G System Developers Handbook

Table of Contents

G System Developers Handbook **1**
Raphael Langerhorst 1
Anton Melser 1
Chapter 1. Introduction 3
Work in Progress 4
What is the G System? 4
Abstract 4
The G System in the World Wide Web 4
G System licenses 4
Overview of the G System Components 4
Project Goals 5
Purpose of the G System Developers Handbook 5
Handbook Structure 6
Ideas backing up the Project 7
What is it all about? 7
A Framework and its Universe 8
Evolution 8
Chapter 2. The Project 8
A little bit of History 9
The Team, Mailing Lists, Contact 9
The G System Team 9
Contact 10
Getting Involved – Joining the Team 10
Introduction 10
Choose what you want to work on 11
Join the appropriate mailing lists 11
Learn the fundamental technologies that you need 11
Read the G System Developers Handbook 11
Guidelines for G System development 11
Introduction 12
Development Environment 12
Libraries used for G System Development 13
Tools used for G System Development 13
Step by step guide for setting up an example environment for G System development 13
Introduction 13
The Operating System 14
Installing pkgsrc 14
Packages to install 15
Tweaking NetBSD 15
Environment variables 16
Setting up the PostgreSQL database management system 17
Setting up the XMPP/Jabber server 18
Getting the G System source code 19
Configuring the G Universe Server and the G Universe Client 19
Running the G Universe Server and the G Universe Client 21
Building the documentation 21
Using KDevelop 21
Chapter 3. Project and Feature Plan, Raodmap 21

Table of Contents

G System Developers Handbook
Roadmap....22
Featurelist....22
- Introduction....23
- G Core System (GCS)....23
- G World Engine (GWE)....23
- G Client Engine (GCE)....24
- G Basic Elements (GBE)....25
- G Universe: The universe created with the G System....25

TODO list....26
- Introduction....26
- To be done during 0.6 release cycle....26
- JUNIOR JOBS....29
- General things to be done, HIGH PRIORITY....29
- Other general things to be done....29
- Documentation....29
- Source Code....29
- OPEN ISSUES....30

Chapter 4. Understanding fundamental G System Technologies....31
Introduction....31
Qt....31
Subversion....32
XMPP – Extensible Messaging and Presence Protocol....32
Chapter 5. Source Management....32
The Subversion Repository....32
- What is Subversion?....32
- The G Source Code Repository....32

Compiling and Installing....33
Documentation....34
Website....34
Chapter 6. Architecture....34
Overview....35
- Introduction....35
- Parts of the G System....35
- Evolution and the G System – putting it together....35

The G Core System (GCS) – defining the element structure....36
- Introduction....36
- Elements....36
- Objects, element data....36
- Agents....37
- Interaction between elements – influence handling....37
- Hierarchical world structuring....38
- Some notes on the GCS....39

The G World Engine (GWE) – connecting elements....39
- The Purpose of the GWE....39
- GWE Design....39
- Network Infrastructure....41
- Elements and the GWE....42

Table of Contents

G System Developers Handbook
- Clients and GWE Servers 43

GWE Sepcifications and Protocol 43
- GWE as a GCS Container 43
- GWE network communication behaviour 44
- XML Constructs for GWE Communication 45

G Basic Elements (GBE) – bringing the system to life 47
- Deterministic Random World Generation 47
- Element hierarchy management 47

The G Client Engine (GCE) – making it visible 47

G Universe Server 48

G Universe Client 48

GOD 48

Agent Plugin Architecture 48
- Overview 48
- Plugin management in the GWE 49
- Developing an agent plugin 49

Chapter 7. Using and Extending the G System Framework 49

Tutorial: Basic Agent Design 49
- Introduction 49
- Step one: sending influences 50
- Step two: receiving influences 52
- Step three: writing the application 53
- Source Files 55

Chapter 8. Contributing to the G System Project 55

The purpose of the G System 55

Joining the team 55

Donations 55

G System Developers Handbook

Raphael Langerhorst

Anton Melser

0.6.0

Copyright © 2004 – 2006 Raphael Langerhorst

Copyright © 2004 Anton Melser

Abstract

The G System is a framework for building large scale virtual realities, simulations, experiments and games.

Table of Contents

1. Introduction
- *Work in Progress*
- *What is the G System?*
  - *Abstract*
  - *The G System in the World Wide Web*
  - *G System licenses*
  - *Overview of the G System Components*
  - *Project Goals*
- *Purpose of the G System Developers Handbook*
- *Handbook Structure*
- *Ideas backing up the Project*
  - *What is it all about?*
  - *A Framework and its Universe*
  - *Evolution*

2. The Project
- *A little bit of History*
- *The Team, Mailing Lists, Contact*
  - *The G System Team*
  - *Contact*
- *Getting Involved – Joining the Team*
  - *Introduction*
  - *Choose what you want to work on*
  - *Join the appropriate mailing lists*
  - *Learn the fundamental technologies that you need*
  - *Read the G System Developers Handbook*
- *Guidelines for G System development*
  - *Introduction*
  - *Development Environment*
  - *Libraries used for G System Development*
  - *Tools used for G System Development*
- *Step by step guide for setting up an example environment for G System development*

        - *Introduction*
        - *The Operating System*
        - *Installing pkgsrc*
        - *Packages to install*
        - *Tweaking NetBSD*
        - *Environment variables*
        - *Setting up the PostgreSQL database management system*
        - *Setting up the XMPP/Jabber server*
        - *Getting the G System source code*
        - *Configuring the G Universe Server and the G Universe Client*
        - *Running the G Universe Server and the G Universe Client*
        - *Building the documentation*
        - *Using KDevelop*
- *3. Project and Feature Plan, Raodmap*
    - *Roadmap*
    - *Featurelist*
        - *Introduction*
        - *G Core System (GCS)*
        - *G World Engine (GWE)*
        - *G Client Engine (GCE)*
        - *G Basic Elements (GBE)*
        - *G Universe: The universe created with the G System*
    - *TODO list*
        - *Introduction*
        - *To be done during 0.6 release cycle*
        - *JUNIOR JOBS*
        - *General things to be done, HIGH PRIORITY*
        - *Other general things to be done*
        - *Documentation*
        - *Source Code*
        - *OPEN ISSUES*
- *4. Understanding fundamental G System Technologies*
    - *Introduction*
    - *Qt*
    - *Subversion*
    - *XMPP – Extensible Messaging and Presence Protocol*
- *5. Source Management*
    - *The Subversion Repository*
        - *What is Subversion?*
        - *The G Source Code Repository*
    - *Compiling and Installing*
    - *Documentation*
    - *Website*
- *6. Architecture*
    - *Overview*
        - *Introduction*
        - *Parts of the G System*
        - *Evolution and the G System – putting it together*
    - *The G Core System (GCS) – defining the element structure*
        - *Introduction*
        - *Elements*

- Objects, element data
- Agents
- Interaction between elements – influence handling
- Hierarchical world structuring
- Some notes on the GCS
The G World Engine (GWE) – connecting elements
- The Purpose of the GWE
- GWE Design
- Network Infrastructure
- Elements and the GWE
- Clients and GWE Servers
GWE Sepcifications and Protocol
- GWE as a GCS Container
- GWE network communication behaviour
- XML Constructs for GWE Communication
G Basic Elements (GBE) – bringing the system to life
- Deterministic Random World Generation
- Element hierarchy management
The G Client Engine (GCE) – making it visible
G Universe Server
G Universe Client
GOD
Agent Plugin Architecture
- Overview
- Plugin management in the GWE
- Developing an agent plugin
7. Using and Extending the G System Framework
Tutorial: Basic Agent Design
- Introduction
- Step one: sending influences
- Step two: receiving influences
- Step three: writing the application
- Source Files
8. Contributing to the G System Project
The purpose of the G System
Joining the team
Donations

Chapter 1. Introduction

Table of Contents

Work in Progress
What is the G System?
- Abstract
- The G System in the World Wide Web
- G System licenses
- Overview of the G System Components
- Project Goals
Purpose of the G System Developers Handbook

Handbook Structure
Ideas backing up the Project
    What is it all about?
    A Framework and its Universe
    Evolution

Work in Progress

The G System Developers Handbook is always a work in progress and adapting to the current state of the G project. As such it is of course always open to contributions and suggestions for improvements. Please feel free to contact us and send us any feedback.

What is the G System?

Abstract

The G System is a C++/Qt based framework for simulation of virtual realities. The core library defines the structure of world content, so called elements, and defines the interface for so called agents which implement the behaviour of elements.

A world engine (GWE) is used for storage (DB), communication between elements and network transparency. The goal for the GWE is to build a hierarchical network distributed system that allows for huge worlds to be simulated. The structure represents the structure of the world itself like solar systems – planets – continents – areas – cities/lands – ...

We also build such a virtual reality with the system, it is called G Universe.

The G System in the World Wide Web

The main website of the G System project is http://www.g–system.at.

G System licenses

BSD style license (that means it's *free and open source* software) for all the source code except external libraries that are used, they have their own license (usually LGPL or BSD).

The documentation is available under the GNU Free Documentation License.

Overview of the G System Components

G System components:

- GCS – G Core System (src/core)
- GBE – G Basic Elements (src/basicelements)
- GWE – G World Engine (src/worldengine)
- GCE – G Client Engine (src/clientengine)
- demo – a demo application for testing purposes (src/demo)
- GOD – G Options Dialog for world engine configuration (src/god)
- G Universe Server – lightweight app for running the GWE Server (src/guniverse)

- G Universe Client – client frontend to the universe (src/guniverseclient)

The GCS (library) defines the element structure. Elements are the basic building blocks for world content.

The GBE (library) is a library of basic element parts (agents, influences, forms,...) that can be used to faster build new worlds.

The GWE (library) provides the execution environment for elements. This also includes the network layer as well as the DB interface. It makes interaction between elements possible in a network transparent way. Note that the interaction itself is handled by agents (part of the GCS).

The GCE (library) is responsible for user interaction. It represents the world content in a 3D environment and allows for user input.

All parts have full API documentation (for further reading).

The demo application shows the basic usage and features of the system and can also serve as a quick testing framework.

GOD serves as an easy to use graphical configuration utility for the GWE. You can create/modify configuration files and integrate a GWE Server into an existing server network infrastructure.

A simple startup application is guniverse. It can load existing configurations, written by hand or created with GOD, into the GWE and initialize it. guniverse is also meant to be usable as system startup application to initialize a GWE Server on system startup (rc script).

guniverseclient is the "user client" for the G Universe. It builds on top of the G System components (libraries) and provides an user interface to interact with user elements.

Project Goals

Goals as a library

Provide a simulation framework.

Goals as an Application

Simulation of evolution/life.

Eternal laws govern our life. They make evolution possible. The G System tries to catch these laws into usable software and create a virtual world that makes it possible to experience these.

Purpose of the G System Developers Handbook

The main goals of this book are:

- *A handbook for users of the G System Framework.*

  The users of the G System Framework are software developers that use the G System to create their own software. This handbook helps them to understand the design of the G System and how to use it.

- *A handbook for developers of the G System themselves.*

  Developing various parts of the G System and G Universe requires a solid understanding of the components and a good general understanding of the overall project architecture. This handbook helps them to ease the learning of various parts of the project and the project as a whole. This is especially useful for new developers and people that want to join the team.
- *A cookbook for current developments.*

  There are always things that are currently worked on. The feature plan also discusses various aspects and design issues of features that are currently worked on. Including all this into the G System Developers Handbook allows everyone to follow the changes within this project.

It is important to remember that this is *not* a handbook on how to use the various G Universe applications. Take a look at the G Universe Handbook for this purpose.

Handbook Structure

The content of the G System Developers Handbook aims at helping developers to understand the G System. As such, different parts might be interesting for different people, in particular depending on whether you are a G System developer yourself or if you just use the G System Framework to build your own applications.

