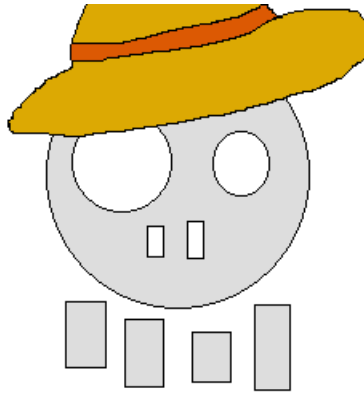


# Strawhat Game Engine



## Introduction:

The Strawhat Game Engine is a framework written in C based on the SDL 2 Library. It is made of several components that work together and provide an API for writing games. These components are structured such that there is a header file for each component and a respective implementation file.

For example, the SGE\_GUI component is made of a header file called SGE\_GUI.h and an implementation file called SGE\_GUI.c. This document describes the structure of the SGE framework in detail and documents the API.

The components of SGE are:

- 1.SGE (core)
- 2.SGE\_GameState
- 3.SGE\_Logger
- 4.SGE\_Texture
- 5.SGE\_GUI
- 6.SGE\_Math
- 7.SGE\_AnimatedSprite

These components are further explained below.

## 1. SGE (Core)

SGE (Core) or just SGE refers to the set of files (header and implementation) that make up the base of the engine, this means that it contains the functions and structures to initialize the engine, create a window and start the main game loop. The game loop is inside of the *SGE\_Run()* function inside of this component. The game loop looks like below (simplified).

```
while(isRunning == true)
{
    handleEvents();
    update();
    render();
}
```

These three functions are called repeatedly until the value of `isRunning` becomes false.

*handleEvents()* is used for handling input events.

*update()* is used for logical updates to game data.

*render()* is used for drawing things to the screen.

The value of `isRunning` is changed to false when *handleEvents()* detects a `SDL_QUIT` event, ending the loop and quitting the application.

The game loop is started by calling *SGE\_Run()* from your applications entry point, the `main()` function. However, before running *SGE\_Run()* to start the game loop, the *SGE\_Init()* function must be called, which also resides in this component. So to begin writing a game in SGE, you would first include the library in your program and then, in your *main()* function call *SGE\_Init()* followed by *SGE\_Run()*

The most basic program created using the SGE engine would be as below.

```
#include <SGE.h>

int main(int argc, char *argv[])
{
    SGE_Init("SGE Game", 800, 600);
    SGE_Run(NULL);
    return 0;
}
```

Running this program will open a window filled with a blue color that is 800 pixels wide and 600 pixels high with the title "SGE Game".



Picture [1A]: The most basic SGE program

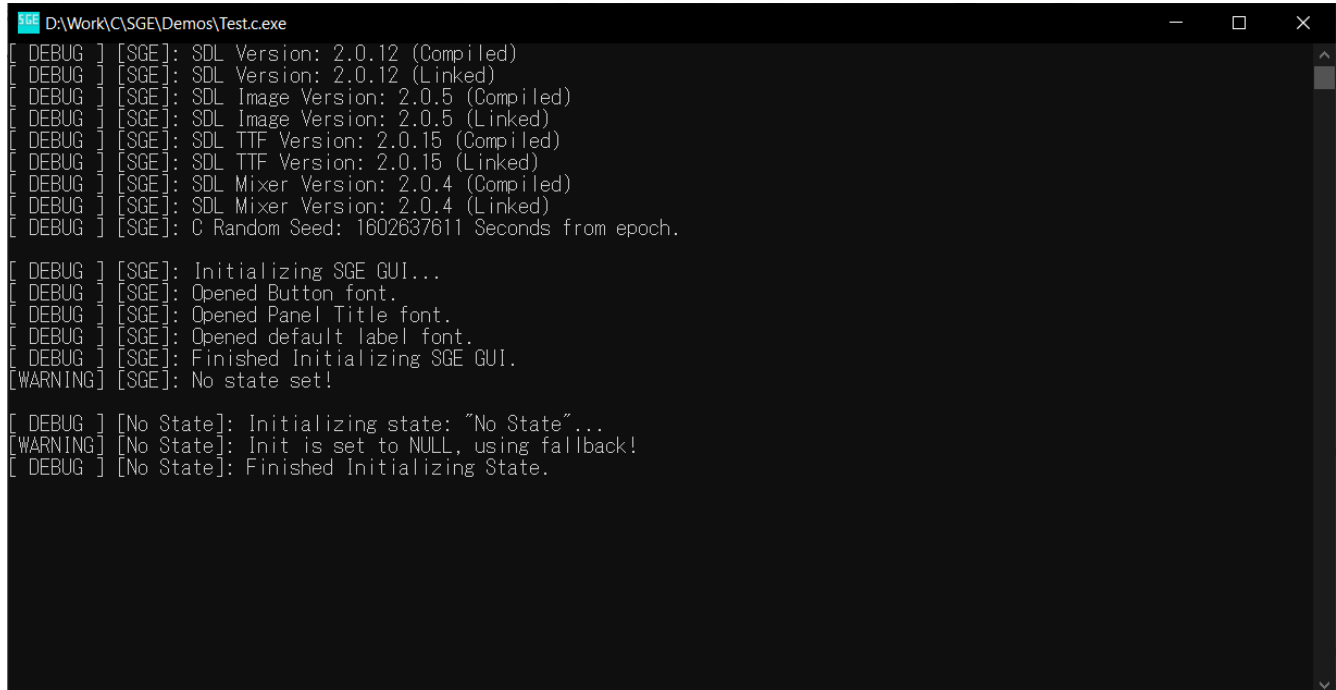
The first line includes the SGE.h header file into your program. Notice that the *main()* function has a specific signature, *int main(int argc, char \*argv[])* that is, it takes two parameters and returns an integer value. It is a requirement of SDL 2 to have the main function take these parameters and return an integer. This means that programs using SGE must also have their main function take these two parameters and return an integer.

