1.1: The Main Function, Part 1

In the Orion Engine, the purpose of the "main" function is to determine the exact conditions the engine is to be run under, and then to spawn the main "loop." The function starts off by starting the clock for benchmarking purposes and instantiating the following variables:

int FPS: The number of frames per second the engine should run in. This should, at max, be the refresh rate of the monitor the game is being run with. In the future, I will find this out programmatically.

bool fullscreen: If true, SDL runs the game in fullscreen

bool debugMode: If true, one of the debug modes will be run. Rather than polling for input from the player through SDL, inputs are read in through a file. This allows past playsessions to be viewed for looking at a bug more closely, observing playtester behavior, or collecting data.

bool keylogger: If true, run the engine in keylogger mode. This causes all input to be encoded in a file so that they may be perfectly recreated later.

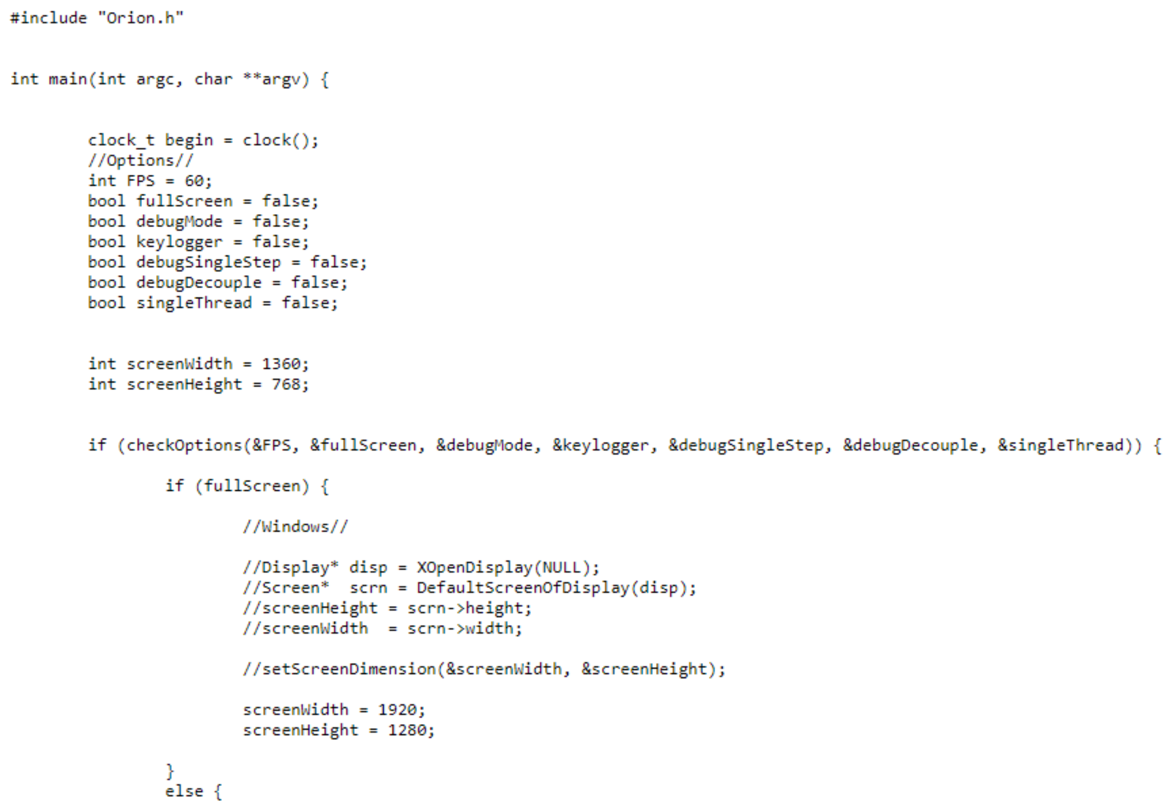
bool debugSingleStep: If debug mode is on, the engine will run in such a way that the spacebar must be pressed to iterate to the next frame. This allows debugging to be conducted much easier; I am able to find the exact frame where the bug shows up.

bool debugDecouple: If debug mode is on, SDL will not be used. This makes using a visual debugger like ddd much easier to use.

bool singleThread: If true, the engine will not use multiple threads. This is handy if I want to run my games through valgrind.

int FPS: The number of frames per second the engine should run in. This should, at max, be the refresh rate of the monitor the game is being run with. In the future, I will find this out programmatically.

