SFMLCollision

Jack Featherstone
Version
03/26/2019

Table of Contents

Table of contents

SFML Collision

This project is an extension of a previous repo (`SFMLHitbox`) where the goal is to create an easily implemented object that allows for accurate hitbox detection on both convex and concave shapes. By default, SFML only has a ConvexShape, for which there is no intersects function.

The user will be able to create a **Polygon** shape in two different ways:

1. By providing a texture to the constructor which will identify important pixels to be included as verticies and create the shape based on these. This will also allow for the user to define the level of accuracy as either pixel perfect or approximate.
2. By proving a vector of pre assembled verticies, possibly from some other SFML class by using the getPoints() method.
3. By directly passing in any of the SFML shape objects (CircleShape, RectangleShape, ConvexShape)

it can do so far:

1. Create a set of verticies from an image file with a varying level of detail to optimize the number of verticies (given by the user)
2. Draw the object and use common SFML transformations (rotate, scale, move)
3. Detect intersection between **Polygon** type and any other SFML shape regardless of concavity (without providing resultant vector)

it can't do (that I am working on):

1. Detecting whether one shape is inside of another
2. Finding the resultant vector when two polygons collide
3. Optimize vertex reduction for objects even more
4. Not have a 700 line constructor for image files (working on refactoring)

There are several aspects to how this library will work, many of which are complicated and so will be overviewed below. This information is not necessary to use the tool, but can be helpful to understand what is going on (especially if you want to contribute to the code).

For information on practical usage of the tool, see the documentation page, which has yet to be created.

This isn't so much mathematical as it is programmatic. This section of code can be found in the polygon constructor that takes in a texture, a level of detail, and an optional list of colors.

Note: the notation for images below is as follows

0 denotes empty space (will not be included)
1 denotes a vertex that will be included
2 denotes the inside of a shape that will not be included
3 denotes a vertex that was included, but no longer will be as it is not necessary

following images are generated from this source image:

The first operation that is performed isolates the important colors in the image. We divide the image into a vertex of colors and then decide whether or not we would like to include a given

pixel in the polygon. By default, any color that isn't (0, 0, 0, 0) will be included, but if a list of ignored colors is provided, anything on that list will be excluded as well.

Now that we have a vertex of 1s representing parts to be included and 0s that represent empty space, we begin removing excess points. The most obvious part to remove here is the inside of the shape. To properly represent our image, we really only need to identify the outline, so we remove anything that isn't accessible to an outside shape. In this stage, we also fill in any inside details such that they are removed. This is done by drawing eight lines in all of the cardinal directions and diagonals, and seeing if they encounter a point that is included. If all eight do, the point must be inside the outline of the object. This can be seen in Images/test6.png and Images/test7.png which will produce the exact same polygon.

The next group of verticies to be removed are those that are in a straight line, of which we really only need to two endpoints of said line. This is really just checking above and below and left and right of a point and removing the intermediate verticies.

The next step of identify verticies is very similar to the previous one, but instead works with diagonal lines.

Finally, we remove any intermediate points along the horizontals, similar to what we did earlier with verticle and horizontal lines.

These steps remove just about all of the excess verticies we have, and provide a rather good, just-about pixel-perfect representation of our texture.

The next step is arguably the hardest, and involves adding our verticies in the proper order. This seems like it should be an easy step, just iterate through our vector and add any points that have a value of 1, but alas. If we were to do this, the order of our verticies would be incorrect, and shape would zig-zag back and forth, and be a terrible representation of the actual shape. Instead, we have to follow one direction, clockwise or counter-clockwise, consistently, which is easier said than done.

One of the saving graces of this process is that due to our vertex reduction, just about every vertex should only be connected to either one two verticies (1), or a place that used to be a vertex (3). The case where this is not true is exmplified in the last image above, where at the top of the shape we have a 2x2 square of all verticies.

We can then keep track of which places (1s or 3s) we have visited, so we don't repeat any values, and each location will only have one other location to go to.

We begin the process by starting at (0, 0) and moving horizontally until we encounter the first pixel marked as a '1'. Once we have this value, we add it to our list of final points, and check the adjacent pixels for either 1s or 3s. The order we check in is clockwise, beginning from the top and ending at the top left.

