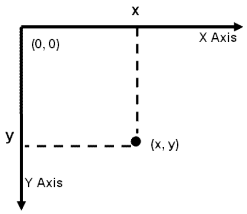
Documentation – Physics

---------- Structure ----------

The **Collider** class is the parent class of every object in the game.
The **RigidBody** inherits the Collider Class. RigidBodies are physical objects that move. Example: Projectiles, Worms.
The **Ground** is a Collider not a RigidBody because it doesn't move and it is not affected by gravity.
The **PhysicsEngine** class is simply a wrapper class so the functionalities of physics can be used in a clean and concise way.

The system of coordinates for the window is as below:



---------- Collider ----------

The **Collider** is a parent class to every object that might collide in the game. It is used to detect collisions. It has the following parameters:

```
private:
int id;
```
All objects have a unique identifier; it will be used by the PhysicsEngine to keep track of the objects in the game.
```
double skin_depth_percent = 0.2;
```
This represents what percentage of the object we allow to overlap with another collider before detecting a collision. It is important for the GUI so the object does not look like it is floating above the collider object.
```
int skin_depth_pixels = 0;
```
Same thing as `skin_depth_percent` but it is in pixels and it is computed when calling the constructor.
```
int countblack = 0;
```
Used to compute the `skin_depth_pixels.`
```
QImage colliding_map = QImage(32,32,QImage::Format_RGB32);
```
The default image colliding map is a 32x32 black square. This image defines the shape of the object and is used to detect collisions and compute response.

In order to access the private parameters (apart from the `id`) one needs to use the get/ set methods:

```
public:
void setSkin(double depth_percent);
```
Sets the `skin_depth_percent` and recomputes the `skin_depth_pixels.` Important: the `skin_depth` is a percentage of the body so if it is more than 20% it will make the body behave weirdly as it will sink on the ground. Also the `skin_depth` is needed to approximate the normal when colliding with the ground therefore do not set a `skin_depth` less than 0.08. The sweet spot is 0.2
```
QImage get_map();
```
gets the colliding map
```
void set_map(QImage map);
```
sets the colliding map to the map you want; recomputes all associated parameters: `skin_depth_pixels` and `countblack`.
```
void change_pixel(int i, int j, QColor color);
```
changes the color of a pixel of the map and recomputes the associated parameters.

The `countblack` is not a parameter which will be used outside this class i.e does not need a function.

```
protected:
   double x, y; The top-left coordinates of the object.
   double cmx, cmy; the coordinates of the central of mass of an object assuming it has uniform density.
```

Every class inheriting the class Collider will be able to access the above parameters directly without the need of set/get methods. Nonetheless, these methods are defined.

```
public:
   double getX();
   double getY();
   void setX(double x1);
   void setY(double y1);
   double getcmX(); get central of mass x coordinate
   double getcmY(); get central of mass y coordinate
   void setcmX(double cx); set cmx to cx
   void setcmY(double cy); set cmy to cy
```

Constructors

```
public:

Collider();
Collider(double ix, double iy, QImage map);
Collider(double ix, double iy); the map is set by default to a black 32x32 square.
```

The methods that can be used by other classes:

- Map methods (important for the GUI and the Ground class)

```
void set_map(QImage map);
void change_pixel(int i, int j, QColor color);
```

- Most used get/ set methods

```
 double getX();            void setX(double x1);
 double getY();            void setY(double y1);
 double getcmX();          void setcmX(double cx);
 double getcmY();          void setcmY(double cy);
```

The methods that should not be used by other classes:

```
double getId();
```
Gets the unique identifier of the object. This method should not be used by other classes. It is for the implementation of PhysicsEngine only.

```
void setId(int iid);
```
Sets the unique identifier of the object. This method should not be used by other classes. It is for the implementation of PhysicsEngine only.

```
QPair<bool, QPair<double, double>> check_collision(Collider &other);
```
This checks for collision with another collider but it is already taken care of by the physics engine and therefore you should never have to call it.

