between the output pin of one node instance and the input pin of another node instance. “Node instances” are often called “nodes” for short (also in this document sometimes), but this is a bit sloppy, because “node” should actually refer to the class to which a node instance belongs.

Goals:

- The node system is not a toy system, but a full-blown alternative/complement to Python for the creation of game logic.
- The node system must be flexible, extensible and multi-layered: there must be an easy-to-use, high level, GUI-controlled layer, while various deeper layers allow more control to more advanced users with various degrees of programming knowledge. More layers will make the initial implementation of the node system more difficult, but this must be accepted in order to get maximum flexibility, extensibility and ease of use.
- The design of these layers and their interaction with the user must be done once and for all. APIs and the node graph description format must not be changed after release, because it would break the node graphs that users have created since then. The underlying implementation of the node system can of course be optimized at any time.
- The coupling to Blender and the BGE should be as loose as possible. Ideally, the system should be extensible for use as game logic engine outside Blender (for example to run on a server that communicates with BGE clients), and also for other flow-related tasks inside or outside Blender.
- Backwards compatibility with current BGE logic brick system is not a primary goal. If the node system would be extensible to provide such backwards compatibility, this would be a bonus.

Design choices, general:

- The node system uses Python 3.1.
- The node system is decentralized, the nodes themselves will do all the work. There will be no “runtime” component of the node system: Scheduling, bookkeeping etc. will be all be done by nodes.
- The node system consists of the following three parts: node declaration, node graph definition, linkage. Node declaration is done in the form of Python classes. Node graph definition is done by the user using a GUI, or possibly using a Python script.
- Every node (every node, not every node instance) must have a unique name.
- During node declaration, GUI-related parameters are determined, such as the number and type of input and output pins, configurable initial values, and perhaps a 2D shape. These parameters are grouped and registered as a node with the GUI. Therefore, a node at the GUI level is nothing but I/O, shape, and configuration, while a node at the Python level contains the full functionality.
- “Linkage” does **not** mean “definition of the connectivity”, that would be part of the node graph definition by the GUI. “Linkage” means that methods from one node are directly exported to

another node. For example, take the following code:

```
“””
Example 1
Linkage in Python of two nodes
Note that the node1 and node2 classes are dummy implementations,
actual node declaration will be different
“””

class node1:
  def __init__(self, value):
    self.value = value
  def set_value(self, value):
    self.value = value
  def get_value(self):
    return self.value

class node2:
  def __init__(self):
    self._inputfuncs = []
  def add_inputfunc(self, inputfunc):
    self._inputfuncs.append(inputfunc)
  def get_value(self):
    value = 0
    for f in self._inputfuncs:
      value += f()
    return value
```

In this case, “linkage” of node1 and node2 means the following:

```
n1 = node1(10)
n1a = node1(8)
n2 = node2()
n2.add_inputfunc(n1.get_value)
n2.add_inputfunc(n1a.get_value)
print(n2.get_value())
```

(By the way, this is an example of pull mode linkage, scroll down a few pages for more details)

The lack of a runtime component doesn't mean that some runtime monitoring cannot be implemented. For example, an alternative linkage implementation could be:

```
count = 0
def monitor(func):
  global count
  count += 1
  if count == 99999:
    raise Exception(“Node system runs too long”)
```

```
  ret = func()
  if not isinstance(ret, float):
    raise Exception("%s has returned the wrong type" % func)
  return ret

n1 = node1(10)
n1a = node1(8)
n2 = node2()
n2.add_inputfunc(functools.partial(monitor, n1.get_value))
n2.add_inputfunc(functools.partial(monitor, n1a.get_value))
print(n2.get_value())
```

Design choices, medium level:

There are two kind of nodes: **environment nodes** and **worker nodes**. The environment nodes implement all functions that are heavy on programming: event handling, the scheduling of cycles ("the pacemaker"), spawning, object registration, collision detection, etc. Environment nodes work behind the scenes and they are linked automatically, not in a GUI. In contrast, worker nodes are meant for maintaining game state and signal integration, using boolean and arithmetic operators, switches, dispatchers, triggers, etc.
Worker nodes are assembled in Python out of a set of **subnodes** (helpers and decorators), used to define input channels, output channels, etc. Additionally, a worker node may implement methods for communication with environment nodes (providing a worker node with sensor/actuator functionality). In the node graph, worker nodes are linked to each other with a standardized channels and these links are specified by the user in a GUI.