The G System Developers Handbook is structured in various parts:

- *General introduction to G*

  Chapters 1 and 2 are intended as a good introduction for anyone interested in the G System.
- *Featureplan, Roadmap and Cookbook*

  Chapter 3 offers a lot of information on feature status, current developments, TODO items and a general roadmap. It is a good resource for keeping track on what's going on with the project and what features are being worked on and will be available in the future.
- *Learning about used Technology*

  In order to get an understand of all the fundamental technology that is utilized by the G System, these technologies are briefly discussed in chapter 4, with suggestions for further reading. If you are not familiar with some of the technology used by the G System you might find some information in this chapter.
- *Installation, Source Management, Documentation, Website*

  Chapter 5 gives instructions on installing the G System and the G Universe as well as a general overview on the source code repository, documentation and the website. If you want to work with development versions, or intend to join the G System team, this is a must–read.
- *Architecture*

  Chapter 6 dives into the design and architecture details of each G System components as well as various frontends that are used for the G Universe. This is the main chapter for actually understanding the design of the G System.
- *Building applications with the G System*

Chapter 7 is all about using the G System in your own application. A must read for all users of the G System Framework that develop their own applications.

There are also tutorials in this chapter that help with understanding how to utilize the G System.

- *Contributing*

  The G System is open source, which means anyone can also contribute to it. Donations are also welcome which make sure that the developers can spend more time on the G System. If you want to contribute, please take a look at chapter 8.

Ideas backing up the Project

What is it all about?

In order to understand the G System and its way of development it is helpful to understand some of the basic ideas that drives this project forward.

The G System (or G for short) was designed primarily for one purpose: the simulation of evolution. But what does this mean? This is a good question to ask and happily it is not too difficult to answer. Before we start, we want to say that the system is in no way limited to this kind of simulation but this is what we have in mind while creating the G System. If you intend to use it for something completely different you will probably find out that it is quite useful for many purposes – because everything actually is part of evolution...

Many scientific or mathematical simulation frameworks and applications are highly specialized in their domain, which is mostly of a technical nature like electronics, mechanics or any combination. The G System does not attempt to bring a complete framework in such domains but tries to fill the gap of a more social, life and evolution framework. Still, it provides the means to construct many highly technical simulations and in fact such implementations could bring even more completeness to the virtual universe that the G System brings forth. The main aspects of the G System do not in any way contradict to technical simulations. In fact, life itself is a wonderfully scientific system. This system can model life, evolution and social aspects of our day to day experiences without taking away the fascination passionate philosophers intuitively feel about the subject.

It is important to remember that the word *simulation* in the context of the G System is not strictly used according to the technical meaning of the word in todays science. Instead, the term is rather extended and applied to a much wider range of phenomas we all encounter in daily life. This means that the G System is a simulation in a broader view of things, trying to provide a simulation and a framework for life, society and evolution besides being a general purpose simulation framework.

The second term that is heavily used and needs to be clarified is *evolution*. In general evolution is understood as evolution of form, meaning genetic changes in bodies of a species through successive generations. But that is truly just a minor part in the larger scheme of things. We define evolution as evolution of life, meaning the change of consciousness of the life that is expressed through the forms. Life never dies or goes away. It's always the same life that, through time, uses different forms to express itself.

Every software project has some origin, a reason why it exists and why developers actually are willing to spend their time on. Thoughts, ideas and a lot of other things are written down in the Philocorner document of the G System. To understand the G System, one should understand the philosophy that backs up this project. In some ways these philosophical considerations opens the project to a group of people normally not involved in software projects at all. Such people are of course highly welcome to join.

A Framework and its Universe

The G System Framework, as such, is not an application in itself, it is rather a set of tools and libraries that can be used to make a simulation. They work by providing a kind of virtual reality whose constituent elements can evolve over time in a realistic way. Users can interact with this reality and thus influence the environment. By being able to interact with others in the environment users themselves are part of the process of evolution.

On the other hand, the G Universe is the virtual reality built with the G System in order to implement the desired evolution simulation. This universe is open to anyone who wants to take part in it. Participation of human entities results in a highly dynamic and interesting behaviour of the overall virtual universe. This also leads to a worldwide virtual community that builds up a complete society in this virtual world. The G Universe can in fact be used to experiement with complex social settings as the working of the virtual world just represents real life – since real life is evolution. The difference is that the G System offers more possibilities, as it is virtual; and, being virtual, it allows for better analysation and thus understanding of the processes involved.

Evolution

Take a quick look at our world, what is there? People, animals, plants ... but also cities, nations, the planet as a whole, many natural areas like forests, continents, rivers, the air, ... On a different level we can consider that our environment is filled with certain ideas, concepts, feelings, etc., whatever makes up the atmosphere we feel or sense around us when we go to work, interact with other people, or just sit in a quiet place. The point is that all things around us are changing in one way or another, and we ourself change in time. We can call this "evolution". How is this accomplished? All systems follow certain laws, physicists know this well. But there are not only physical laws which accurately describe the behaviour of physical matter, there are also laws that apply to the latter categories (emotions, thoughts, ideas, the perceived ambiance of the place we are right now,...). Under such laws changes take place inside the system according to how the individual parts influence each other. Now, this is a very rational approach but I think it shows the basic concept.

Now let's take a look at the universe. Even this enormous system is changing/evolving in some way. Solar systems come into existence and fade away. And inside such solar systems are planets. On some such planets we may find evolution to be quite active...

I think the most interesting way of looking at evolution is to think about human beings, which is probably a very good subject for these studies. For now, this is left to the reader as an exercise...

Summing things up we can say that everything is evolving and evolution is possible through influences between parts of a system. Designing the influences themselves and the reactions are two major steps in creating a simulation with the G System.

This was a basic overview of what the G System aims at. It tries to give you a context for the rest of the documentation.

Chapter 2. The Project

Table of Contents

A little bit of History
The Team, Mailing Lists, Contact
    The G System Team

- Contact
- Getting Involved – Joining the Team
  - Introduction
  - Choose what you want to work on
  - Join the appropriate mailing lists
  - Learn the fundamental technologies that you need
  - Read the G System Developers Handbook
- Guidelines for G System development
  - Introduction
  - Development Environment
  - Libraries used for G System Development
  - Tools used for G System Development
- Step by step guide for setting up an example environment for G System development
  - Introduction
  - The Operating System
  - Installing pkgsrc
  - Packages to install
  - Tweaking NetBSD
  - Environment variables
  - Setting up the PostgreSQL database management system
  - Setting up the XMPP/Jabber server
  - Getting the G System source code
  - Configuring the G Universe Server and the G Universe Client
  - Running the G Universe Server and the G Universe Client
  - Building the documentation
  - Using KDevelop

A little bit of History

To be written.

The Team, Mailing Lists, Contact

The G System Team

Introduction

Team information and roles are available on the homepage.

G System development has lots of different aspects, not just writing code. Thus what team members do varies greatly, but still everything is necessary in the end to make the project complete. The team has so far been very stable, which means there is hardly any team member fluctuation.

The G System develops slowly but consistently. There is usually not tons of code changed every day. Rather, actual changes are well thought about and sometimes there is no code change for weeks. Thus, code that is written hardly gets thrown away later, as it is well thought about in the first place. Thus, our way of developing the G System is slow, but steady and consistent.

It is important for us not just to write the code, but also to make sure the project as a whole is well

documented, that the web page is maintained and so on. So there is lots of things going on not reflected in the code itself.

Current Team Members and Activities within the Project

Anton Melser created the documentation infrastructure and writes some documentation as well as various parts of the source code.

Gerald Degeneve is the webmaster. He did all the website design and implementation (three times, we owe him a lot). Apart from mastering the web he writes some source code, in particular physics agent design and implementation. He's also the main graphics artist.

Martin Reinsprecht handles all the infrastructure for the project: web and project servers, mailinglists and so on.

Raphael Langerhorst does most of the project coordination and architectural design and writes lots of code and documentation.

People somehow involved with the project

Todd Burnham is the man behind the idea of a virtual online 3D space for communities.

Contact

Mailing Lists

We mainly use mailing lists to communicate with each other, although often we directly communicate via instant messaging, telephone or personal meetings.

Everyone that wants to contact us is encouraged to sign up on one of our mailing lists which are separated for various topics. Simply choose the one where you think your issue or topic fits best, sign up and send a mail to it. If you are unsure which list to use, then you should use the users mailing list.

Personal Contact

You can also contact Raphael directly for anything related to the G System although you're encouraged to use the mailing lists if possible. Raphael's email addresses are raphael–langerhorst@gmx.at and raphael.langerhorst@kdemail.net .

Getting Involved – Joining the Team

Introduction

If you intend to join the G System team, you are very welcome to do so. We are very happy for every contribution made to the project.

In this section you will find some guidelines and information on how to best procede to get involved.

An important aspect to consider is that the G System is fully an open source project, meaning it is completely free to be used by everyone for any purpose without restrictions. So it is a project for everyone, for common

good. Whatever you invest enriches all that are related to this project. Likewise everything that other people invest comes to your good – which is basically the whole idea behind open source: you give a tiny bit and you get all the contribution that other people give.

There is also a more general section about contributing at the end of the G System Developers Handbook. In particular, if you don't intend to join the team, but want to support the G System you can donate to the project, thus allowing the team of developers to spend more time on the project.

Choose what you want to work on

The first thing to do is to choose what you want to work on. There are really many options for helping the project:

- Being an User
- Giving Feedback
- Source code development, architecture and design
- Documentation writing and proofreading
- Translating
- Artists, graphical as well as musical
- Project Coordination
- Providing and managing infrastructure
- Binary package building and maintaining
- Testing, also for various plattforms
- Sharing your philosophical insights.
- Sharing your scientific insights.
- Financial support, donations (also hardware, or books).
- Working on something from the TODO list.

Even if you have no suitable abilities or simply no time to spend, then at least you can help us to be able to spend more time on the project by donating money.

Join the appropriate mailing lists

After deciding what you want to work on you should sign up for the appropriate mailing lists, which can be found on the G System homepage. Please discuss what you want to do on these mailing lists.

Learn the fundamental technologies that you need

In any case you should read the section on the fundamental technologies that the G System utilizes and get familiar with what you need.

Read the G System Developers Handbook

And please fully read this G System Developers Handbook, there is lots of valuable information to be found in it.

Guidelines for G System development

Introduction

This section gives you some advice on how to work with this project: what tools to use, how to prepare patches and so on. Most of these guidelines are not obligatory, but usually provide the best way to work with the project. If you have other or better ways to get things done, in particular if you use different tools, then you should just carry on using them as long as the results are usable.

Development Environment

Operating System

G System development can best be done on POSIX compliant operating systems. This usually means to use some kind of Unix, like Linux, FreeBSD or NetBSD which are all freely available. All of the tools used are available on these systems. Still, the source code itself is not platform specific and can be compiled on any operating system supported by Qt and KDE.

MAC OS X is based on FreeBSD and thus also provides a suitable environment for G System development, although none of the current developers uses it. This means you might encounter some initial troubles with this operating system. Nevertheless, if you are using a MAC and can work out the initial issues we would be grateful to add MAC OS X to our list of possible development platforms.

Windows is a very difficult development platform, as it is not standards compliant. Still, you might at least get the source code to compile with gcc and Qt for Windows. The Borland C++ compiler might work as well, but probably has some issues. Building the documentation or doing any reasonable development on Windows is probably difficult, likely more difficult than on MAC OS X. If you get some setup to work please let us know.

Installing a suitable operating system for G System development

You might already have a suitable platform installed for G System development. If not then you should get and install one of the following platforms, whichever suits you best.

It is very important to note that the only difference in the listed operating systems is the base system. Basically all of these systems provide the same set of software, like KDE. So it is a matter of taste what you prefer to install, you will have all required tools available on all the listed operating systems.

All of theses systems are not just available on Intel x86 hardware, but also on x86_64 (AMD64), PPC (MAC), Sparc and many more. NetBSD is available on over 50 such platforms! So if you have exotic hardware you will still be able to run these systems.

- *Linux*

  If you want to install Linux you can choose between various distributions, that provide a full Linux based operating system which usually includes all required tools for G System development out−of−the−box.