Depending on whether or not we find a 3 or a 1 changes the next step slightly. In either case, we will move to that pixel and repeat the process of searching adjacent pixels, but will only add the coordinate to our final vector if the value is a 1.

One of the options that is presented to the user is the level of detail in which to model the texture. There are currently three different levels: Less, More, Optimal. Each of these values represents a percent difference the final polygon is allowed to be from the initial one. No matter the level of detail, the previous conversion is always the same, and this trimming is only done afterwards.

Essentially, we iterate through every points on the shape and find the difference in area if we were to remove it. If we find that the difference is below a certain value, which is based on the level of detail (Less - 15%, More, 7.5%, Optimal - 1%), we actually remove that point and continue. This allows for us to remove insignificant points systematically.

Since our polygon intersection method is based on the lines that surround each shape, one of the most important parts of intersection is to properly be able to address line intersections. The method we use can be seen in (1), but can be summarized as follows: we take ratios of the start and end coordinates of each of the two lines to calculate the percent along each line where they intersect. If these two percents are between 0 and 1, it means that they properly intersect within the domain of both lines, and actually do intersect. This allows to to not only find whether two line segments intersect on their given domain, but also allows us to find the intersection point quite easily. Another benefit of using this method is that if we want to extend a line segment to a full line (which will be useful in finding whether one shape is inside another), we can easily do this by only checking one of the percent values, and ignoring the other.

Before we get to the main intersection method described below, we want to do a simpler test to determine if collision between the two objects is possible. We do this by using SFML's built in FloatRect intersection method, using the absolute bounds of the shape. This means that we use the getGlobalBounds() method for each shape to find whether any of the lines could possibly overlap. This should be taken into account when looking at benchmarks between our total collision method vs. the built in one in SFML, as ours uses the latter.

The main way we detect collisions between two polygons is by taking the vector of lines contained within each polygon class, and iterating through every line with every other line. This may seem to be a rather inefficient method, but since our line intersection method is so simple, the O(n^2) complexity doesn't spiral out of control. Since we also optimize the number of verticies, we rarely have a shape with more than 50 lines, normally we have between 10 and 20. In the future, I would like to optimize this somehow, but as of now, it works and there are more pressing issues to deal with

Not quite sure how to do this just yet, but so far my thoughts are as follows. If we can take the point of intersection for each line (which we can easily find) along with the vertex end of the segment that is inside of the shape (this is the harder part). We can take the vector between these two, as a resultant, and average it with any other resultant vectors, and place them at the average of the verticies found inside the other shape. This is a highly theoretical process and I and have no way to test it as of now, but hopefully a more concrete solution (either this or another one) will stick.

1. `Parametric line intersection method`

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Class Documentation

Line Class Reference

The line class is integral to detecting collisions between polygon classes, and thus is used internalling in the intersection method, and **Polygon** objects. A note about these is that the name is a slight misnomer, in that by default, the lines are actually line segments, having a finite length and a start and end point.

```
#include <Line.hpp>
```

Public Member Functions

Line (Vector2f p1, Vector2f p2)

*Construct a new **Line** object from two endpoints. They don't necessarily have to be in order of higher/lower x or y.*

Line ()

*Construct a new **Line** object with no information, using for comparing lines to see if they exist.*

float **y** (float x)

Find the y value at any given x on the line (doesn't deal with endpoints/domain)

bool **intersects** (**Line** line)

Wrapper for our more involved intersection method that doesn't deal with any other parameters but the line.

bool **intersects** (**Line** line, Vector2f &intersectionPoint, bool extendLine=false)

Check to see if two lines intersect within their domain (between the endpoints)

float **getAngle** ()

Return the angle the line makes with the horizontal x axis.

float **getIntercept** ()

Return the intercept, or when the line crosses the y axis.

float **getSlope** ()

Return the slope, or rise over run, of the line.

Vector2f **getStart** ()

Return the first point used in the creation of the line object. Does not necessarily need to be before (in either x or y) the second point. Returned point does have any offset effects applied to it.

Vector2f **getEnd** ()

Return the second point used in the creation of the line object. Does not necessarily need to be after (in either x or y) the first point. Returned point does have any offset effects applied to it.