A **node graph** is always running inside a **box**. A box provides a set of environment nodes which are by default hidden from the user, but that can be edited by the advanced user. The outer box contains the environment nodes that interface with the BGE, such as keyboard/mouse detection and a pacemaker that is controlled by the BGE tick rate. However, it is possible to define other boxes where these things are redefined, for example to implement turn-based game logic (a pacemaker under control of the outer box), to implement loops (the pacemaker under control of an iterator node), to implement graph spawning (a box running multiple instances of the embedded graph, which can transmit signals to those graphs if the signals are labeled with an instance identifier), asynchronous boxes for network communication, or just boxes to make the game logic more modular.

Design choices, specific

Linkage

I have developed a low-level linkage library in Python called libcontext. This library implements so-called contexts, classes which contain plugins, sockets, and subcontexts. Sockets are declared as context.socket(identifier, socketclass(A)). A must be a callable object. Plugins are declared as context.plugin(identifier, pluginclass(B)). When a context is closed, the following actions are taken automatically: first all subcontexts are closed. Then,
for all identifiers i:
  for all sockets A registered under i:
    for all plugins B registered under i:

A is called with B as argument.
Socketclass and pluginclass can be just wrappers around A and B, but mixins can be added specifying that the socket/plugin must be filled at least once, exactly once, at most once, that there may be no socket/plugin with exactly the same name, etc. Socket plugin/pairs can be also used as flags, meaning that the socket just requires that a certain plugin is defined without doing anything with the actual value of the plugin.

A box will be implemented as a subclass of libcontext.context, providing and/or requiring certain environment nodes through plugins/sockets. The whole node graph is contained within one outer box, which interfaces to the BGE. Environment nodes are instantiated and placed directly in the box and linked through each other automatically using their sockets and plugins. In contrast, every worker node inside the box has a subcontext on its own. The context library supports an import command, where (references to) plugins and sockets are imported from one context into another. All environment node sockets and plugins are imported automatically into the worker node contexts. Links between worker nodes are implemented as an explicit import command. For example:

```
“””
Example 2
Linkage of two worker nodes
Note that the workernode1 and workernode2 classes are dummy
implementations, actual worker node declaration will be different
“””
class workernode1:
  def __init__(self):
    self._outputtargets = []
  def add_outputtarget(self, outputtarget):
    self._outputtargets.append(outputtarget)
  def output(self):
    for f in self._outputtargets: f(self.outputvalue)
  ...
  def place(self):
    signature = (“output1”, (“push”, “float”, “red”))
    socketclass = libcontext.socketclasses.socket_multi_required
    mysocket = socketclass(self.add_outputreceiver)
    libcontext.socket(signature, mysocket)

class workernode2:
  def get_input(self, value):
    ...
  def place(self):
    signature = (“input1”, (“push”, “float”))
    pluginclass = libcontext.pluginclasses.plugin_single_required
    myplugin = pluginclass(self.get_input)
    libcontext.plugin(signature, myplugin)

subcontext1.place(workernode1())
subcontext2.place(workernode2())
```

```
signature1 = (“output1”, (“push”, “float”, “red”))
signature2 = (“input1”, (“push”, “float”))

#import, from subcontext1 into subcontext2, socket <signature1> as
socket <signature2>

subcontext2.import_socket(subcontext1, signature1, signature2)
```

Node declaration

Node declaration is done using Python classes. Python files containing node declarations should be placed in a special directory where they can be discovered by Blender/the GUI, similar to what is currently done for Blender UI scripts.

Node declaration of environment nodes

Environment nodes must have nodesystem.envnode as a metaclass. This metaclass is extremely simple:

- It checks that the class definition has a __nodename__ attribute (perhaps __author__, __copyright__, etc. as well)
- It checks that the class definition has a place() method
- It creates the class without any modification (type(name, bases, dict))
- It registers the class with Blender/the GUI under the value of __nodename__
- It returns the class

Every box contains a sheet of environment nodes together with a an argument list to instantiate that node. When the node graph is assembled, the place() method of every environment node is called.

