Simulating LHC events with object-oriented programming in C++

Aodhán Burke

9965247

School of Physics and Astronomy

University of Manchester

Computing project report

April 2020

Abstract

A program was written with the concept of object-oriented programming (OOP) in the C++ language to attempt a simulation of particle events produced by proton collisions at the Large Hadron Collider (LHC). This involved the design and implementation of three main classes: `particle`, `detector` and `collsion`. Included in the method of coding were various advanced OOP features, such as the use of smart pointers to avoid memory leaks and substantial amounts of time devoted to debugging these, and the use of different containers to simplify the manipulation of data.

1 Introduction

Programming is becoming an ever more important skill in the modern workplace as we become more and more reliant on computing and technology. We are in an age where data security and management is implemented with almost everything we interact with. Object-oriented programming is a powerful method of securing and managing data. The notion of 'objects' is familiar to us and the programming style is intuitive, even to beginner programmers.
The C++ language in used extensively in experimental particle physics. Using the object-oriented style is natural to the storage and analysis of particle physics data, of which there are substantial amounts of. The efficiency of the compilable C++ language lends itself well to managing the hundreds of terabytes of data in individual job runs [1]. Particle physics experiments, for example the ATLAS experiment, have each developed their own object-oriented framework in C++ [2]. In this project I have developed a simplified simulator of particle events at the Large Hadron Collider (LHC) with a general-purpose detector such as the ATLAS detector.

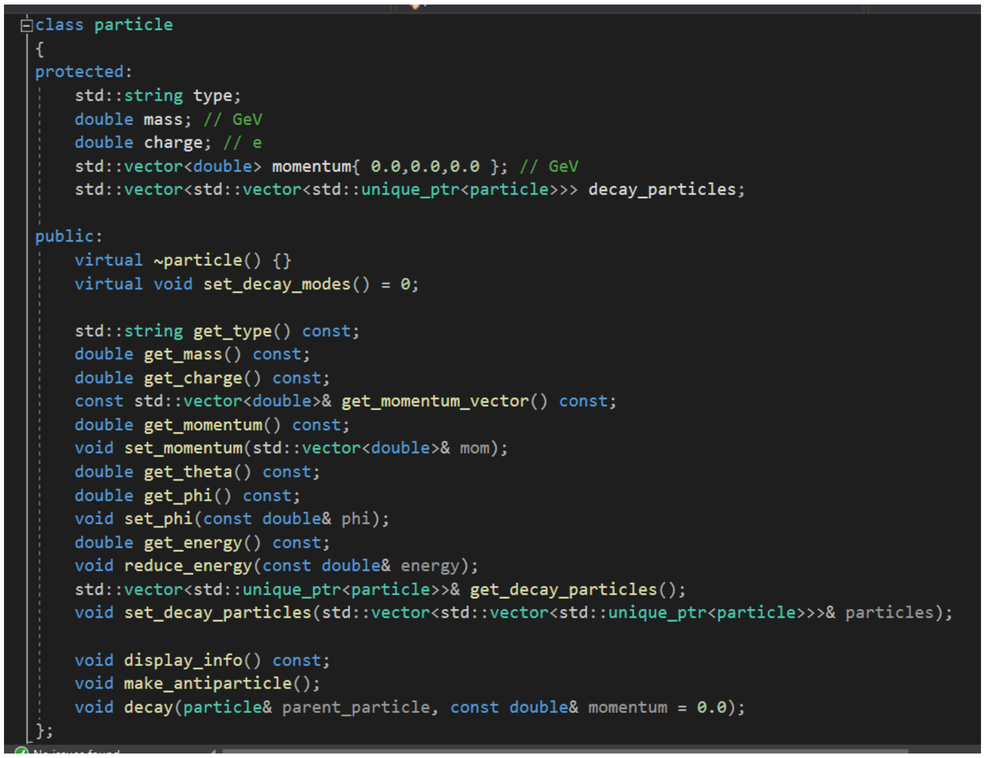
2 Code design and implementation

2.1 Particles

Figure 1 shows the abstract base `particle` class. The class contains protected member data which is important to simulating collisions, including `particle` smart pointers which represent the daughter particles if the given particle were to decay at some point. These pointers are contained in a `vector` of `vectors`, `decay_particles`. They are stored in this way since a particle may have multiple decay channels each containing multiple decay particles.
The majority of the public member functions are applicable to all derived `particle` classes, for example all of the getter and setter functions, with the exception of `particle::set_decay_modes()`. This fuction is `virtual` since the `decay_particles` are unique to individual particles and so it must be overridden for each particle. The non-trivial functions will be described in Sections 2.1.1-2.1.3.

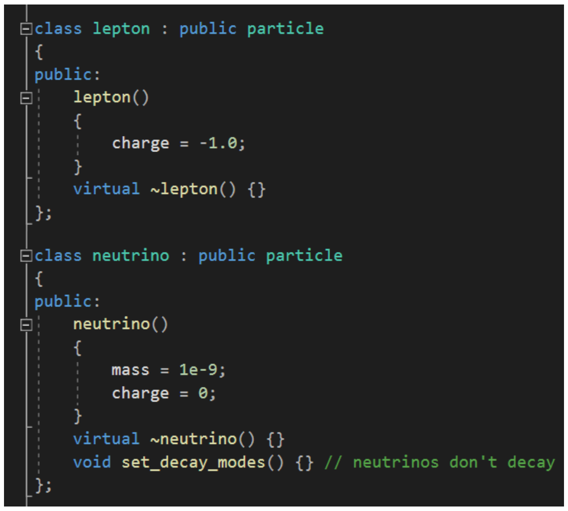
The abstract derived classes are `lepton`, `neutrino` and `quark`, shown in Figure 2. The `charge` is always defined in these classes through their constructors as the charge is common to each of the possible particles in their respective family. The `neutrino` class also defines the mass and overrides `particle::set_decay_modes()` to an empty function as none of the neutrinos decay. The quarks can be split into two sub-families based on their charge, hence two namespaces, `up_type` and `down_type`, were used to create two abstract derived `quark` classes.