The first parameter is an integer which represents the number of command line arguments passed to *main()* by the OS. The second is an array of strings which represents the actual commands passed to *main()*. These parameters are used for command line applications in C, since SGE is for making games, which are GUI applications we don't have to worry about using these parameters. The return value is used by the OS as an error code, it should always be 0 for a successful execution of the program.

The *SGE\_Init()* function takes three parameters, the first is a string that will be the title of the window and the second and third parameters are integers, which are the width and the height, respectively of the created game window.

The *SGE\_Run()* function takes one parameter that is a pointer to a SGE\_GameState structure which acts as a starting state for the game. SGE's state system is explained later, so passing NULL will result in an empty window with a blue background. This window will remain open and continuously fill the screen with the blue color until the close button on the window is clicked.

The return statement is reached after the window is closed and returns a zero to the OS signifying a successful execution.



```
D:\Work\CSGE\Demos\Test.c.exe
[ DEBUG ] [SGE]: SDL Version: 2.0.12 (Compiled)
[ DEBUG ] [SGE]: SDL Version: 2.0.12 (Linked)
[ DEBUG ] [SGE]: SDL Image Version: 2.0.5 (Compiled)
[ DEBUG ] [SGE]: SDL Image Version: 2.0.5 (Linked)
[ DEBUG ] [SGE]: SDL TTF Version: 2.0.15 (Compiled)
[ DEBUG ] [SGE]: SDL TTF Version: 2.0.15 (Linked)
[ DEBUG ] [SGE]: SDL Mixer Version: 2.0.4 (Compiled)
[ DEBUG ] [SGE]: SDL Mixer Version: 2.0.4 (Linked)
[ DEBUG ] [SGE]: C Random Seed: 1602637611 Seconds from epoch.

[ DEBUG ] [SGE]: Initializing SGE GUI...
[ DEBUG ] [SGE]: Opened Button font.
[ DEBUG ] [SGE]: Opened Panel Title font.
[ DEBUG ] [SGE]: Opened default label font.
[ DEBUG ] [SGE]: Finished Initializing SGE GUI.
[WARNING] [SGE]: No state set!

[ DEBUG ] [No State]: Initializing state: "No State"...
[WARNING] [No State]: Init is set to NULL, using fallback!
[ DEBUG ] [No State]: Finished Initializing State.
```

Picture [1B]: Console Window

You will also have noticed the console window that opens along with the game window. It is used for debugging and has lots of information about the internal workings of SGE. Later on, we will see how to use the SGE\_Logger component to print information to the console.

## 2. SGE\_GameState

SGE is based around a concept of states, which are a collection of functions that are grouped together and executed in a specific order. This state system allows switching between states which act as a useful tool to have different sections in a game such as a title screen, a settings screen, different levels and other such areas.

An SGE\_GameState is a structure defined in the SGE\_GameState.h header file.

```
typedef struct
{
    char name[20];
    bool (*init)();
    void (*quit)();
    void (*handleEvents)();
    void (*update)();
    void (*render)();
} SGE_GameState;
```

It is a collection of five function pointers, along with a name for the state. An object of type SGE\_GameState must be created and a pointer to it must be passed to *SGE\_Run()* to define it as the starting state of the game. A game can have multiple states which can be switched between, but a game must have at least one state in order to make anything more than an empty window.

In order to create a state, its five functions must also be created. These functions can have any names but their signatures must match the function pointers in the SGE\_GameState structure. Once these functions are created a variable of type SGE\_GameState must be made and initialized with these functions as its function pointers. This initialization is done using the *SGE\_SetStateFunctions()* function. This state variable can then be passed to *SGE\_Run()* and *SGE\_SwitchToState()* as a pointer.