Example for a (hypothetical) environment node to interface with a physics library:

```
#Example 3

import myphysicslibrary
class physics(metaclass=nodesystem.envnode):
  __nodename__ = “physics_myphysicslibrary”
  def __init__(self):
    self._identifiers = {}
    self._collisioncallbacks = {}
  def add_object(self, identifier, obj):
    assert identifier not in self.identifiers
    phobj = convert_object_to_myphysics_format(obj)
    handle = myphysicslibrary.add_object(phobj)
    self._identifiers[identifier] = handle
    self._collisioncallbacks[identifier] = []
  def _collisionhandler(self, identifier, otherobjecthandle):
    otherobjectidentifier = self._identifiers[otherobjecthandle]
    for c in self._collisioncallbacks[identifier]:
      c(otherobjectidentifier)
  def add_collisioncallback(self, identifier, callback):
```

```
    assert identifier in self._identifiers
    if not self._collisioncallbacks[identifier]:
      handle = self._identifiers[identifier]
      collisionhandler = functools.partial(self._collisionhandler,
identifier)
      myphysicslibrary.set_collisionhandler(handle, collisionhandler)
    self._collisioncallbacks[identifier].append(callback)
  def set_force(self, identifier, force):
    ...
  def set_position(self, identifier, position):
    ...
  def place(self):
    pluginclass = libcontext.pluginclasses.plugin_supplier
    libcontext.plugin("physics", pluginclass(self))
```

Node declaration of worker nodes

Worker nodes have nodesystem.node as the metaclass. This metaclass is considerably more complicated than nodesystem.envnode.

- It checks that the class definition has a __nodename__ attribute (perhaps __author__, __copyright__, etc. as well)
- It checks that at least one input pin or output pin has been declared
- It checks that no __init__ function has been defined.
- It creates an empty *runtime class dictionary* (rundict)
- Inside the rundict, it defines an _place() method. This method contains plugin and socket declarations that depend on the declared subnodes, plus a call to self.place(). If no place() method has been defined, it defines an empty one.
- Inside the rundict, it defines an __init__method that depends on the declared subnodes.
- All methods that are not subnodes are copied into the rundict.
- Out of the rundict, a runclass is built using the type() function.
- A runbox class is built, as a subclass of libcontext.context. The runbox will be a subcontext of the box in which the node graph is running, so all environment nodes will be automatically available. The runbox__init__ function is redefined to accept the same parameters as the runclass, these parameters are buffered. A runbox place() method is defined, which instantiates a runclass using the buffered parameters and then calls the _place() function of the instantiated runclass.
- A GUI node class is built and registered with the GUI. The contents of the GUI node class are dependent on the input pins and output pins and other subnodes of the class definition. The exact implementation will depend on the interface with Blender, but an example could be:

```
GUInode = Blender.nodesystem.GUI.node()
GUInode.add_inputpin("input1",  ("push", "float"), "NW")
GUInode.add_inputpin("input2",  ("push", "float"), "SW")
GUInode.add_outputpin("output",  ("push", "float", "red"), "E")
GUInode.set_shape("myshape.png")
GUInode.close()
```

- The runbox class is registered with the GUI under __nodename__. The GUI can then instantiate runbox instances and connect them to each other. When the node graph is assembled, the place() method of every runbox instance is called. The exact implementation will depend on the

interface with Blender, but an example could be:

```
runbox = type(...)
GUInode = Blender.nodesystem.GUI.node()
... #see above
Blender.nodesystem.GUI.new_worker_node(__nodename__, GUInode, runbox)
```

- The runbox class is returned.

Channels

Channels define the communication between worker nodes. Channels are opened by input pins and output pins. Every channel has a signature consisting of three parts: mode, data type and color.

Mode

The mode of a channel can be push or pull. Push channels transmit the data to the receiver on the initiative of the sender, while pull channels ask the sender for data on the initiative of the receiver. In Example 1, nodes are linked in pull mode, while Example 2 uses push mode linkage.

Data types

The data type is a string label (or a tuple of string labels) describing the format of the data associated with a channel. Note that it is just a descriptor: the nodes themselves are responsible for returning that conform to the described data format.
The exception is the startvalue_GUI node (see below): the data type of the buffer or recorder must then be known to the GUI (Blender.RNA).

The following data types are suggested (more can always be added):
int, string, float, bool, count (unsigned int)
matrix, orientation, vector, unitvector: 4x4 matrix, 3x3 matrix, xyz vector, xyz unit vector
angle (degrees or radians)?, quaternion?
object: a Blender object name
label: a string, or a tuple of 1 or more strings, used to label signals (used to send signals to dispatcher nodes or spawning boxes)

All of these types can also be combined into tuples.
Array types will probably not be necessary, they can be implemented better by iterator nodes.

In addition, the following types can be used with push data only:
trigger: a call without data
pulse: a call without data, with the first call (odd calls) representing an activation signal, and the second call (even calls) representing a deactivation signal.

Color

The color of a channel or a subnode indicates how often they fire or update. For push channels, the color is determined by the sender. For pull channels, the color is determined by the receiver. Colors are meant to make it more tractable to the user how the node graph works, it is up to the implementation of the subnode.