Finally, Figure 3 shows the derived `muon` class which inherits from the `lepton` class. There is a default constructor and a parametrised constructor which takes momentum as an argument. Both constructors have an optional boolean argument, `is_antimuon`, which is set by default to `false`. If `true` is passed, then the `particle::make_antiparticle()` function is called which will change the muons `charge` and `type` member data. The constructors and destructor are identical for all other derived `particle` classes, differing only by what values are passed to the member data and the printed message.

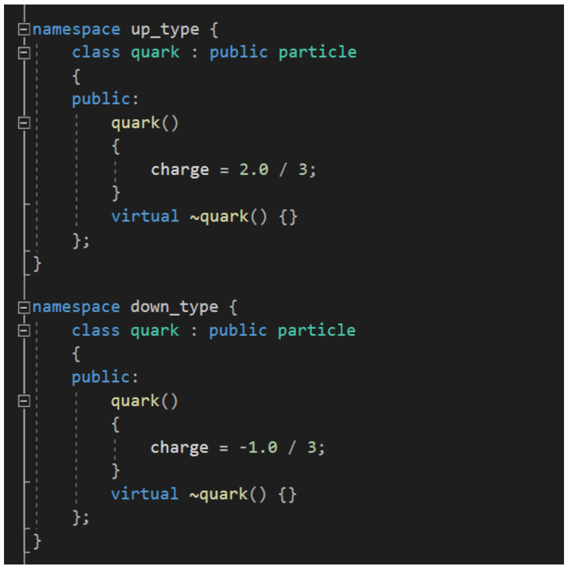


```cpp
class particle
{
protected:
    std::string type;
    double mass; // GeV
    double charge; // e
    std::vector<double> momentum{ 0.0,0.0,0.0 }; // GeV
    std::vector<std::vector<std::unique_ptr<particle>>> decay_particles;

public:
    virtual ~particle() {}
    virtual void set_decay_modes() = 0;

    std::string get_type() const;
    double get_mass() const;
    double get_charge() const;
    const std::vector<double>& get_momentum_vector() const;
    double get_momentum() const;
    void set_momentum(std::vector<double>& mom);
    double get_theta() const;
    double get_phi() const;
    void set_phi(const double& phi);
    double get_energy() const;
    void reduce_energy(const double& energy);
    std::vector<std::unique_ptr<particle>>& get_decay_particles();
    void set_decay_particles(std::vector<std::vector<std::unique_ptr<particle>>>& particles);

    void display_info() const;
    void make_antiparticle();
    void decay(particle& parent_particle, const double& momentum = 0.0);
};
```

Figure 1: Class design for particles involved in collisions at the LHC. The mass, momentum and energy of particles are in units of GeV throughout, and the charge in units of the electron charge e. The momentum is stored as a `vector` of the x, y and z components to simplify access in various functions.

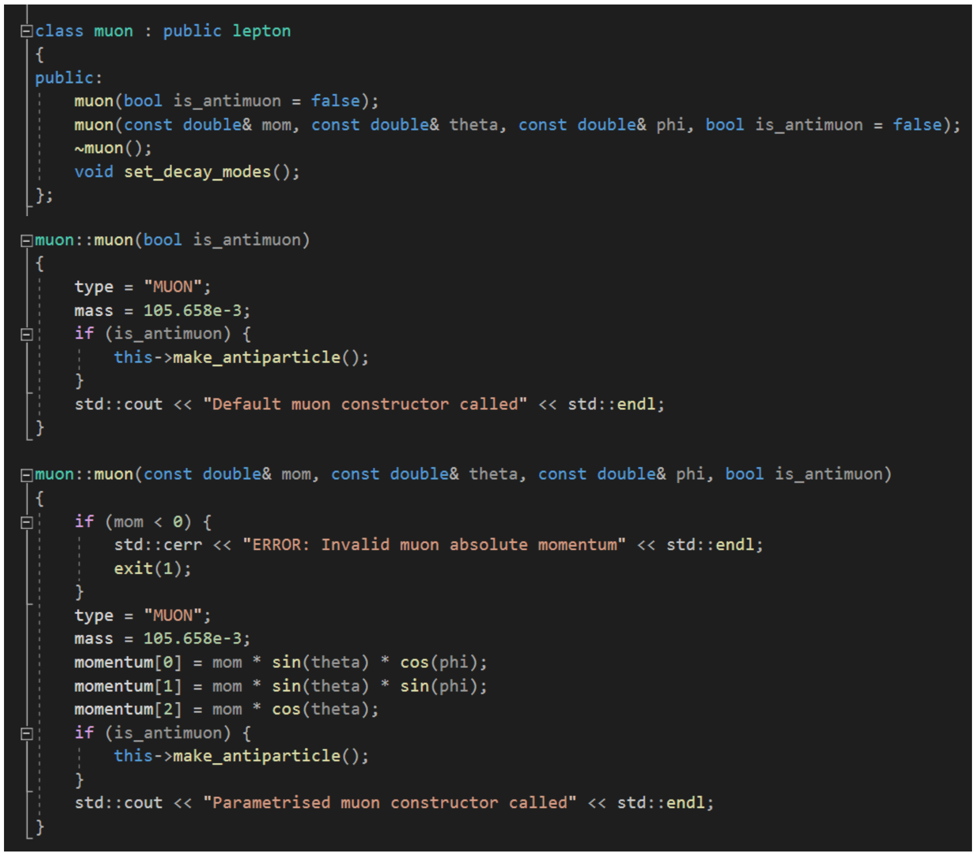


```cpp
class lepton : public particle
{
public:
    lepton()
    {
        charge = -1.0;
    }
    virtual ~lepton() {}
};

class neutrino : public particle
{
public:
    neutrino()
    {
        mass = 1e-9;
        charge = 0;
    }
    virtual ~neutrino() {}
    void set_decay_modes() {} // neutrinos don't decay
};
```



```cpp
namespace up_type {
    class quark : public particle
    {
    public:
        quark()
        {
            charge = 2.0 / 3;
        }
        virtual ~quark() {}
    };
}

namespace down_type {
    class quark : public particle
    {
    public:
        quark()
        {
            charge = -1.0 / 3;
        }
        virtual ~quark() {}
    };
}
```