After these variables are initialized, they are passed by reference to the "checkOptions" function, which is found in "sdlHandlers.cpp."



```
#include "Orion.h"


int main(int argc, char **argv) {


        clock_t begin = clock();
        //Options//
        int FPS = 60;
        bool fullScreen = false;
        bool debugMode = false;
        bool keylogger = false;
        bool debugSingleStep = false;
        bool debugDecouple = false;
        bool singleThread = false;


        int screenWidth = 1360;
        int screenHeight = 768;


        if (checkOptions(&FPS, &fullScreen, &debugMode, &keylogger, &debugSingleStep, &debugDecouple, &singleThread)) {

                if (fullScreen) {

                        //Windows//

                        //Display* disp = XOpenDisplay(NULL);
                        //Screen*  scrn = DefaultScreenOfDisplay(disp);
                        //screenHeight = scrn->height;
                        //screenWidth  = scrn->width;

                        //setScreenDimension(&screenWidth, &screenHeight);

                        screenWidth = 1920;
                        screenHeight = 1280;

                }
                else {
```

Figure 1: Relevant Code for 1.1

1.2 The “checkOptions” function

The checkOptions function parses the file “Options.txt” for specific keywords that indicate the boolean values from 1.1 being flipped to true. While it might not be ideal, the keywords I used are not the same as the names of the Booleans. They are as follows:

fullscreen – true

fullscreen – false

FPS – FPS followed by numbers

debugMode – debug

logKeys – keylogger

debugSingleStep – dSingleStep

decouple – decouple

singleThread – single

The function operates by reading in input from “Options.txt” one character at a time through the ifstream “options.” This is to account for newlines, and the fact that the target framerate comes immediately after the characters “FPS.” The function continues to read in characters until the end of the file. If “Options.txt” does not exist, the function immediately returns false, true otherwise.

At the end of this function, assuming “Options.txt” exists, the booleans corresponding to the way the engine is to be operated will be initialized.

We then step back up to the main function.

```
bool checkOptions(int * FPS, bool * fullscreen, bool * debugMode, bool * logKeys, bool * debugSingleStep, bool * decouple, bool * singleThread) {

        string line;
        ifstream options("Options.txt");
        stringstream s;

        if (options.is_open()) {

                char c;

                while (options.get(c)) {

                        if (c == '\n') {

                                line.clear();
                        }
                        else {

                                line += c;

                        }

                        if (line.compare("true") == 0) {


                                (*fullscreen) = true;


                        }
                        else if (line.compare("false") == 0) {


                                (*fullscreen) = false;


                        }

                        if (line.compare("FPS") == 0) {

                                line.clear();
                                options.get(c);

                                while (c != '\n') {

                                        line += c;
                                        options.get(c);

                                }

                                *FPS = atoi(line.c_str());

                                line.clear();

                        }

                        if (line.compare("debug") == 0) {


                                (*debugMode) = true;


                        }

                        if (line.compare("keylogger") == 0) {


                                (*logKeys) = true;


                        }
                        if (line.compare("dSingleStep") == 0) {


                                (*debugSingleStep) = true;

                        }

                        if (line.compare("decouple") == 0) {

                                (*decouple) = true;

                        }

                        if (line.compare("single") == 0) {

                                *(singleThread) = true;
                        }

                }

                return true;

                //cout << line << endl;

        }
        else {

                return false;

        }

        return false;

}
```

Figure 2: The checkOptions function

1.3: The Main Function, Part 2

If "Options.txt" did not exist, the FPS is default 60. In the future, I will need to change this to find the refresh rate of the monitor programmatically. The same should be done with the monitor dimensions. If checkOptions returns true and fullscreen is true, the window dimensions are set to the monitor dimensions. This has not been implemented fully yet; I am in the process of having the engine adapt to different operating systems for this step. If fullscreen is false, the screensize will be 1360x768; I will allow the user to change the window dimensions manually when not in fullscreen in the future.

At this point, the engine branches into several gameloops based on function. These take up most of the runtime of the engine. To avoid having many checks for different options in the main game loop, I decided to split off into multiple functions representing the way the engine is to be used as early as possible so as to prevent redundancy.

The "loop" function is chosen if the keylogger and all possible debugModes are turned off. This is what the standard player will be using.

The "loopWrite" function is a standard game loop, but with a keylogger turned on. Every input from every frame WITH input will be logged to a file.

The "loopDebug" function is a gameloop in debug mode. A file is polled for input, and every frame with inputs in the file follows those inputs.

The "loopDebugSingleStep" function is a game loop in debug mode, but the spacebar must be pressed to advance to the next frame.

The "loopDebug_SDLDecoupled" function is a game loop in debug mode, but SDL subsystems are turned off; no visuals or audio will take place (other than through the command line).

