2020

Autonomous Route and Mapping 2D - 3D Lidar Scanning

TECHICAL MANUAL

CALIN DORAN

Table of Contents

1. Introduction

The purpose of this Technical Manual is to document, outline the requirements, and to show the installation of the application. This will also show all relevant code for the A.R.M Lidar System. The current state of the application is still heavily in development, however, the link to the project as well as the SDK and driver for the Slamtec Lidar is included in this GitHub repository.

The C++ code for this document is displayed as is in Visual Studio 2019 to provide the most accurate representation of the current state of the project. Some code has been omitted from this document as it is either code taken from an external source, which has been referenced in the file itself or documented here. To see the full code please see the GitHub project linked below.

All A.R.M Lidar System code can be found at the link below:

https://github.com/calindoran/year4Project

2. Project Code

2.1 Shared project code

Some files were the result of a combination fo code take from multiple tutorials on C++. Most files in the helpers folder are functions taken from multiple sites that provide assistance in solving problems regarding C++, i.e stackoverflow.com, habr.com, isocpp.org and cplusplus.com.

2.2 src/main.cpp

```
#include <sfml/graphics.hpp>
#include <sfml/window.hpp>
#include <sfml/system.hpp>

#include <iostream>
#include <string>

#include "..\lidar_draws\window_constants.h"
#include "..\lidar_draws\lidarscene.h"
#include "..\sfml_wrap\winrend.h"
#include "..\lidar\lidardevicewrapper.h"
#include "..\lidar\lidar_params.h"

using namespace std;

int main(int argc, const char* argv[])
{
    std::cout << "A.R.M LIDAR System" << std::endl;
    std::cout << "Version: " << RPLIDAR_SDK_VERSION << std::endl;
    sf::ContextSettings settings;
    settings.antialiasingLevel = 8;

    sf::RenderWindow window(sf::VideoMode(DESIGNED_WIDTH, DESIGNED_HEIGHT), "A.R.M Lidar System", sf::Style::Default, settings);

    //TODO: add something like boost command line parser, so can use like
    //-d com3 -b 300

    LidarParams params;
    //this is delay between lidar readings, so 1 reading will be at least this value or slower
    //it is not the same as draw frame rate, those 2 things are independant +/-
    params.update_delay_ms = 100;

    if (argc > 1)
        params.device = std::string(argv[1]); // or set to a fixed value: e.g. "com3"

    if (argc > 2)
    {
        try
        {
            params.bauds = std::max(0, std::stoi(argv[2])); // no negatives
        }
        catch (std::invalid_argument const& e)
        {
            std::cerr << "Bad input: std::invalid_argument thrown" << '\n';
        }
        catch (std::out_of_range const& e)
        {
            std::cerr << "Integer overflow: std::out_of_range thrown" << '\n';
        }
    }

    LidarScene scene(params);
    return WinRend::Main(window, scene, false, 60.f);
}
```

