GGE Manual

Daniel Johansson

July 2, 2023

Contents

Part I
GGE in short

GGE stands for Grid Game Engine (very creative, I know). It is inteded to be used as a game engine for grid based games such as chess or Sid Meier's Civilization. It is not only restricted to turn based games.

The engine itself is written in C++ and depends on SDL2. Games for the engine are written in Guile, a variant of lisp.

There are tools for generating code written in Perl5 though they are not necessary for using the engine.

The following two parts will cover the inner and outer workings of the engine.

The inner working of the engine is not strictly necessary to know when writing a game but it may help understand what happens under the hood. This part should be helpful in case of modifying the engine or adding custom game engine modules.

The outer working of the engine is the (Guile) scripting part of it. This is necessary for understanding the process of making a game. It will also thuroughly describe the example "game" provided.

Part II
Inner Workings

1 Coding Standards

Or rather the lack there of. Before anyone gets an aneurysm from reading the code, be aware that this is a project that has been spanning years during which I have developed and changed. What I thought was good coding at the begninning of the project may not be something I think holds true anymore.

So why have a whole section on it? Aside from personal change there are also two other big contributing factors that I think are important to consider before you burn me at the stake.

First of all there is the fact that my job is to work with ancient legacy Perl code that is almost as old as I am. Trying to force coding standards on decade old code where there are none will either take a life time or drive you insane, one does not exlude the other. Which leads me to the second factor.

In order to not go insane from the jungle that legacy code can be, especially in a language like Perl, I have a Taoist approach. Rather than forcing what I consider "good" upon the code I adjust and try my best to the standard(s) of the file(s) I'm currently working in. Rather than pulling my hair because I can't understand what stupid idiot wrote this piece of shit code I take a deep breath I accept that this might just be what was best at the time of writing. Adapt to the structures that are there already. Be like water. The downside with this approach that sometimes a suboptimal structure remains longer that necessary because the workaround is still easier to work with than to actually properly solve the problem. But every once in a while the water will be a tsunami.

The little code standards that exists:

Type	Style
Class	`Class_name`
Variable	`variable_name`
Const	`CONST_NAME`
Function	`function_name`

Data referencing standard:

1. Reference (not for storage)
2. Smart pointer
3. Pointer (for SDL related data only) (only one object is responsible for freeing the memory pointed to)

I could of course try to make SDL work with smart pointers but using normal pointers works fine. It is not that much of an issue to use. Just keep of track of the pointer. How hard can it be?

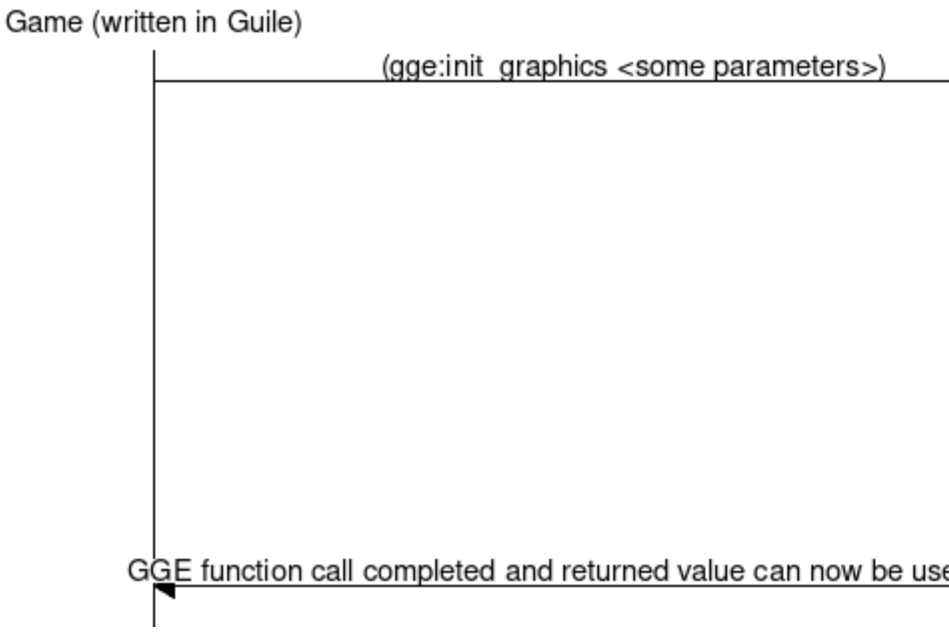

Figure 1: A call from the game to initialize the graphics module.

2 The Core

At the core of GGE is the `Core` object. The `Core` contains `Moduler` for storing and `Runner` running GGE-modules various functions in desired order. Without the core much of the necessary coordination between objects would not be possible.

3 Guile - GGE Communication

Communication between the game, written in Guile, and GGE, written i C++, is done via code found under `script_handling/` and in the center stands the `Scripter` object.

There are several steps to script communication as illustrated by figure 1.

First the `Script` object needs to initialize a script engine object i.e. some class that inherts from the `Script_engine` class, at the moment of writing only Guile is available, but GGE is open for more.

The script engine object must implement functions necessary for direct communications with the script itself and whatever is necessary for the script to communicate with a C++ program and the `GGE_API`. Guile can communicate with C++ via C so a simple C wrapper (C-linked code to be more precise) is necessary.

GGE communicates with the game by having modules exposing their functions to the `GGE_API`. This is done entirely in C++. Any functionality not exposed to the `GGE_API` can be seen as private member functions of GGE no available to the game. Certain core functions are always exposed such as `add_command` and the initialization functions.

The `Scripter` object is not a part of the `Core` object, neither is the `GGE_API` object.

