```
/****************************************************************************

** Author: J.-O. Lachaud, University Savoie Mont Blanc

** (vaguely adapted from Qt colliding mices example)

**

** Copyright (C) 2015 The Qt Company Ltd.

** Contact: http://www.qt.io/licensing/

****************************************************************************/


#include <cmath>

#include <QtWidgets>

#include "objects.hpp"

#include <QBitmap>

#include <QPixmap>

static const int AsteroidCount = 10;

static const int SpaceTruckCount = 5;

int main(int argc, char **argv)

/* Creating a pixmap object with the image of the asteroid. */

{


  // Initializes Qt.

  QApplication app(argc, argv);

  QPixmap asteroid_pixmap(":/images/asteroid.gif");

  // Initializes the random generator.

  qsrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));


  // Creates a graphics scene where we will put graphical objects.
```

```
QGraphicsScene graphical_scene;

graphical_scene.setSceneRect(0, 0, IMAGE_SIZE, IMAGE_SIZE);

graphical_scene.setItemIndexMethod(QGraphicsScene::NoIndex);


// We choose to check intersection with 100 random points.

logical_scene = new LogicalScene(100);


int SpaceTruckCount = 10;

// Creates a few SpaceTruck


for (int i = 0; i < SpaceTruckCount; ++i)

{

  // Qcolor green in cok

  QColor cok(0, 255, 0);

  QColor cko(255, 240, 0);


  // A master shape gathers all the elements of the shape.

  MasterShape *spaceTruck = new SpaceTruck(cok, cko, (qrand() % 20 + 20) / 10.0);


  // Set direction and position

  spaceTruck->setRotation(qrand() % 360);

  spaceTruck->setPos(IMAGE_SIZE / 2 + ::sin((i * 6.28) / SpaceTruckCount) * 200,

                    IMAGE_SIZE / 2 + ::cos((i * 6.28) / SpaceTruckCount) * 200);

  // Add it to the graphical scene

  graphical_scene.addItem(spaceTruck);
  // and to the logical scene

  logical_scene->formes.push_back(spaceTruck);

}



// Creates a few asteroids...

for (int i = 0; i < AsteroidCount; ++i)

{

  QColor cok(150, 130, 110);

  QColor cko(255, 240, 0);



  // A master shape gathers all the elements of the shape.

  MasterShape *asteroid = new Asteroid(cok, cko,

                                       (qrand() % 20 + 20) / 10.0 /* speed */

                                       ,

                                       (double)(10 + qrand() % 40) /* radius */);

  // Set direction and position

  asteroid->setRotation(qrand() % 360);

  asteroid->setPos(IMAGE_SIZE / 2 + ::sin((i * 6.28) / AsteroidCount) * 200,

                   IMAGE_SIZE / 2 + ::cos((i * 6.28) / AsteroidCount) * 200);

  // Add it to the graphical scene

  graphical_scene.addItem(asteroid);

  // and to the logical scene

  logical_scene->formes.push_back(asteroid);

}



/* ADD 4/11/2022 */
```

```
int EnterpriseCount = 3;

// create few enterprise object

for (int i = 0; i < EnterpriseCount; ++i)

{

  QColor cok(150, 130, 110);

  // red color

  QColor cko(255, 0, 0);



  // A master shape gathers all the elements of the shape.

  MasterShape *enterprise = new Enterprise(cok, cko, (rand() % 20 + 20) / 10.0 /* speed */);

  // Set direction and position

  enterprise->setRotation(rand() % 360);

  enterprise->setPos(IMAGE_SIZE / 2 + ::sin((i * 6.28) / EnterpriseCount) * 200,

                     IMAGE_SIZE / 2 + ::cos((i * 6.28) / EnterpriseCount) * 200);

  // Add it to the graphical scene

  graphical_scene.addItem(enterprise);

  // and to the logical scene

  logical_scene->formes.push_back(enterprise);

}



// QColor _cok(150, 130, 110);

// QColor _cko(255, 0, 0);

// NiceAsteroid *niceAsteroid = new NiceAsteroid(_cok, _cko, (rand() % 20 + 20) / 10.0);



// add new niceAsteroid
```

```
int niceAsteroidCount = 1;

// create few enterprise object

for (int i = 0; i < niceAsteroidCount; ++i)

{

  QColor cok(150, 130, 110);

  // yellow color

  QColor cko(255, 255, 0);

  // A master shape gathers all the elements of the shape.