The *SGE\_SetStateFunctions()* function takes seven parameters, the first of which is a pointer to a SGE\_GameState variable for which we want to set the state function pointers to the functions that we write ourselves. The second parameter is a string that defines a name for the state, and the next five parameters are the names of the functions we create for this state.

These functions are executed in a specific order. First, the init function is called before the game loop starts, it is intended for setting up the state data such as loading assets and setting initial values. Next, the handleEvents, update and render functions are executed in a loop until the program either exits or a state switch is triggered by calling *SGE\_SwitchToState()*, in which case, the quit function is triggered freeing all the data loaded by the init function. If a switch was triggered, then the init function of the next state is called after quitting the previous state and the game loop starts for the new state.

The *SGE\_SwitchToState()* function just takes a pointer to the SGE\_GameState variable of the next state to switch to as a parameter. It can be called from within the handleEvents and update loop to switch from one state to another in real time. However, this will cause all the resources loaded by the previous state to be freed. This means if the same state is loaded again at a later time, it will reload all its resources. This can be avoided by having persistent states in a future version of SGE.

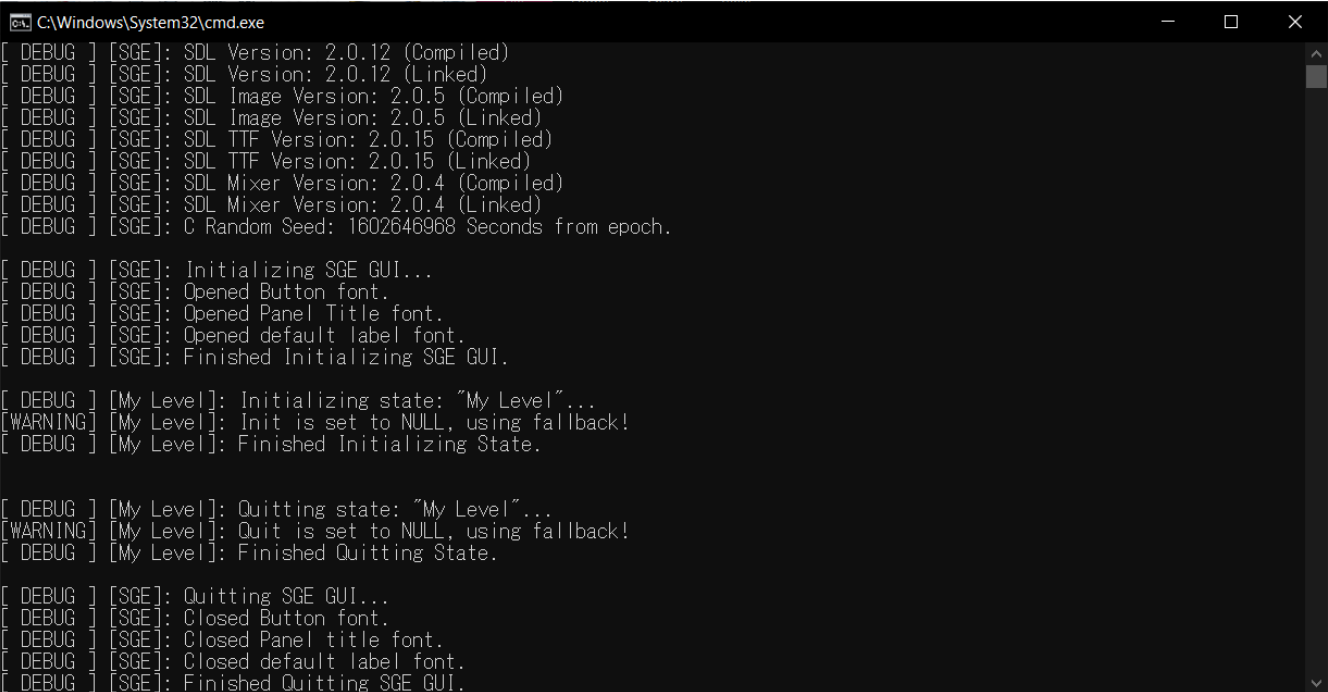
The following program demonstrates the usage of the state system to write a game with a single state called Level which changes the screen color to red and quits when the escape key is pressed.

```
#include <SGE.h>

void LevelHandleEvents()
{
    if(SGE_GetEngineData()->event.type == SDL_KEYDOWN)
    {
        if(SGE_GetEngineData()->event.key.keysym.sym == SDLK_ESCAPE)
            SGE_GetEngineData()->isRunning = false;
    }
}

void LevelRender()
{
    SGE_ClearScreenColor(SGE_COLOR_RED);
}

int main(int argc, char *argv[])
{
    SGE_GameState levelState;
    SGE_SetStateFunctions(&levelState, "My Level", NULL, NULL,
LevelHandleEvents, NULL, LevelRender);
    SGE_Init("SGE Game", 800, 600);
    SGE_Run(&levelState);
    return 0;
}
```



```
C:\Windows\System32\cmd.exe
[ DEBUG ] [SGE]: SDL Version: 2.0.12 (Compiled)
[ DEBUG ] [SGE]: SDL Version: 2.0.12 (Linked)
[ DEBUG ] [SGE]: SDL Image Version: 2.0.5 (Compiled)
[ DEBUG ] [SGE]: SDL Image Version: 2.0.5 (Linked)
[ DEBUG ] [SGE]: SDL TTF Version: 2.0.15 (Compiled)
[ DEBUG ] [SGE]: SDL TTF Version: 2.0.15 (Linked)
[ DEBUG ] [SGE]: SDL Mixer Version: 2.0.4 (Compiled)
[ DEBUG ] [SGE]: SDL Mixer Version: 2.0.4 (Linked)
[ DEBUG ] [SGE]: C Random Seed: 1602646968 Seconds from epoch.

[ DEBUG ] [SGE]: Initializing SGE GUI...
[ DEBUG ] [SGE]: Opened Button font.
[ DEBUG ] [SGE]: Opened Panel Title font.
[ DEBUG ] [SGE]: Opened default label font.
[ DEBUG ] [SGE]: Finished Initializing SGE GUI.

[ DEBUG ] [My Level]: Initializing state: "My Level"...
[WARNING] [My Level]: Init is set to NULL, using fallback!
[ DEBUG ] [My Level]: Finished Initializing State.

[ DEBUG ] [My Level]: Quitting state: "My Level"...
[WARNING] [My Level]: Quit is set to NULL, using fallback!
[ DEBUG ] [My Level]: Finished Quitting State.

[ DEBUG ] [SGE]: Quitting SGE GUI...
[ DEBUG ] [SGE]: Closed Button font.
[ DEBUG ] [SGE]: Closed Panel title font.
[ DEBUG ] [SGE]: Closed default label font.
[ DEBUG ] [SGE]: Finished Quitting SGE GUI.
```