Vector2f **getPerpendicular** ()

Get a vector that represents the perpendicular to the line, always has a magnitude of 1.

void **offset** (Vector2f offset)

Offset the endpoints of our line (and recalculate the intercept) in a way that does NOT stack with previous offsets (See source for more info)

RectangleShape * **getDrawable** (Color color=Color::Cyan)

This method was created for the sole purpose of dubugging line intersections, and should never really be used to draw a line It is for this reason that we don't hold the rectangle shape as a member variable, because once this issue has been solved, we should never need this again.

void **rotate** (Vector2f center, float angle)

Detailed Description

The line class is integral to detecting collisions between polygon classes, and thus is used internalling in the intersection method, and **Polygon** objects. A note about these is that the name is a slight misnomer, in that by default, the lines are actually line segments, having a finite length and a start and end point.

Dependencies: <SFML/Graphics.hpp> <iostream> <tgmath.h>

Namespaces: sf (SFML) std (Standard)

Constructor & Destructor Documentation

Line::Line (Vector2f *p1*, Vector2f *p2*)

Construct a new **Line** object from two endpoints. They don't necessarily have to be in order of higher/lower x or y.

Parameters:

p1	Endpoint 1
p2	Endpoint 2

Line::Line ()

Construct a new **Line** object with no information, using for comparing lines to see if they exist.

Member Function Documentation

float Line::getAngle ()

Return the angle the line makes with the horizontal x axis.

Returns:

float The line's angle

RectangleShape * Line::getDrawable (Color *color* = `Color::Cyan`)

This method was created for the sole purpose of dubugging line intersections, and should never really be used to draw a line It is for this reason that we don't hold the rectangle shape as a member variable, because once this issue has been solved, we should never need this again.

@depracated

Parameters:

color	The color to draw the line as

Returns:

RectangleShape* A drawable rectangle that represents our line

Vector2f Line::getEnd ()

Return the second point used in the creation of the line object. Does not necessarily need to be after (in either x or y) the first point. Returned point does have any offset effects applied to it.

Returns:

Vector2f The second point (x, y)

float Line::getIntercept ()

Return the intercept, or when the line crosses the y axis.

Returns:

float Return the intercept

Vector2f Line::getPerpendicular ()

Get a vector that represents the perpendicular to the line, always has a magnitude of 1.

Returns:

Vector2f A vector in the perpendicular direction to the line, with a magnitude of 1

float Line::getSlope ()

Return the slope, or rise over run, of the line.

Returns:

float Return the slope of the line

Vector2f Line::getStart ()

Return the first point used in the creation of the line object. Does not necessarily need to be before (in either x or y) the second point. Returned point does have any offset effects applied to it.

Returns:

Vector2f The first point (x, y)

bool Line::intersects (Line *line*)

Wrapper for our more involved intersection method that doesn't deal with any other parameters but the line.

Parameters:

line	The other line

Returns:

true The two lines are intersecting
false The two lines are not intersecting

bool Line::intersects (Line *line*, Vector2f & *intersectionPoint*, bool *extendLine* = `false`)

Check to see if two lines intersect within their domain (between the endpoints)

Parameters:

line	The other line we are checking
intersectionPoint	A reference to a point that will hold the intersection point if it exist, or (-1, -1) if it doesn't
extendLine	If this is true, we extend the first line from a line segment to an infinite line and check collision with that

Returns:

true The two lines are intersecting
false The two lines are not intersecting

void Line::offset (Vector2f *offset*)

Offset the endpoints of our line (and recalculate the intercept) in a way that does NOT stack with previous offsets (See source for more info)

Parameters:

offset	The amount we want to offset x and y by

float Line::y (float *x*)

Find the y value at any given x on the line (doesn't deal with endpoints/domain)

Parameters:

x	Our x value

Returns:

float The y value

The documentation for this class was generated from the following files:

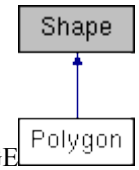
include/Line.hpp
src/Line.cpp

Polygon Class Reference

The polygon object is the most important aspect of our collisions class, and accounts for most of what will likely be used externally. Polygons can be created through either a texture, a vector of points, or any other child class of sf::Shape (CircleShape, RectangleShape, etc.). **Polygon** objects act very similarly to the other sf::Shape classes, being able to be drawn by a RenderWindow and are able to be transformed by any method from sf::Transformable. On top of these stock attributes, Polygons are also able to be either convex or concave (unlike convexshape) and have a method to detect collision between two instances of the class, or between the class and any other sf::Shape class. With this ability comes a few more paramters that can be changed about the shape, including the density, rigidity, moment of inertia, etc. that all affect how a shape reacts to colliding with another shape.