The "loopDebugSingleThreaded" function is a game loop in debug mode, but only a single CPU core is used.

```
        if (checkOptions(&FPS, &fullScreen, &debugMode, &keylogger, &debugSingleStep, &debugDecouple, &singleThread)) {

                if (fullScreen) {

                        //Windows//

                        //Display* disp = XOpenDisplay(NULL);
                        //Screen*  scrn = DefaultScreenOfDisplay(disp);
                        //screenHeight = scrn->height;
                        //screenWidth  = scrn->width;

                        //setScreenDimension(&screenWidth, &screenHeight);

                        screenWidth = 1920;
                        screenHeight = 1280;

                }
                else {


                }

        }
        else {

                FPS = 60;

        }

        if (debugMode == false && keylogger == false && debugSingleStep == false && debugDecouple == false) {

                loop(FPS, screenWidth, screenHeight, fullScreen);

        }
        else if (debugMode == false && keylogger == true) {

                loopWrite(FPS, screenWidth, screenHeight, fullScreen);

         }
         else if (debugMode == true && keylogger == false && singleThread == false) {


                 loopDebug(FPS, screenWidth, screenHeight, fullScreen);

         }
         else if (debugSingleStep == true && keylogger == false) {

                 loopDebugSingleStep(FPS, screenWidth, screenHeight, fullScreen);

         }
         else if (debugDecouple == true && keylogger == false) {

                 loopDebug_SDLDecoupled(FPS, screenWidth, screenHeight, fullScreen);

         }
         else if (debugMode == true && keylogger == false && singleThread == true) {

                 loopDebugSingleThreaded(FPS, screenWidth, screenHeight, fullScreen);


         }

         clock_t end = clock();

         double elapsed_secs = double(end - begin)/CLOCKS_PER_SEC;
         cout << "Elapsed Secs: " << elapsed_secs << endl;

         return 0;
}
```

Figure 3: Relevant Code for 1.3

1.4: The loop function:

The loop function has two major parts to it. The first is setting up SDL and the other components needed in the main game loop. The second is the actual gameloop. The first component starts by initializing a number of variables used in the main game loop or in SDL subsystems:

bool hasController is used in the initialization of SDL later to store whether a controller has been plugged in and is recognized.

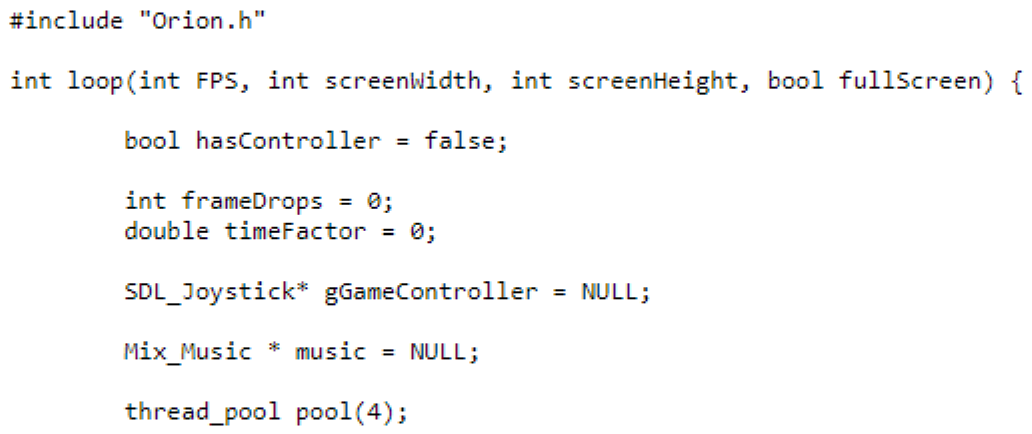
int frameDrops keeps track of the number of times the main game loop took over (1/FPS) seconds to complete. This is mainly used as debug information for myself; if a game has over 2-3 framedrops, this is a signal that something is wrong. I have found that a few framedrops will always be recorded no matter what is being run, but if many are being recorded, I run a profiler and fix what is slowing things down.

double timeFactor, initialized later as 1/(double)FPS, is used in keeping things like movement consistent across different target framerates.

SDL_Joystick * gGameController is used during and after SDL initialization to interact with a game controller.

Mix_Music * music is used during and after SDL initialization for interacting with SDL's audio subsystems.

thread_pool pool is an implementation of a threadpool by Vitaliy Vitsentiy. I intend to build my own implementation later on. I have included their implementation below.