  Linux Distributions:

  - OpenSuse: http://www.opensuse.org
  - Kubuntu: http://www.kubuntu.org
  - Fedora: http://fedora.redhat.com

    - Mandriva: http://www.mandriva.org
    - Debian: http://www.debian.org
- *FreeBSD*

  FreeBSD is a full operating system in itself, and probably the most complete available anywhere. Together with NetBSD and OpenBSD it is derived from the original BSD code (Berkeley Software Distribution). You should have some basic knowledge of Unix if you want to use FreeBSD.

  You can find FreeBSD in the internet at http://www.freebsd.org.
- *NetBSD*

  NetBSD, like FreeBSD, is a full operating system in itself, but more lean and mean. You should have some basic knowledge of Unix if you want to use NetBSD.

  You can find NetBSD in the internet at http://www.netbsd.org.

Libraries used for G System Development

All these operating systems come with a large number of libraries and application that can be installed right away. If one of the required libraries or tools are not included with the installation CDs of the operating system, you might need to download and install them yourself.

You should make sure that you have the following libraries installed:

- Qt 4
- Qt SQLite database plugin (or any other supported DB plugin)

Tools used for G System Development

You should make sure that you have the following tools and applications installed:

- Subversion
- KDevelop (recommended IDE)
- Doxygen
- graphviz (used by doxygen)
- docbook–xml
- docbook–xsl
- htmldoc
- xsltproc (libxslt)

Step by step guide for setting up an example environment for G System development

Introduction

Here you will find an example setup that can be used for G System development. It is by no means the one and only possible setup, but you may find the instructions helpful to set up your own environment. The final setup will be useful for both working on the source code as well as documentation. You will also be able to run a local XMPP server to use for the G Universe Server and the G Universe Client and have a database set

up. This makes it possible to perform all development related activities on a single host that does not even need to be connected to the internet.

Please note that not every step is discussed in detail. You should have some basic knowledge to be able to deal with your operating system. This is intentional, you should have to think yourself to get aquainted with the system, which will help you a lot for the actual development work itself that you will do. If you think something important is missing from these instructions or something is simply wrong, please contact the G System team.

The overall setup that is presented uses NetBSD together with pkgsrc. With this it is rather easy to install all required packages, you basically only need to know what you need. The advantage of pkgsrc is that it works on many different unix–like operating systems, not just NetBSD, so you could use pkgsrc for whatever operating system you're using. You can find details on supported platforms on the pkgsrc website – http://www.pkgsrc.org.

After installing all required packages we will take a look at tweaking the environment to suit your needs – environment variables, database configurations, importing the project into KDevelop and other things.

If possible you should have a spare computer to do the actual installation on. If that is not possible for you, partitioning the hard disk might serve your needs.

All system related configurations must be performed as super user (root) of course.

At the time of writing the available versions were in general NetBSD 3.0, xorg 6.9.0, PostgreSQL 8.0 and 8.1, Qt 4.1, Jabberd 2.0s10 and G System pre–0.6 (svn trunk).

The Operating System

The first thing to install is the operating system, NetBSD in this case. I'm not going to duplicate any documentation, so just take a look at the NetBSD documentation to get yourself a working installation: http://www.netbsd.org/guide/en/.

Installing pkgsrc

After you have successfully installed NetBSD (wasn't that easy after all? Just putting in the CD and walking through the multilinugal setup...), you should install pkgsrc. Have a look at the documentation on how to do this: http://www.netbsd.org/Documentation/pkgsrc/.

Currently it is recommended to use the latest CVS version of pkgsrc, only this includes the required Qt 4. As an alternative, you can manually install the latest Qt version yourself. Please note that at the time of writing, pkgsrc does not yet provide the PostgreSQL database plugin for Qt 4. You can also use the SQLite plugin, which is available. You could also download and install Qt yourself, this way you can install all database plugins you want, including PostgreSQL.

After installing pkgsrc, you should edit /etc/mk.conf to set some options, I have found the following useful:

```
ACCEPTABLE_LICENSES+=lame-license
PKG_OPTIONS.jabberd2=pam pgsql
PGSQL_THREAD_SAFETY=yes
X11_TYPE=xorg
```

Please note that this uses xorg as the X11 server, you might have other preferences. The lame−license is required for the lame mp3 encoder, which in turn is required for kdemultimedia, in case you want a full K Desktop Environment. Also note that we use the pgsql option for jabberd2, which is different from the default, which is mysql. But we will use pgsql for all our database needs (three databases in total).

Packages to install

With pkgsrc you can install a package just by typing *make install* in the directory of the desired package. This list is hopefully somewhat complete, please tell us if you think something is missing.

packages to install from pkgsrc:

- textproc/docbook−xsl
- textproc/libxslt
- devel/doxygen
- databases/postgresql81
- meta−pkgs/xorg
- meta−pkgs/kde3
- chat/jabberd2
- devel/kdevelop
- x11/qt4
- x11/qt4−pgsql (does not exist at the time of writing)
- devel/subversion−base

Some of these packages require you to install rc scripts. The install procedure will tell you to do so. Make sure you don't miss to do this. In particular this applies for the PostgreSQL and the Jabberd2 packages.

You will find all the required rc scripts in */usr/pkg/share/examples/rc.d/*. The ones you need to copy to */etc/rc.d/* are:

- pgsql
- kdm
- jabberd
- c2s
- sm
- s2s
- router
- resolver

Tweaking NetBSD

After installing all the packages, there are a few things you should do with your fresh installation to make NetBSD suit your needs.

X.org (X11) graphical environment configuration

The first thing to configure is your graphical environment − the X server. pkgsrc installs the xorg package into /usr/pkg/xorg by default. Usually we want to have the X11 installation in /usr/X11R6. but this directory is used by the NetBSD X11 server. We can just delete that and then make that a link to the xorg directory:

```
rm -rf /usr/X11R6
ln -s /usr/pkg/xorg /usr/X11R6
```

After that you can just run *xorgcfg* to create a configuration, save the configuration in */etc/X11/xorg.conf*.

After the file has been saved, edit it and look for *GLX*. You should enable (uncomment) the line that says *Load "glx"*. This is needed for OpenGL support in the X Server. If your system provides DRI, just enable that, too.

If *xorgcfg* does not work, just run *xorgconfig*.

Note: the xorg project is planning a next protocol version, so X11R6 might be replaced by X11R7 in the above instructions.

Another important thing is to have correct permissions for the communication directories for xorg, created on startup. For this put the following into */etc/rc.local*:

```
mkdir -p -m 1777 /tmp/.ICE-unix
mkdir -p -m 1777 /tmp/.X11-unix
```

Creating an NetBSD user

You should create an user account to use for the G System development, have a look at the *useradd* command. After creating the user, run *passwd <username>* to set a suitable password.

Running the K Desktop Environment (KDE)

By default NetBSD does not run any desktop environment at all – a shell is all it needs for many systems. Still, we want to run a graphical environment.

There are more possibilities:

- put *startkde* into *$HOME/.xinitrc*,
- run the K Display Manager (KDM) that gives you a graphical login screen on startup. This can either be done with the kdm rc script, or by using *rc.local*

To use the rc.local script, edit the file */etc/rc.local* and add the following lines at the end (but before the last echo):

```
echo "Starting the K Display Manager (kdm)"
/usr/pkg/bin/kdm
```

/etc/rc.local is basically a shell script that is run at boot time after the system rc scripts were run. So it is the last script run at startup. With all this in place, KDM will greet you on system startup. Choose the correct session type from the menu and log in as your user.

Environment variables

I found it useful to put various environment variables into my *$HOME/.profile* file.

A very useful variable is *LD_LIBRARY_PATH*. To avoid the need of installing the G System, this variable can be used to point to the location of the compiled libraries. Thus it is possible to start the compiled G System

binaries without the need to install the libraries into the system. So everything can stay in the development tree.

Other variables include variables used for Qt and path settings.

I found the following content in $HOME/.profile useful:

```
PATH=$HOME/bin:/bin:/sbin:/usr/bin:/usr/sbin:/usr/X11R6/bin:/usr/pkg/bin
PATH=${PATH}:/usr/pkg/sbin:/usr/games:/usr/local/bin:/usr/local/sbin

QTDIR=/usr/pkg/qt4
QMAKESPEC=netbsd-g++

PATH=$QTDIR/bin:$PATH

LD_LIBRARY_PATH=/home/raphael/devel/G/trunk/lib:/usr/X11R6/lib

export PATH QTDIR QMAKESPEC LD_LIBRARY_PATH

EDITOR=vi
export EDITOR
```

Please note that these settings are only useful for shells that evaluate $HOME/.profile. I would recommend you to use bash. You can find it in pkgsrc as well, if it's not already installed. Note, you can use *vipw* to set the login shell.

Additionally, the *LD_LIBRARY_PATH* variable expects the G System development sources to be in $HOME/devel/G.

Setting up the PostgreSQL database management system

PostgreSQL is a fully featured, robust ORDBMS (Object Relational Database Management System). It is the most advanced free DBMS available. Look at http://www.postgresql.org for details and documentation.

PostgreSQL is needed for the following databases:

- jabberd2 authentication
- G Universe Server database
- G Universe Client database

Preparing PostgreSQL

First, make sure you have copied the pgsql rc script to */etc/rc.d/pgsql*. If not, look above.

With this in place, add *pgsql=YES* to your */etc/rc.conf* file.

Then run the PostgreSQL server: */etc/rc.d/pgsql start*

Creating the jabberd2 database

```
# su pgsql
$ createuser jabberd2
$ createdb -O jabberd2 jabberd2
$ exit
```

```
# cd /usr/pkgsrc/chat/jabberd2
# make extract
# cd work/jabberd*/tools
# su pgsql
$ psql -U jabberd2 jabberd2
jabberd2=> \i db-setup.pgsql
jabberd2=> \q
$ exit
# cd /usr/pkgsrc/chat/jabberd2
# make clean
```

We will use this database later in the configuration for jabberd2.

Creating the G Universe databases

Before creating the databases themselves, you should create an PostgreSQL user with the same username as your unix username. Because PostgreSQL also identifies users by ther system user id. I recommend using the same user as you use for G System development. Then you should create two databases owned by that user, we will call them *gclient* and *gserver*. As usual, replace *<username>* with the actual username you intend to use.

```
# su pgsql
$ createuser <username>
$ createdb -O <username> gserver
$ createdb -O <username> gclient
$ exit
#
```

The created user does not have any password set, which is fine because PostgreSQL only allows connections from localhost by default.

We will use these databases later for running the G Universe Server and the G Universe Client on localhost (this computer).

Setting up the XMPP/Jabber server

jabberd2 is an XMPP server. XMPP is the protocol used by the G World Engine for network communication. Look at http://www.xmpp.org and http://www.jabber.org for details.

Please note that there are probably better XMPP servers out there, especially fully XMPP compliant servers like ejabberd are preferred. So using jabberd2 is just one possibility – jabberd2 is also integrated in pkgsrc, so it's easier to set up.

Configuring jabberd2 startup

For jabberd2 we need to slightly tweak the requirements for the rc scripts. In particular c2s and sm need as requirement the pgsql service, to make sure PostgreSQL is started before jabberd2.

To do so, add *pgsql* to the *REQUIRE:* settings of */etc/rc.d/c2s*. The complete header will look like this:

```
#!/bin/sh
#
# $NetBSD: c2s.sh,v 1.2 2004/06/26 11:21:46 abs Exp $
#
```

```
# PROVIDE: c2s
# REQUIRE: DAEMON pgsql
```

This will make sure that PostgreSQL is started before the c2s component of jabberd2.

Finally we need to actually enable jabberd2 in the rc configuration. Add the following lines to */etc/rc.conf*:

```
jabberd=YES
c2s=YES
sm=YES
s2s=YES
router=YES
resolver=YES
```

For this to work you must have the appropriate rc scripts in */etc/rc.d/*. Look at Packages to install to learn where you get the rc scripts from.