Figure 2: All of the abstract derived `particle` classes. The `charge` is defined in these along with extra definitions for the `neutrino` class specifically. Namespaces are used to distinguish quarks with positive and negative charge, with the up-type and down-type nomenclature used in particle physics.

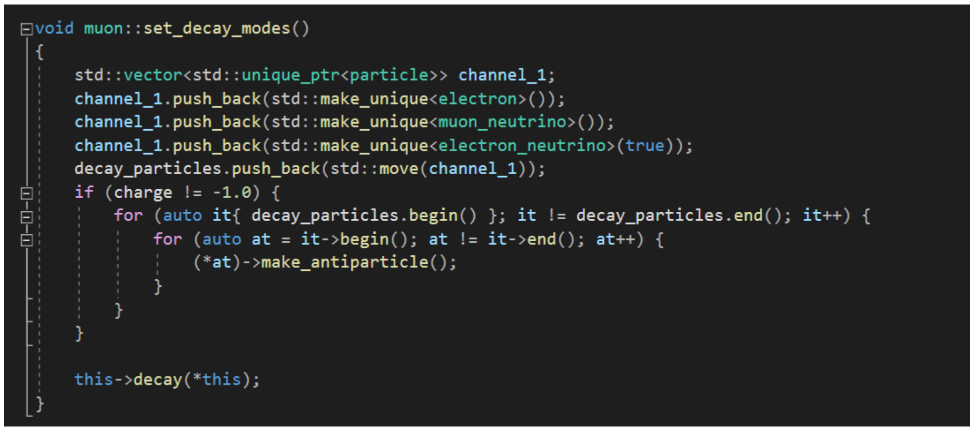


```cpp
class muon : public lepton
{
public:
    muon(bool is_antimuon = false);
    muon(const double& mom, const double& theta, const double& phi, bool is_antimuon = false);
    ~muon();
    void set_decay_modes();
};

muon::muon(bool is_antimuon)
{
    type = "MUON";
    mass = 105.658e-3;
    if (is_antimuon) {
        this->make_antiparticle();
    }
    std::cout << "Default muon constructor called" << std::endl;
}

muon::muon(const double& mom, const double& theta, const double& phi, bool is_antimuon)
{
    if (mom < 0) {
        std::cerr << "ERROR: Invalid muon absolute momentum" << std::endl;
        exit(1);
    }
    type = "MUON";
    mass = 105.658e-3;
    momentum[0] = mom * sin(theta) * cos(phi);
    momentum[1] = mom * sin(theta) * sin(phi);
    momentum[2] = mom * cos(theta);
    if (is_antimuon) {
        this->make_antiparticle();
    }
    std::cout << "Parametrised muon constructor called" << std::endl;
}
```

Figure 3: Derived `muon` class with constructor declarations and definitions.

2.1.1 `particle::set_decay_modes()`

Figure 4 shows the overridden `particle::set_decay_modes()` function for the `muon` class. For each possible decay channel, a `vector` of `unique_ptrs` to `particle` objects is created. Then the relevant objects are pushed onto these `vectors`, before the `vectors` themselves are pushed onto `decay_particles`. The `vectors` must be pushed with `std::move()` as the pointers are unique. The particles which are added to `decy_particles` are assumed to be those corresponding to a muon decay rather than an antimuon decay. However, there is then a check on whether the muon is actually an antimuon, in which case all decay particles are set to be antiparticles. Finally, the `particle::decay()` function is called on the `particle`.



```cpp
void muon::set_decay_modes()
{
    std::vector<std::unique_ptr<particle>> channel_1;
    channel_1.push_back(std::make_unique<electron>());
    channel_1.push_back(std::make_unique<muon_neutrino>());
    channel_1.push_back(std::make_unique<electron_neutrino>(true));
    decay_particles.push_back(std::move(channel_1));
    if (charge != -1.0) {
        for (auto it{ decay_particles.begin() }; it != decay_particles.end(); it++) {
            for (auto at = it->begin(); at != it->end(); at++) {
                (*at)->make_antiparticle();
            }
        }
    }

    this->decay(*this);
}
```

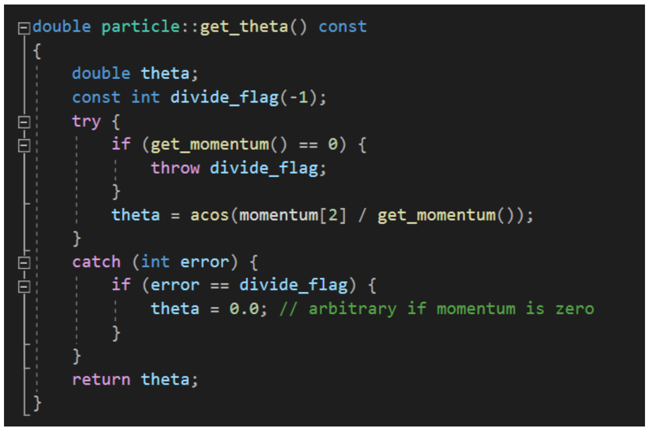
Figure 4: Definition of `particle::set_decay_modes()` for the `muon` class. The muon decays to an electron, emitting a muon-neutrino and an electron-antineutrino.