---------- **RigidBody** ----------

The **RigidBody** inherits the **Collider** class. It is an object that moves i.e is subject to physics.
It has the below parameters:

```
private:
double mass = 0, vx = 0, vy = 0, ax = 0, ay = 0;
QPair<double, double> currentForce = QPair<double, double>(0, 0);
```
The force is a tuple of x-coordinate and y-coordinate
```
double bounciness_f = 0;
```
The scalar by which the velocity is multiplied after a collision. It is between [0, 1). If you have a bounciness_f = 1 then the object will bounce forever.
```
bool stable = false;
```
By default the object is not stable. The object is stable when the velocity and the currentForce are very small.

```
public:
bool is_colliding = false;
```
By default the object is not colliding. This parameter is updated by the UPDATE method of the PhysicsEngine.

To access these parameters(apart from the public one) one needs to use the get/set methods:

```
void setbounciness(double b);
void setm(double m);
void setvx(double v_x);
void setvy(double v_y);
void setax(double a_x);
void setay(double a_y);
void setstable(bool a);
```

```
bool getstable();
double getbounciness();
double getm();
double getvx();
double getvy();
double getax();
double getay();
```

Constructors:

```
public:
    RigidBody(double imass, double ix, double iy, double vx0, double vy0,
double ax0, double ay0, QImage map);
    RigidBody(double imass, double ix, double iy);
```
The acceleration and velocity are set by default to 0, the map is the 32x32 black square and the id = -1.

The methods that can be used by other classes:

```
void addForce(QPair<double, double> F);
```
The parameter given to this function is a pair of (Fx, Fy). This pair is added to the current Force acting on the RigidBody.
```
double distance(RigidBody other);
```
It computes the distance between two centers of masses between your RigidBody and another one.

The methods should not be used by other classes:

```
void bounce(QPair<double, double> normal, double dt);
```
This creates the Implusive Force which makes the bounce of the RigidBody after detecting a collision. This method is already implemented in the PhysicsEngine, so you do not need to call it at any point.
```
void simulate(double dt);
```
After a certain time dt(miliseconds), this function simulates the simple kinematic equations to calculate the new position, acceleration and velocity of the RigidBody. It is implemented in the PhysicsEngine, so you do not need to call it at any point.

---------- **PhysicsEngine** ----------

The **PhysicsEngine** class is the class that you should be using in the game loop to implement physics. It has the following parameters:

```
    private:
        QMap<int, RigidBody*> rigidbodies = QMap<int, RigidBody*>();
```
This is a list of rigidbodies (pointers) so that the Engine can keep track of what are the objects that need to be updated.
```
        QMap<int, Collider*> colliders = QMap<int, Collider*>();
```
This is a list of colliders (pointers) that are not rigidbodies so that the Engine can keep track of the objects for which it needs to check collision. Example: the ground.
```
        QPair<double, double> general_force = QPair<double, double>(0, 0);
```
These are the general forces that should be applied at all time (gravity – wind) etc.

Constructors :

```
public:
PhysicsEngine();
PhysicsEngine(QPair<double, double> gf);
```
This creates a new physics Engine with an initial general force which pair of (Fx, Fy).

All Methods should be used by other classes/codes especially the game loop :

```
void add_RigidBody(RigidBody* other);
```
This adds an existing rigidbody to the list of rigidbodies in the simulation. This also gives the rigidbody a unique Id.
```
void add_Collider(Collider* other);
```
This adds an existing collider to the list of colliders in the simulation. This also gives the collider a unique Id.
```
void update(double dt);
```
This method is the one you should really be needing all the time. This updates all the objects of the simulation by a time period dt (milliseconds). This updates the rigidbody parameter is_colliding.
```
RigidBody* create_rigidbody(double imass, double ix, double iy, double vx0,
double vy0, double ax0, double ay0, QImage map);
```
This method creates a new Rigidbody, adds it to the simulation using the `add_RigidBody` function and returns the pointer.
```
RigidBody* create_rigidbody(double imass, double ix, double iy);
```
Same as the previous one.
```
Collider* create_collider(double ix, double iy, QImage map);
```
This method creates a new Rigidbody, adds it to the simulation using the `add_RigidBody` function and returns the pointer.
```
Collider* create_collider(double ix, double iy);
```
Same as the previous one.
```
RigidBody* get_rigidbody(int id);
```
Gets the RigidBody through the ID.
```
Collider* get_collider(int id);
```
Gets the Collider through the ID.