Configuring jabberd2

Configuration of the Jabber2 daemon is actually rather easy. The default configuration is just fine except that we need to use pgsql for authentication instead of mysql. This needs to be set in the c2s and sm configuration.

Go to the directory */usr/pkg/etc/jabberd/* and edit the file *c2s.xml*. Look for the section called *Authentication/registration database configuration* and within <authreg> set *pgsql* as the backend module to use. Do the same for the *s2s.xml* configuration file. The settings for the PostgreSQL database itself is already correct, as it uses the jabberd2 user and jabberd2 database name by default. The password has no effect as the jabberd2 user is not password protected.

Getting the G System source code

Before you proceed it is probably a good idea at this point to reboot your machine and see if the whole startup procedure works fine, especially with regards to the PostgreSQL database and Jabberd2. Another reason is to make sure that the environment variables are all set when you log in as user. If you know your system well enough you'll certainly be able to achieve all this faster without rebooting, of course.

Getting the sources and compiling is explained in detail in the chapter Source Management. It is recommended to check out the sources into the *$HOME/devel/G/trunk* directory.

Please also make sure that you are able to compile all of the G System source code. You should be able to perform this step with *./scripts/compile* from within the trunk directory of the sources. Please note that the 3.4 version of gcc will not work correctly, use a different version. NetBSD 3 uses gcc 3.3, so that works just fine.

And make sure that you do *not* install the G System into the system. That is, do not call *./scripts/makeinstall*.

Configuring the G Universe Server and the G Universe Client

Registering Jabber accounts

So far you cannot register a Jabber/XMPP account from within the G System applications. For this you need to use any of the available Jabber clients, like *Kopete* or *Psi*.

Make sure you register two accounts:

Accounts to register

- **G Universe Server account**
  - server: localhost
  - Jabber ID: gserver@localhost
  - password: abc
- **G Universe Client account**
  - server: localhost
  - Jabber ID: gclient@localhost
  - password: abc

G Universe configurations

Both the client and the server use an xml configuration file for network and database settings. After you have compiled the G System you can use *./bin/god* to create and edit any of these configurations.

Put the server configuration into */usr/local/etc/gweconfig.xml* and the client configuration into *$HOME/.guniverseclient.xml*.

Within the configurations, please replace *gdev* with the actual username that you use. For better clarity the desc attributes have been removed.

The content of /usr/local/etc/gweconfig.xml looks like this:

```
<gwe_controller value="advanced" >
<db_driver value="QPSQL7" />
<db_host value="" />
<db_name value="gserver" />
<db_password value="" />
<db_port value="" />
<db_username value="gdev" />
<init_network value="yes" />
<master_server_jid value="" />
<xmpp_jid value="gserver@localhost" />
<xmpp_password value="abc" />
</gwe_controller>
```

The content of *$HOME/.guniverseclient.xml* looks like this:

```
<gwe_controller value="advanced" >
<db_driver value="QPSQL7" />
<db_host value="" />
<db_name value="gclient" />
<db_password value="" />
<db_port value="" />
<db_username value="gdev" />
<init_network value="yes" />
<master_server_jid value="gserver@localhost" />
<xmpp_jid value="gclient@localhost" />
<xmpp_password value="abc" />
</gwe_controller>
```

Running the G Universe Server and the G Universe Client

We are now ready to run the G Universe Server and the G Universe Client on localhost, connected to the local PostgreSQL database and communicate through the local jabber/XMPP server.

First start the server: *./bin/guniverse* and then the client: *./bin/guniverseclient*.

Building the documentation

If you have the docbook packages and libxslt installed, you can also build the documentation. To do so, run *./scripts/makedocs* from the base directory of the G System sources. You can set various parametres for the script, depending on what you actually want to build, the output of running the script without parametres will tell you what options you have.

Using KDevelop

If you want to use KDevelop you can simply do so by running KDevelop and choosing *Import Project...* from the *Project* menu. Then select the *$HOME/devel/G/trunk* directory, set *GSystem* as project name and choose *Qt C++ Application (QMake based)* as project type.

You can make your life easier if you enable code completion, also for the Qt library. Go to Project settings and add Qt to code completion in the C++ settings. Note that KDevelop won't provide you with code completion for the x3dtoolkit, the qglviewer and the xmpp libraries, as these are *included* in the project files – KDevelop does not detect this.

When using KDevelop you should be aware of the following guidelines:

- Do *not* edit qmake project files from within KDevelop
- Never add or remove files from the project from within KDevelop, which implies changes to the qmake project files. If you have to change qmake project files after you have added or removed classes, close KDevelop, edit the project file and restart KDevelop.

Chapter 3. Project and Feature Plan, Raodmap

Table of Contents

Roadmap
Featurelist
    Introduction
    G Core System (GCS)
    G World Engine (GWE)
    G Client Engine (GCE)
    G Basic Elements (GBE)
    G Universe: The universe created with the G System
TODO list
    Introduction
    To be done during 0.6 release cycle
    JUNIOR JOBS
    General things to be done, HIGH PRIORITY

Other general things to be done
Documentation
Source Code
OPEN ISSUES

Roadmap

Further releases are planned with focus on the following features:

Features for G System 0.6.0

- *Targeted release date: 2006*
- **Already implemented**
    - Improved rendering capabilities for complex forms (done, uses X3D)
    - Migrating from KDE/Qt 3 to KDE/Qt 4 (done, KDE dependency temporarily removed)
    - Further client improvements for improved interaction (action interface)
    - Information Interface
    - Action Interface
    - Standalone Network implementation
- **Still to be done for 0.6.0**
    - Testing
    - Some minor beautifications
    - Live CD
    - Packaging
- **Planned features for 0.6.1**
    - Persistent world content and user elements
    - Next step network capabilities (hierarchical servers)
- **Planned features for 0.6.2**
    - Agent Plugin Architecture for the GWE
    - Rendering Plugin Architecture for the GCE
    - Scriptable Agents with Ruby and/or SPL or Kross
- **Planned features for 0.7**
    - Webinterface
    - Infrastructure for automated downloading and installing of patches
    - Infrastructure for automated distribution of agent and rendering plugins (both Qt–native and scripts)
    - Physics Agent
    - Proper User management (registration, accounts, passwords)
    - Realistic world content for G Universe (solar systems, planets, landscapes, ...)

Featurelist

Introduction

This feature list describes the desired functionality for the G System that should be implemented for the 1.0 release. The G System is made up of various parts, each part will be documented independently. Only source code related features will be discussed. Nevertheless documentation is just as important as source code so we work hard to provide good documentation for the whole system. Documentation and other aspects are included in the raodmap.

Please note that many already implemented features are still subject to further improvements.

G Core System (GCS)

State: stable

Abstract

The purpose of the G Core System is to provide an abstraction of world content of any kind. It should also provide reasonable interfaces for element handling by the GWE, agent specialisation as well as generic data extension possibilities.

Feature list

Implemented features:

- General definition of element structure
- Object (holds data, also energy and form)
- Energy (element characteristic)
- Form (element position and geometry)
- Agent interface (agents define element behaviour)
- Influence
- Element interface (signals and slots for element management by the GWE)
- General purpose element wide XML data storage that is shared among all agents of one element
- XML specification of element representation for serialization
- Proper influence energy amount management for received influences

Current Activities

The GCS is fully implemented, currently no further features planned.

G World Engine (GWE)

State: Element management done, XML based data management including network already usable.

Abstract

The GWE is responsible for Element management, it should provide all means to allow execution of elements including network management and persistence. The GWE also provides the communication infrastructure for elements.

Feature list

Implemented features:

- Abstract world engine interface (factory, controller, ...)
- Database interface for persistent element storage
- Local influence handling
- Element storage (database)
- Serialisation of elements (for databases, network transportation)
- Network layer (based on direct TCP/IP connections; XMPP currently disabled)
- Local clients directly using the GWE library are possible

Missing features (most of them are currently being worked on!):

- GCS::GAgent Plugin infrastructure (see GWE docs)
- Network protocol for clients
- Network protocol improvements for multi–hierarchy–levels servers
- Advanced network protocol capabilities like moving elements, update subscriptions
- GWE Modules (see old docs from first svn repository)

Current Activities

Further improvements in the network layer to allow multi–hierarchy server infrastructure. Persistent content, moving element responsibility between servers. Implementing element subscriptions. Agent plugin architecture is important and needs to be implemented soon.

G Client Engine (GCE)

State: usable, improvements still to be done in user–element interaction

Abstract

The GCE is responsible for defining a standard for the client – server interface to allow anyone to write individual client implementations.

The G System provides a standard client that includes libraries for 3D visualisation and client – server communication. These libraries can be used to develop custom clients.

Feature list

Implemented features:

- OpenGL interface
- Camera
- Local (library based) GWE interface
- Communication between users (XMPP messages)
- Rendering of static X3D models
- Interface for showing information about user controlled elements
- Interface for user controlable agents (action interface)

Missing features:

- Network capable GWE interface and protocol
- Optimization for user input
- GUI to enable the user to manage customized actions (customized action buttons)
- Infrastructure for rendering plugins

Current Activities

Interface beautification and usability, action interface.

G Basic Elements (GBE)

State: World generation and a few basic agents done.

Abstract

The G Basic Elements Library implements common agents, forms, influences which provides basic element functionality. All element functionality that is required for a virtual universe to function is provided in the GBE.

The GBE is a collection of agents.

Feature list

Implemented features:

- Very basic agents for movement
- Very basic agents for influencing
- A very basic agent for form manipulation according to an element's energy
- Framework for deterministic random universe creation

Missing features:

- Agents for physics calculations, like collisions, ...
- More sophisticated user controlable agents
- Agents for evolution
- Scriptable agents (with SPL, Ruby, ...)
- Agents to interface external systems

Current Activities

Scriptable Agents. Agents for working with Kolab groupware.

G Universe: The universe created with the G System

State: since version 0.5 of the G System it's possible to run a multi–user virtual universe.

Abstract

Packs the G System, which is a framework, into an application.

Feature list

Implemented features:

- Server
- Client
- GOD, a basic configuration utility
- Translatable strings in GUI

Missing features:

- Realistic world content (landscapes, rivers, solar systems, ...)
- Webinterface: GWI (G Web–Interface)
- ...

Current Activities

The client is being improved to allow more interaction with the simulation as a whole and to show more information about connected elements, energy and so on. See also the GCE feature list.

Work on the GWI has started, it already provides functionality to query a guniverse server for information.

Basically the G Universe grows together with the G System.

TODO list

Introduction

This section includes a summary of the things that need to be done in the near future, if you would like to work on something, please discuss it on the g–devel mailing list. Look at the homepage for joining the mailing list.

To be done during 0.6 release cycle

The 0.6.x release series focuses on making the G System usable for real–world applications.

BUGS

- There have been some quirks with jabberd 1.4.4 servers which (wrongly) advocate XMPP 1.0 compliance. This should be handled in GXmppNetwork (if at all). This will be fixed in one of two ways: with moving to the current Qt4 based libiris version or using fully compliant jabber servers, like ejabberd. Please note that by now the Qt4 based iris library is not yet available, thus XMPP is disabled by default in the G System.
- MySQL (or the qt–mysql plugin) can't manage elementdata. look for MySQL todos in GStorage. DO NOT USE MYSQL DATABASES FOR ANY GUNIVERSE SERVER DEALING WITH PERSISTENT DATA (not yet relevant since nothing is yet persistently stored accross restarts).

- XMPP presence subscriptions are not working properly due to lack of XMPP server support (jabberd 1.4.3), look for HACKs in GXmppNetwork. Again, move to ejabberd, or similar XMPP compliant servers.
- Update Subscriptions are not yet in use. Look for HACKs in GXmlDataController and todos in GStorage.

DESIGN

- Add a signal for connection changed in the element and use it in the GElementInfoWidget of the client.
- **Proper load management**
  - Introduce sorted LastUpdated list in GWE in order to close elements that do not show activity for a long time
  - Additionally provide a `void mayPark();` method in `GCS::GElement` and `GCS::GAgent` that checks all agents whether the element may be closed. In `GCS::GAgent` this should simply default to true.