`#include <Polygon.hpp>`

Inheritance diagram for Polygon:

IMAGE 

Public Member Functions

Polygon (Texture *texture, Detail detail=Detail::Optimal, vector< Color > ignoredColors={})

*Construct a new **Polygon** object from a given texture (image).*

Polygon (vector< Vector2f > points)

*Construct a new **Polygon** object from a vector of points.*

Polygon (CircleShape shape)

*Construct a new **Polygon** object from a sf::CircleShape object.*

Polygon (RectangleShape shape)

*Construct a new **Polygon** object from a sf::RectangleShape object.*

Polygon (ConvexShape shape)

*Construct a new **Polygon** object from a sf::ConvexShape object.*

virtual size_t **getPointCount** () const

Get the number of verticies on our polygon.

virtual Vector2f **getPoint** (size_t index) const

Get the vertex at index in the vector m_points.

vector< Vector2f > **getPoints** ()

Returns the entire vector of points that represent the shape, without any modifications from transformations (rotate, move, scale)

vector< **Line** > **getLines** ()

Return the lines that represent the polygon's outline/border.

float **getFarthestVertex** ()

Returns the distance of the farthest vertex from the centroid. Calculated in findCentroid()

Vector2f **getCentroid** ()

Returns the centroid of the shape (does not recalculate it)

void **setSolid** (bool state)

Set whether the shape is solid (can collide with other shapes)

bool **isSolid** ()

Check whether or not the shape can collide with other shapes.

void **setRigidity** (float value)

Set how much energy is conserved when this object collides with another. 0 for no energy conserved (completely inelastic collision) and 1 for completely elastic (all energy conserved)

float **getRigidity** ()

Get how much energy is conserved when this object collides with another. 0 for no energy conserved (completely inelastic collision) and 1 for completely elastic (all energy conserved)

void **setMovableByCollision** (bool value)

Set whether the shape can be moved by being collided with by another object.

bool **isMovableByCollision** ()

Get whether the shape can be moved by being collided with by another object.

void **setDensity** (float newDensity)

Set the density of the object, used in calculate its mass and moment of inertia (default is 1) and recalculate both values.

float **getDensity** ()

Get the relative density of the polygon.

float **getMass** ()

Return the mass of the polygon, using the density and area to calculate.

float **getMomentOfInertia** ()

Return the moment of inertia of the polygon, using the density and vertex distribution.

void **setVelocity** (Vector2f newVelocity)

Changes the linear velocity of the polygon to the paramter provided.

Vector2f **getVelocity** ()

Returns the current linear velocity of the shape.

void **setAngularVelocity** (float newAngularVelocity)

Changes the angular velocity of the polygon to the paramter provided.

float **getAngularVelocity** ()

Returns the current angular velocity of the polygon to the paramter provided.

void **update** (float elapsedTime)

Updates the shape and applies both linear and angular velocity to update the position and rotation of the polygon.

void **adjustVelocityFromCollision** (Vector2f resultant, **Polygon** shape)

Currently WIP!!

void **setScale** (const Vector2f &**scale**)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

void **setScale** (float xScale, float yScale)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

void **scale** (const Vector2f &scale)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

void **scale** (float xFactor, float yFactor)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

void **setRotation** (float angle)

An overriden method from sf::Shape that changes the rotation like its super-counterpart and also recreates the lines that represent the shape.

void **rotate** (float angle)

An overriden method from sf::Shape that changes the rotation like its super-counterpart and also recreates the lines that represent the shape.

void **setPosition** (const Vector2f &position)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

void **setPosition** (float x, float y)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

void **move** (const Vector2f &offset)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

void **move** (float xOffset, float yOffset)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

bool **intersects** (**Polygon** shape)

Check the intersection between two polygon shapes. Has three levels of detection, to reduce unecessary calculations and resource usage. Returns no information about collision after-effects or resultants. For that, see ***intersects(Polygon shape, Vector2f& resultant)****.*

bool **intersects** (RectangleShape shape)

