

G Tutorial Series – A Practical Guide to G System Development

Table of Contents

<u>G Tutorial Series – A Practical Guide to G System Development</u>	1
<u>Raphael Langerhorst</u>	1
<u>Anton Melser</u>	1
<u>Chapter 1. Tutorial 1 – Basic Agent Design and Implementation</u>	1
<u>Introduction</u>	2
<u>Step one: sending influences</u>	2
<u>InfluenceSender class declaration</u>	3
<u>Step two: receiving influences</u>	4
<u>Step three: writing the application</u>	5
<u>Source Files</u>	7
<u>Chapter 2. Copyright and Licencing Information</u>	7
<u>Documentation Licencing</u>	7
<u>Code Licencing</u>	7

G Tutorial Series – A Practical Guide to G System Development

Raphael Langerhorst

Anton Melser

Version 0.3.1

Copyright © 2004 Raphael Langerhorst

Copyright © 2004 Anton Melser

Abstract

This Tutorial Series gives a hands-on introduction to development with the G System. It is work in progress and additions or improvements are welcome. Contact the mailing list or have a look at the forum on the G System [website](#) for any comments. We appreciate feedback.

Table of Contents

1. Tutorial 1 – Basic Agent Design and Implementation

Introduction

Step one: sending influences

InfluenceSender class declaration

Step two: receiving influences

Step three: writing the application

Source Files

2. Copyright and Licencing Information

Documentation Licencing

Code Licencing

Chapter 1. Tutorial 1 – Basic Agent Design and Implementation

Table of Contents

Introduction

Step one: sending influences

InfluenceSender class declaration

Step two: receiving influences

Step three: writing the application

Source Files

Introduction

This tutorial is based on version 0.3 of the G System.

This tutorial will guide you through the process of agent design and implementation. Agents are used to provide behaviour or functionality for elements. The agent interface and the element structure as a whole are defined in the G Core System (GCS) library. All needed information will be provided in this tutorial, but you still might want to take a look at the [API documentation](#) while working through the tutorial.

When we have created the agents we will also write the application that puts it all together, but we will see that this is a simple step.

Step one: sending influences

By subclassing the `GCS::GAgent` class, we already have a compilable agent. Of course it does not do much out of itself, but we only need to implement what we really need.

There are two important methods, that the agent interface offers: `GAgent::run()` and `GAgent::receiveInfluence()`.

The `run()` method is executed as an independent thread. `GAgent` inherits `QThread` to make this possible. The `receiveInfluence()` method is called whenever an influence is received by the element.

`GAgent::run()` should hold all the code that the agent should do out of itself. It is possible to create any level of complexity here. The agent could even access information from the internet and act accordingly, there is no restriction. Also it is perfectly fine to create more threads, but then starting and parking the agent must be implemented carefully. See the API documentation for details.

Our first agent will be called `InfluenceSender` and inherits `GAgent`. It should periodically radiate an influence, we can achieve this with a simple reimplement of `GAgent::run()`.

```
void InfluenceSender::run()
{
    while(!shutdown)
    {
        GEnergy* e = this->requestEnergy();

        GElementInfluence influence( this->getElementID(), e->take(0.01));
        emit radiateInfluence( influence );

        sleep(2);
    }
}
```

Let's walk this through line by line.

```
while(!shutdown)
```

`shutdown` is a protected attribute of `GAgent`. By default it is `FALSE` as long as the agent should execute, if the element should stop to execute then this variable is set to `TRUE`. With this while loop we make sure the thread stops, that means the `run()` method returns, when it is supposed to do so. Basically every agent that runs as a thread has this while loop. More complex agents might have to implement their own way of shutting

down though.

```
GEnergy* e = this->requestEnergy();
```

Every element has energy that gives the element a certain characteristic. When we want to influence something, this "personal" characteristic of the element is always included in the influence we generate.

```
GElementInfluence influence( this->getElementID(), e->take(0.01));
```

The `GCS::GElementInfluence` class is used as a base class for all influences. It even implements a complete influence in itself without having to create a subclass. The required parameters are the source ID, which is represented by a `GCS::GElementID` object. This ID of the element can be obtained by `GAgent::getElementID()`. The second parameter of an influence is the energy that we load the influence with. We already obtained a pointer to the element's energy and `GEnergy::take()` creates a `GEnergy` object with the given fraction of the original energy. This means, we take 1% (0.01) of our own energy and load the influence with it. Normally the more energy we put into an influence the stronger is the effect. But we need to consider that every element only has a certain amount of energy so we need to be careful about how much we really use for an influence. As we will see later, by receiving influence the energy amount of the element can increase.

```
emit radiateInfluence( influence );
```