2.1.2 `particle::decay()`

The function first iterates over all channels in `decay_particles` and sums the mass of the daughter particles. It checks which channels (if any) have a total daughter mass less than the energy of the parent particle. From those which do, a random channel is selected which the particle will decay via.
In the rest frame of the parent particle, a momentum vector is generated with magnitude p, which is half the parent mass, in a random direction $\theta \epsilon [0, \pi]$, $\phi \epsilon [-\pi, \pi]$. This is then Lorentz boosted back to the lab frame by using the momentum of the parent particle. If the decay channel contains only two particles then a second momentum vector is calculated as the difference between the initial parent momentum and the Lorentz boosted momentum vector. These are then assigned to the decay particles.
Alternatively, if there are more than two decay particles then the Lorentz boosted momentum vector is assigned to the first decay particle. Then a `recoil` object is instantiated, which inherits from `particle`. The remaining decay particles are moved onto the `recoil` using the `particle::set_decay_particles()` function and `particle::decay()` is called recursively on the `recoil`. The decay particles, now with altered momenta, are then moved back onto the parent particle and the other channels are removed so that the parent particle has one decay channel with correctly calculated momenta associated with each decay particle.
To be able to calculate the correct Lorentz boost recursively, the parent particle is passed by reference as an argument in `particle::decay()`. A momentum magnitude is also passed which is used in the calculation of any recoil mass.

2.1.3 `particle::get_theta()`

In the getter function `particle::get_theta()`, θ is found using the z-component of the momentum and the total momentum. There is a chance of dividing by zero if the total momentum is zero, hence error catching was implemented as shown in Figure 5. If the momentum is zero then θ holds no meaning and zero is returned. There is a similar design in the `particle::get_phi()` function.



```cpp
double particle::get_theta() const
{
    double theta;
    const int divide_flag(-1);
    try {
        if (get_momentum() == 0) {
            throw divide_flag;
        }
        theta = acos(momentum[2] / get_momentum());
    }
    catch (int error) {
        if (error == divide_flag) {
            theta = 0.0; // arbitrary if momentum is zero
        }
    }
    return theta;
}
```

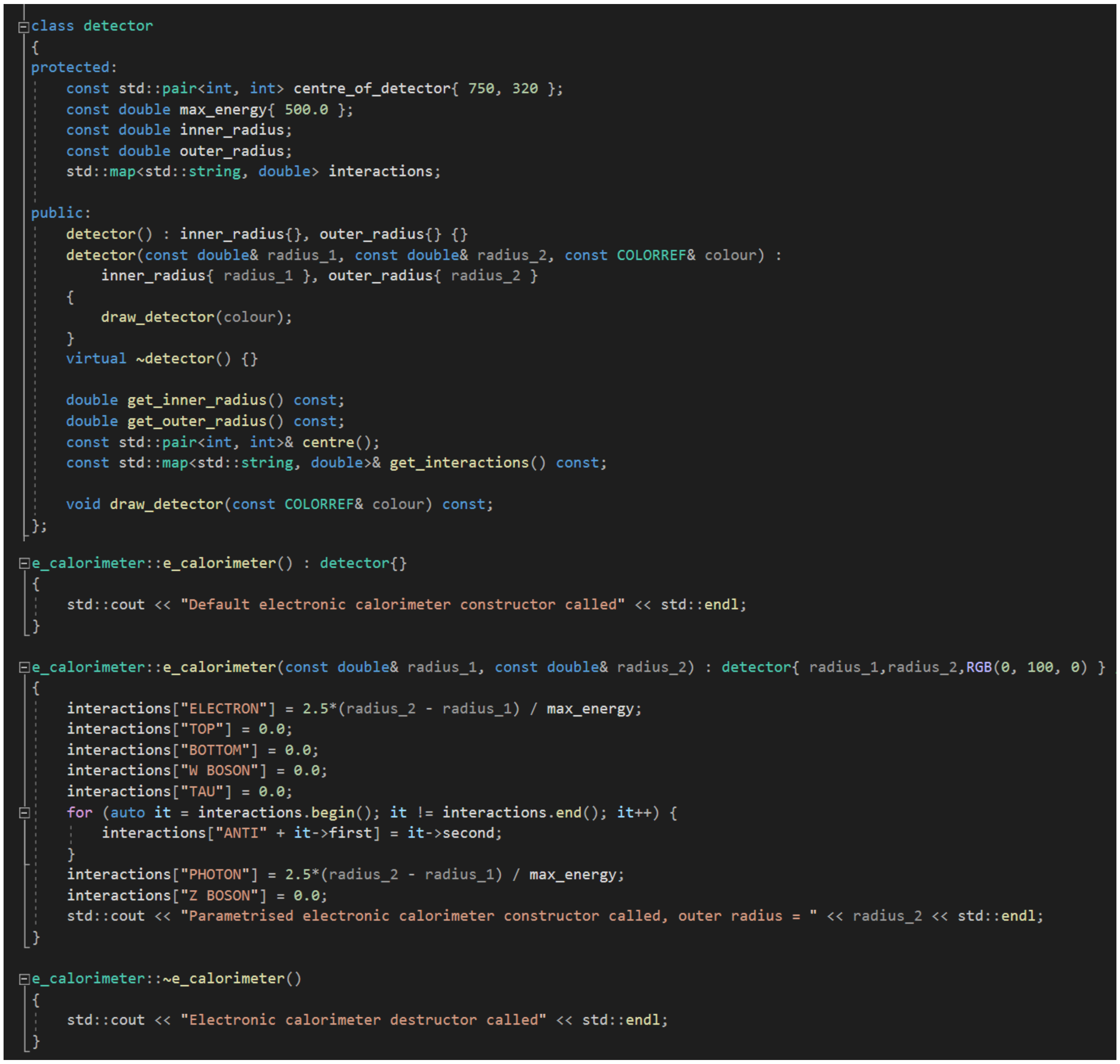
Figure 5: Using try-throw-catch to avoid dividing by zero when calculating the polar angle θ of a momentum vector.