2.3 lidar/lidardevicewrapper.cpp

```
#include "lidardevicewrapper.h"
#include <iostream>
#include <exception>
#include <sstream>
#include <iomanip>
#include <algorithm>

//TODO: change this to any object (stream) which has overloaded << operator
#define ERROR_OUT std::cerr

LidarDeviceWrapper::LidarDeviceWrapper(const LidarParams& params) :
    orig_params(params),
    devinfo{}
{
    init(params.device, params.bauds);
}

LidarDeviceWrapper::~LidarDeviceWrapper()
{
    cleanup();
}

std::string LidarDeviceWrapper::toString() const
{
    std::stringstream cout;
    //Print out the device serial number, firmware and hardware version number
    cout << "RPLIDAR S/N: ";
    for (const auto v : devinfo.serialnum)
        cout << std::uppercase << std::setfill('0') << std::setw(2) << std::hex << static_cast<int>(v);
    cout << std::endl;
    cout << "Firmware Ver: " << devinfo.firmware_version << std::endl;
    cout << "Hardware Rev: " << static_cast<int>(devinfo.hardware_version);

    return cout.str();
}

bool LidarDeviceWrapper::checkRPLIDARHealth() const
{
    bool res = false;
    if (drv)
    {
        rplidar_response_device_health_t healthinfo;
        const auto op_result = drv->getHealth(healthinfo);
        if (IS_OK(op_result))
            res = (healthinfo.status != (RPLIDAR_STATUS_ERROR));
        else
            ERROR_OUT << "Error, cannot retrieve the lidar health code: " << op_result << std::endl;
    }
    else
        ERROR_OUT << "Trying to check health for not connected device." << std::endl;
    return res;
}
```

```
bool LidarDeviceWrapper::init(std::string dev_path, const uint32_t baud_rate)
{
    cleanup();

    if (dev_path.empty())
    {
#ifdef _WIN32
        // use default com port
        // opt_com_path = "\\\\.\\com57";
        dev_path = "COM3";
#elif __APPLE__
        dev_path = "/dev/tty.SLAB_USBtoUART";
#else
        dev_path = "/dev/ttyUSB0";
#endif
    }
    std::vector<uint32_t> bauds{ 115200, 256000 };
    if (baud_rate)
    {
        bauds.clear();
        bauds.push_back(baud_rate);
    }

    for (const auto br : bauds)
    {
        //FIXME: not sure why allocation is for each baud
        //took from example, move prior the loop maybe
        drv = allocLidarDriver();
        if (!drv)
            throw std::runtime_error("Failed to allocate driver!");
        if (IS_OK(drv->connect(dev_path.c_str(), br)) && IS_OK(drv->getDeviceInfo(devinfo)))
            break;
    }

    const bool ok = drv && drv->isConnected() && checkRPLIDARHealth();
    if (!ok)
        ERROR_OUT << "Error, cannot bind to the specified serial port: " << dev_path << std::endl;
    else
        std::cout << toString() << std::endl;
    return ok;
}

void LidarDeviceWrapper::cleanup()
{
    stopScan();
    drv.reset();
    memset(&devinfo, 0, sizeof(devinfo));
}

void LidarDeviceWrapper::stopScan() const
{
    if (drv)
    {
        //stop...
        drv->stop();
        //stop motor...
        drv->stopMotor();
    }
}
```

```
void LidarDeviceWrapper::runScan(uint32_t options) const
{
    if (drv)
    {
        //start motor...
        drv->startMotor();
        //start scan...
        drv->startScan(false, true, options);
    }
}

LidarValuesVector LidarDeviceWrapper::readOnce(size_t count) const
{
    LidarValuesVector res;
    if (drv)
    {
        pools::PooledVector<rplidar_response_measurement_node_hq_t> nodes;
        nodes.resize(count);
        if (IS_OK(drv->grabScanDataHq(nodes.data(), count)))
        {
            nodes.resize(count); //make vector of same size as returned
            drv->ascendScanData(nodes.data(), count);
            res.reserve(count); //making memory allocation in front, so loop is fast
            std::transform(std::begin(nodes), std::end(nodes), std::back_inserter(res), [](const auto& v)
                {
                    return LidarValues::fromDriverData(v);
                });
        }
    }
    return res;
}

bool LidarDeviceWrapper::testHealthAndReinitIfNeed()
{
    return checkRPLIDARHealth() || init(orig_params.device, orig_params.bauds);;
}
```

2.4 lidar/lidardevicewrapper.h

```
#pragma once

#include <string.h>
#include "..\helpers\cm_ctors.h"
#include "lidar_device.h"
#include "lidar_data.h"
#include "lidar_params.h"

//this class wraps LidarDriver to our task

class LidarDeviceWrapper
{
private:
    RPlidarDriverPtr drv{ nullptr };
    rplidar_response_device_info_t devinfo;

    LidarParams orig_params;
    bool init(std::string dev_path, const uint32_t baud_rate);

public:
    NO_COPYMOVE(LidarDeviceWrapper);
    LidarDeviceWrapper() = delete;
    LidarDeviceWrapper(const LidarParams& params);
    ~LidarDeviceWrapper();

    bool checkRPLIDARHealth() const;
    bool testHealthAndReinitIfNeed();
    void cleanup();

    void runScan(uint32_t options = 0) const;
    void stopScan() const;

    //grabs scan data of count samples, i.e. 1 full circle divided into count pieces
    LidarValuesVector readOnce(size_t count = 8192) const;

    std::string toString() const;
};
```