As soon as we have created the influence object we are ready to send it out. The `GAgent` class defines two signals for influence sending: `GAgent::radiateInfluence()` and `GAgent::sendInfluence()`. The first is used to radiate the influence into the surrounding environment and the second is used to send the influence to a specific destination element. We simply want to radiate the influence into our surrounding. What elements are affected by radiating influences depends on the implementation of the G World Engine (GWE). The implementation we will use in our application is the `GWEInterfaceSimple`. In this class radiation of influences affects the element's parent, all children of the source element and all elements that are touching the source element (forms overlap).

```
sleep(2);
```

This method is provided by `QThread` and puts the agent to sleep for given number of seconds. After that execution starts just after this call, which means that we are at the beginning of the while loop.

InfluenceSender class declaration

We already discussed the implementation details for the `InfluenceSender` class. Now we will take a brief look at how a subclass of `GAgent` basically looks. Keep in mind that reimplemented methods should always be declared virtual.

```
#ifndef INFLUENCESENDERH
#define INFLUENCESENDERH

#include <GAgent.h>

class InfluenceSender : public GCS::GAgent
{
    Q_OBJECT
public:
    InfluenceSender(QObject* parent = 0, const char* name = 0);
    virtual ~InfluenceSender();
};
```

```

protected:
    virtual void run();
};

#endif

```

To define a `GAgent` subclass we simply inherit it. Please note that `GAgent` is in the `GCS` namespace. Since the

```
using namespace XXX;
```

should be avoided in header files we simply put the namespace in front of used classes. Also every subclass should have a virtual destructor as this allows clean freeing of memory. This especially applies for custom agents and forms.

Step two: receiving influences

Now that we have seen how to send an influence in the `run()` method we can take a look at how we handle influence receiving in the `receiveInfluence()` slot. This slot gets called whenever the element receives an influence, just like the one we have created in the previous section.

We will stick to a very simple implementation so it is easy to follow. The agent we use for receiving will be called `InfluenceReceiver`, and all we want to do is to print information about the received influence. So this is how the implementation looks:

```

void InfluenceReceiver::receiveInfluence(GElementInfluence& influence)
{
    const GElementID& source_id = influence.source();
    qDebug("influence received from " + QString::number(source_id.getID())
        + " by " + QString::number(getElementID().getID()) + "...");

    GEnergy& inf_energy = influence.Energy;
    qDebug(" energy (level,amount,sigma): "
        + QString::number(inf_energy.level()) + " "
        + QString::number(inf_energy.amount()) + " "
        + QString::number(inf_energy.sigma()));

    GEnergy* own_energy = this->requestEnergy();
    own_energy->put(inf_energy);
}

```

This implementation shows how to access the data of an received influence. We will also write this data to the console so we can see what our elements are doing when we run the application.

`qDebug()` is used for output and simply accepts a string which it writes to the standard output of the application which in turn is our console. `QString::number()` converts a given number to a string.

```
const GElementID& source_id = influence.source();
```

`GElementInfluence::source()` returns a constant `GElementID` reference to the source element. With it we can identify from where the influence came. In complexer environments the receiving agent could create a child (subelement) with a connection ID of the source element's `GElementID`. A connection ID of an element can be seen as the element this element is concerned with. In particular it could send (in contrast to

radiate!) influences of certain specialised types to it to achieve some desired result. See the API documentation of `GCS::GObject::Connection` for additional information. This attribute will play an important role in complex settings with intelligent agents.

```
GEnergy& inf_energy = influence.Energy;
```

The energy of an influence is a publicly accessible attribute. This essentially means that we can even modify it. Here we store a reference to it in a local variable.

```
GEnergy* own_energy = this->requestEnergy();  
own_energy->put(inf_energy);
```

And this is exactly what we do here. We "put" the energy from the influence into the element's own energy. This is a process of mixing two energies of different level, amount and sigma into one energy. This is perfectly comparable to mixing two bottles of water with different colours (the colour being the energy level).

So we see that the `GCS::GElementInfluence` class offers two pieces of information:

- The ID of the influence source.
- The energy that the influence is loaded with.

Step three: writing the application

After writing our two agents, we will put it all together in a small application. For this we will create two elements, a parent and its child. Radiated influences always influence the source's parent and children, so in this setting the elements will influence each other.