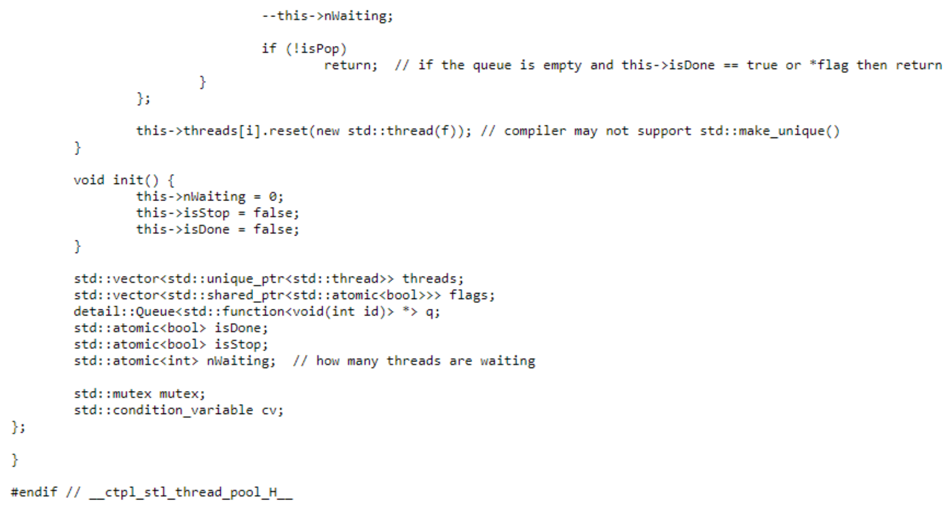


```
#include "Orion.h"

int loop(int FPS, int screenWidth, int screenHeight, bool fullScreen) {

        bool hasController = false;

        int frameDrops = 0;
        double timeFactor = 0;

        SDL_Joystick* gGameController = NULL;

        Mix_Music * music = NULL;

        thread_pool pool(4);
```