Picture [2A]: Console Output With State Set

In this example, *LevelHandleEvents()* and *LevelRender()* are the only functions we are creating and sending into the state. For *init()*, *quit()* and *update()*, we are sending NULL pointers to *SGE\_SetStateFunctions()*. This will work and the init, quit and update function pointers in levelState will be set to pointers to fallback functions that just print a warning to the console when initializing and quitting the state. This is similar to how we passed NULL to *SGE\_Run()* which causes the console to output the state as “[No State]”.

The *SGE\_GetEngineData()* function is a very useful function that returns the address of the main engine data structure, that is, the variable of type *SGE\_EngineData*. It contains engine level data such as the *SDL\_Window*, *SDL\_Renderer*, the *screenWidth* and *screenHeight* variables and other globally accessed data variables. Only one instance of this *SGE\_EngineData* structure exists and it is statically stored in the *SGE.c* file, with the *SGE\_GetEngineData()* function in *SGE.h* providing global access to it.

The *SGE\_Init()* function returns the same address when it is called, so it can be stored in a local variable in your own code. For example, instead of calling *SGE\_GetEngineData()* multiple times, the address can be stored as follows:

```
#include <SGE.h>

SGE_EngineData *Engine = NULL;

void LevelHandleEvents()
{
    if(Engine->event.type == SDL_KEYDOWN)
    {
        if(Engine->event.key.keysym.sym == SDLK_ESCAPE)
            Engine->isRunning = false;
    }
}

void LevelRender()
{
    SGE_ClearScreenColor(SGE_COLOR_RED);
}

int main(int argc, char *argv[])
{
    Engine = SGE_Init("SGE Game", 800, 600);
    SGE_GameState levelState;
    SGE_SetStateFunctions(&levelState, "My Level", NULL, NULL,
LevelHandleEvents, NULL, LevelRender);
    SGE_Run(&levelState);
    return 0;
}
```