2.4 lidar/lidar_params.h

```
#pragma once

#include <stdint.h>
#include <string>

struct LidarParams
{
    std::string device;
    uint32_t bauds{ 0 };

    //this is delay between 2 lidar readings as it goes in parallel
    uint64_t update_delay_ms{ 1000 / 60 };
};
```

2.5 lidar/lidar_device.h

```
#pragma once

#include <memory>
#include <memory.h>
#include <algorithm>
#include "rplidar.h" //RPLIDAR standard sdk, all-in-one header

using namespace rp::standalone::rplidar;
using RPlidarDriverPtr = std::shared_ptr<rp::standalone::rplidar::RPlidarDriver>;

inline auto allocLidarDriver()
{
    return RPlidarDriverPtr(RPlidarDriver::CreateDriver(DRIVER_TYPE_SERIALPORT), [](RPlidarDriver* p)
        {
            if (p)
                RPlidarDriver::DisposeDriver(p);
        });
}
```

2.6 lidar/lidar_data.h

```
#pragma once

#include <stdint.h>
#include <sstream>

#include "..\helpers\cm_ctors.h"
#include "..\helpers\type_checks.h"
#include "..\helpers\pooled_shared.h"

class LidarValues
{
public:
    //TODO: update here to match SDK / sense
    using flag_t = uint32_t;
    using qual_t = uint32_t;
    using float_t = float;

    flag_t  rFlag{ 0 };
    float_t rAngle{ 0.f };
    float_t rDistance{ 0.f };
    qual_t  rQuality{ 0 };

public:
    DEFAULT_COPYMOVE(LidarValues);

    LidarValues() = default;
    ~LidarValues() = default;

    LidarValues(flag_t rFlag, float_t rAngle, float_t rDistance, qual_t rQuality) :
        rFlag(rFlag), rAngle(rAngle), rDistance(rDistance), rQuality(rQuality)
    {
    }

    bool operator < (const LidarValues& c) const
    {
        return rAngle < c.rAngle;
    }

    //doing template so compiler will accept anything with proper fields present
    template <class T>
    static inline LidarValues fromDriverData(const T& data) noexcept
    {
        constexpr static auto div = static_cast<float_t>(static_cast<uint32_t>(1) << 14);
        LidarValues r;
        CASTSET2FIELD(r.rFlag, data.flag);
        CASTSET2FIELD(r.rAngle, data.angle_z_q14 * 90.f / div);
        CASTSET2FIELD(r.rDistance, data.dist_mm_q2 / 4.0f);
        CASTSET2FIELD(r.rQuality, data.quality);
        return r;
    }

    std::string toString() const
    {
        std::stringstream cout;
        cout << "Flag: " << rFlag << " | Angle: " << rAngle << " | Distance: " << rDistance << " | Quality: " << rQuality;
        return cout.str();
    }
};

//checking moving is allowed, so compiler will do fast code. on some changes it may get prohibited
TEST_MOVE_NOEX(LidarValues);

using LidarValuesQueue = pools::PooledDeque<LidarValues>;
using LidarValuesVector = pools::PooledVector<LidarValues>;
```