  MasterShape *niceAsteroid = new NiceAsteroid(cok, cko, 1.0);

  // advance niceAsteroid

  // niceAsteroid->advance(2);

  // niceAsteroid->setRotation(qrand() % 360);

  niceAsteroid->setPos(IMAGE_SIZE / 2 + ::sin((i * 6.28) / niceAsteroidCount) * 200,

                       IMAGE_SIZE / 2 + ::cos((i * 6.28) / niceAsteroidCount) * 200);


  // Add it to the graphical scene

  graphical_scene.addItem(niceAsteroid);

  // and to the logical scene

  logical_scene->formes.push_back(niceAsteroid);

}


// Standard stuff to initialize a graphics view with some background.

QGraphicsView view(&graphical_scene);

view.setRenderHint(QPainter::Antialiasing);

view.setBackgroundBrush(QPixmap(":/images/stars.jpg"));

view.setCacheMode(QGraphicsView::CacheBackground);
```

```
  view.setViewportUpdateMode(QGraphicsView::BoundingRectViewportUpdate);

  view.setDragMode(QGraphicsView::NoDrag); // QGraphicsView::ScrollHandDrag

  view.setWindowTitle(QT_TRANSLATE_NOOP(QGraphicsView, "Space - the final frontier"));

  view.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

  view.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

  view.resize(IMAGE_SIZE, IMAGE_SIZE);

  view.show();



  // Creates a timer that will call `advance()` method regularly.

  QTimer timer;

  QObject::connect(&timer, SIGNAL(timeout()), &graphical_scene, SLOT(advance()));

  timer.start(30); // every 30ms



  return app.exec();

}
```

```
/****************************************************************************
** Author: J.-O. Lachaud, University Savoie Mont Blanc
** (adapted from Qt colliding mices example)
**
** Copyright (C) 2015 The Qt Company Ltd.
** Contact: http://www.qt.io/licensing/
****************************************************************************/

#include <cmath>
#include <cassert>
#include <QGraphicsScene>
#include <QRandomGenerator>
#include <QPainter>
#include <QStyleOption>
#include "objects.hpp"
#include <ostream>
#include <iostream>

using namespace std;
static const double Pi = 3.14159265358979323846264338327950288419717;
// static double TwoPi = 2.0 * Pi;

// Global variables for simplicity.
QRandomGenerator RG;
LogicalScene *logical_scene = 0;
```

```
////////////////////////////////////////////////////////////////////////////////

// class Disk

////////////////////////////////////////////////////////////////////////////////


Disk::Disk(qreal r, const MasterShape *master_shape)

    : _r(r), _master_shape(master_shape) {}


QPointF

Disk::randomPoint() const

{

  QPointF p;

  do

  {

    p = QPointF((RG.generateDouble() * 2.0 - 1.0),

                (RG.generateDouble() * 2.0 - 1.0));

  } while ((p.x() * p.x() + p.y() * p.y()) > 1.0);

  return p * _r;

}


bool Disk::isInside(const QPointF &p) const

{

  return QPointF::dotProduct(p, p) <= _r * _r;

}


QRectF

Disk::boundingRect() const
```

```
{
  return QRectF(-_r, -_r, 2.0 * _r, 2.0 * _r);
}

void Disk::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
  painter->setBrush(_master_shape->currentColor());
  painter->drawEllipse(QPointF(0.0, 0.0), _r, _r);
}

///////////////////////////////////////////////////////////////////////////////
// class MasterShape
///////////////////////////////////////////////////////////////////////////////

MasterShape::MasterShape()
{
}

MasterShape::MasterShape(QColor cok, QColor cko)
    : _f(0), _state(Ok), _cok(cok), _cko(cko)
{
}

void MasterShape::setGraphicalShape(GraphicalShape *f)
{
  _f = f;
  if (_f != 0)

    _f->setParentItem(this);

}



QColor

MasterShape::currentColor() const

{

  if (_state == Ok)

    return _cok;

  else

    return _cko;

}



MasterShape::State

MasterShape::currentState() const

{

  return _state;

}



void MasterShape::paint(QPainter *, const QStyleOptionGraphicsItem *, QWidget *)

{

  // nothing to do, Qt automatically calls paint of every QGraphicsItem

}



void MasterShape::advance(int step)

{
```

```
if (!step)
  return;


// (I) Garde les objets dans la scene.
auto p = scenePos(); // pareil que pos si MasterShape est bien à la racine.