Red channels and subnodes can fire or update **at any time**. Some of them update as soon as they are being pushed or pulled, others are under control of an external trigger channel.
Blue channels and subnodes fire or update **exactly once per cycle**. They compute their value when they update and they keep this value until the next cycle.
Green channels and subnodes fire or update **at most once per cycle**. Like red channels/subnodes, they update as soon as they are being pushed, pulled or triggered. Like blue channels/subnodes, they keep this value until the next cycle.

On the low level, input pins in pull mode are implemented as sockets, and output pins in pull mode are implemented as plugins. At link time, the input pins receive a reference to the output pin, which they can call at runtime, without arguments, to receive the transmitted value.
Input pins in push mode are implemented as plugins, and output pins in push mode are sockets. At link time, the output pins receive a reference to the input pin, which they can call at runtime with the transmitted value as argument.

Subnodes

Subnodes can be divided into helpers, named helpers and decorators. The syntaxes are as follows:

```
class node(metaclass=nodesystem.node):

  helper(<args>)
  or:
  name = helper(<args>)

  name = named_helper(<args>)

  @decorator
  def func(<args>):
    ...
```

All subnodes are defined under nodesystem.subnodes, it is therefore recommended to import * from this module before starting a node declaration. Within a node declaration, every subnode must be linked to another subnode at least once.

The following subnodes will be implemented. Additional subnodes can be thought of later, especially subnodes for GUI style directives.

inputpin
Named helper. Takes as argument a channel signature: (mode, type, color). For push input pins, the color is meaningless and can (should) be omitted.

outputpin
Named helper. Takes as argument a channel signature: (mode, type, color). For pull output pins, the color is meaningless and can (should) be omitted.

weaver
Helper. Only one weaver can be unnamed. Takes as argument a channel signature: (mode, type, color),

and a list of inputs, which are instances or names of input pins, properties and/or other weavers. Multiple subnodes can pull from a pull weaver. When triggered, it will weave all inputs into a single value corresponding to the channel signature. Design question: to what extent will implicit type conversions be supported?.
Red push weavers will be triggered by any push input received, as well as by any associated trigger subnodes. If the weaver is red push and none of the inputs is push, there must be at least one associated trigger or triggerfunc subnode.

unweaver
Helper. Only one unweaver can be unnamed. Takes as argument a channel signature: (mode, type, color), and a list of outputs, which are instances or names of output pins, properties and/or other unweavers. When triggered, it will unweave its input value and spread it over its outputs. Multiple subnodes can push into a push unweaver. Red unweavers will be triggered by a push input value, a pull output or an associated trigger. If the unweaver is red push, it needs to have a value supplied before it can be triggered by pull output or triggers, else a runtime error results; so it is recommended to associate a startvalue subnode with red push unweavers. If the unweaver is red pull and none of the outputs is pull, there must be at least one associated trigger or triggerfunc subnode.

operator
Helper. Takes as argument a callable Python object, an input and an output. Input must be the name or instance of an input pin, a property or a weaver. Input may be omitted, in which case the unnamed weaver is used (which must then be defined). Output must be the name or instance of an output pin, a property or an unweaver. Output may be omitted, in which case the unnamed unweaver is used (which must then be defined). The input and output subnode must have the same mode (push or pull). If they are push, the operator will be triggered by the weaver (or an associated trigger or triggerfunc subnode). If they are pull, by the unweaver.

modifier
Decorator. Takes as argument a color (red or blue).
The decorated member function must not take any other arguments than self. The member function is called when scheduled.
Red modifiers must have a trigger or triggerfunc subnode associated with them.

property
Named helper. Takes as argument a color (red or blue) and an input. An input is the name or instance of an input pin or a weaver. When triggered, it will assign the input value to self.propname, where propname is the name of the property. For example:

```
inp = inputpin(“push”, “int”)
inpvalue = property(“red”, inp)
```

This will update self.inpvalue whenever inp is activated.

Instead of a subnode input argument, a data type argument can also be supplied. In that case, the property must be set through modifiers and other member functions.