2.2 Detectors

Figure 6(a) shows the `detector` class design. Each detector segment has a position, size and a `map` containing particle types and their strength of interaction, if any, with that detector segments medium. The position is stored with a `const pair` of `ints` which is used in the `detector::draw_detector()` function to display the detector in the console window. Altering the `centre_of_detector` member will move the detector around the console window. The `detector::draw_detector()` function also takes a `COLORREF` argument, which is passed to it from the parametrised `detector` constructor, so that each detector segment can be coloured differently. An example of a class which inherits from `detector`, `e_calorimeter`, is shown in Figure 6(b). Electrons and photons are solely added to the `interactions` member of this class so that decays and interactions of `electron` or `photon` objects can be triggered when one enters this region of the detector.

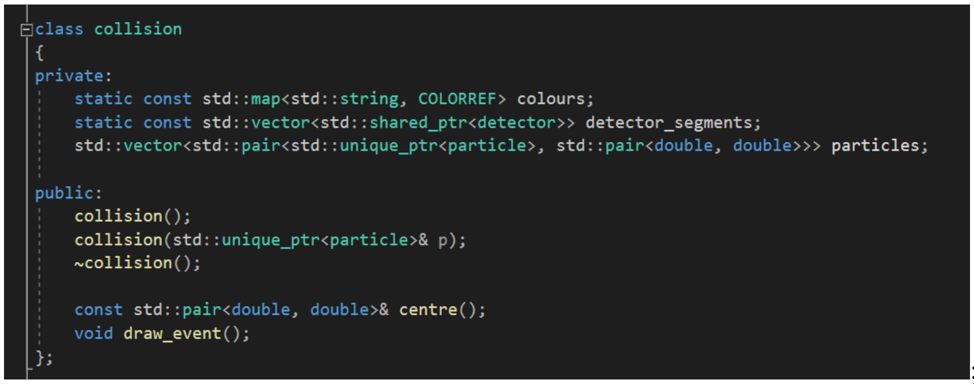
2.3 Collisions

The `collision` class encapsulates the main functionality of the program. As shown in Figure 7, the class contains a polymorphic `vector` of `detector` objects. This `vector` was made `static` so that all `collision` instances created during the lifetime of the program have access to the same `detector` objects. For this to be possible the `vector` contains `shared_ptrs` to the `detector` objects. The `static vector` is defined in the collision.cpp file with a set inner and outer radius for each segment. This calls the constructors for the `detector` segments, each of which call the `detector::draw_detetor()` function with a set colour such that the detector is drawn only once at the start of the program and there it remains. The class also contains another polymorphic `vector` of `pairs` of `particle` objects and coordinates, which are represented by a `pair` of `ints`.



```
class detector
{
protected:
    const std::pair<int, int> centre_of_detector{ 750, 320 };
    const double max_energy{ 500.0 };
    const double inner_radius;
    const double outer_radius;
    std::map<std::string, double> interactions;

public:
    detector() : inner_radius{}, outer_radius{} {}
    detector(const double& radius_1, const double& radius_2, const COLORREF& colour) :
        inner_radius{ radius_1 }, outer_radius{ radius_2 }
    {
        draw_detector(colour);
    }
    virtual ~detector() {}

    double get_inner_radius() const;
    double get_outer_radius() const;
    const std::pair<int, int>& centre();
    const std::map<std::string, double>& get_interactions() const;

    void draw_detector(const COLORREF& colour) const;
};

e_calorimeter::e_calorimeter() : detector{}
{
    std::cout << "Default electronic calorimeter constructor called" << std::endl;
}

e_calorimeter::e_calorimeter(const double& radius_1, const double& radius_2) : detector{ radius_1,radius_2,RGB(0, 100, 0) }
{
    interactions["ELECTRON"] = 2.5*(radius_2 - radius_1) / max_energy;
    interactions["TOP"] = 0.0;
    interactions["BOTTOM"] = 0.0;
    interactions["W BOSON"] = 0.0;
    interactions["TAU"] = 0.0;
    for (auto it = interactions.begin(); it != interactions.end(); it++) {
        interactions["ANTI" + it->first] = it->second;
    }
    interactions["PHOTON"] = 2.5*(radius_2 - radius_1) / max_energy;
    interactions["Z BOSON"] = 0.0;
    std::cout << "Parametrised electronic calorimeter constructor called, outer radius = " << radius_2 << std::endl;
}

e_calorimeter::~e_calorimeter()
{
    std::cout << "Electronic calorimeter destructor called" << std::endl;
}
```

Figure 6: Structure of the `detector` abstract base class and the implementation of an electronic calorimeter class, `e_calorimeter`, which inherits from `detector`. The only particles which interact with an electronic calorimeter are electrons and photons, hence these are added to `interactions`.