    Be careful, this might be a bad idea as from time to time elements simply must be closed for example when the application itself closes. But the LastUpdated list is a good idea.

    Improvement: do implement this mayPark infrastructure – to use it when going through the last updated list and looking for elements to be closed. Simply close those that return true. In situations where elements must be closed don't evaluate mayPark but simply do it.
- Jabber Servers with pre XMPP 1.0 protocol do not handle presence subscriptions. We should definitely *just* allow XMPP 1.0, also put up a server that clients can use as Jabber Server (universe.g–system.at). Currently it works with both :) but we should limit to XMPP 1.0 in due time.
- *User elements:*

  Getting an user element for a client should not be the responsibility of the client but rather that of the server to provide a pool of available elements that can be requested/created by a new client. Technically this would mean to transport element responsibility (see protocol requirements). Once an user has chosen an element this information is kept since unregistering and presence are two distinct things (see protocol requirements). As soon as the user is present again, that very element is transported to the user again. It should also be possible for the user to query for those elements that belong to the user (again, see protocol requirements).

  I would suggest to implement this in 0.6, the G Universe client should start out without any client and be able to select one from the given world (raphael).
- *Universe creation and initialization:* Currently the server creates one big big universe object on startup. This needs to change since with a DBMS backend we get persistent storage at no cost. Then injecting the first element should also not be the responsibility of the server application but rather that of an administration tool like GOD. GOD should connect to the server through XMPP (needed at some point anyway) and be able to inject elements as well as fetch information about elements. Please note that with the dawning of the G Web–Interface such administrative things can also be done through the GWI.
-

  Make persistent content possible. This includes:
  - Improve the protocol to run from existing database content (utilize the gweserver hierarchy, last column of the gweserver DB table)

    - Improve the client to fetch its element from the server.
    - Possibly improve the client to create its persistent client element.
- Forward changed signals also to the agents of the element. Probably provide predefined slots that can be reimplemented if desired. Connect these slots in GCS::GElement::addAgent()

GENERAL

- TEST THE NETWORK!
- *Installer improvements:* automatically check for patches in patches subdirectory and apply them before compiling – user doesn't need to know how to handle patches. This is generally postponed to the 0.7 release.

CLIENT

- Make it possible to slow down rendering as desired (the msleep(x) variable between rendering updates). Create a slider for this.
- Add global information interface It should show things like current parent – like in which solar system the user is in and maybe server status, whatever. Element independent information. This should be placed at the top (a long, thin widget).
- Add element information interface to display information about the element and/or the universe. Similar to the existing communication widget. This should be placed at the left border.

  Information should include:

    - Current size
    - Energy

PROTOCOL REQUIREMENTS:

- *Move element responsibility* (this is certainly a non−trivial thing to do really right, but *basic* support for it should be done already). This is required to be able to select a client element from a pool of existing elements.

  This can be done with the "owner" attribute of a GElement message. The original owner sends the element with the updated owner attribute to the new owner who recognizes that the incoming element needs to be taken over. There should be a confirmation message that the element has been accepted or denied and a timeout which defaults to denied (don't lose data!).

  Different approach: subscriptions need to be transported as well. Thus I would suggest sth like:

```
<moveelement>
  <GElement/>
  <subscriptiondata/>
</moveelement>
The receiving server should send some kind of acknowledge message, like:
<elementtaken id="523"/>
```

  id should be the element id that has been moved from one server to the other.

  Likewise there can be a <elementrejected id="523"/>

See GXmlDataController::receiveData() for protocol stuff!
- The user should be able to query for elements belonging to the user (see design todos)

JUNIOR JOBS

These jobs are not necessarily fast todo, but should be doable without solid knowledge of the system. If you want to work on one of these jobs, you can discuss this on the mailing list.

- *Write a coding style guide based on the G Core System (src/core).* Coordinate this with the mailing list. Doing this would be appreciated and should be done before anything else as it helps newbies to write clean code.

General things to be done, HIGH PRIORITY

- *Add a tutorial for advanced agent creation*

  This tutorial should probably create a solar system with agents orbiting the center – a GOrbitingAgent, this would also be required to create planets in a solar system. See below for todos in the documentation. This tutorial should go into the chapter on using the G System.
- Implement test cases for the whole API. Qt 4 includes an unit testing framework.
- Also update the iris XMPP library when porting to Qt4. The new iris library was already in *src/worldengine/iris* but it is not yet functional with Qt4 (does not even compile, thus removed). Look at http://delta/affinix.com/iris/ for details. It would be useful to investigate the current state of iris (contact the PSI developers!) and see if it can be reintegrated into the G System, maybe as an external library similar to the X3DToolKit.

Other general things to be done

- Most of the time there are things that need to be done, ask on the g–devel mailing list.
- Make progress towards the ultimate goal of the G Universe, look at HACKING and the docs in doc/gdocs. We've also written a vision paper in October 2006 which gives an outline on the vision of the G Universe. It is currently not included in the G System documentation, but might end up in the philocorner. For now, contact Raphael to get a copy.

Documentation

- Always improve the docs, in particular the philocorner is of high interest. It represents the thoughts behind the G System and the G Universe.
- Make documentation i18n capable.
- Provide much more documentation for the users of the G System framework (they are developers). See the chapter on using the G System. The tutorial needs the be replaced by a newer version, it is very misleading by now (originally written for the 0.3 version of the G System).

Source Code

- Make all strings i18n capable. Use the Qt translation infrastructure for this, which basically means to use the `void tr();` method. This is already done to a large extend, look through the application and complete this task.
- The network layer is now implemented, see GXmlNetwork and GXmppNetwork and

GXmlDirectNetwork for details. The information below is kept for reference.

This layer goes into the GWE library.

Possible implementations:

- What about grid systems?
- What about GLED?
- What about libMAGE?
- Other suggestions are welcome.
- *Latest suggestion:* Jabber, it also makes it potentially possible to take primitive actions from ANY(?) Jabber client. Also Jabber is a distributed network(!). This solution is currently the preferred one.
- *One more latest suggestion: GLOBUS, a grid system.* This means that all network related components should be implemented as grid services, take a look at GLOBUS related papers, they include pretty much information about grid architecture.
- With regards to changes in element data (`GCS::GElementData`): only send partial data of elementdata (that is, if /dynamics changed, only send /dynamics, not the whole elementdata... saves bandwidth).

- *Refactor GWE::GXmlDataController::receiveData*

  Each message should be interpreted by a separate message interpreter object. Such objects can be added to the data controller externally. Interpreter classes should implement a specified interface by the GXmlDataController class. The objects can be stored in a map where the key is the actual message tag the object can interpret.

  Maybe use threads for message interpreters to not block the network for too long. Be careful here though. In particular you should import the QDomNodes that were received in the message to a separate document before forwarding it to the interpreter as a thread.
- Use multithreading for GWE more extensively to not block too long. This can now be done, the G System is ported to Qt4.
- Now that we use Qt4, the Agent plugin infrastructure can be put in place by using Qt Low–Level Plugins – http://doc.trolltech.com/4.1/plugins–howto.html

  This also works with C++ agents, not just interpreted (scripted) agents.

  This basically affects the GWE.
- Network code for the client–server interface (build client directly on GWE like it is done now?)
- Have a look at the todo list of the API Reference for more things.
- You'll also find lots of @todo items in the sourcecode itself (grep for it).
- **Investigate possibilities for using mathematical expressions for element geometry**

  - This would make LOD (level of detail) easier as well.
  - The muParser might be useful for this purpose.
  - This is mainly a rendering plugin for the GCE, the agents just need to put the desired expressions into the element data.

OPEN ISSUES

- Time/Space scaling, what about "fast" elements? (tunneling effect)

- Collision detection and response Difficult if every element has to implement it itself, though this would be natural. A different approach would be to give the parent element better control over its children, but this is unnatural. There is a thread in raphael's blog about a suggestion. This suggestion is generally a bad idea.
- Energy level and sigma – all elements that somehow relate to each other must share the same energy level range in order to be able to respond to influences that affect physical matter as such.
- Time synchronization between servers, especially important to add a timestamp on influences, etc. so the receiver exactly knows when an influence was sent. Note: if the receiver looks up data from the sender the fetched data might be of a different time point than when the influence was sent (different time–difference).
- Events and listeners between elements (formChanged(), energyChanged(),...). Does this comply with the influence mechanism? Especially if elements don't have the same parent (influences usually don't go through that).

Chapter 4. Understanding fundamental G System Technologies

Table of Contents

Introduction
Qt
Subversion
XMPP – Extensible Messaging and Presence Protocol

Introduction

The G System does not invent everything on its own. Existing technology is reused as much as possible. This section briefly describes the used technologies that any G System developer needs to know. There are also many links included to further reading for more detailed information.

Qt

Qt is *the* fundamental technology that the G System is based on. Qt is a cross plattform C++ toolkit with many advanced features, bringing C++ development to an even higher level.

You need to know Qt for both, using the G System Framework to build your own applications as well as for developing the G System itself.

Qt is developed by Trolltech and freely available for all plattforms for use with open source projects. Trolltech also provides the source code for Qt. KDE, a powerful desktop environment, is also based on Qt.

You can learn about Qt on http://doc.trolltech.com. The currently used version is 4.1 since the G System 0.6 release. The old 0.5 version of the G System uses version 3.3 of Qt. It is recommended to learn about Qt4 and not Qt3 of course, unless you have to deal with older G System versions.

It is highly recommended to work through the provided tutorials.

There is an independent Qt tutorial available which is also a very good resource for learning Qt.

Qt is based on C++, this means you should also be familiar with C++ itself and should be aware of the C++ syntax and general object orientated programming (OOP) concepts. There is a good freely available online–book available, Thinking in C++ (TICPP): http://www.briceg.com/ticpp/one/, you only need to read book one, the second covers many aspects that you do not need for the G System.

Subversion

Subversion is a revision control system. It supersedes CVS, providing a better design and thus fixing many flaws of CVS.

There is a very detailed online book about subversion at http://svnbook.red–bean.com. It is constantly updated along with subversion itself and is *the* handbook. There are also printed versions of it available that you can bye in any (reasonable) bookstore – or at amazon.

XMPP – Extensible Messaging and Presence Protocol

XMPP is the network protocol used by the G System. You can find more information and further links, including the specifications at http://www.xmpp.org.

Chapter 5. Source Management

Table of Contents

The Subversion Repository
    What is Subversion?
    The G Source Code Repository
Compiling and Installing
Documentation
Website

The Subversion Repository

What is Subversion?

All project data, including the website, is kept in a subversion repository. Subversion is a revision control system, which is an essential tool for any software project, or even any kind of data. If you are unfamiliar with subversion, or even with revision control systems in general you should take a look at http://subversion.tigris.org. You can also get subversion from this site for any platform. All required information can be found in the subversion book, which can be found at http://svnbook.red–bean.com.

The actual place where subversion stores all the data is called a *repository*.

The G Source Code Repository

Repository Layout

The URL is *svn://svn.g–system.at/G*. It is further divided into these subdirectories:

- trunk
- branches
- tags

The current development source code is located in *trunk*, stable versions are in *branches* and *tags*, which are both further divided into individual version directories. You usually don't need to access these to do normal development, but it might be important for backports and bugfixes.

All documentation is included within this repository.

The website is located in a separate repository, located at *svn://svn.g−system.at/G−www*, which also includes the *trunk* and *tags* subdirectories. The www.g−system.at website is automatically extracted from trunk. Thiw is implemented with a post−commit hook of the subversion repository.

Working with the Repository

Here are some basic instructions on using the repository, for details please consult the subversion book.

- *Checking out the latest sources*

  svn co svn://svn.g−system.at/G/trunk

  This creates a local directory *./trunk*, which includes a copy of the current sources.
- *Updating a checkout*

  svn up

  This command must be run within the checked out directory. It updates the local copy to the latest version.
- *Creating a patch with your local changes*

  svn diff > /tmp/my_local_changes.diff

  This command must be run within the checked out directory. svn diff creates an unified diff that can be used as a patch. You can do local changes and use this command to create a patch and send it to the developers.