2.7 lidar_draws/lidarcontainer.cpp

```
#include <SFML/Graphics/RectangleShape.hpp>
#include <SFML/Graphics/RenderWindow.hpp>
#include "lidarcontainer.h"
#include "..\lidar\lidardevicewrapper.h"
#include "..\helpers\block_delay.h"
#include "..\helpers\guard_on.h"
#include "..\helpers\containers_helpers.h"

constexpr static float minDist = 200;
constexpr static float maxDist = 600;
constexpr static float halfFOV = 30.f; //half of field-of-view in degrees
constexpr static uint32_t initial_nodes_amount = 1024;//that was 8192 reduced for better managment
constexpr static size_t do_health_check_each_N_reads = 100;

LidarContainer::LidarContainer(const LidarParams& params) :
    scanDensity(initial_nodes_amount)
{
    using DelayMeasuredIn = std::chrono::milliseconds;

    //this is lidar reader thread
    lidarThread = utility::startNewRunner([this, params](const auto need2stop)
        {
            const DelayMeasuredIn DELAY = std::chrono::duration_cast<DelayMeasuredIn>(std::chrono::milliseconds(params.update_delay_ms));
            try
            {
                //creating device inside thread as many OS dislike cross-thread handles
                LidarDeviceWrapper lidar(params);
                lidar.runScan();
                size_t counter = 0;
                while (!(*need2stop)) //checking if program terminates (need2stop is shared pointer to boolean value)
                {
                    //this will ensure 1 iteration takes at least DELAY
                    DelayBlockMs<DelayMeasuredIn> delay(DELAY);
                    (void)delay;

                    //each couple steps lets check health and restart / reconnect if needed
                    if (((++counter) % do_health_check_each_N_reads) == 0)
                    {
                        lidar.testHealthAndReinitIfNeed();
                        lidar.runScan();
                    }

                    auto readings = lidar.readOnce(scanDensity.load());
                    if (readings.size())
                    {
                        const auto static cannot_see = [](const LidarValues& v)->bool
                        {
                            //took "can see" from example and inverted
                            return !(v.rDistance > minDist && v.rDistance < maxDist && (v.rAngle < halfFOV || v.rAngle > 360 - halfFOV));
                        };

                        //removing all readings we cannot use
                        types_ns::remove_if(readings, cannot_see);

                        //making left side angles negatives, so it is proper sorted
                        std::transform(std::begin(readings), std::end(readings), std::begin(readings), [](LidarValues& a)
                            {
                                static_assert(halfFOV < 45.f, "FOV must be less then 90 degree.");
                                if (a.rAngle >= 270.f)
                                    a.rAngle -= 360.f;
                                return a;
                            });

                        //sort by angle as LidarValues has comparator by angle
                        std::sort(std::begin(readings), std::end(readings));
```

```
                    //then sort by distance in reverse order
                    std::sort(std::begin(readings), std::end(readings), [](const LidarValues& a, const LidarValues& b)
                        {
                            return a.rDistance > b.rDistance;
                        });

                    //pushing values to "global", so drawer thread may use it
                    LOCK_GUARD_ON(valuesLock);
                    lastToDraw.swap(readings);
                }
                else
                {
                    counter = do_health_check_each_N_reads - 1; //no readings, try to restart ASAP

                    //FIXME: if lidar fails to read drop whats shown, however can comment out 2 lines below
                    //then drawer will show last update permanent
                    LOCK_GUARD_ON(valuesLock);
                    lastToDraw.clear();
                }

                //here pause may happen issued by delay destructor
            }
        }
        catch (...)
        {
            std::cerr << "Exception in lidar reader thread." << std::endl;
            std::exit(255);
        }
    });
}

LidarContainer::~LidarContainer()
{
    lidarThread.reset(); //terminating thread
}

void LidarContainer::update(Scene& scene, float time, const InputSource& input)
{
    (void)scene;
    (void)time;
    (void)input;
    //TODO: here can do something which changes draw logic, for example zoom by keyboard

    //example if initial_nodes_amount is 1024
    const bool plus = input.getNavigator()->isTop();
    const bool minus = input.getNavigator()->isBottom();
    if (plus != minus)
    {
        auto val = scanDensity.load();
        val += (plus) ? 3 : -5;
        scanDensity = std::max(10u, std::min(8192u * 2, val));
    }
}

void LidarContainer::draw(sf::RenderTarget& where, sf::RenderStates states) const
{
    //this function generates visuals accodring to data and draws it. Visual objects are not stored.
    //expect lidar thread pushed only visible items to here, so it wont spend time on calculations

    //doing local copy of latest lidar readings. so lidar reader can keep update for
    //also need a copy (not move) because may draw faster then lidar makes new data, so we need to draw something
    //a must of {}, keeping thread lock short
    LidarValuesVector curr;
    {
        LOCK_GUARD_ON(valuesLock);
        curr = lastToDraw;
    }
```

```
    const sf::Vector2f drawSz(size.x * scale.x, size.y * scale.y);

    const float x_per_degree = drawSz.x / (2 * halfFOV);
    const float y_mid = drawSz.y / 2.f;
    const float distance_0_size = std::fmin(drawSz.x, drawSz.y) * 0.9f;

    //drawing, making visual rectangle for each piece of data

    const static sf::Color colors[] = { sf::Color::Red, sf::Color::Green, sf::Color::Blue };
    size_t clr_index = 0;
    for (const auto& v : curr)
    {
        sf::RectangleShape rectangle;
        rectangle.setSize({ 1.0f, 1.0f });
        rectangle.setOrigin(0.5f, 0.5f);

        rectangle.setFillColor(colors[clr_index++]);
        clr_index = clr_index % types_ns::countof(colors);

        const float sz = distance_0_size / ((v.rDistance - minDist) - 1.f);
        rectangle.scale(sz, sz);
        rectangle.setPosition(x_per_degree * (v.rAngle + halfFOV), y_mid);

        where.draw(rectangle, states);
    }
}

void LidarContainer::setScreenSize(float width, float height)
{
    size = sf::Vector2f(width, height);
}

void LidarContainer::setScale(float mx, float my)
{
    scale = sf::Vector2f(mx, my);
}
```