// pos() est dans les coordonnées parent et setPos aussi.
if (p.x() < -SZ_BD)
{
  auto point = parentItem() != 0
                  ? parentItem()->mapFromScene(QPointF(IMAGE_SIZE + SZ_BD - 1, p.y()))
                  : QPointF(IMAGE_SIZE + SZ_BD - 1, p.y());
  setPos(point);
}
else if (p.x() > IMAGE_SIZE + SZ_BD)
{
  auto point = parentItem() != 0
                  ? parentItem()->mapFromScene(QPointF(-SZ_BD + 1, p.y()))
                  : QPointF(-SZ_BD + 1, p.y());
  setPos(point);
}
if (p.y() < -SZ_BD)
{
  auto point = parentItem() != 0 ? parentItem()->mapFromScene(QPointF(p.x(), IMAGE_SIZE + SZ_BD - 1))
                                 : QPointF(p.x(), IMAGE_SIZE + SZ_BD - 1);
  setPos(point);
}
```

```
  else if (p.y() > IMAGE_SIZE + SZ_BD)
  {
    auto point = parentItem() != 0
                      ? parentItem()->mapFromScene(QPointF(p.x(), -SZ_BD + 1))
                      : QPointF(p.x(), -SZ_BD + 1);
    setPos(point);
  }

  // (II) regarde les intersections avec les autres objets.
  if (logical_scene->intersect(this))
    _state = Collision;
  else
    _state = Ok;
}

QPointF
MasterShape::randomPoint() const
{
  assert(_f != 0);
  return mapToParent(_f->randomPoint());
}

bool MasterShape::isInside(const QPointF &p) const
{
  assert(_f != 0);
  return _f->isInside(mapFromParent(p));
```

```
}


QRectF

MasterShape::boundingRect() const

{

  assert(_f != 0);

  return mapRectToParent(_f->boundingRect());

}


/////////////////////////////////////////////////////////////////////////////////

// class MasterShape

/////////////////////////////////////////////////////////////////////////////////


Asteroid::Asteroid(QColor cok, QColor cko, double speed, double r)

    : MasterShape(cok, cko), _speed(speed)

{

  // This shape is very simple : just a disk.

  Disk *d = new Disk(r, this);

  // Tells the asteroid that it is composed of just a disk.

  this->setGraphicalShape(d);

}


void Asteroid::advance(int step)

{

  if (!step)

    return;
  setPos(mapToParent(_speed, 0.0));
  MasterShape::advance(step);
}

////////////////////////////////////////////////////////////////////////////////
// class LogicalScene
////////////////////////////////////////////////////////////////////////////////

LogicalScene::LogicalScene(int n)
    : nb_tested(n) {}

bool LogicalScene::intersect(MasterShape *f1, MasterShape *f2)
{
  for (int i = 0; i < nb_tested; ++i)
  {
    if (f2->isInside(f1->randomPoint()) || f1->isInside(f2->randomPoint()))
      return true;
  }
  return false;
}

bool LogicalScene::intersect(MasterShape *f1)
{
  for (auto f : formes)
    if ((f != f1) && intersect(f, f1))
      return true;
  return false;
}

Rectangle::Rectangle(QPointF upLeft, QPointF downRight, const MasterShape *master_shape)
    : _downRight(downRight), _upLeft(upLeft), _master_shape(master_shape) {}

QPointF Rectangle::randomPoint() const
{

  // calculate the norm of the vector _downRight.x() - _upLeft.x() _downRight.y() - _upLeft.y()

  double x = RG.generateDouble() * (_downRight.x() - _upLeft.x()) + _upLeft.x();
  double y = RG.generateDouble() * (_downRight.y() - _upLeft.y()) + _upLeft.y();

  return QPointF(x, y);
}

bool Rectangle::isInside(const QPointF &p) const
{
  return (p.x() >= _upLeft.x()) && (p.x() <= _downRight.x()) && (p.y() >= _upLeft.y()) && (p.y() <= _downRight.y());
}

QRectF Rectangle::boundingRect() const
{
  return QRectF(_upLeft, _downRight);
}
```

```
void Rectangle::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
  painter->setBrush(_master_shape->currentColor());
  painter->drawRect(QRectF(_upLeft, _downRight));
}

SpaceTruck::SpaceTruck(QColor cok, QColor cko, double speed)
    : MasterShape(cok, cko), _speed(speed)
{
  // QPointF upLeft = QPointF(0.0, 0.0);
  // QPointF downRight = QPointF(100.0, 20.0);

  // Rectangle *rectangle = new Rectangle(downRight, upLeft, this);

  // this->setGraphicalShape(rectangle);

  // In SpaceTruck constructor
  Rectangle *d1 = new Rectangle(QPointF(-80, -10), QPointF(0, 10), this);
  Rectangle *d2 = new Rectangle(QPointF(10, -10), QPointF(30, 10), this);
  Rectangle *d3 = new Rectangle(QPointF(0, -3), QPointF(10, 3), this);
  Union *u23 = new Union(d2, d3);
  Union *u = new Union(d1, u23);

  // Tells the space truck that it is composed of the union of the previous shapes.