There are of course many more things you can do with subversion, please consult the subversion book for this.

Compiling and Installing

To avoid duplication of information you are advised to consult the *doc/INSTALL* file for details on the installation process.

A simple way of installing the G System is to use the graphical and interactive installer located in the base directory.

Documentation

All the documentation is located in the *doc* directory of the sources. Extended documentation written in docbook format is in *doc/gdocs*. You can use the *./scripts/makedocs* script to build all the docbook based documentation, including the API reference, which is extracted from the *src* directory.

You can usually find prebuilt documentation in HTML and PDF format of the latest stable release on the G System website.

Website

All the website is contained in a separate subversion repository. Since a subversion repository is not a regular filesystem at all (it is rather a database), the latest version needs to be extracted from the repository. This is done directly on the server by a hook script which is executed whenever the *G−www* repository is updated and copied into the webserver directory.

The website sources are not readable to the public, so you cannot do anonymous checkouts. If you have suggestions or ideas for improvements for the website please contact the developers.

Chapter 6. Architecture

Table of Contents

Overview
- *Introduction*
- *Parts of the G System*
- *Evolution and the G System − putting it together*

The G Core System (GCS) − defining the element structure
- *Introduction*
- *Elements*
- *Objects, element data*
- *Agents*
- *Interaction between elements − influence handling*
- *Hierarchical world structuring*
- *Some notes on the GCS*

The G World Engine (GWE) − connecting elements
- *The Purpose of the GWE*
- *GWE Design*
- *Network Infrastructure*
- *Elements and the GWE*
- *Clients and GWE Servers*

GWE Sepcifications and Protocol
- *GWE as a GCS Container*
- *GWE network communication behaviour*
- *XML Constructs for GWE Communication*

G Basic Elements (GBE) − bringing the system to life
- *Deterministic Random World Generation*
- *Element hierarchy management*

The G Client Engine (GCE) − making it visible

G Universe Server
G Universe Client
GOD
Agent Plugin Architecture
Overview
Plugin management in the GWE
Developing an agent plugin

Overview

Introduction

When reading about the architecture of the G System, you will probably find many uncommon and new aspects of software development. This is mostly because of the quite uncommon purpose this project works on. Nevertheless, we feel that the G System fills a gap in both social and scientific areas that require some attention. So the purpose of the technical implementation is to model life (or evolution, both are the same), and is thus not a means by itself. This can result in a somewhat uncommon but highly fascinating software design. Note that this is the purpose of the G Universe, you are free to model any kind of simulation with the G System Framework.

Parts of the G System

First, we should take a look at what parts constitute the whole system. The G System is modularised to allow for easy and flexible application development. Here is a brief overview.

- The G Core System (GCS) defines the basic structure of the world. This includes the representation of objects in a world (called "elements") and the structuring of the world itself with those elements.
- The G Basic Elements (GBE) already provides basic parts for element creation, allowing for faster application development.
- The G World Engine (GWE) is responsible for element management. This means communication between elements, network transparency and database connectivity.
- The G Client Engine (GCE) contains all things that have to do with the user interface.

There are also some individual applications that build on the G System framework and are part of the project:

- The G Demo, which serves as a testing framework and for showing the basic functionality of the system.
- The G Universe is the virtual world implementing all aspects and concepts of the G System in a living application. It is a WAN–distributable, highly scalable and flexible client–server architecture, consisting of the G Universe Client and the G Universe Server applications that utilize the G System Framework The G Options Dialog (GOD) application is a configuration and management utility for the G Universe.

Evolution and the G System – putting it together

The G System was designed to represent evolution. As such it has to retain incredible flexibility to represent almost everything in a consistent way. The high level of abstraction which is reached in the G Core System provides this flexibility and still allows for consistent rule systems. Energy represents matter, form represents form, agents represent the interactions that occur between the elements of the system. The hierarchical

structure ultimately leads to the conclusion that every element is after all part of the universe and is thus in some way one with and/or connected to all existing elements. This is only a very basic interpretation of the Core System but is valid in general.

It is a long way from a view of life/evolution to creating software from it. The G System tries to provide an *as perfect as possible* framework for this modelling work. It does not dictate how the modelling of evolution should be done on a detailed level. Nor does it dictate how any particular simulation should be implemented. Implementation is thus left to the philosophical thinking and scientific modelling capabilities of the involved people. If you *feel* the need to take part in such an effort to find/create a simulation of an issue so fundamental to humankind, you probably should get involved in this project.

The G Core System (GCS) – defining the element structure

Introduction

The central library is the GCS (G Core System). It defines the basic structure of world content. Every such piece of world content is called an "element", represented by the class `GCS::GElement`. This class uses other classes like `GCS::GObject` to store information about the element.

Elements

Basically an element consists of three things:

- Energy,
- Form,
- Any number of agents.

Energy gives an element a certain characteristic and is fully defined in the core system as `GCS::GEnergy`. Energy can take an important role for agent behaviour as some agents behave differently for different energies.

Form defines the element's geometry, the basic form attributes are defined in the GCS as `GCS::GForm`. The `GCS::GElementData` object optionally contains a setting for a 3D datafile, usually a 3D modell in the X3D file format.

Objects, element data

The `GCS::GObject` class holds all data of an element. This includes the energy, the form, the ID of the element, the IDs of children and the parent ID as well as the ID of the element this element is connected to. There is also a generic data container which can hold any desired additional data of the element. This data is stored in an XML structure. All agents have access to this data and should use it to store ALL persistent data. This data gets also automatically transported by the GWE through the network and into the database.

Connected Elements

The `GCS::GObject` class also stores the ID of the connected element. Every element usually has an element it is particularly connected with. This is always the element of interest and towards which the actions of this element are directed.

The concept of connected elements is quite fundamental to the G System. Everything in the world is somehow connected with each other. This idea is represented by the connection ID of the element.

Agents

Agents perform all of the behaviours of elements, which includes all laws that the element is affected by. No agents are defined in the GCS itself, it only provides the framework in `GCS::GAgent` to allow easy creation of new agents. Have a look at the basic elements library for some examples.

What the agent exactly does is completely up to the agent designer. Every agent is executed as a separate thread. Additionally the agent has read and write access to all data of its own element and read only access to other elements for which the IDs are known. When working with element data the agent designer has to keep in mind that there can be more active agents in the element which are also working with element data. For proper synchronisation most element data can be locked. It is highly recommended to use this locking mechanism when developing agents. To avoid deadlocks the order of locking is always:

- Energy
- Form
- ElementData
- Object

Please keep in mind that all data that should be persistent must be stored stored in the `GCS::GElementData` object that is contained in the element's `GCS::GObject`. The `GCS::GElementData` class already provides methods to work with element data. These methods have the prefix *xml*.

Agents basically have two responsibilities:

- creating influences
- responding to influences

The response mechanism is implemented by reimplementing a virtual receive method of the `GCS::GAgent` class. The parameter of this method is the influence itself.

`GCS::GAgent` inherits `QThread` which means that the agent itself is a thread. In its execution the agent can work with and process element data and can radiate influences or send influences to specific destinations (often the connected element). This is where most of the actual logic of the agent is usually defined.

An element can globally be started or parked, which means that all agent threads of that element get started or parked.

See the documentation of the `GCS::GAgent` class for more details.

Interaction between elements – influence handling

An influence is represented by the `GCS::GElementInfluence` class. Agents can send out influences, either by radiation of influence or by influencing a definite destination element (this is where the ID of the "connected" element comes in). As soon as the agent has created such an influence, the agent can send it out and the GWE is then responsible for bringing the influence to the destination(s). When an influence is sent out, some other element will receive an influence. A virtual method of every agent of every receiver element

is called and the influence is the parameter. Using this method the agent can define the reactions to the influence. The reaction can either be a simple modification of element attributes – or even just internal agent attributes – or more complex things that are handled in the agent's thread (instead of handling the reaction directly in this influence receiving method).

The `GCS::GElementInfluence` class is independent of an `GCS::GElement` instance, but is used by elements for interaction. By limiting interaction to one general way, it is possible to create general rule systems that apply to all kinds of interaction.

The only attribute that is transported with an influence itself is energy. In early versions of the G System it was common to subclass `GCS::GElementInfluence` to provide additional information to the receiving agent. But this brings a close dependency between sender and receiver of the influence. Starting with version 0.5 agents are able to access data of remote elements (for reading only) and thus it is not necessary to transport any additional information with the influence itself. If for example an receiving element needs to react to the sender's position then the agent would simply fetch the additional information through the `GCS::GWorldData` interface.

Influences always carry energy from the source element with them. This means that an element that sends out an influence puts a certain amount of its own energy into that influence, which can be thought of as the strength of the influence. If more than one element receive the influence, the GWE automatically distributes the energy among all receivers. Every receiving element absorbs the received energy amount.

The kind of energy that is received with an influence also determines whether the influence is at all recognised. Only if the energies are somewhat alike the influence is received at all. Further improvements on the GWE could result in the GWE making decisions about how much energy is received by what element.

Purpose of influencing

Agent behaviour depends on the energy of the element. If the energy changes then the behaviour of the agent can change as well. This depends on the design and implementation of the agent.

Sending or radiating influences can thus be a means to change the behaviour of other elements. If the rules the other element adhers to are known, then it is possible to control the behaviour of other elements. This is just the way reality works! Everything acts according to its energy, and things can only be modified by influence. Please note that this is a quite radical view of reality and there are many different ways to look at it!! The G System just tries to build a model–able simulation framework of reality and has also more to offer than just this very point of view. In particular you should have a look at the Philocorner. A lot of thoughts are collected there about philosophical aspects of life and evolution.

Hierarchical world structuring

The element data of IDs proposes a hierarchical world structure, and in fact this is what is used. Every element is in itself a complete unit to the outside and can have a very complex subhierarchy – solar systems are good examples. First, we have the whole solar system that can be seen as a complete unit ("one element") to the rest of the galaxy. This element has some child elements, which are included in itself (in terms of position). These children are the planets, asteroids, large space stations and space ships and so on. Every such element again holds a complex subhierarchy – continents – oceans. Continents hold landscapes and cities. They hold either houses, individual beings or whatever. Individual beings are again a very complex entity – which is probably a topic for our Philosophical Corner (philocorner).

Some notes on the GCS

Please note that an element does not have any direct reference (or pointer) to other elements, only their ID. Thus it is not necessary to have the other elements in the same system process – they can even be on a remote machine connected through the network. Large applications probably even should be distributed hierarchically among a network. Reflecting the element hierarchy of the virtual world in the hierarchy of the computer systems used is often a good idea. See the chapter about GWE for further details.

The G World Engine (GWE) – connecting elements

The Purpose of the GWE

The G System is designed to simulate huge virtual realities. This can include many solar systems and planets, down to small details such as houses, environments, and even individual plants. All these things are not static. This means everything can change, humans (often controlled by real human users) move around, interact with other elements in their environment, ...

To allow for such a large scale simulation, some infrastructure needs to provide the computer processing power, the storage, and the communication. This infrastructure is provided by the GWE – the G World Engine, it is the container framework for all elements.

