GGE Manual

Daniel Johansson

August 16, 2022

Contents

Part I
GGE in short

GGE stands for Grid Game Engine. 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/C++ and depends on SDL2. Games for the engine are written in Chai script, a C++-header scripting language.

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 (chai) scripting part of it. This is necessary for understanding the processes 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. 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 reference standards:

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?

2 The Core

At the core of GGE is the Core object. The core in of itself does not do much in the way of running a game, for that modules are used such as the graphics module. However, without the core none of the many parts of GGE would be able to function.

3 Chai - GGE Communication

Communication between the game, written in Chai script, and GGE is done via the `Scripter` object. This is done by exposing the `GGE_API` object to the chai script which in turn communicates with the rest of GGE.

4 Modules

GGE performs anything but the bare essentials through modules. 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 nothing can happen. If the game does not add any commands (see 7.2) for initalized modules nothing will happen. You will not even be able to quit the game since no event module that handles input exists. 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.

The standard modules that can be used. Internal module name and class names are only used within the engine. In the script the external module name is used, sometimes as a suffix (e.g. `init_events`).

Internal Module Name	Class Name	External Module Name
GRAPHICS	`Graphics`	graphics
EVENTS	`Events`	events
GRID	`Hex_grid`	grid
SCROLLER	`Scroller`	scroller
TEXTER	`Texter`	texter

4.1 Registering a Module

In order for GGE to run modules they must be registerd in the file `registerd_gge_modules.hpp`. Next modules must be initializable, either via another module's initialization function or its own initialization function. These functions should reside in the `GGE_API` for exposure 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.

In order to create command classes one can either write them manually or use the Perl scripts to generate the code needed. See the section 5 for how to do so.

4.2 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 Chai script. This enables GGE to call these functions in the desired order given by the Chai script.

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.

4.3 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 chai script. First one needs to specifiy in chai 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. Module function execution order is handled by the `Runner` class.

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. First the modules' commands are executed then the game loop, the one defined in Chai script, will be executed.

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 Chai script

The games for GGE are written in Chai script.

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 file called `init.chai`. This is where module initialization of modules should take place. It should return a string containing the name of the object where the main loop of the game is, i.e. where game logic takes place. This object's function `game_loop` will be called by GGE.

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 beginning with `init_` e.g. `init_graphics(<ARGUMENTS>)`. Some modules are initialized together with the same function such as graphics and scrolling modules. This will be changed or made more clear in the future.

7.2 Adding a Command

Once modules have been initiated commands can be added to the module through the `gge_api` function `add_command` The argument is a string in the form of `<CLASS_NAME>.<CLASS_FUNCTION>(ARGUMENT)`.
Example: `gge_api.add_command("graphics.draw(grid)")`

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.