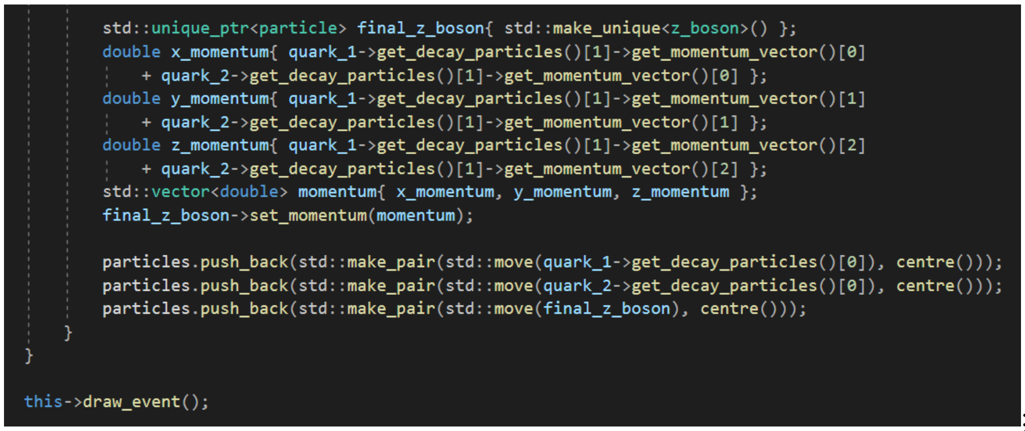


```
class collision
{
private:
    static const std::map<std::string, COLORREF> colours;
    static const std::vector<std::shared_ptr<detector>> detector_segments;
    std::vector<std::pair<std::unique_ptr<particle>, std::pair<double, double>>> particles;

public:
    collision();
    collision(std::unique_ptr<particle>& p);
    ~collision();

    const std::pair<double, double>& centre();
    void draw_event();
};
```

;

Figure 7: Class structure of a `collision` designed to replicate LHC events. The class contains both `particle` and `detector` objects.

The class has only a default constructor where the `particles vector` gets filled. The LHC collides protons; the majority of the time it is gluons, up quarks or down quarks which are involved in the interactions. Two partons are drawn at random from these options and, depending on which are drawn, these are interacted in one of a various number of ways. The outgoing particles will include up or down quarks and electroweak bosons. The innermost detector segment, the tracker, then has the weak bosons in its `interactions map` so that these decay immediately to a range of possible particles.
Figure 8 shows an example of one the possible collision outcomes. The colliding partons are quarks which interact via vector boson fusion. Both quarks emit a Z boson which fuse to a single Z boson so that the final state is two quarks + one Z boson. Since the emission of a Z boson is not a defined channel in `particle::set_decay_modes()` for the quarks, the daughter particles are created manually and pushed onto the parent quarks using `particle::set_decay_particles()`. The `particle::decay()` function is then called on the quarks and a new Z boson is created with a momentum defined as the vector sum of the two daughter Z boson momenta. The particles are then all moved back on to `particles`.

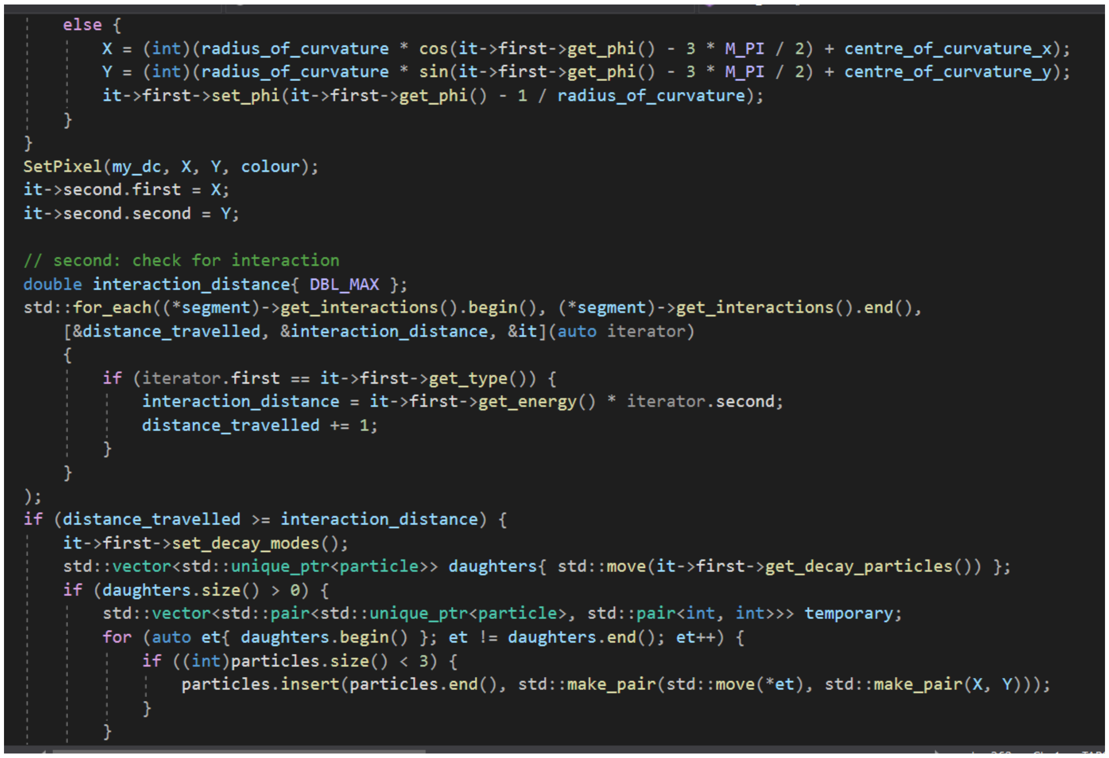


```
            std::unique_ptr<particle> final_z_boson{ std::make_unique<z_boson>() };
            double x_momentum{ quark_1->get_decay_particles()[1]->get_momentum_vector()[0]
                + quark_2->get_decay_particles()[1]->get_momentum_vector()[0] };
            double y_momentum{ quark_1->get_decay_particles()[1]->get_momentum_vector()[1]
                + quark_2->get_decay_particles()[1]->get_momentum_vector()[1] };
            double z_momentum{ quark_1->get_decay_particles()[1]->get_momentum_vector()[2]
                + quark_2->get_decay_particles()[1]->get_momentum_vector()[2] };
            std::vector<double> momentum{ x_momentum, y_momentum, z_momentum };
            final_z_boson->set_momentum(momentum);

            particles.push_back(std::make_pair(std::move(quark_1->get_decay_particles()[0]), centre()));
            particles.push_back(std::make_pair(std::move(quark_2->get_decay_particles()[0]), centre()));
            particles.push_back(std::make_pair(std::move(final_z_boson), centre()));
        }
    }

    this->draw_event();
```