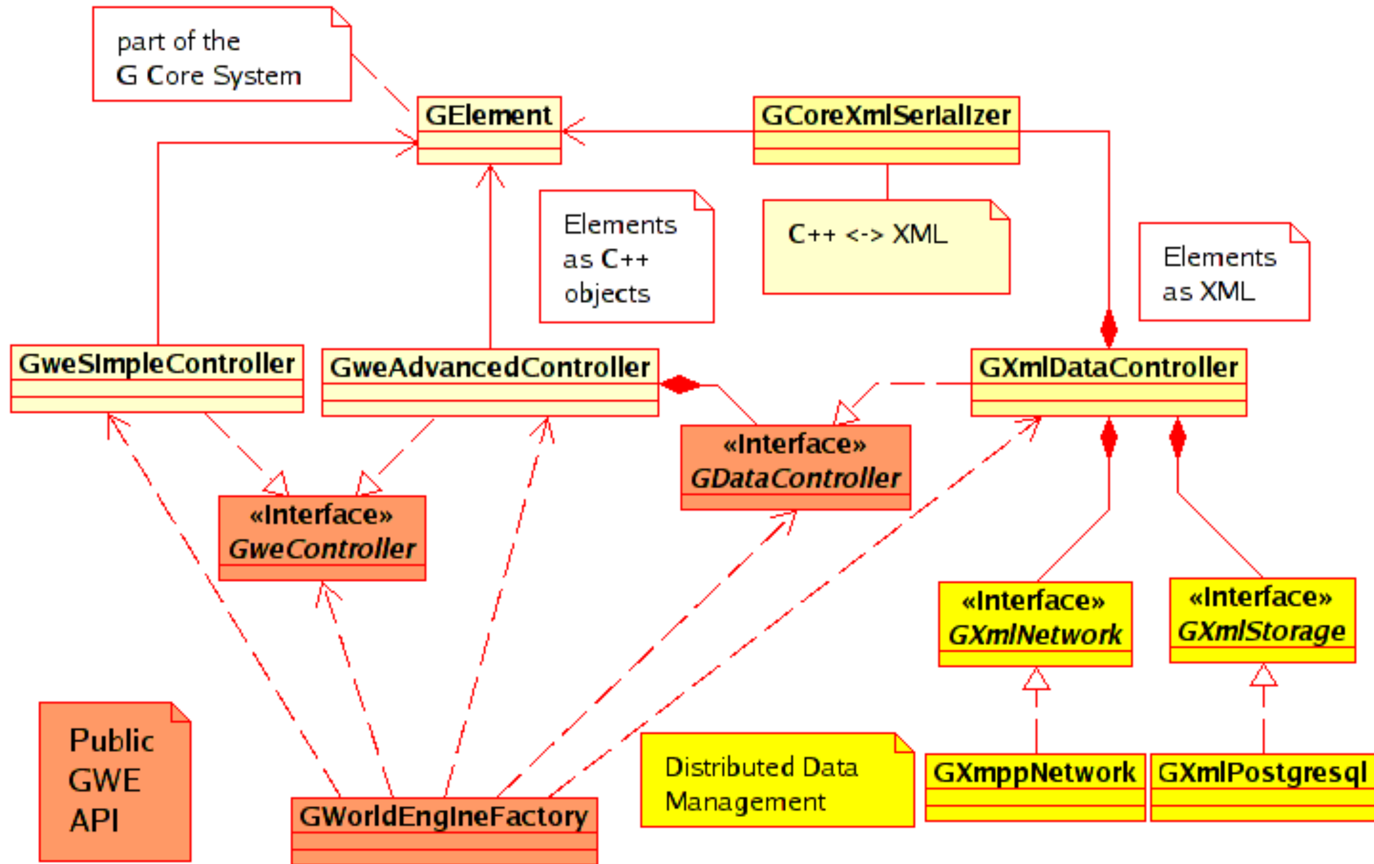
Elements simply operate. To them, the complete simulated universe is just a huge (hierarchical) collection of elements. They are not concerned with the physical location of other elements when communicating. Providing this seamless network transparency of a distributed server infrastructure is the purpose of the GWE.

GWE Design

As hinted in the previous section, the GWE is a distributed server infrastructure, or in other words, an element container and thus the framework the elements rely on for their operation.

The following UML diagram should give an idea of the GWE structure.

G World Engine Architecture



Component Interfaces

To provide a flexible implementation a general interface for each functionality is defined. This currently includes the `GweController` which is responsible for element execution management and logic, and the `GDataController` which is responsible for persistent and network transparent data management and GWE Server network communication.

The abstract interfaces, including the GWE Factory class, are available as public API, thus enabling application developers to control the GWE behaviour without being dependent on the actual implementation classes being used by the different components (for example whether a simple or advanced GWE Controller is used) because the implementation is not part of the API. The Factory class is used to bootstrap a GWE Server which uses the differen implementation classes.

Distributed Data Management

An important part is the complete abstraction of physical data location. This is accomplished by a data controller class that manages all data and data transfer. This is also where the other GWE Servers in the network are known and intelligent load and data balancing is done according to the logical hierarchical structure of the virtual universe.

Currently there is one XML based data controller available. This controller uses a separate serialisation class to convert between C++ objects and XML representation of elements. The C++ objects are required for execution and the XML representation is used for sending element data across the network and managing

database content.

Element Execution Management

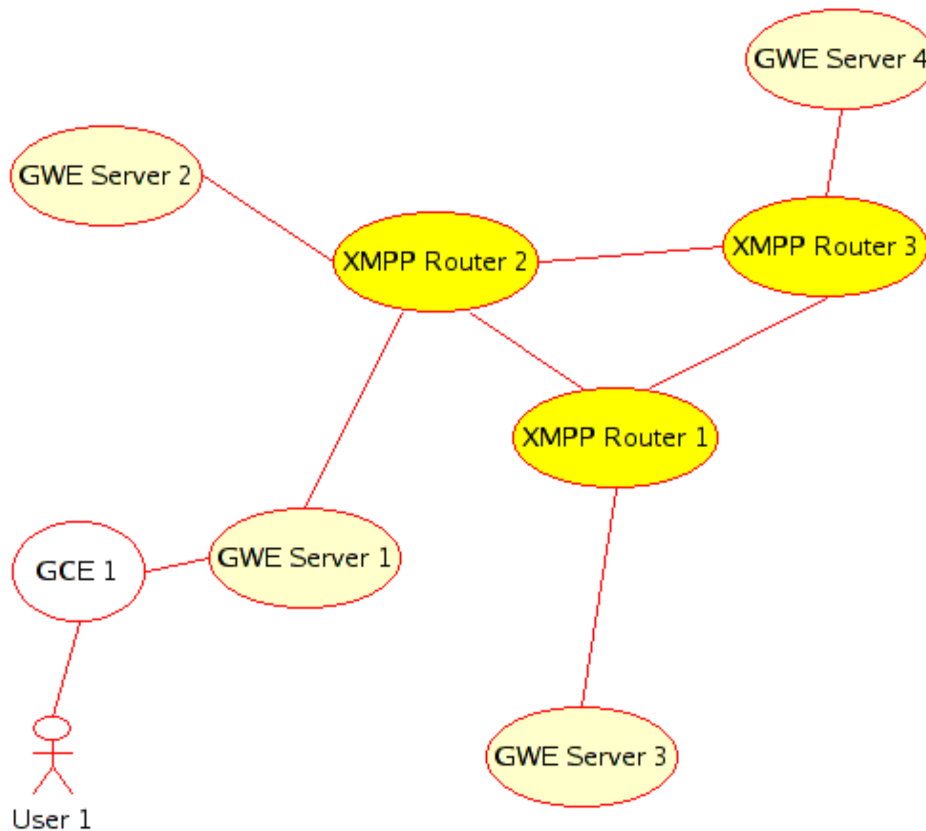
The GWE Controller classes provide all execution management functionality that is required by elements. This includes influence routing, tracking data changes. Starting and stopping element execution, requesting and forwarding data from the data controller, etc.

Network Infrastructure

Currently there is one Network implementation available, the `GXmppNetwork`. This implementation builds on the standardised XMPP protocol. XMPP is essentially a real–time XML router, as Justin K. has put it in an email. And this is exactly what we need. We have XML data that needs to be routed through the network to some destination (another GWE Server).

Speaking in XMPP language a GWE Server is an XMPP client and uses an XMPP server to communicate with other GWE Servers which are also XMPP clients (possibly from other XMPP servers). Thus we do not even have hardwired peer–to–peer connections but rather a transportation framework that takes care of message routing and delivering.

The network structure basically looks like the following:



Elements and the GWE

Element Hierarchy

The elements that make up the universe are structured hierarchically. Every element, such as a planet, includes all its subelements and presents itself as one complete unit. This way it does not matter how complex an element is internally. Planets, complete landscapes, cities, or even houses can be represented by one element, altough they consist of many subelements internally. This hierarchical structure can be thought of as logical layering. Communication between elements usually only takes place between adjecant layers. This makes the complete GWE Server infrastructure enormously scalable because each GWE Server only needs to know about adjecant Servers!

Element Responsibility

For every element exactly one GWE is responsible. Responsibility distribution is usually handled according to communication load between elements. Elements that communicate a lot with each other normally reside on the same physical GWE Server.

GWE Servers that contain elements that send influence to elements not managed on the same GWE Server need to know where to find these elements. This information can be obtained from GWE servers higher in the hierarchy.

Network transparent element management

The network capabilities of the G System include distributed element management and in particular element responsibility. This means that responsibility can also shift between GWE Servers. This raises the requirement for data transport and for control messages. Data messages can contain any element data, either complete elements or just partial data which can then be updated on the receiving GWE Server. Elements that are known on multiple GWE Servers can only have one GWE Server that actually is responsible for a particular element. Passing on the responsibility can only originate from a server that currently possesses the responsibility of a particular element.

Network latency implicitly requires save transactions and no influence loss. Here is the general procedure for shifting element responsibility described:

First, the elements themselves need to be made known to the target GWE Server. This is a simple data transfer. The source GWE Server needs to keep track of all received influences after the element data has been transfered, since influences are only forwarded to the currently responsible GWE Server. This can be done by sequentially numbering received influences and remembering the last number that the elements have received before the last data transfer to the target host.

As soon as the data has been sent to the target host, the source GWE Server asks the target host if it has enough capacity to take the responsibility for the specified elements. If the target GWE Server rejects the elements, then no further actions are taken. All influences are further delivered to the elements on the source GWE Server.

In case the target GWE Server accepts the responsibility for the elements all influences that were received by the source GWE Server in the meantime are forwarded to the target host, the responsibility is implicitly shifted and all GWE Servers that are directly higher in the hierarchy of the shifted elements are notified about the responsibility shift. Note that this shift implies that the new responsible GWE Server activates influence

emitting of the taken elements while the source GWE Server disables influence emitting for these elements.

Element Predictions

It is also possible that other GWE Servers do execute elements they are not responsible for to get adequate predictions of states. Such predicted elements are called secondary elements wheras the real elements are called primary elements. Each GWE Server can thus have primary and secondary elements.

Secondary elements are usually found on a GWE to which clients are connected. Such secondary elements never send out influences but still can receive them from local influence sources to improve prediction accuracy. Such influences are then multiplexed to the primary element and the corresponding secondary element. Influences are never sent to remote secondary elements but only to the primary elements and local secondary elements. The data of predicted elements are constantly updated from the GWE Server that hosts the corresponding primary element.

Clients and GWE Servers

A client is a GWE Server with an user interface. To the network a client is thus nothing special. In general a GWE Server that acts as a client has few elements it is responsible for and has a lot of secondary (predicted) elements.

GWE Sepcifications and Protocol

All information about required GWE behaviour in terms of service for the G Core System as well as network protocol can be found in this section. It should provide enough information to build your own compatible G World Engine. You are still advised to know about the general GWE design discussed in earlier sections.

This specification is work in progress. Information will be added while the specifications are being discussed and implemented by the developers. You can of course join the development team to add valuable considerations for this important part of the G System.

Although it is a very important aspect to fully document the functionality requirements, the current focus is to provide an implementation on the basis of C++ and the Qt toolkit.

This specification is definitely not as *normative* as it could or should be. If you have suggestions on improving the style of the specification you are very welcome to contact us on the mailinglists and discuss your ideas. Productive criticism is certainly welcome.

GWE as a GCS Container

The GCS, which defines the elements, works almost by itself. Still, it requires the GWE to provide some services, in particular the communication infrastructure between elements.

The GWE and the GCS implementation must work together and be of a compatible implementation. If you plan to implement a GWE according to this specification, you must also provide a GCS that is compliant to the GCS specification.

GWE network communication behaviour

The network protocol builds directly on the XMPP Core protocol. You can find additional information for this on http://www.xmpp.org. The protocol is thus completely XML based. Here, all valid XML messages are defined as well as the roles of the various parts of the G System.