  this->setGraphicalShape(u);
```

```
}


void SpaceTruck::advance(int step)
{
  if (!step)
    return;
  setPos(mapToParent(_speed, 0.0));
  setRotation(rotation() + 2.0);
  MasterShape::advance(step);
}


// Union(GraphicalShape *f1, GraphicalShape *f2);
Union::Union(GraphicalShape *f1, GraphicalShape *f2)
{
  _f1 = f1;
  _f2 = f2;

  _f1->setParentItem(this);
  _f2->setParentItem(this);
}


Union::Union() : _f1(nullptr), _f2(nullptr){};
QPointF Union::randomPoint() const
{
  // take random point with bool b mutable change everytime
  if (_b)
  {
    _b = false;
    return _f1->randomPoint();
  }
  else
  {
    _b = true;
    return _f2->randomPoint();
  }
}

bool Union::isInside(const QPointF &p) const
{
  return _f1->isInside(p) || _f2->isInside(p);
}

QRectF Union::boundingRect() const
{
  return _f1->boundingRect() | _f2->boundingRect();
}

void Union::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
  // painter->setBrush(Qt::red);
  // _f1->paint(painter, nullptr, nullptr);
  // _f2->paint(painter, nullptr, nullptr);
```

```
}


Transformation::Transformation(GraphicalShape *f, QPointF dx, qreal angle)
{
  _f = f;
  _f->setParentItem(this);
  _dx = dx;
  _angle = angle;
  this->setPos(dx);
  this->setRotation(angle);
}


void Transformation::paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget)
{
  //   painter->save();
  //   painter->translate(_dx);
  //   painter->rotate(_angle);
  //   _f->paint(painter, option, widget);
  //   painter->restore();
}


QPointF Transformation::randomPoint() const
{
  return _f->randomPoint();
}
```

```
bool Transformation::isInside(const QPointF &p) const

{

  return _f->isInside(QPointF(cos(_angle) * (p.x() - _dx.x()) - sin(_angle) * (p.y() - _dx.y()), sin(_angle) * (p.x() - _dx.x

}



QRectF Transformation::boundingRect() const

{

  return mapRectToParent(_f->boundingRect());

}



void Transformation::setAngle(qreal angle)

{

  _angle = angle;

  setRotation(angle);

}



Enterprise::Enterprise(QColor cok, QColor cko, double speed)

    : MasterShape(cok, cko), _speed(speed)

{

  Rectangle *r1 = new Rectangle(QPointF(-100, -8), QPointF(0, 8), this);

  Rectangle *r2 = new Rectangle(QPointF(-100, -8), QPointF(0, 8), this);

  Rectangle *rb = new Rectangle(QPointF(-40, -9), QPointF(40, 9), this);

  Rectangle *s1 = new Rectangle(QPointF(-25, -5), QPointF(25, 5), this);

  Rectangle *s2 = new Rectangle(QPointF(-25, -5), QPointF(25, 5), this);

  Disk *d = new Disk(40.0, this);

  Transformation *t1 = new Transformation(r1, QPointF(0., 40.0));
  Transformation *t2 = new Transformation(r2, QPointF(0., -40.0));

  Transformation *td = new Transformation(d, QPointF(70., 0.0));

  Transformation *ts1 = new Transformation(s1, QPointF(-30.0, 0.0), 0.0);

  Transformation *us1 = new Transformation(ts1, QPointF(0.0, 0.0), 45.0);

  Transformation *ts2 = new Transformation(s2, QPointF(-30.0, 0.0), 0.0);

  Transformation *us2 = new Transformation(ts2, QPointF(0.0, 0.0), -45.0);

  Union *back = new Union(t1, t2);

  Union *head = new Union(rb, td);

  Union *legs = new Union(us1, us2);

  Union *body = new Union(legs, back);

  Union *all = new Union(head, body);

  this->setGraphicalShape(all);

}


void Enterprise::advance(int step)

{

  if (!step)

    return;

  setPos(mapToParent(_speed, 0.0));

  setRotation(rotation() + 2.0);

  MasterShape::advance(step);

}


// ImageShape definition

ImageShape::ImageShape(const QPixmap &pixmap, const MasterShape *master_shape)

    : _pixmap(pixmap), _master_shape(master_shape)
```

```
{
  _mask = _pixmap.mask();
  _mask_img = QImage(_mask.toImage().convertToFormat(QImage::Format_Mono));
}


QPointF ImageShape::randomPoint() const
{
  int x = qrand() % _pixmap.width();
  int y = qrand() % _pixmap.height();
  while (_mask_img.pixelIndex(x, y) == 0)