;

Figure 8: An example of a type of interaction occurring in proton collisions. If the interacting partons are quarks, vector boson fusion of Z's can occur.

When an event is generated, the outgoing `particle` objects are propagated through the `detector` segments by the `collision::draw_event()` function which is called at the end of the constructor. The function loops over the `particles` of the `collision` instance drawing each onto the console window, using the `momentum` and `charge` of the `particle` to propagate it in the correct direction with the correct curvature. This is demonstrated in Figure 9 where a `while` loop and the `SetPixel()` function, which is included in the Windows.h header, are used to draw the path of a particle with non-zero charge. At every step of the propagation, the function finds in which detector segment, if any, the particle is currently located using `std::find_if()` and a lambda function. A second lambda function is used with `std::for_each()` to check whether the particle interacts with the current detector segment or not, as shown in Figure 9. If that is the case, after the particle has traversed more than the interaction distance the `particle::decay()` function is called on the `particle` and its daughter particles are pushed onto the `particles vector` with the coordinates of the current `particle`.

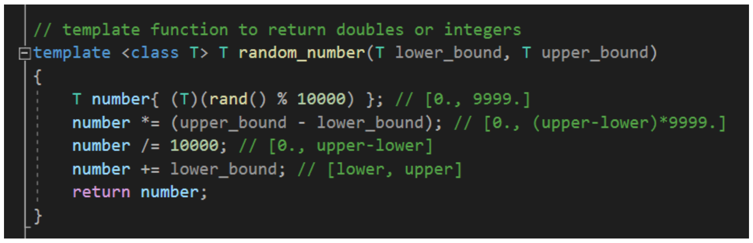


```
    else {
        X = (int)(radius_of_curvature * cos(it->first->get_phi() - 3 * M_PI / 2) + centre_of_curvature_x);
        Y = (int)(radius_of_curvature * sin(it->first->get_phi() - 3 * M_PI / 2) + centre_of_curvature_y);
        it->first->set_phi(it->first->get_phi() - 1 / radius_of_curvature);
    }
}
SetPixel(my_dc, X, Y, colour);
it->second.first = X;
it->second.second = Y;

// second: check for interaction
double interaction_distance{ DBL_MAX };
std::for_each((*segment)->get_interactions().begin(), (*segment)->get_interactions().end(),
    [&distance_travelled, &interaction_distance, &it](auto iterator)
    {
        if (iterator.first == it->first->get_type()) {
            interaction_distance = it->first->get_energy() * iterator.second;
            distance_travelled += 1;
        }
    }
);
if (distance_travelled >= interaction_distance) {
    it->first->set_decay_modes();
    std::vector<std::unique_ptr<particle>> daughters{ std::move(it->first->get_decay_particles()) };
    if (daughters.size() > 0) {
        std::vector<std::pair<std::unique_ptr<particle>, std::pair<int, int>>> temporary;
        for (auto et{ daughters.begin() }; et != daughters.end(); et++) {
            if ((int)particles.size() < 3) {
                particles.insert(particles.end(), std::make_pair(std::move(*et), std::make_pair(X, Y)));
            }
        }
```

Figure 9: A section of the `collision::draw_event()` function showing how charged particles are drawn as curves using the `SetPixel()` function, and how decays in different detector segments are implemented with lambda functions.

2.4 Randomness

The program often uses random numbers, for example to decide on θ, ϕ or on which channel to decay to. The random numbers are required to be of type `double` or `int`. Rather than creating a separate random number function for each type, a template function, `random_number()` was created which can generate both a `double` and an `int`. The function is shown in Figure 10; it takes two arguments which control the range in which the random number will be generated. A random integer is drawn in the range [0, 9999] which is then cast to the class type `T`, meaning that the random number will have a maximum of 4 significant figures. The number is then scaled and transformed to the required range before being returned.