A red property will trigger upon push input or by an associated trigger. If the property is red and the input is pull, there must be at least one trigger or triggerfunc subnode associated with the property.

init

Decorator. The decorated function will be called at the start of node graph execution..

triggerfunc
Named helper. Takes as a subnode argument that is the name or instance of an input pin, property, weaver, modifier, operator, unweaver, or output pin. The subnode must be red. Returns a function that when called, triggers that subnode.
For example:

```
inp = inputpin("pull", "int", "red")
inpvalue = property("red", inp)
inptrigger = triggerfunc(inpvalue)
```

Calling self.inptrigger() from a modifier function will then trigger the inpvalue property, causing inp to be pulled and its result written to self.inpvalue.

trigger
Helper. Takes as two subnode arguments, each argument being the name or instance of an input pin, property, weaver, modifier, operator, unweaver, or output pin. The second subnode must be red. Whenever the first subnode is activated, the second subnode is activated afterwards.

bind
Helper. Takes as two subnode arguments, the first argument being an input, the name or instance of a property, weaver or input node, the second of an output node. Both subnodes must either be push or pull. The subnodes are connected by a channel in the same way as an output pin to an input pin, but in reverse order (the data goes from input to output).

startvalue
Helper. Takes a subnode argument and a value argument. The subnode argument must be the name or instance of a property or unweaver that can accept <value>. This value is assigned to the subnode at the start of node graph execution.

startvalue_GUI
Helper. Takes a subnode argument and a defaultvalue argument. The subnode argument must be the name or instance of a property or unweaver that can accept <defaultvalue>. The datatype of the subnode is determined, and a value of this type will be an editable property in the GUI (under the name of the startvalue_GUI subnode, if such a name is provided; else, under the name of the property or unweaver, if provided). <defaultvalue> is the initial value of editable property. The value of this editable property is assigned to the subnode at the start of node graph execution.

The place() function
Worker nodes may also have a place() function, defining sockets and plugins to communicate with environmental nodes. This will basically turn the worker node into a sensor or actuator.

The scheduling of subnodes
Scheduling is implemented by a number of environment nodes that are part of the default box
When a cycle ends, the following events happen in order:

- The uptodate flag of all green and blue subnodes is set to false
- All blue input pins are evaluated in undefined order; other blue and green subnodes are evaluated if they are triggered and their uptodate flag is false; evaluation of a blue/green

subnode sets its uptodate flag to true. If a blue/green node is triggered when its uptodate flag is true, it returns its pre-computed value.

- The other blue subnodes are evaluated as above in the following order: properties, weavers, modifiers, unweavers, output pins.

Examples of a subnode declaration

A red node for push-adding two floats

```
"""
Example 4
"""
from nodesystem.subnodes import *
"""
imports nodesystem.subnodes.inputpin,
  nodesystem.subnodes.outputpin, ...
"""

class mynode(metaclass=nodesystem.node):
  inp1 = inputpin("push", "float")
  inp2 = inputpin("push", "float")
  outp = outputpin("push", "float", "red")
  weaver(("push", ("float", "float"), "red"), (inp1, inp2))
  unweaver(("push", "float"), outp)
  operator(float.__add__)
```

A red node for pull-adding two floats

```
"""
Example 5
"""
from nodesystem.subnodes import *
"""
imports nodesystem.subnodes.inputpin,
  nodesystem.subnodes.outputpin, ...
"""

def pull_add(func1, func2):
  return func1() + func2()

class mynode(metaclass=nodesystem.node):
  inp1 = inputpin("pull", "float", "red")
  inp2 = inputpin("pull", "float", "red")
  outp = outputpin("pull", "float")
  weaver(("pull", ("float", "float")), (inp1, inp2))
  unweaver(("push", "float", "red"), outp)
  operator(pull_add)
```

A red node for push-adding two floats, using a modifier
(A modifier is not the most efficient way here, but it gives an example of buffering using properties, which may be very useful in other cases)

```
"""
Example 6
"""
from nodesystem.subnodes import *
"""
imports nodesystem.subnodes.inputpin,
  nodesystem.subnodes.outputpin, ...
"""

class mynode(metaclass=nodesystem.node):
  inp1 = inputpin("push", "float")
  inp2 = inputpin("push", "float")
  outp = outputpin("push", "float", "red")
  prop_inp1 = property("red", inp1)
  prop_inp2 = property("red", inp2)
  prop_outp = property("red", "float")
  bind(prop_outp, outp)
  trigger_outp = triggerfunc(prop_outp)
  @modifier("red")
  def add_input(self):
    self.prop_outp = self.prop_inp1 + self.prop_inp2
    self.trigger_outp()
  trigger(inp1, add_input)
  trigger(inp2, add_input)
```