Other protocols might be implemented. In such cases GWE Servers capable of both the XMPP based protocol and the other protocol must act as gateway. It is highly recommended to contact the G System team to discuss the integration of new protocols. The rest of this section only describes the official XMPP based protocol.

The role of the GWE in an XMPP network

The G World Engine Servers represent an XMPP client as defined by the XMPP protocol. XMPP Servers are used to establish a communication between the GWE Servers, the XMPP Servers are out of scope for this project, they are only used as a transport mechanism.

Although the GWE basically adhers to the XMPP–Core (RFC 3920) specification some elements of the XMPP–IM (RFC 3921) specifications are implemented to allow presence handling and basic message exchange with so called Jabber instant messaging clients. This allows for interesting interaction between the GWE and pure instant messaging applications.

GWE communication is totally accomplished through *message* stanzas. Each message stanza uses the appropriate to attribute for destination identification. In every such stanza additional xml data can be transported which is specified by the actual protocol.

GWE Network initialization

Network communication is initialized according to the rules specified in the XMPP–Core protocol. Additionally, after authentication a presence stanza is transmitted with an empty show element. This changes the state of the GWE Server to online in the sense of XMPP and actually enables message forwarding to the GWE Server from the XMPP Server.

After XMPP initialization the master server of the established GWE Server infrastructure is contacted to register the GWE Server and integrate it into the hierarchy. The master server usually stores the registered GWE Server as a child server.

After server registration basic element data is exchanged. When element information is exchanged for the first time normal communication is taken up.

Normal GWE communication

Normal GWE communication is taken up after successfull initialization and registration. In this state only relevant element data is exchanged between servers.

Further cases to be described in the future:

- Server hierarchy restructuring
- Element responsibility exchange
- Proper shutdown

XML Constructs for GWE Communication

This section describes all known XML constructs that the GWE server must be able to understand. A short description of the expected behaviour is also given.

We refer to each such XML construct as *message*. Be careful not to confuse it with the message stanza kind of XMPP. Still, it is true that the messages described here can only be contained in message stanzas, so there is some connection at least.

Remember that particularly this section is subject to changes, especially extensions. The G System developers are not bound to ensure compatibility between different versions of development releases. Only with the release of version 1.0 can this protocol specification be considered stable and compatibility will be ensured at least through all 1.x releases.

<GElement>

See the GCS XML Schema for the structure of a valid GElement XML message.

Additional to the given structure in the XML Schema, the following attribute is allowed in the <GElement> tag:

Additional attributes in <GElement> tag:

- *owner* (string, XMPP JID), defines the GWE Server that manages this element as primary element.

A GElement message is used to exchange information about elements. The additional owner attribute can be used to find the GWE Server that manages this element, which is very important for sending influences and other element related messages to the correct server. This kind of message is probably the most commonly used.

<GElementInfluence>

See the GCS XML Schema for the structure of a valid GElementInfluence XML message.

Additional to the given structure in the XML Schema, the following attribute must be included in the <GElementInfluence> tag:

Additional attributes in <GElementInfluence> tag:

- *target* (GCS::GElementID, target element), defines the target element to which the influence should be forwarded.

A GElementInfluence message is used to transport influencing information. The additional target attribute must specify the actual target element since the influence itself does not contain target information.

<reparent>

<reparent> <element> element id which is reparented (GCS::GElementID) </element> <from> old parent GElementID (GCS::GElementID) </from> <to> new parent GElementID (GCS::GElementID) </to> <transformation> transformation matrix (GCS::GMatrix44) </transformation> </reparent>

The IDs are in the form of GCS::GElementID XML representations, the transformation matrix is in the form of GCS::GMatrix44 XML representation. The <transformation> tag is optional, if it is not present, an identity matrix is assumed (usually the case with formless elements).

For reparenting notifications this message is used. The same construct is sent to all servers that manage one of the three affected elements. Each server modifies the parent−child relations of its primary elements.

Whenever a server managing a particular element detects that reparenting is necessary, this message is used to notify the old and new parent. Such a message should only be generated by a server which manages the reparented element (as primary element).

<register>

<register> descriptive text </register>

The registration construct is simply empty or can contain some description. The sender can be identified through the XMPP protocol, so it doesn't need to be provided through this message.

The use of this message is to register a new child server at a master GWE server. This is usually done after server startup.

<unregister>

<unregister> descriptive text </unregister>

Like the register message, the unregister message is also an empty element with optional descriptive text in the body.

The use of this message is to unregister a GWE server that is about to shutdown from a master server.

<requestfreeids>

<requestfreeids> unsigned long number, holding the amount of requested free element IDs </requestfreeids>

The requestfreeids message only holds the amount of requested IDs.

If a G World Engine server is running low on available element IDs it usually sends a requestfreeids message to its master server to request more free element IDs. Such a request is usually answered by a <freeids> message.

<freeids>

<freeids> <from> range 1, lower bound </from> <to> range 1, upper bound </to> <from> range 2, lower bound </from> <to> range 2, upper bound </to> <!−− and so on, any number of ranges allowed −−> </freeids>

The freeids message ranges of element IDs that can be used for additional element creation.

The <freeids> message is usually an answer to a <requestfreeids> message. These two messages make it possible to distribute free element IDs within the server infrastructure.

Note that it is not uncommon for a GWE server to send free IDs to its master server in case many IDs get available because elements are removed. This way used element IDs get reused.

G Basic Elements (GBE) – bringing the system to life

As we have seen the GCS does nothing out of itself. In order to get functionality one has to create agents. Some basic classes of this kind are already available and can be found in the GBE (G Basic Elements) library.

The kind of classes you will find in this library mostly have to do with virtual worlds – world creation, element hierarchy management, physics, ... soon this library will also provide you with agents for evolution aspects. Some of the more important agents will be discussed in this section.

Deterministic Random World Generation

World creation is an interesting topic, some of the required functionality is already implemented. The class `GBE::GDynamicGeneratorAgent` class provides functionality to generate a randomised universe based on a deterministic random generator. As this allows for a huge universe to be created, the generator can create just as much as the current active elements (whether they are intelligent computer driven beings or human users) require. As these active elements start to explore other parts of the universe the generator brings these parts into existence – and by the time passed, these parts can determine in what state they have evolved (age of solar systems or planets,...). Parts of the universe that are not inhabited by active elements are just put to "sleep" to save computing power. This is sadly a limitation we have, but it will have only a minimal (if at all) effect on the overall simulation if handled correctly.

Now, if the G System is about evolution, how does this get along with a randomised universe? This is rather a philosophical question but as it must occur to some readers we will answer (part of it) it here. First we need to distuingish between form representation and physiological aspects of the universe. The deterministic random world generation is in the first sense a means to determine where to place what kind of elements. It is a completely different topic how these elements behave after their creation, which is not in any way determined by the generator. Also, if for example the generator determines that on a certain location in a solar system a planet should be placed then the planet is not necessarily just there but the created element can just be a divine energy in the area of the planet that slowly evolves into a planet – for example by attracting asteroids, fragments of dead planets, ... So the determination that on a certain location a planet should be, does not clash with the way how this planet comes into existance and evolves.

Element hierarchy management

Another agent, `GBE::GReparentAgent`, handles the hierarchical structure of elements. As discussed earlier, the universe is represented in a highly hierarchical structure – which is indeed necessary to allow for huge universes to be functional. Thus a human being may be part of a house while being inside. But if the human being walks out, it will be part of the city or the surrounding landscape. In order to handle the hierarchy changes properly, the reparent agent observes positional changes and notifies all affected elements when a change occurs. Details can be found in the API documentation.

The G Client Engine (GCE) – making it visible

In order to allow user interaction an user interface has to be provided – this is the job of the G Client Engine. It allows for the rendering of elements (that is, the forms of the elements) and should forward user input to the elements controlled by the user.

The GCE builds directly on the GWE, this means that the user actually has a local GWE Server running which in turn connects to the other servers.

GWE Server can handle primary and secondary elements. Clients will naturally have many secondary elements.

As the current implementation is mostly for testing purposes, we would appreciate it very much if some volunteer with some experience in 3D programming would assist in writing a good client framework.

We target at a very reusable client library. That means everyone should be able to write his/her own client or extend the given standard client to his own needs. The protocol will be completely open – as the whole software is. Thus everyone is able to create clients to personal preferences. Even clients without a GWE seem possible if the GWE protocol is implemented all the way.

G Universe Server

TO BE WRITTEN

G Universe Client

TO BE WRITTEN

GOD

TO BE WRITTEN

Agent Plugin Architecture

Overview

Attention: This concept is not yet fully implemented. It is also considered to provide scriptable agents that can load scripts that define their actual behaviour. Various scripting languages are possible, like Ruby or SPL. It is still considered which is best suitable. A possibility would also be to provide more than one scripting language, like `GBE::GRubyAgent` and `GBE::GSplAgent`. Thus, more scripting languages are usable.

Agents are the parts of an element that define their behaviour. The behaviour of elements ultimately defines the physiology or the behaviour of the overall simulation created with the G System.

This makes agents a very important part of the G System. In particular agents are exactly what is used to customize the simulation to the needs of the particular usage. This also means that agents need to be created by people not directly developing the G System itself. To allow everyone to develop custom agents, a plugin architecture is used.

Agents can thus be developed independently of the G System Framework distribution. The plugin architecture takes care of dynamically loading agents into the simulation. It is even possible to update agents during runtime by unloading and reloading a particular plugin.

Plugin management in the GWE

The GWE actually takes care of loading and creating `GCS::GAgent` objects from libraries. The responsible class is the `GWE::GAgentPluginManager` which implements dynamic library loading and plugin instantiation. Any kind of agents can be loaded this way.

The `GWE::GAgentPluginManager` class is also used by the `GWE::GCoreXmlSerializer` class to actually create agents specified by the XML document.

Distributing the plugin libraries

Since the G System is a distributed simulation, it is necessary to somehow inject a plugin into a running simulation.

This works by installing the plugin into the master GWE Server, this server automatically further distributes the library file to all direct child GWE Servers which in turn do the same. This way a new plugin gets automatically known to the complete server infrastructure.

Developing an agent plugin

TO BE WRITTEN

Chapter 7. Using and Extending the G System Framework

Table of Contents

Tutorial: Basic Agent Design
- *Introduction*
- *Step one: sending influences*
- *Step two: receiving influences*
- *Step three: writing the application*
- *Source Files*

Tutorial: Basic Agent Design

Introduction

This tutorial is based on version 0.3 of the G System. As such it is very much outdated and needs to be updated for the current version.

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 in general are provided by the GCS library. All required information will be provided in this tutorial, but you still might want to take a look at the API documentation while working through the tuorial.

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 reimplementation 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);
  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 8. Contributing to the G System Project

Table of Contents

The purpose of the G System
Joining the team
Donations

The purpose of the G System

The G System is completely free and open source software. This opens the G System to anyone who is interested. Not just everyone is able to use it, but the technology that lies within it is available to everyone down to every detail. It is truly a project for common good.

The reason for creating such an open project lies in its purpose, which is also a global effort of research on the field of evolution, which is in the end an aid for every human being because it is the very nature of humanity.

As such a global effort it would only hinder its purpose if access to the project is restricted. Thus the G System is designed as a project for everyone. And thus everyone interested in this aspect of life and science is welcome to take part.

Joining the team

There are many aspects in which you can contribute, not just as a developer. There is a section dedicated to joining the team, you can look there for the possibilities on what you can work on and how you can actually join the G System team.

Probably the most important contribution is to be an user.

Since the G System has to do with many specialised technical domains, like networking, databases, client programming and even artificial intelligence a wide variety of specialised software engineers can help the G System. On the other hand specialists in social domains as well as scientists of any sort are welcome. If in doubt whether you can be useful or not, just contact us to find out.

Donations

It would also help a lot if you are interested in supporting the project financially with donations, which helps us pay our infrastructure and especially allows us to spend *more* time working on the project because we don't need to earn money with other work. Feel free to contact us if you are interested in any way.

G System Documentation Home