## Creating and using multiple states:

The program above demonstrates the process of creating a single state for a game and using it as the starting state by passing it to *SGE\_Run()*. A program like this is good for writing a small game where the entire game can be written within the five functions of a single state. However, in order to write bigger games, it is useful to have an option to split the game into multiple sets of functions that are grouped together in a logical way. For example, a game with many different levels can have a set of five functions for each level and a level select screen that allows the player to switch between these levels. Such a game can be created by using one *SGE\_GameState* variable called *levelSelectState* and then, one such as *level\_01\_State* for each level. The *handleEvents* function for *levelSelectState* can check what level to start and switch to that level by calling *SGE\_SwitchToState(&level\_01\_State)*. This is demonstrated in the below example program.

```
#include <SGE.h>

SGE_EngineData *Engine = NULL;
SGE_GameState levelSelectState;
SGE_GameState level_01_State;

void LevelSelectHandleEvents()
{
    if(Engine->event.type == SDL_KEYDOWN)
    {
        if(Engine->event.key.keysym.sym == SDLK_RETURN)
            SGE_SwitchToState(&level_01_State);
    }
}

void LevelSelectRender()
{
    SGE_ClearScreenColor(SGE_COLOR_GREEN);
}

void Level_01_Render()
{
    SGE_ClearScreenColor(SGE_COLOR_BLUE);
}

int main(int argc, char *argv[])
{
    SGE_SetStateFunctions(&levelSelectState, "Level Select", NULL,
    NULL, LevelSelectHandleEvents, NULL, LevelSelectRender);
    SGE_SetStateFunctions(&level_01_State, "Level 01", NULL, NULL,
    NULL, NULL, Level_01_Render);

    Engine = SGE_Init("State System Demo", 800, 600);
    SGE_Run(&levelSelectState);
    return 0;
}
```

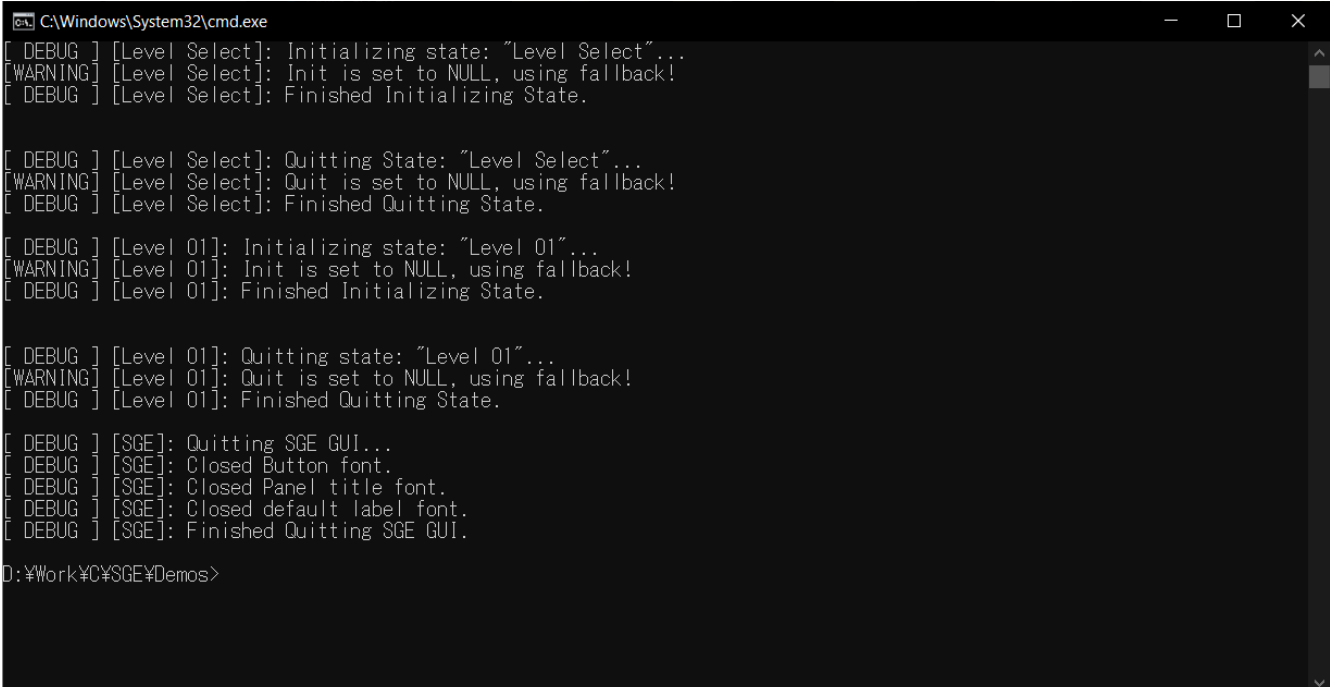
This example program will start with a green screen, indicating our level selection screen and switch to a blue screen, to indicate Level 01 when we press Enter on the keyboard. Again, in this example we are not creating the init, quit and update functions for the states since we aren't doing much more than filling the screen with colors. The level\_01\_State doesn't even need handleEvents as it doesn't do any event handling. But for actual games, you would always use init and quit to load and free assets, and the update functions to update the game logic. How to actually load assets will be covered when we talk about SGE\_Texture and rendering text and images to the screen.