2.8 lidar_draws/lidarcontainer.h

```
#pragma once

#include <mutex>
#include <atomic>
#include "..\helpers\spinlock.h"
#include "..\sfml_wrap\drawableentity.h"
#include "..\helpers\runners.h"
#include "..\lidar\lidar_params.h"
#include "..\lidar\lidar_data.h"
#include <SFML/System/Vector2.hpp>

//this class completely deals with lidar - draws anything inside own borders
//creates driver and access it

class LidarContainer : public DrawableEntity
{
private:
    utility::runner_t lidarThread{ nullptr };

    mutable spinlock valuesLock;
    LidarValuesVector lastToDraw;
    sf::Vector2f size{ 0.f, 0.f };
    sf::Vector2f scale{ 1.f, 1.f };

    //this is not requested example of communication with program
    std::atomic<uint32_t> scanDensity;
public:
    LidarContainer(const LidarParams& params);
    ~LidarContainer() override;

    void update(Scene& scene, float time, const InputSource& input) override;
    void draw(sf::RenderTarget& where, sf::RenderStates states) const override;

    virtual void setScreenSize(float width, float height);
    virtual void setScale(float mx, float my);
};
```

2.9 lidar_draws/lidarscene.cpp

```
#include "lidarscene.h"
#include "lidarcontainer.h"
#include "..\helpers\pooled_shared.h"

LidarScene::LidarScene(const LidarParams& params)
{
    auto pc = pools::allocShared<LidarContainer>(params);
    entities.push_back(pc);
}

void LidarScene::viewWasChanged(const sf::View& view) const
{
    const sf::Vector2f curSize(view.getSize()); //copy, to make sure it remains while we're in function
    for (auto& e : entities)
        e->setScreenSize(curSize.x, curSize.y);
}
```

2.10 lidar_draws/lidarscene.h

```
#pragma once

#include "..\sfml_wrap\scene.h"
#include "..\lidar\lidar_params.h"

class LidarScene : public Scene
{
public:
    LidarScene(const LidarParams& params);
    ~LidarScene() override = default;

    void viewWasChanged(const sf::View& view) const override;
};
```

2.11 lidar_draws/window_constants.h

```
#pragma once

#define DESIGNED_WIDTH  (1280)
#define DESIGNED_HEIGHT (720)
```

2.12 sfml_wrap/scene.cpp

```
#include "scene.h"
#include "SFML/Graphics/Texture.hpp"
#include <iostream>
#include "..\helpers\pooled_shared.h"

Scene::Scene()
{
    clock.restart();
}

void Scene::processEvents(EventsQueue& e)
{
    events.consumeEvents(e);
}

float Scene::getElapsed()
{
    auto r = clock.restart().asMicroseconds() * 0.001f;
    return r;
}

void Scene::viewWasChanged(const sf::View& view) const
{
}

const DrawableEntityList& Scene::getEntities() const
{
    return entities;
}

void Scene::win() const
{
}

void Scene::processUpdatedEntitesBeforeDraw(const DrawableEntityList&)
{
}

bool Scene::updateViewBeforeRender(sf::View&)
{
    return false;
}
```

```
void Scene::render(sf::RenderTarget& where, bool paused)
{
    const auto set_new_view = [this, &where](const auto& v)
    {
        where.setView(v);
        viewWasChanged(v);
    };

    if (needUpdateView)
    {
        needUpdateView = false;
        //doing "copy", I think initialy is empty here
        const sf::View v(where.getDefaultView());
        set_new_view(v);
    }

    const float time = getElapsed();
    if (!paused)
    {
        for (auto it = entities.begin(); it != entities.end(); ++it)
        {
            const auto& e = *it;
            e->update(*this, time, events);
        }

        processUpdatedEntitesBeforeDraw(entities);

        //fixme: possible copy of view maybe slow ...
        auto v = where.getView();
        if (updateViewBeforeRender(v))
            set_new_view(v);
    }

    const auto& render_state = sf::RenderStates::Default;

    for (const auto& e : entities)
        where.draw(*e, render_state);
}
```