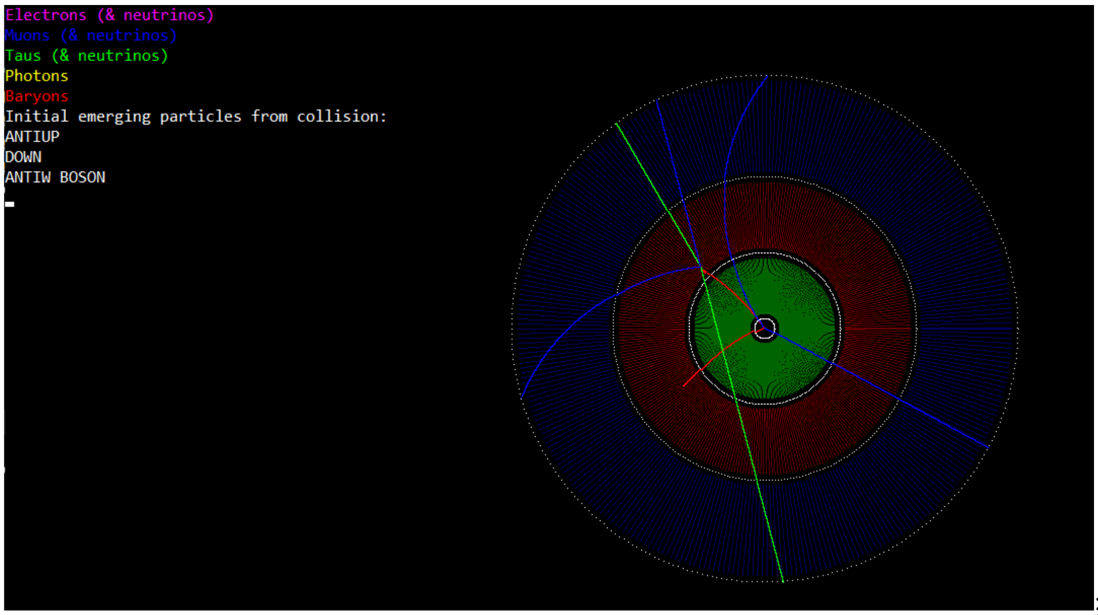


```
// template function to return doubles or integers
template <class T> T random_number(T lower_bound, T upper_bound)
{
    T number{ (T)(rand() % 10000) }; // [0., 9999.]
    number *= (upper_bound - lower_bound); // [0., (upper-lower)*9999.]
    number /= 10000; // [0., upper-lower]
    number += lower_bound; // [lower, upper]
    return number;
}
```

Figure 10: Template function returning a randomly generated `double` or `int` in a range specified by the input parameters `lower_bound` and `upper_bound`. The return value type is governed by the type of input parameters.

3 Results

Creating one `collision` object in the `main()` function gives the result shown in Figure 11. A legend is printed in the top left corner so that the user can identify particle types from the track colours and curvature. In this particular example, the initial emerging particles are an $\bar{u}$ quark, a d quark and a W^+ boson. This will have been the result of two initial gluons; one of which decayed to $d\bar{d}$, and the other to $u\bar{u}$. The u and $\bar{d}$ quarks collide to form the W^+ boson and the remaining two quarks leave as hadronic jets.
Following the tracks and using the legend, we can work out that the initial W^+ boson decayed immediately via $W^+ \rightarrow \mu^+\nu_\mu$, giving the two blue trajectories originating at the centre of the detector. The down quark decayed in the hadronic calorimeter via $d \rightarrow W^-u$, the W^- boson resulting from this going on to decay via $W^- \rightarrow \tau\bar{\nu_\tau}$ giving one of the green tau neutrino trajectories. The tau then rapidly decayed via $\tau \rightarrow \mu\bar{\nu_\mu}\nu_\tau$ to give the remaining green tau neutrino track and the other blue muon tracks.
Using the μ^+ emitted from the initial W^+ boson, we can make the assumption that positively charged particles curve clockwise. Then, checking the curvature of the other tracks against the charge of their indicated particle identities, we can verify that decays and particles are produced correctly.



```
Electrons (& neutrinos)
Muons (& neutrinos)
Taus (& neutrinos)
Photons
Baryons
Initial emerging particles from collision:
ANTIUP
DOWN
ANTIW BOSON
```

;

Figure 11: Example output of one `collision` instance, showing the drawn detector and particles created from the interaction. On the left is a legend showing the colour each particle type is drawn in as well as a list of the initial particles created in the collision.

4 Discussion and conclusion

The constructor of the `collision` class contains a substantial number of lines of code. This is simply due to the number possible interactions which can occur during proton collisions. The constructor could easily be shortened; rather than considering the interactions between a randomly generated pair of colliding partons, the initial emerging particles could be generated with random kinematics from a `list` containing common initial state particles. This could then include Higgs states, which were not included in the project.
The `collision::draw_event()` also contains many lines of code since there are various options for drawing a particular particle's trajectory, for instance its charge and whether or not it interacts with a particular detector medium. To improve on this, functions could be made for charged and non-charged trajectory drawing. The storage of `particle` objects along with coordinates in a `vector` of `pairs` also looks quite messy. Perhaps the coordinates could be moved to be a data member of the particle class instead, further cleaning up the `collision::draw_event()` function.
Object-oriented programming in C++ makes for a relatively simple way of managing data necessary to simulate particle events at the LHC. Using run-time polymorphism allows for concise code to iterating over many types of particle and perform the same job on each.

References

[1] Paterno, M., Kowalkowski, J., and Green, C. Improving robustness and computational efficiency using modern C++. *Journal of Physics: Conference Series*, **513**(5), 2014.

[2] Catmore, J., Cranshaw, J., Gillam, T., Gramstad, E., Laycock, P., Ozturk, N., and Stewart, G.A. A new petabyte-scale data derivation framework for ATLAS. *Journal of Physics: Conference Series*, **664**(7), 2015.

Appendices

A Impact of COVID19 on the work completed and report

Time lost travelling home:
At most 2 days of work were lost.

Lack of access to computers and the internet:
My laptop supports Visual Studio 2019, however, internet connection at home can be slow at times.

Limitation in the outcome:
Not much impact.

B Length

The total number of words in this document is 2561.