A collision detection sensor node
It depends on the environment node of example 3 and on a scenemanager environment node that can retrieve Blender objects by identifier (simple to implement).
The name of the Blender object to watch for collisions is static, and set in the GUI when the node graph is defined.

```
"""
Example 7
"""
from nodesystem.subnodes import *
"""
imports nodesystem.subnodes.inputpin,
  nodesystem.subnodes.outputpin, ...
"""

class mynode(metaclass=nodesystem.node):
  obj = property("red", "object")
  collisionobj = property("red", "object")
  send_output = triggerfunc(collisionobj)
```

```
  outp = outputpin("push", "object", "red")
  bind(collisionobj, outp)
  startvalue_GUI(obj, "Cube")
  @init
  def register_object(self):
    object = self.scenemanager.get_object(self.obj)
    self.physics.add_object(object, self.obj)
    self.physics.add_collisioncallback(self.obj, self.collision)
  def collision(self, collisionobj):
    self.collisionobj = collisionobj
    self.send_output()
  def set_physics(self, physics):
    self.physics = physics
  def set_scenemanager(self, scenemanager):
    self.scenemanager = scenemanager
  def place(self):
    socketclass = libcontext.socketclasses.socket_single_required
    libcontext.socket("physics", socketclass(self.set_physics))
    libcontext.socket("scenemanager",
socketclass(self.set_scenemanager))
```

Advanced topics

These are meant to be implemented after the initial implementation is working.

I. Game state box

Many online games use client prediction: the game state is updated by the client according to a prediction, and later, the game state is corrected by authority of the game state data sent by the server. This means that it must be possible to save, rewind and restore game state.

It is easy to develop a special "game state box", based on the standard box, but with two additional environment nodes: get_state and set_state. Every worker node placed into this box **must** connect to these environment nodes (easy to implement with the appropriate socket subclass). This would make client prediction easy and would also help a game developer in the implementation of save games.

II. Fast linkage of (sub)nodes coded in C++

Since the whole node system is coded in Python, the linkage will also be done in Python. However, there is no reason why the nodes themselves need to be fully coded in Python. It is possible to define nodes in C++ and link them using Python, **without any Python overhead at runtime**. This means that except for a lack of inlining, C++ nodes linked to each other will be as fast as if they were linked by the C++ compiler. Therefore, a node graph consisting of C++ nodes will run at full C++ speed, about two orders of magnitude faster than Python. Here a proof of principle is provided, using Example 1.

The nodes in Example 1 can be re-coded in C++ as follows:

```
typedef boost::function<float (void)> floatfunc;
```

```
typedef boost::function<void (floatfunc *)> floatregisterfunc;

class node1 {
  public:
  float value;
  node1(float val): value(val) {}
  void set_value(float val) {
    value = val;
  }
  float get_value() {
    return value;
  }
};

class node2 {
  private:
  std::vector<floatfunc *> _inputfuncs;
  public:
  node2() {}
  void add_inputfunc(floatfunc *inputfunc){
    _inputfuncs.push_back(inputfunc);
  }
  float get_value() {
    float value = 0;
    std::vector<floatfunc *>::iterator it;
    for (it = _inputfuncs.begin(); it != _inputfuncs.end(); it++) {
      value += (**it)();
    }
    return value;
  }
};
```

The Boost library is used here so that member functions can be bound and exposed to other nodes and to Python. This will be explained step by step below.

Let's start with what we would do if we were linking inside C++:

```
int main() {
  node1 n1(10);
  node1 n1a(8);
  node2 n2;
  floatfunc f1 = boost::bind(&node1::get_value, &n1);
  n2.add_inputfunc(&f1);
  floatfunc f1a = boost::bind(&node1::get_value, &n1a);
  n2.add_inputfunc(&f1a);
  std::printf("%.3f\n", n1.get_value());
  std::printf("%.3f\n", n1a.get_value());
  std::printf("%.3f\n", n2.get_value());
```

```
}
```

The main problem is that we need the class definitions of node1 and node2, which would be very hard to expose to Python natively, without wrapping in Pyobjects. The following helper functions get rid of the class definition requirement through the use of instancing and binding:

```
extern "C" node1 *node1_get_instance(float value) {
  node1 *ret = new node1(value);
  return ret;
}

extern "C" node2 *node2_get_instance() {
  node2 *ret = new node2();
  return ret;
}

extern "C" floatfunc *node1_get_value (node1 *self) {
  floatfunc *ret = new floatfunc;
  *ret = boost::bind(&node1::get_value, self);
  return ret;
}

extern "C" floatregisterfunc *node2_add_inputfunc (node2 *self) {
  floatregisterfunc *ret = new floatregisterfunc;
  *ret = boost::bind(&node2::add_inputfunc, self, _1);
  return ret;
}

extern "C" floatfunc *node2_get_value (node2 *self) {
  floatfunc *ret = new floatfunc;
  *ret = boost::bind(&node2::get_value, self);
  return ret;
}
```