  {
    x = qrand() % _pixmap.width();
    y = qrand() % _pixmap.height();
  }
  return QPointF(x, y);
}


bool ImageShape::isInside(const QPointF &p) const
{


  QRectF rect = boundingRect();
  if (rect.contains(p))
  {
    return _mask_img.pixelIndex(p.x(), p.y()) != 0;
  }
  else
  {
    return false;
  }
}

QRectF ImageShape::boundingRect() const
{
  return _pixmap.rect();
}

void ImageShape::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
  painter->drawPixmap(QPointF(0.0, 0.0), _pixmap);
  if (_master_shape->currentState() == MasterShape::Collision)
  {
    painter->setOpacity(0.5);
    painter->setBackgroundMode(Qt::TransparentMode);
    painter->setPen(_master_shape->currentColor());
    painter->drawPixmap(QPointF(0.0, 0.0), _mask);
  }
}

// NiceAsteroid definition constructor
NiceAsteroid::NiceAsteroid(QColor cok, QColor cko, double speed)
    : MasterShape(cok, cko), _speed(speed)
{
  _asteroid = new ImageShape(QPixmap(":/images/asteroid.gif"), this);

  Transformation *t1 = new Transformation(_asteroid, _asteroid->boundingRect().center(), 0);

  Transformation *t2 = new Transformation(t1, QPointF(0.0, 0.0), 0);

  _t = t2;

  this->setGraphicalShape(_asteroid);

}


void NiceAsteroid::advance(int step)

{

  if (!step)

    return;

  //_t->setAngle(_t->_angle + 0.01);

  setPos(mapToParent(_speed / 400, 0.0));

  // generate a qrand between 0.0 and 1.0

  double r = (double)qrand() / (double)RAND_MAX;

  // generate a qrand between 0 and 360

  double angle = (double)qrand() / (double)RAND_MAX * 360.0;

  this->setRotation(angle);

  MasterShape::advance(step);

}

//  END ADD
```

```
/***************************************************************************
** Author: J.-O. Lachaud, University Savoie Mont Blanc
** (adapted from Qt colliding mices example)
**
** Copyright (C) 2015 The Qt Company Ltd.
** Contact: http://www.qt.io/licensing/
***************************************************************************/

#ifndef OBJECTS_HPP
#define OBJECTS_HPP

#include <vector>
#include <QGraphicsItem>
#include <QBitmap>

static const int IMAGE_SIZE = 600;
static const int SZ_BD = 100;

/// @brief Abstract class that describes a graphical object with additional
/// methods for testing collisions.
struct GraphicalShape : public QGraphicsItem
{
  virtual QPointF randomPoint() const = 0;
  virtual bool isInside(const QPointF &p) const = 0;

  // Already in QGraphicsItem
  // virtual QRectF  boundingRect() const override;
};


/// @brief Polymorphic class that represents the top class of any complex
/// shape.
///
/// It takes care of memorizing collisions and storing the
/// current main coordinates of a shape.
struct MasterShape : public GraphicalShape
{
  enum State
  {
    Ok,
    Collision
  };
  MasterShape();
  MasterShape(QColor cok, QColor cko);
  void setGraphicalShape(GraphicalShape *f);
  virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                     QWidget *widget) override;
  virtual QPointF randomPoint() const override;
  virtual bool isInside(const QPointF &p) const override;
  virtual QRectF boundingRect() const override;

  // Checks if this shape collides with another shape and forces also
  // the shapes to stay in the graphical view.
  virtual void advance(int step) override;
  State currentState() const;
  QColor currentColor() const;

protected:
  GraphicalShape *_f;
  State _state;
  QColor _cok, _cko;
};

/// @brief An asteroid is a simple shape that moves linearly in some direction.
struct Asteroid : public MasterShape
{
  Asteroid(QColor cok, QColor cko, double speed, double r);
  // moves the asteroid forward according to its speed.
  virtual void advance(int step) override;

protected:
  double _speed;
};

/// @brief A disk is a simple graphical shape.