The state switching mechanism is very simple and is implemented as below:

```
void SGE_SwitchToState(SGE_GameState *newState)
{
    currentState->quit();
    currentState = newState;
    currentState->init();
}
```

This just calls the quit function for the current state, freeing all memory loaded by it's init function when the program started. Then it changes the current state to the new state passed as the parameter, and then calls the init function for this new state so that the resources for it are available when the handleEvents, update and render functions for this state are called in a loop.

Of course, since we didn't write any init and quit functions, no memory for assets was loaded or freed and the calls to init and quit just resulted in the console printing a warning, that a fallback was used. The next section will explain how to use the console window to print information and warnings like this one for your own games.



```
C:\Windows\System32\cmd.exe
[ DEBUG ] [Level Select]: Initializing state: "Level Select"...
[WARNING] [Level Select]: Init is set to NULL, using fallback!
[ DEBUG ] [Level Select]: Finished Initializing State.

[ DEBUG ] [Level Select]: Quitting State: "Level Select"...
[WARNING] [Level Select]: Quit is set to NULL, using fallback!
[ DEBUG ] [Level Select]: Finished Quitting State.

[ DEBUG ] [Level 01]: Initializing state: "Level 01"...
[WARNING] [Level 01]: Init is set to NULL, using fallback!
[ DEBUG ] [Level 01]: Finished Initializing State.

[ DEBUG ] [Level 01]: Quitting state: "Level 01"...
[WARNING] [Level 01]: Quit is set to NULL, using fallback!
[ DEBUG ] [Level 01]: Finished Quitting State.

[ DEBUG ] [SGE]: Quitting SGE GUI...
[ DEBUG ] [SGE]: Closed Button font.
[ DEBUG ] [SGE]: Closed Panel title font.
[ DEBUG ] [SGE]: Closed default label font.
[ DEBUG ] [SGE]: Finished Quitting SGE GUI.

D:\Work\C\SGE\Demos>
```

Picture [2B]: Console output with multiple states



### 3. SGE\_Logger

The SGE\_Logger component allows the use of the console window to be used for debugging by offering functions to print information to the console during run time. We have already seen this logging functionality in use when we start even the most basic SGE program. The console window starts before the game window is created and starts showing information that is output by the engine. This can be seen in the above screenshots of the console window.

The console first prints some information about the SDL library version that the engine was compiled with, as well as the version of SDL that it is dynamically linking against. Next, it prints the C random seed that is being used for this particular run of the program, which is the number of seconds since epoch, this is used for random number generation with the C standard library. Next, it prints information about the SGE\_GUI component being initialized.

After which the starting state is initialized by the engine. This has been seen so far, as the three lines,

```
[ DEBUG ] [My Level]: Initializing state "My Level"...  
[WARNING] [My Level]: Init is set to NULL, using fallback!  
[ DEBUG ] [My Level]: Finished Initializing State.
```

Since there is no init function set, the engine uses a fallback function that calls an SGE\_Logger function to print a warning to the console. When we create an init function and send it to the state with *SGE\_SetStateFunctions()*, that function will be used instead of the fallback and this warning line will disappear from the console output.

Now, let's see how to send our own such messages to the console.

The SGE\_Logger.h file declares the functions that we are interested in. These functions are:

```
void SGE_LogPrintLine(SGE_LogLevel level, const char *format, ...);  
void SGE_LogPrint(SGE_LogLevel level, const char *format, ...);  
void SGE_printf(const char *format, ...);
```

#### ***SGE\_LogPrintLine():***

This function takes in an SGE\_LogLevel as the first parameter, which is an enum that indicates the level of the message you want to print. There are three log levels: DEBUG, WARNING and ERROR. These indicate the type of message you are sending.

The DEBUG level indicates that the message is intended for debugging, so we can use this option when printing messages for inspection such as the values of certain variables. This is also used by the engine, as we have seen for indicating state initialization and other debugging information.

The WARNING level indicates that the message is a warning about something that is wrong with our program but it isn't fatal, in other words the program will keep running without crashing. An example of this that we have seen, is the warning message displayed when we don't send a starting state to *SGE\_Run()*, or when we don't set functions for a state and a fallback is used.

The ERROR level indicates that something has gone wrong and the program has to exit. This could be mean an asset failed to load or some other fatal error that means the program cannot continue running. When we learn to load textures, we will check for errors and use the ERROR level to indicate a failure and exit the program.

If you look at a line in the console, you will notice the structure in which each message is printed.

```
[ DEBUG ] [My Level]: Finished Initializing State.
```

First there are two square brackets , followed by a colon and then, a string that represents the message. The first square bracket indicates the log level of the message, the second bracket indicates which state is active while the message is being sent, and finally, the whole part after the colon is the actual message that was sent to the function as the second parameter.