2.13 sfml_wrap/scene.h

```
#pragma once

#include "SFML/Window/Event.hpp"
#include "SFML/Graphics/Drawable.hpp"
#include "SFML/System/Clock.hpp"
#include "SFML/Graphics/RenderTarget.hpp"
#include "drawableentity.h"
#include <map>
#include <string>
#include "inputsource.h"
#include "..\helpers\cm_ctors.h"

class Scene
{
public:
    NO_COPYMOVE(Scene);
    Scene();
    virtual ~Scene() = default;

public:
    //API for entities
    const DrawableEntityList& getEntities() const;

    //signals winning condition
    virtual void win() const;
protected:
    friend class WinRend;
    DrawableEntityList entities;
    void processEvents(EventsQueue& e);
    void render(sf::RenderTarget& where, bool paused);
    inline float getElapsed();
    //"signal", called once new view set
    virtual void viewWasChanged(const sf::View& view) const;

    virtual void processUpdatedEntitesBeforeDraw(const DrawableEntityList& entities);

    //view parameter has current view on call, if function returns true - view will be updated
    //and will call viewWasChanged
    virtual bool updateViewBeforeRender(sf::View& view);

private:
    bool needUpdateView{ true };
    sf::Clock clock{};
    InputSource events;
};
```

2.14 sfml_wrap/drawableentity.cpp

```
#include "drawableentity.h"

void DrawableEntity::setScreenSize(float width, float height)
{
    (void)width;
    (void)height;
}

void DrawableEntity::setScale(float mx, float my)
{
    (void)mx;
    (void)my;
}
```

2.14 sfml_wrap/drawableentity.h

```
#ifndef DRAWABLEENTITY_H
#define DRAWABLEENTITY_H

#include <memory>
#include <vector>
#include <cstdint>
#include "inputsource.h"
#include "SFML/Graphics/Drawable.hpp"

class Scene;
class DrawableEntity : public sf::Drawable
{
protected:
    DrawableEntity() = default;
public:
    ~DrawableEntity() override = default;
    virtual void update(Scene& scene, float time, const InputSource& input) = 0;
    virtual void setScreenSize(float width, float height);
    virtual void setScale(float mx, float my);
};

using DrawablePtr = std::shared_ptr<sf::Drawable>;
using DrawableEntityPtr = std::shared_ptr<DrawableEntity>;
using DrawableEntityList = std::vector<DrawableEntityPtr>;

#endif // DRAWABLEENTITY_H
```