The `main()` function could look like this:

```
#include <qapplication.h>  
  
#include "InfluenceSender.h"  
#include "InfluenceReceiver.h"  
#include <GWEInterfaceSimple.h>  
#include <GElement.h>  
#include <GObject.h>  
#include <GEnergy.h>  
#include <GElementID.h>  
  
using namespace GCS;  
using namespace GWE;  
  
int main(int argc, char** argv)  
{  
    QApplication a(argc,argv);  
    GWEInterfaceSimple* gwe = new GWEInterfaceSimple();  
  
    GObject* object_parent = new GObject(  
        new GEnergy(5,10,3), //level,amount,sigma  
        NULL, //no form  
        GElementID(1), //the element is it's own parent  
        GElementID(1), //ID is 1  
        GElementID(1)); //connection ID is 1 - itself  
  
    GElement* element_parent = new GElement(object_parent);
```

G Tutorial Series – A Practical Guide to G System Development

```
element_parent->addAgent(new InfluenceSender());
element_parent->addAgent(new InfluenceReceiver());

gwe->add(element_parent);

GElement* child = new GElement(
    new GObject(
        new GEnergy(3,1,2), //level,amount,sigma
        NULL, //no form
        GElementID(1), //parent
        GElementID(2), //ID is 2
        GElementID(1)); //connection ID is 1 - the parent

child->addAgent(new InfluenceSender());
child->addAgent(new InfluenceReceiver());
object_parent->addChild(child->getElementID());

gwe->add(child);

element_parent->executeElement();
child->executeElement();

return a.exec();
}
```

The `main()` function is in general very flexible in design. It is important that a `QApplication` object is present before any G related classes are instantiated, because many of these classes are derived from `QObject`. To be able to create a `QApplication` instance we need to include `qapplication.h`. Additionally we include the two header files from our self-made agents. Also it is important to include every used class of the G libraries. And in implementation files,

```
using namespace XXX;
```

should be used so we can use these classes without namespaces.

After creating the `QApplication`, we can create a `GWE` instance. This is used to handle all elements and manage influence sending and radiating between elements. Without a `GWE` influences wouldn't work.

The next step is to create the `GObject` of the parent element. The object holds all the data of an element, so we have to give an energy, a form and IDs for the parent, for itself and for the connected element. We store the created object in a pointer as we need it later to add the child.

Creating the element itself is then very simple, we only need to pass it the object.

In order to set agents for an element we can use `GElement::addAgent()`. The element takes care of initialisation of the agent, which means setting the `GAgent::Object` and `GAgent::Agents` attributes and starting the agent thread when the element is executed.

After the element itself is created and the agents are added, we add the element to the `GWE`. Note that when an agent creates a child element it only needs to emit a `GAgent::childElementCreated()` signal and the `GWE` recognises the child automatically, as long as the parent itself was added to the `GWE`. This means that only the top-most element needs to be added to the `GWE` manually and if this top-most element creates it's own children they will be automatically registered with the `GWE` through this signal. In our tutorial we don't create the child with an agent in the parent, so we will also have to do the parenting and the registration

with the GWE ourselves.

It is important that every element has a parent. The top-most element is essentially its own parent. This is currently the convention used by the GWE, but it might change in the future. The API Reference is your friend in this case.

Then we create the child element with a slightly different energy level. After adding the agents we also add this element as a child to the parent's object. Then we add to the GWE.

As soon as all initialisation is done we can execute the elements and start the Qt event loop. Our application is running now. When an element is executed, it also starts all agent threads, just as it parks all agents when the element is parked. So no direct communication to the agents is necessary.

You could now try to complete the source code yourself with missing header files and implementations and compile it. It is highly recommended you go for it yourself. If you do encounter problems you cannot solve, you are welcome to use the forums on our website or the mailing list for your questions. You could also download the source files for this tutorial, which includes a qmake project file and short installation instructions, but this is only recommended when you don't manage to create the application yourself.

Source Files

The complete source code for the tutorial can be found at <ftp://ftp.g-system.at/data/tutorials/>.

Chapter 2. Copyright and Licencing Information

Table of Contents

[Documentation Licencing](#)

[Code Licencing](#)

Documentation Licencing

The documentation is provided under the GNU Free Documentation Licence.

Copyright (c) 2004 Raphael Langerhorst.

Copyright (c) 2004 Anton Melser.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the appendix entitled "GNU Free Documentation License".

Code Licencing

All code released by the G System Team is available under a BSD licence. A copy of the license is included in the appendix entitled "BSD License"

Copyright (c) 2004 Rapheal Langerhorst All rights reserved.

[G System Documentation Home](#)