Declare these functions as declspec(dllexport) under Windows or compile as -fPIC under Linux. The C++ (actually, it's almost C now) linking code would become as follows:

```
int main() {
  node1 *n1 = node1_get_instance(10);
  node1 *n1a = node1_get_instance(8);
  node2 *n2 = node2_get_instance();

  floatfunc *f1 = node1_get_value(n1);
  floatfunc *f1a = node1_get_value(n1a);
  floatregisterfunc *add_inputfunc = node2_add_inputfunc(n2);
  (*add_inputfunc)(f1);
  (*add_inputfunc)(f1a);
```

```
  floatfunc *f2 = node2_get_value(n2);
  std::printf("%.3f\n", (*f1)());
  std::printf("%.3f\n", (*f1a)());
  std::printf("%.3f\n", (*f2)());
}
```

We are still using the type declaration of node1 and node2 to compile this code, but that is because we are respecting type-safety: we could re-declare the (node1 *) and (node2 *) return arguments as (void *) and it would still work. This is exactly what we will do in Python, using the ctypes library. However, because floatfunc and floatregisterfunc are C function pointers not callable from Python, we need first two helper functions to wrap their execution:

```
extern "C" float evalfloatfunc(floatfunc *f) {
  return (*f)();
}

extern "C" void evalfloatregisterfunc(floatregisterfunc *f, floatfunc
*arg) {
  (*f)(arg);
}
```

After that, we can port all of the above C++ linkage code in main() to Python. First, the import of the library:

```
import ctypes

nodes = ctypes.CDLL("nodes.dll")
nodes.node1_get_instance.restype = ctypes.c_void_p
nodes.node1_get_value.restype = ctypes.c_void_p
nodes.node2_get_instance.restype = ctypes.c_void_p
nodes.node2_get_value.restype = ctypes.c_void_p
nodes.node2_add_inputfunc.restype = ctypes.c_void_p
nodes.evalfloatfunc.restype = ctypes.c_float
nodes.evalfloatregisterfunc.restype = ctypes.c_void_p
```

And then, the actual linkage:

```
n1 = nodes.node1_get_instance(ctypes.c_float(10.0))
n1a = nodes.node1_get_instance(ctypes.c_float(8.0))
n2 = nodes.node2_get_instance()

f1 = nodes.node1_get_value(n1)
f1a = nodes.node1_get_value(n1a)

add_inputfunc = nodes.node2_add_inputfunc(n2)
nodes.evalfloatregisterfunc(add_inputfunc, f1)
nodes.evalfloatregisterfunc(add_inputfunc, f1a)
```

```
f2 = nodes.node2_get_value(n2)

print(nodes.evalfloatfunc(f2))
print(nodes.evalfloatfunc(f1a))
print(nodes.evalfloatfunc(f2))
```

The nodes.evalfloatfunc can be wrapped, for example using functools.partial, thus exposing the C++ nodes for linkage to Python nodes. The reverse, exposing Python nodes to C++ nodes, is also possible, but ctypes can only generate naked function pointers, not boost::function pointers. So what we need is a wrapper on the C++ side:

```
typedef float (*nakedfloatfunc)(void);

extern "C" floatfunc *wrapnakedfloatfunc(nakedfloatfunc fx) {
  floatfunc *ret = new floatfunc(fx);
  return ret;
}
```

(this wrapper can also be used to expose operators coded in C to C++ nodes).
Then we can expose to C++ any Python function that takes nothing and returns float:

```
nakedfloatfunc = ctypes.CFUNCTYPE(ctypes.c_float)
nodes.wrapnakedfloatfunc.argtypes = [nakedfloatfunc]
nodes.wrapnakedfloatfunc.restype = ctypes.c_void_p

...

def get_value():
  return 5.0

pygetvalue = nodes.wrapnakedfloatfunc(nakedfloatfunc(get_value))
nodes.evalfloatregisterfunc(add_inputfunc, pygetvalue)

...

print nodes.evalfloatfunc(pygetvalue)
```

In short, it is possible to link (sub)nodes written in C++ to each other and to Python nodes, without any unnecessary overhead at runtime.
The implementation of C++ (sub)nodes will consist of a number of C++ helper functions, typedefs, and perhaps a few macros. On the Python side, these helper functions will then be wrapped inside the same kind of subnodes as those for Python nodes, and a nodesystem.node_cpp metaclass will be developed that looks much like nodesystem.node. Actually, for a node developer, the syntax for building a node consisting of C++ subnodes will be exactly the same as those for Python nodes, except that not every type signature will be available (for example, a weaver emitting a (float, float, int, float) type signature must have been implemented in C++ before it is available for node building; to deal with non-implemented type signatures, a void*[] calling convention can be implemented).