Figure 4: Relevant code for 1.4

```
#ifndef __ctpl_stl_thread_pool_H__
#define __ctpl_stl_thread_pool_H__

#include <functional>
#include <thread>
#include <atomic>
#include <vector>
#include <memory>
#include <exception>
#include <future>
#include <mutex>
#include <queue>



// thread pool to run user's functors with signature
//      ret func(int id, other_params)
// where id is the index of the thread that runs the functor
// ret is some return type


namespace ctpl {

namespace detail {
template <typename T>

class Queue {

 public:
        bool push(T const & value) {
                std::unique_lock<std::mutex> lock(this->mutex);
                this->q.push(value);
                return true;
        }

        // deletes the retrieved element, do not use for non integral types
        bool pop(T & v) {
                std::unique_lock<std::mutex> lock(this->mutex);

                if (this->q.empty())
                        return false;

                v = this->q.front();

                this->q.pop();

                return true;
        }

        bool empty() {
                std::unique_lock<std::mutex> lock(this->mutex);
                return this->q.empty();
        }

 private:
        std::queue<T> q;
        std::mutex mutex;
 };
 }


class thread_pool {

public:

        thread_pool() {
                this->init();
        }

        thread_pool(int nThreads) {
                this->init();
                this->resize(nThreads);
        }

        // the destructor waits for all the functions in the queue to be finished
        ~thread_pool() {
                this->stop(true);
        }

        // get the number of running threads in the pool
        int size() {
                return static_cast<int>(this->threads.size());
        }

        // number of idle threads
        int n_idle() {
                return this->nWaiting;
        }

        std::thread & get_thread(int i) {
                return *this->threads[i];
        }

        // change the number of threads in the pool
        // should be called from one thread, otherwise be careful to not interleave, also with this->stop()
        // nThreads must be >= 0
        void resize(int nThreads) {
                if (!this->isStop && !this->isDone) {
                        int oldNThreads = static_cast<int>(this->threads.size());

                        if (oldNThreads <= nThreads) {  // if the number of threads is increased
                this->threads.resize(nThreads);
                this->flags.resize(nThreads);

                for (int i = oldNThreads; i < nThreads; ++i) {
                    this->flags[i] = std::make_shared<std::atomic<bool>>(false);
                    this->set_thread(i);
                }
            }
            else {  // the number of threads is decreased
                for (int i = oldNThreads - 1; i >= nThreads; --i) {
                    *this->flags[i] = true;  // this thread will finish
                    this->threads[i]->detach();
                }

                {
                    // stop the detached threads that were waiting
                    std::unique_lock<std::mutex> lock(this->mutex);
                    this->cv.notify_all();
                }

                this->threads.resize(nThreads);  // safe to delete because the threads are detached

                this->flags.resize(nThreads);  // safe to delete because the threads have copies of shared_ptr of the flags, not originals
            }
        }
    }

    // empty the queue
    void clear_queue() {
        std::function<void(int id)> * _f;

        while (this->q.pop(_f))
            delete _f; // empty the queue
    }

    // pops a functional wrapper to the original function
    std::function<void(int)> pop() {
        std::function<void(int id)> * _f = nullptr;
        this->q.pop(_f);
        std::unique_ptr<std::function<void(int id)>> func(_f); // at return, delete the function even if an exception occurred
        std::function<void(int)> f;

        if (_f)
            f = *_f;

        return f;
    }


    // wait for all computing threads to finish and stop all threads
    // may be called asynchronously to not pause the calling thread while waiting
    // if isWait == true, all the functions in the queue are run, otherwise the queue is cleared without running the functions
    void stop(bool isWait = false) {
        if (!isWait) {
            if (this->isStop)
                return;

            this->isStop = true;

            for (int i = 0, n = this->size(); i < n; ++i) {
            *this->flags[i] = true;  // command the threads to stop
        }

        this->clear_queue();  // empty the queue
    }
    else {
        if (this->isDone || this->isStop)
            return;

        this->isDone = true;  // give the waiting threads a command to finish
    }

    {
        std::unique_lock<std::mutex> lock(this->mutex);
        this->cv.notify_all();  // stop all waiting threads
    }

    for (int i = 0; i < static_cast<int>(this->threads.size()); ++i) {  // wait for the computing threads to finish
        if (this->threads[i]->joinable())
            this->threads[i]->join();
    }

    // if there were no threads in the pool but some functors in the queue, the functors are not deleted by the threads
    // therefore delete them here
    this->clear_queue();

    this->threads.clear();

    this->flags.clear();
}
```

```
 template<typename F, typename... Rest>
 auto push(F && f, Rest&&... rest) ->std::future<decltype(f(0, rest...))> {
                auto pck = std::make_shared<std::packaged_task<decltype(f(0, rest...))(int)>>(
                        std::bind(std::forward<F>(f), std::placeholders::_1, std::forward<Rest>(rest)...)
                );
                auto _f = new std::function<void(int id)>([pck](int id) {
                        (*pck)(id);
                }

                                                                                  );
                this->q.push(_f);
                std::unique_lock<std::mutex> lock(this->mutex);
                this->cv.notify_one();
                return pck->get_future();
        }

        // run the user's function that excepts argument int - id of the running thread. returned value is templatized
        // operator returns std::future, where the user can get the result and rethrow the catched exceptins
        template<typename F>
        auto push(F && f) ->std::future<decltype(f(0))> {
                auto pck = std::make_shared<std::packaged_task<decltype(f(0))(int)>>(std::forward<F>(f));
                auto _f = new std::function<void(int id)>([pck](int id) {
                        (*pck)(id);
                }

                                                                                  );
                this->q.push(_f);
                std::unique_lock<std::mutex> lock(this->mutex);
                this->cv.notify_one();
                return pck->get_future();
        }



private:

        // deleted
        thread_pool(const thread_pool &);// = delete;
        thread_pool(thread_pool &&);// = delete;
        thread_pool & operator=(const thread_pool &);// = delete;
        thread_pool & operator=(thread_pool &&);// = delete;

        void set_thread(int i) {
                std::shared_ptr<std::atomic<bool>> flag(this->flags[i]); // a copy of the shared ptr to the flag
                auto f = [this, i, flag/* a copy of the shared ptr to the flag */]() {
                        std::atomic<bool> & _flag = *flag;
                        std::function<void(int id)> * _f;
                        bool isPop = this->q.pop(_f);

                        while (true) {
                                while (isPop) {  // if there is anything in the queue
                                        std::unique_ptr<std::function<void(int id)>> func(_f); // at return, delete the function even if an exception occurred
                                        (*_f)(i);

                                        if (_flag)
                                                return;  // the thread is wanted to stop, return even if the queue is not empty yet
                                        else
                                                isPop = this->q.pop(_f);
                                }

                                // the queue is empty here, wait for the next command
                                std::unique_lock<std::mutex> lock(this->mutex);

                                ++this->nWaiting;

                                this->cv.wait(lock, [this, &_f, &isPop, &_flag]() {
                                        isPop = this->q.pop(_f);
                                        return isPop || this->isDone || _flag;
                                }
                                                                  );
```



```
                                --this->nWaiting;

                                if (!isPop)
                                        return;  // if the queue is empty and this->isDone == true or *flag then return
                        }
                };

                this->threads[i].reset(new std::thread(f)); // compiler may not support std::make_unique()
        }

        void init() {
                this->nWaiting = 0;
                this->isStop = false;
                this->isDone = false;
        }

        std::vector<std::unique_ptr<std::thread>> threads;
        std::vector<std::shared_ptr<std::atomic<bool>>> flags;
        detail::Queue<std::function<void(int id)> *> q;
        std::atomic<bool> isDone;
        std::atomic<bool> isStop;
        std::atomic<int> nWaiting;  // how many threads are waiting

        std::mutex mutex;
        std::condition_variable cv;
};

}

#endif // __ctpl_stl_thread_pool_H__
```

Figure 5: Vitaliy Vitsentiy's implementation of a queue and thread pool

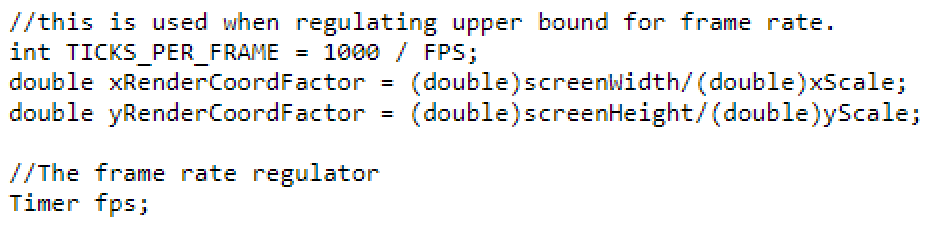
---

1.4.1: The loop function (contd):

int TICKS_PER_FRAME is used in regulating framerate, and is set to 1000/FPS.

double xRenderCoordFactor and yRenderCoordFactor are used for scaling movement for different screen resolutions. This feature is a work in progress and still has some bugs.