///
/// It points to its master shape
/// in order to know in which color it must be painted.
struct Disk : public GraphicalShape
```

```
{
  Disk(qreal r, const MasterShape *master_shape);
  virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                     QWidget *widget) override;
  virtual QPointF randomPoint() const override;
  virtual bool isInside(const QPointF &p) const override;
  virtual QRectF boundingRect() const override;
  const qreal _r;
  const MasterShape *_master_shape;
};

struct Rectangle : public GraphicalShape
{
  QPointF _upLeft;    // haut gauche
  QPointF _downRight; // bas droite

  const MasterShape *_master_shape;

  Rectangle(QPointF p, QPointF q, const MasterShape *master_shape);

  virtual QPointF randomPoint() const override;
  virtual bool isInside(const QPointF &p) const override;
  virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                     QWidget *widget) override;
  virtual QRectF boundingRect() const override;
};
```

```
struct SpaceTruck : public MasterShape
{
  SpaceTruck(QColor cok, QColor cko, double speed);
  virtual void advance(int step) override;

protected:
  double _speed;
};

struct Union : public GraphicalShape
{
  mutable bool _b;
  // two variable _f1 and _f2
  GraphicalShape *_f1;
  GraphicalShape *_f2;
  Union(MasterShape *master_shape, GraphicalShape *f,
        GraphicalShape *g);
  Union(GraphicalShape *f1, GraphicalShape *f2);
  Union();
  // Union(GraphicalShape &f1, GraphicalShape &f2);
  virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                     QWidget *widget) override;
  virtual QPointF randomPoint() const override;
  virtual bool isInside(const QPointF &p) const override;
  virtual QRectF boundingRect() const override;
```

```
};


/* Add 4/11/2022 */


struct Transformation : public GraphicalShape

{

  GraphicalShape *_f;

  QPointF _dx;

  qreal _angle;

  Transformation(GraphicalShape *f, QPointF dx, qreal angle = 0);

  virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,

                  QWidget *widget) override;

  virtual QPointF randomPoint() const override;

  virtual bool isInside(const QPointF &p) const override;

  virtual QRectF boundingRect() const override;


  // setAngle

  void setAngle(qreal angle);

};


// create class Enterprise public MasterShape

struct Enterprise : public MasterShape

{

  Enterprise(QColor cok, QColor cko, double speed);

  virtual void advance(int step) override;
```

```
private:

  double _speed;

};


class ImageShape : public GraphicalShape

{

public:

  ImageShape(const QPixmap &pixmap, const MasterShape *master_shape);


  virtual void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,

                     QWidget *widget) override;

  virtual QPointF randomPoint() const override;

  virtual bool isInside(const QPointF &p) const override;

  virtual QRectF boundingRect() const override;


private:

  const QPixmap _pixmap;

  const MasterShape *_master_shape;

  QBitmap _mask;

  QImage _mask_img;

};


// create class NiceAsteroid who take two Transformations and speed


struct NiceAsteroid : public MasterShape

{
  NiceAsteroid(QColor cok, QColor cko, double speed);

  // moves the asteroid forward according to its speed.

  virtual void advance(int step) override;



  GraphicalShape *_asteroid;

  Transformation *_t;

  double _speed;

};



/// @brief A class to store master shapes and to test their possible

/// collisions with a randomized algorithm.

struct LogicalScene

{

  std::vector<MasterShape *> formes;

  int nb_tested;



  /// Builds a logical scene where collisions are detected by checking

  /// \a n random points within shapes.

  ///

  /// @param n any positive integer.

  LogicalScene(int n);

  /// Given two shapes \a f1 and \a f2, returns if they collide.

  /// @param f1 any master shape.

  /// @param f2 any different master shape.

  /// @return 'true' iff they collide, i.e. have a common intersection.

  bool intersect(MasterShape *f1, MasterShape *f2);
  /// @param f1 any master shape.

  /// @return 'true' iff it collides with a different master shape stored in this logical scene.

  bool intersect(MasterShape *f1);

};


extern LogicalScene *logical_scene;


#endif
```