The second parameter is a formatted string similar to the one we pass to *printf()*. For example, to print the value of a variable of type integer named “number” to the console, we would generally use:

```
printf("Value = %d\n", number);
```

We can do the same with the *SGE\_LogPrintLine()* function. So to print the same message using *SGE\_Logger*, we will use:

```
SGE_LogPrintLine(SGE_LOG_DEBUG, "Value = %d", number);
```

Output:

```
[ DEBUG ] [My Level]: Value = 42
```

Notice how in the second parameter, we don't need to add “\n” at the end to insert a newline. This is because this function adds a newline after printing the message that was passed.

### ***SGE\_LogPrint():***

This function takes the same parameters as *SGE\_LogPrintLine()*, the only difference between the two is that this function does not add a newline at the end of the message, giving us control over how we want to format the output on the console window.

### ***SGE\_printf():***

This function is the exact same as the *printf()* from the C standard library, meaning it will not print the message as a structured line with information about the log level and the current state. It will work the same as including `<stdio.h>` and using *printf()*. The reason this function exists, is because there is a feature in *SGE\_Logger* that allows you to send the output of the console to a file instead of the console window. This means this function should be used instead of the C function, as you can change the current output stream between different messages, and the output will go to the last set output stream. This is further explained in the next section.

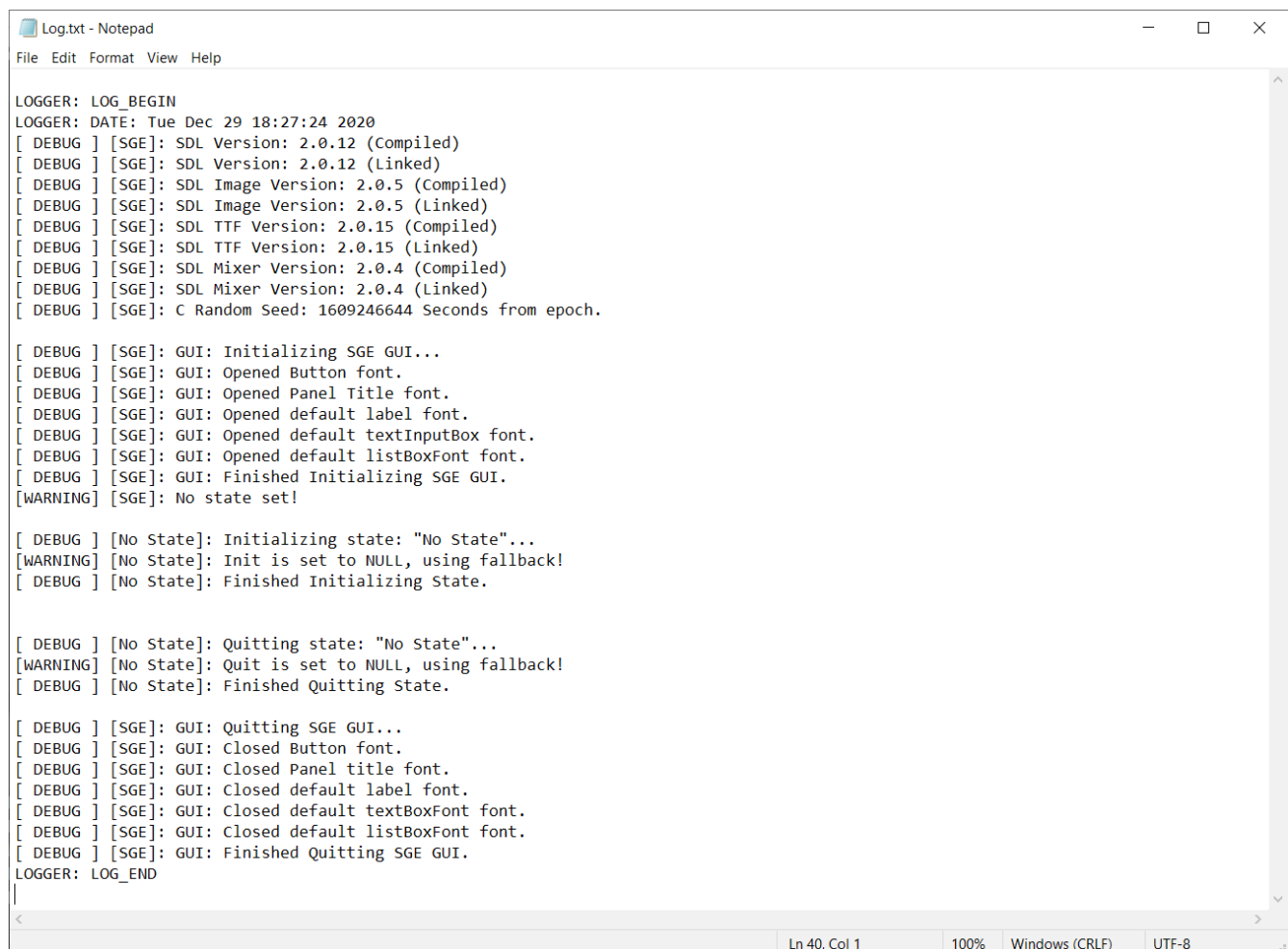
### ***SGE\_LogSetFile(const char \*filePath);***