Timer fps is an instance of the Timer class as written by LazyFooProductions (credit to them for their excellent tutorials on SDL). It is used for measuring time using SDL subsystems. I use it in regulating framerate, measuring the length of a frame in particular.



```
//this is used when regulating upper bound for frame rate.
int TICKS_PER_FRAME = 1000 / FPS;
double xRenderCoordFactor = (double)screenWidth/(double)xScale;
double yRenderCoordFactor = (double)screenHeight/(double)yScale;

//The frame rate regulator
Timer fps;
```

Figure 6: Relevant code for 1.4.1

```
#include "Timer.h"

Timer::Timer() {
        //Initialize the variables
        startTicks = 0;
        pausedTicks = 0;
        paused = false;
        started = false;
}

void Timer::start() {
        //Start the timer
        started = true;

        //Unpause the timer
        paused = false;

        //Get the current clock time
        startTicks = SDL_GetTicks();
}

void Timer::stop() {
        //Stop the timer
        started = false;

        //Unpause the timer
        paused = false;
}
void Timer::pause() {
        //If the timer is running and isn't already paused
        if ((started == true) && (paused == false)) {
                //Pause the timer
                paused = true;

                //Calculate the paused ticks
                pausedTicks = SDL_GetTicks() - startTicks;
        }
}

void Timer::unpause() {
        //If the timer is paused
        if (paused == true) {
                //Unpause the timer
                paused = false;

                //Reset the starting ticks
                startTicks = SDL_GetTicks() - pausedTicks;

                //Reset the paused ticks
                pausedTicks = 0;
        }
}

int Timer::get_ticks() {
        //If the timer is running
        if (started == true) {
                //If the timer is paused
                if (paused == true) {
                        //Return the number of ticks when the timer was paused
                        return pausedTicks;
                }
                else {
                        //Return the current time minus the start time
                        return SDL_GetTicks() - startTicks;
                }
        }

        //If the timer isn't running
        return 0;
}

bool Timer::is_started() {
        return started;
}

bool Timer::is_paused() {
        return paused;
}
```

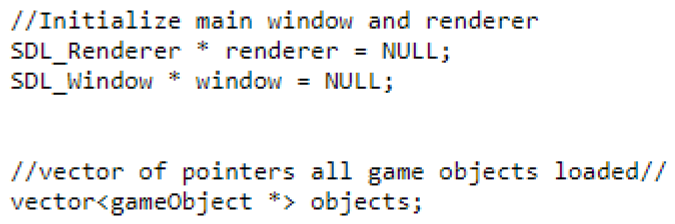
Figure 7: LazyFooProduction's Timer class

1.4.2: The loop function (contd):

SDL_Renderer * renderer is SDL's renderer. It is initialized later in the init function.

SDL_Window * window is an SDL class handling the game window.

Vector<gameObject *> objects is a vector intended to hold every instance of a gameObject in the game at a time. A gameObject is a class of my own representing any entity in the game. The vector holds pointers to these objects rather than the objects themselves due to the fact that they hold virtual functions.



```
//Initialize main window and renderer
SDL_Renderer * renderer = NULL;
SDL_Window * window = NULL;


//vector of pointers all game objects loaded//
vector<gameObject *> objects;
```