4 GGE-Modules

GGE performs anything but the bare essentials through GGE-modules, or simply modules. The bare essenatials in this case is memory management and script communication.

A module is really just a class that performs some specfic function. See 4.1 for list of standard modules. In order for GGE to be able to use a class as a module it must fullfil certain conditions listed in 4.2. An example of a module is the graphics module. It handles all the graphics related functionality of the engine such as rendering graphics on the screen.

If the game does not initializes any modules GGE can not perform any actions. The `Scripter` object is not a GGE module, it is always initiated. If the game does not add any commands (see 7.2) for initalized modules GGE will not perform any actions with the modules intialized. You will not even be able to quit the game since no event module that handles input exists, unless it has been written into the game to quit on its own.

In order for the game to make use of a module in the game engine

1. Initialize the module to be used.
2. Add a command for the module if necessary.
3. When the game is finished initializing and the `Core` object will run the commands in the order given by the script.

4.1 Standard Modules

Internal module name and class names are only used within the engine.

Internal Module Name	Class Name
GRAPHICS	`Graphics`
EVENTS	`Events`
GRID	`Hex_grid`
SCROLLER	`Scroller`
TEXTER	`Texter`
GAME_LOOP	`Game_loop`

4.2 Creating a Module

In order for GGE to see a module it must be registerd in the file `registerd_gge_modules.hpp` in the enum `registered_gge_module`. This enum, or rather its integer representation, is returned to the game when the module is initialized. In order for GGE to use a module it must inherit from the `GGE_module` class. This enables GGE to use polymorphism to handle modules in a generic way, both the `Moduler` and `Runner` use this genericity to handle memory of modules and calling commands (funcitons of modules) respectively.

Next modules should be initializable via a function in `GGE_initializer`. These functions should be callable from the `GGE_API` for exposure to the outside.

Now the module can be loaded by the engine. In order to actually make use of the module it is necessary to create command classes that make use of the module's member functions.

Both intialization functions and command classes one can either be written manually or generated via the Perl scripts in `build_tools/`. See the section 5 for how to do so.

4.3 Command Classes

In order for GGE to know what functions to call of what module in which order the command pattern is used. Each module should have a so-called command class that inherits from the class `Command`. For example, the `Graphics` module has a `Graphics_command` class.

This command class should contain all the functions one can call from the game. This enables GGE to call these functions in the desired order given by the game.

The class `Command` contains

- a pointer to the module.
- a pointer (that may be null) to the argument.
- command.

These can be used by its children.

These command classes are used by the `Runner` object by calling each command class' `exeute()` function. This function will run whichever command it was set to execute when the command class was instantiated.

A command class may contain multiple functions that can be called, but only one can be executed by each command class. Thus several of each command class my be needed in order to call different functions of the same module.

4.4 Module Handling

Once all modules are registered, their functions exposed through the GGE API and have the corresponding the command classes they can now be used in the

game script. First one needs to specifiy in the game script which modules to use and in what order the modules functions are to be called.

Module memory management is handled by the `Moduler` class. Any module can be obtained from the `Moduler` by providing the module's enum.

Module function execution order is handled by the `Runner` class. The `Core` calls the command class functions by calling `Runner.execute()`. Polymorphism is used to call module specific function.

GGE will check to ensure the initialization and call order is correct. A failed check will result in an error. If the check passed the game will start and GGE will run the modules in the given order.

4.5 Modules Containing Components

There are multiple classes making use of the module `Componenter` in order to handle multiple sub structures. These classes can be seen in the table below.

Class	sub structure
`Agenter`	`Agent`
`Spriter`	`Sprite`
`Texter`	`Text`

5 Generating Code

Perl5 is used to generate a bunch of code. This is not necessary in order to run the engine. While not strictly necessary to use when modding the engine there are unfortunately some places of the code that require repetitive coding which can be skipped with the scripts in `build_tools/`. Simply run all scripts ending in `.pl` and all necessary code will be generated. The table below describes the different scripts and their functions.

Script Name	Writes to	Description
`expand_gge_module.pl`	`gge_module.hpp`	Generates cases for stringifying a modules name
`expand_modules.pl`	`moduler.hpp` and `moduler.cpp`	Generates getters and setters for modules
`expand_modules.pl`	`moduler.hpp` and `moduler.cpp`	Generates getters and setters for modules in the Moduler
`generate_commands.pl`	`commands/<specific_command>.hpp`	Generates classes for specific commands
`generate_get_gge_modules.pl`	`generated_get_gge_modules`	DEPRECATED
`generate_gge_api_defaults.pl`	`gge_api_defaults.generated`	Generates template code for exposing functions in the `GG`

Note: the Perl scripts to not check for errors nor always generate nicely formated code. Make sure your code is correct or finding erros will be a hassle.

Part III
Outer Workings

6 Guile

The games for GGE are written in Guile is an implementation of Scheme, which is a dialect of Lisp.

The game should be all contained in one directory. This directory should be used as a parameter to GGE.

GGE will be looking for a two important files.

First `init.scm`. This is where module initialization of modules and addition of commands should take place.

Second `game.scm` where a function called `game_loop` should exist. In this function game logic should reside.

7 Modules

If you want to use a non-standard module please read through section 4.

7.1 Initializing a Module

Modules are initialized through the `gge_api` functions found in the `gge` module. The function will return an integer that represents the ID (enum) of the module used inside GGE. This can be useful for when adding commands. Example: `(init_graphics "GGE Test" 640 480))`

7.2 Adding a Command

Once modules have been initiated commands can be added to the module through the `gge_api` function `add_command` Example: `(add_command game_loop)`

GGE will check to ensure the initialization and call order is correct. A failed check will result in an error and not continue running the script.