2.14 sfml_wrap/winrend.cpp

```
#include "SFML/Window/Event.hpp"
#include <thread>
#include <chrono>
#include "winrend.h"
#include <iostream>
#include "..\helpers\block_delay.h"
#include "..\helpers\guard_on.h"

using DelayMeasuredIn = std::chrono::milliseconds;

//static defs
RendererLockType WinRend::renderMutex;
std::atomic<uint32_t> WinRend::clearColor(sf::Color::White.toInteger());

void WinRend::setClearColor(const sf::Color& color)
{
    clearColor = color.toInteger();
}

int WinRend::Main(sf::RenderWindow& window, Scene& game, const bool makeNewViewIfResized, const float desiredFPS)
{
    using namespace std::chrono_literals;
    window.setActive(false); //detaching window from thread
    EventsQueue events;
    std::atomic<bool> lostFocus(false);

    const DelayMeasuredIn DELAY = std::chrono::duration_cast<DelayMeasuredIn>(std::chrono::milliseconds(static_cast<int32_t>(1000.f / desiredFPS)));

    std::thread renderThread([&window, &events, &game, &lostFocus, &DELAY]
        {
            window.setActive(true); //attaching window to this thread
            while (window.isOpen())
            {
                DelayBlockMs<DelayMeasuredIn> delay(DELAY);//defines FPS, however it is MS delay ...
                (void)delay;

                const bool p = lostFocus;
                if (!p)
                    game.processEvents(events);

                {
                    LOCK_GUARD_ON(renderMutex);
                    window.clear(sf::Color(clearColor));
                    game.render(window, p);
                    window.display();
                }
            }
        });

    // run the program as long as the window is open
    while (window.isOpen())
    {
        DelayBlockMs<DelayMeasuredIn> delay(DELAY);//defines FPS, however it is MS delay ...
        (void)delay;
```

```
        // check all the window's events that were triggered since the last iteration of the loop
        sf::Event event;
        while (window.pollEvent(event))
        {
            // catch the resize events
            if (event.type == sf::Event::Resized)
            {
                if (makeNewViewIfResized)
                {
                    LOCK_GUARD_ON(renderMutex);
                    // update the view to the new size of the window
                    sf::FloatRect visibleArea(0.f, 0.f, event.size.width, event.size.height);
                    const sf::View v(visibleArea);
                    window.setView(v);
                    game.viewWasChanged(v);
                }
            }
            else
                if (event.type == sf::Event::Closed)// "close requested" event: we close the window
                {
                    LOCK_GUARD_ON(renderMutex);
                    window.close();
                }
                else
                    if (event.type == sf::Event::LostFocus)
                    {
                        lostFocus = true;
                        events.clear();
                    }
                    else
                        if (event.type == sf::Event::GainedFocus)
                            lostFocus = false;
                        else
                            events.push(event);
        }
    }

    renderThread.join();
    return 0;
}
```

2.15 sfml_wrap/winrend.h

```
#pragma once

#include "SFML/System.hpp"
#include "SFML/Graphics/RenderWindow.hpp"
#include <mutex>
#include "mthread_config.h"
#include "scene.h"

/*
    This class implements basic game loop to be called from the programme
 */

class WinRend
{
private:
    WinRend() = delete;
    static RendererLockType renderMutex;
    static std::atomic<uint32_t> clearColor;
public:
    static void setClearColor(const sf::Color& color);

    //makeNewViewIfResized == false  - view will remain same, all content will be scaled to new window proportions
    //makeNewViewIfResized == true  - everything will retain size, view will be recreated to match new window size
    int  static Main(sf::RenderWindow& window, Scene& game, const bool makeNewViewIfResized = false, const float desiredFPS = 60.f);
};
```

2.16 sfml_wrap/mthread_config.h

```
#pragma once

#ifndef _MTHREAD_CONFIG_H_
#define _MTHREAD_CONFIG_H_

#include <mutex>
#include "..\helpers\spinlock.h"

//typedef SpinLock LockType;
using LockType = std::recursive_mutex;
using LockGuard = std::lock_guard<LockType>;

using RendererLockType = recursive_spinlock;

#endif //_MTHREAD_CONFIG_H_
```

3. Appendix

3.1 Bibliography

[1] Wiki Lidar [ONLINE].
Available at:
https://en.wikipedia.org/wiki/Lidar
[Accessed 14-10-2019]

3.2 Plagiarism Declaration

INSTITUTE of TECHNOLOGY CARLOW

Work submitted for assessment which does not include this declaration will not be assessed.

DECLARATION

*I declare that all material in this submission e.g. thesis/essay/project/assignment is entirely my/our own work except where duly acknowledged.

*I have cited the sources of all quotations, paraphrases, summaries of information, tables, diagrams or other material; including software and other electronic media in which intellectual property rights may reside.

*I have provided a complete bibliography of all works and sources used in the preparation of this submission.

*I understand that failure to comply with the Institute's regulations governing plagiarism constitutes a serious offence.

Student Name: (Printed) CALIN DORAN

Student Number(s): C00220175

Signature(s): Calin Doran

Date: 3/4/2020

Please note:

a) * Individual declaration is required by each student for joint projects.

b) Where projects are submitted electronically, students are required to type their name under signature.

c) The Institute regulations on plagarism are set out in Section 10 of Examination and Assessment Regulations published each year in the Student Handbook.