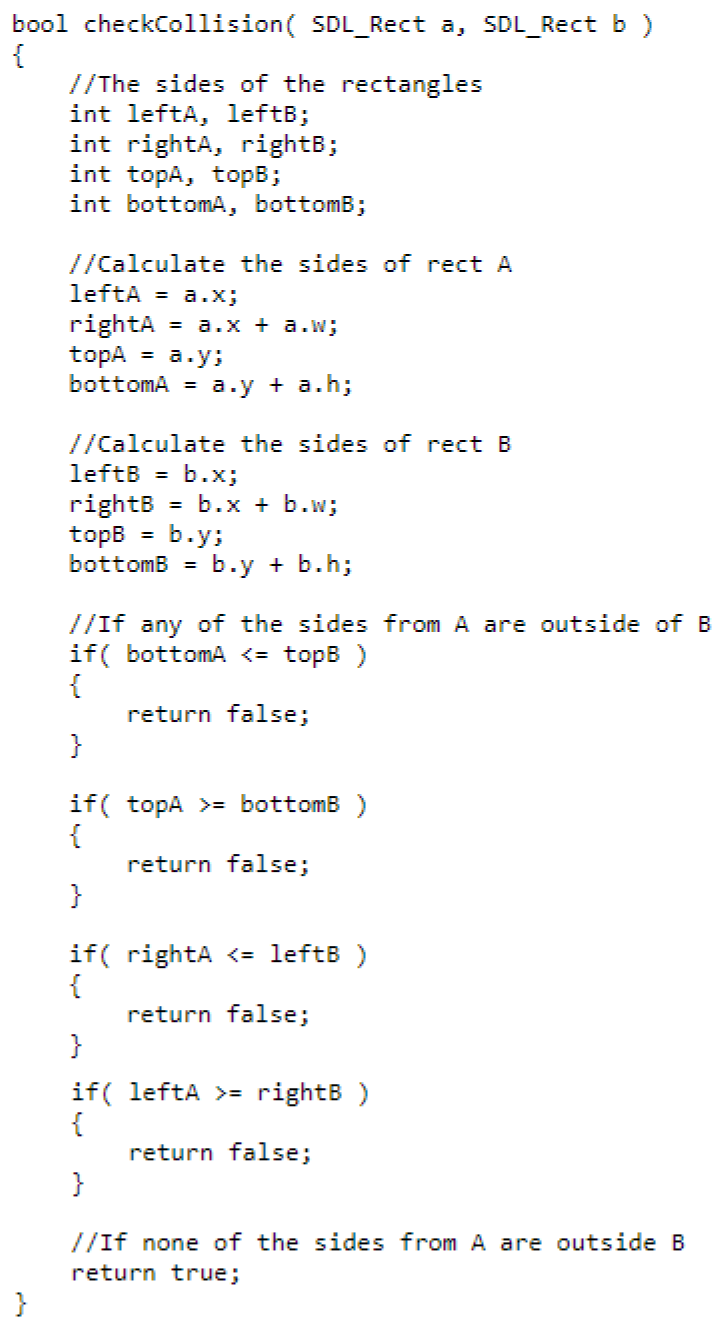
Figure 8: Relevant code for 1.4.2

2: The gameObject Class Part 1

A gameObject is a superclass for any entity in the game. It will generally have visuals associated with it, but it doesn't have to. A gameObject could be used to group multiple gameObjects together and coordinate their behavior. The gameObject class includes a good number of variables and functions that most, if not all, subclasses of gameObject will need (ex: the coordinates X and Y). The Orion Engine is designed around the game entities' behavior being in the virual functions 'handleInput', 'handleStateChanges', and 'enactStateChanges'. handleInput should include all the logic needed to make the game entity do what is in the vector 'inputs'. This will be elaborated upon when discussing the 'handleEvents' and 'addInput' functions. The handleStateChanges and enactStateChanges functions are based on a unique property of how game logic works. All changes in state for every entity in the game must happen at the exact same time. This is due to the fact that the logic determining how an object acts might be dependent on the state of another object. To give a simple example, let's say we had an engine where state changes happen in an arbitrary order and are only handled by a simple function, and, in this engine, we programmed a baseball video game. The ball would be thrown and the bat would swing and hit it. Let's say that, in this engine, the ball logic was processed first. The ball object would determine that it is touching the bat through some collision detection algorithm, figure out that the bat is in the "swinging" state, which in turn would register a hit. Since, in this theoretical engine there is only one function handling state change logic, the ball must start to fly away after being hit. The X and Y coordinates of the ball would be changed immediately. Now, let's say that the player's controller should vibrate when they hit the ball, and that this is handled in the bat object's logic. The bat object should follow a similar logic as the ball; when it touches the ball and is in the “swinging” state, a hit is registered. However, the ball has already started flying away; its X and Y coordinates are out of range of the bat. Therefore, no hit will be registered for the bat and the controller will not vibrate. The solution to this problem is having two functions handling state changes. The first should log what state changes should occur but not actually change the variables. The way I do this is have a corresponding “future” variable (ex: futureX, future) for each variable that needs to be protected. In the Orion Engine, this is handled in the handleStateChanges function. Then, in the enactStateChanges function, all variables labeled “future” are copied over to their counterparts. This is also where the “move” function of a gameObject is placed, which changes the object’s coordinates, as well as its hitboxes’ coordinates.

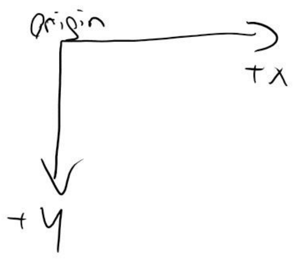
2.1: The hitbox Structure

A key structure in almost every gameObject is the hitbox. A hitbox is simply a structure keeping track of the axis of a box relative to the gameObject. It is used to capture all of or part of the area represented by a gameObject in the game space. Using the baseball example, the hitbox for the baseball would be the square covering all of the baseball as tightly as possible. Then, to check for a collision between the ball and the bat, a simple algorithm to compare the axis’ between the ball and bat hitboxes is used. The following code is from lazyfoo.net.