This function sets the path to a text file to be used as the destination for all console output. The parameter can be a file path or NULL. If a path to a file is specified, all log messages and console output is redirected to this file and written to it as plain text. If the file at the path doesn't exist, it is created. Else if the file already exists, then the output is appended to the contents of the pre-existing file. Below is an example program called `Logger_Demo.c` that demonstrates the use of this function.

```
#include "SGE.h"
#include "SGE_Logger.h"

int main(int argc, char *argv[])
{
    SGE_LogSetFile("Log.txt");
    SGE_Init("SGE Logger Demo", 800, 600);
    SGE_Run(NULL);
    SGE_LogCloseFile();
    return 0;
}
```

Running this program will create a file named "Log.txt" in the same folder as the program executable. The contents of this file will be the messages that would have been printed to the console window. The console window will be empty, since the first thing we do in our program is redirect the output to the log file. The contents of "Log.txt" are shown below.



```
Log.txt - Notepad
File Edit Format View Help

LOGGER: LOG_BEGIN
LOGGER: DATE: Tue Dec 29 18:27:24 2020
[ DEBUG ] [SGE]: SDL Version: 2.0.12 (Compiled)
[ DEBUG ] [SGE]: SDL Version: 2.0.12 (Linked)
[ DEBUG ] [SGE]: SDL Image Version: 2.0.5 (Compiled)
[ DEBUG ] [SGE]: SDL Image Version: 2.0.5 (Linked)
[ DEBUG ] [SGE]: SDL TTF Version: 2.0.15 (Compiled)
[ DEBUG ] [SGE]: SDL TTF Version: 2.0.15 (Linked)
[ DEBUG ] [SGE]: SDL Mixer Version: 2.0.4 (Compiled)
[ DEBUG ] [SGE]: SDL Mixer Version: 2.0.4 (Linked)
[ DEBUG ] [SGE]: C Random Seed: 1609246644 Seconds from epoch.

[ DEBUG ] [SGE]: GUI: Initializing SGE GUI...
[ DEBUG ] [SGE]: GUI: Opened Button font.
[ DEBUG ] [SGE]: GUI: Opened Panel Title font.
[ DEBUG ] [SGE]: GUI: Opened default label font.
[ DEBUG ] [SGE]: GUI: Opened default textInputBox font.
[ DEBUG ] [SGE]: GUI: Opened default listBoxFont font.
[ DEBUG ] [SGE]: GUI: Finished Initializing SGE GUI.
[WARNING] [SGE]: No state set!

[ DEBUG ] [No State]: Initializing state: "No State"...
[WARNING] [No State]: Init is set to NULL, using fallback!
[ DEBUG ] [No State]: Finished Initializing State.

[ DEBUG ] [No State]: Quitting state: "No State"...
[WARNING] [No State]: Quit is set to NULL, using fallback!
[ DEBUG ] [No State]: Finished Quitting State.

[ DEBUG ] [SGE]: GUI: Quitting SGE GUI...
[ DEBUG ] [SGE]: GUI: Closed Button font.
[ DEBUG ] [SGE]: GUI: Closed Panel title font.
[ DEBUG ] [SGE]: GUI: Closed default label font.
[ DEBUG ] [SGE]: GUI: Closed default textBoxFont font.
[ DEBUG ] [SGE]: GUI: Closed default listBoxFont font.
[ DEBUG ] [SGE]: GUI: Finished Quitting SGE GUI.
LOGGER: LOG_END
```

There is a C FILE\* pointer variable called the logStream in Logger.c that is either, set to NULL, or to the address of a file which is opened when LogSetFile() is given a file path. Notice the call to the function *SGE\_LogCloseFile()* at the end of the program. This function must be called before the program exits if a file is set as the current log stream. This is to ensure that the file that is opened by the logStream pointer is closed. Calling this function is the same as passing NULL to the *SGE\_LogSetFile()* function, which just resets the log stream to be the console window.

The reason that *SGE\_printf()* should be used, instead of the standard C *printf()* is because it uses the currently set log stream for output. If the standard C function is used, it will always send output to the console, ignoring the current log stream set by SGE. Nevertheless, the standard C version of *printf()* can still be used for output if it is absolute that the output must be sent directly to the console and not to a log file.