Here is an example of the building of a node consisting of C++ subnodes, based on example 4:

```
from nodesystem.subnodes_cpp import *
“””
imports nodesystem.subnodes_cpp.inputpin,
  nodesystem.subnodes_cpp.outputpin, ...
“””

class mynode(metaclass=nodesystem.node_cpp):
  inp1 = inputpin(“push”, “float”)
  inp2 = inputpin(“push”, “float”)
  outp = outputpin(“push”, “float”, “red”)
  weaver((“push”, (“float”, “float”), “red”), (inp1, inp2))
  unweaver((“push”, “float”), outp)
  operator(ctypes.CDLL(“nodes.dll”).add)
```

Subnodes and subnode features specific to C++ nodes:

“operator”: this subnode takes a boost::function pointer instead of a Python callable object.
“operator_c”: a version of “operator” that accepts naked function pointers instead of boost::function pointers. Note that naked function pointers can be wrapped as boost::function pointers but not vice versa without the use of a runtime Python wrapper, so it is better to use this subnode only with push data.

C++ nodes can have an additional helper subnode called “constructor”, which takes a pointer to a get_instance() C function, e.g. “constructor(ctypes.CDLL(“nodes.dll”).node1_get_instance)“. At node graph startup, this function is evaluated and the returned class instance is stored. Functions registered using the subnode “modifier” do not receive the Python “self” object, but this class instance instead.

“modifier”: this subnode is implemented differently than the nodesystem.node version. It is not a decorator, but a helper. It takes a C++ pointer to a binding function. This binding function must accept a C++ class instance obtained from the “constructor” subnode and must return a boost::function method pointer bound to this class instance, no parameters taken or returned.

“propertybind”: a helper subnode that takes a property subnode object or name, and a C++ pointer to a binding function (i.e. a function that accepts a C++ class instance obtained from the “constructor” subnode and returns a bound class data member (e.g. “boost::bind(&node1::value, node1instance)”). Recorders for C++ nodes do not assign their value to self.<propertyname>, but instead call the registered propertybind function pointer with the recorded value as argument.

Long-term development

In the future, it should be possible to develop a tool to automatically convert Python signal nodes into C++ nodes, based on Shedskin or RPython, by compiling all Python modifiers and operators as C++ and then change the Python node definition into using nodesystem.node_cpp instead of nodesystem.node.

III. Metanodes

The node system here defines nodes with static types, which can be annoying for users because they will have many nodes to select from, only differing in type, and when they change the type they have to destroy the node and build a new one with different types.

The idea is to define metanodes: the nodesystem.metanode metaclass returns not a runbox, but a metarunbox: a class that takes type arguments from the GUI and then returns a runbox. This will require a smart GUI that knows how these metanodes work.

The implementation will be similar to C++ templates: there will be a template class, that can be used instead of static type identifiers such as “int”, “float”, etc.

Here is an example:

A meta red node for push-adding two floats, based on example 4

```
from nodesystem.metasubnodes import *

“””
imports nodesystem.metasubnodes.inputpin,
  nodesystem.metasubnodes.outputpin, ...
  AND nodesystem.metasubnodes.template
“””

def add(v1, v2):
  return type(v1)(v1 + v2)

class mynode(metaclass=nodesystem.metanode):
  t1 = template(“int”, “float”, “string”, “vector”)
  t2 = template(“int”, “float”, “string”, “vector”)
  inp1 = inputpin(“push”, t1)
  inp2 = inputpin(“push”, t2)
  outp = outputpin(“push”, t1, “red”)
  weaver((“push”, (t1, t2), “red”), (inp1, inp2))
  unweaver((“push”, t1), outp)
  operator(add)
```

In principle, templates can be used not only for types, but also for mode and color. Even operator selection should be possible, so that the user can select “add”, “multiply”, etc. in the GUI and the node with the correct operator is built from the template value.

Even C++ metanodes would be possible, but probably quite complicated to define (register not single C++ operators but a dictionary of C++ operators, each for a different combination of template values).