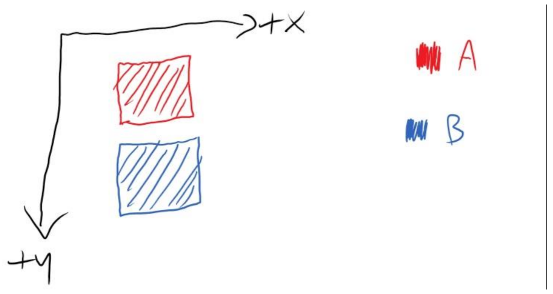


```
bool checkCollision( SDL_Rect a, SDL_Rect b )
{
    //The sides of the rectangles
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    //Calculate the sides of rect A
    leftA = a.x;
    rightA = a.x + a.w;
    topA = a.y;
    bottomA = a.y + a.h;

    //Calculate the sides of rect B
    leftB = b.x;
    rightB = b.x + b.w;
    topB = b.y;
    bottomB = b.y + b.h;

    //If any of the sides from A are outside of B
    if( bottomA <= topB )
    {
        return false;
    }

    if( topA >= bottomB )
    {
        return false;
    }

    if( rightA <= leftB )
    {
        return false;
    }
    if( leftA >= rightB )
    {
        return false;
    }

    //If none of the sides from A are outside B
    return true;
}
```

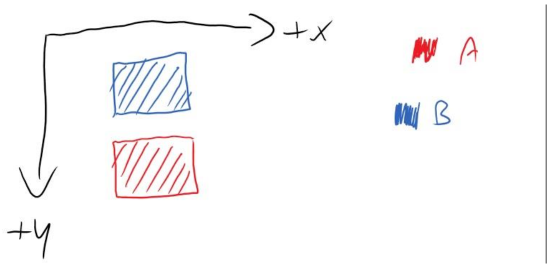
Only four comparisons need to be made between the sides of the hitbox in order to determine if they are overlapping. Note that in this scenario, a coordinate system like the following is used:



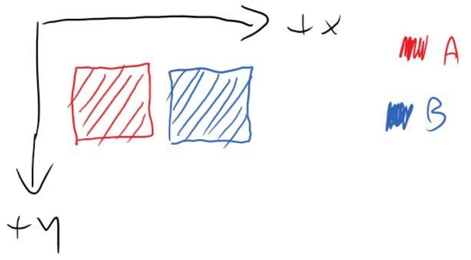
The first comparison compares the bottom of hitbox A and the top of hitbox B. If the bottom of hitbox A is less than or equal to the top of hitbox B, there is no scenario in which the two hitboxes can be colliding.



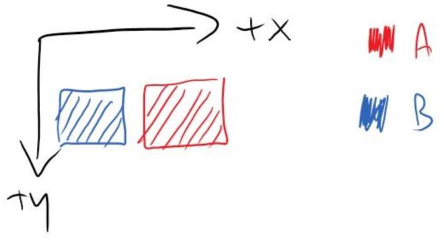
The second comparison compares the top of hitbox A with the bottom of hitbox B. If the top of hitbox A is greater than or equal to the bottom of B, there is no scenario in which the hitboxes can be colliding.



The third comparison compares the right side of hitbox A with the left side of hitbox B. If the right side of hitbox A is less than or equal to the left side of hitbox B, there is no scenario in which the hitboxes can be colliding.



The fourth comparison compares the left side of hitbox A and the right side of hitbox B. If the left side of hitbox A is greater than or equal to the right side of hitbox B, there is no scenario in which the hitboxes can be colliding.



If any of these comparisons are true, the function returns false. Otherwise, it returns true, indicating a collision.

In my implementation of a hitbox, all axis’ are unsigned integers; an axis should never have a negative value as it will lead to a segmentation fault in my data structure handling collision detection.

I also include the axis of the hitbox in the previous frame. This allows a fancy trick to be used to speed up collision detection; this will be explained when discussing my collision detection data structure.

Each hitbox has its own ID represented by an integer. This makes searching for a specific hitbox in my collision detection data structure much faster compared to a string ID.

Every hitbox stores the name and type of its parent (the gameObject this hitbox is attached to). Every gameObject has a name, a unique identifier, and a type, an identifier that may be shared. A ball gameObject might have the name “ball_1” and type “ball”. This is used in my data structure for collision detection; the gameObjects themselves are not stored in this data structure, but the hitboxes. “name” and “type” are stored so that, in the event of a collision, a hitbox knows exactly what it collided with.