A wrapper method to check the intersection between a ***Polygon*** *shape and a RectangleShape by converting it to a polygon and then using our full intersection method See* ***intersects(Polygon shape)*** *for the full intersection method.*

bool **intersects** (CircleShape shape)

A wrapper method to check the intersection between a ***Polygon*** *shape and a CircleShape by converting it to a polygon and then using our full intersection method See* ***intersects(Polygon shape)*** *for the full intersection method.*

bool **intersects** (ConvexShape shape)

A wrapper method to check the intersection between a ***Polygon*** *shape and a ConvexShape by converting it to a polygon and then using our full intersection method See* ***intersects(Polygon shape)*** *for the full intersection method.*

bool **intersects** (**Polygon** shape, Vector2f &resultant)

Currently WIP!!

bool **intersects** (RectangleShape shape, Vector2f &resultant)
bool **intersects** (CircleShape shape, Vector2f &resultant)
bool **intersects** (ConvexShape shape, Vector2f &resultant)
bool **contains** (**Polygon** shape)

Not implemented!

bool **contains** (RectangleShape shape)

Not implemented!

bool **contains** (CircleShape shape)

Not implemented!

bool **contains** (ConvexShape shape)

Not implemented!

float **getArea** ()

Return the area of the polygon.

Static Public Member Functions

static void **getArea** (vector< Vector2f > points, float &value)

This is a static method that finds the area of any given shape (vector of points) Ngl, I don't remember where I found this method for finding the area of a polygon, but will post when I find it.

Detailed Description

The polygon object is the most important aspect of our collisions class, and accounts for most of what will likely be used externally. Polygons can be created through either a texture, a vector of points, or any other child class of sf::Shape (CircleShape, RectangleShape, etc.). **Polygon** objects act very similarly to the other sf::Shape classes, being able to be drawn by a RenderWindow and are able to be transformed by any method from sf::Transformable. On top of these stock attributes, Polygons are also able to be either convex or concave (unlike convexshape) and have a method to detect collision between two instances of the class, or between the class and any other sf::Shape class. With this ability comes a few more paramters that can be changed about the shape, including the density, rigidity, moment of inertia, etc. that all affect how a shape reacts to colliding with another shape.

Dependencies: <SFML/Graphics.hpp> <iostream> <tgmath.h>

Namespaces: sf (SFML) std (Standard)

Constructor & Destructor Documentation

Polygon::Polygon (Texture * *texture*, Detail *detail* = `Detail::Optimal`, vector< Color > *ignoredColors* = `{}`)

Construct a new **Polygon** object from a given texture (image).

Parameters:

texture	The texture for the shape/sprite we want to model
detail	The level of detail to keep in the shape, from least to most: Less, More, Optimal, Exact
ignoredColors	By default, all pixels that arent (0, 0, 0, 0) will be included, any colors specified here will also be ignored

Polygon::Polygon (vector< Vector2f > *points*)

Construct a new **Polygon** object from a vector of points.

Parameters:

points	The points that constitute our shape

Polygon::Polygon (CircleShape *shape*)

Construct a new **Polygon** object from a sf::CircleShape object.

Parameters:

shape	The CircleShape object whose points we will use

Polygon::Polygon (RectangleShape *shape*)

Construct a new **Polygon** object from a sf::RectangleShape object.

Parameters:

shape	The RectangleShape object whose points we will use

Polygon::Polygon (ConvexShape *shape*)

Construct a new **Polygon** object from a sf::ConvexShape object.

Parameters:

shape	The ConvexShape object who points we will use

Member Function Documentation

void Polygon::adjustVelocityFromCollision (Vector2f *resultant*, Polygon *shape*)

Currently WIP!!

Parameters:

resultant	The unit vector in the direction of the new motion
shape	The other colliding shape

bool Polygon::contains (Polygon *shape*)

Not implemented!

Parameters:

shape	

Returns:

true
false

bool Polygon::contains (RectangleShape *shape*)

Not implemented!

Parameters:

shape	

Returns:

true
false

bool Polygon::contains (CircleShape *shape*)

Not implemented!

Parameters:

shape	

Returns:

true
false

bool Polygon::contains (ConvexShape *shape*)

Not implemented!

Parameters:

shape	

Returns:

true
false

float Polygon::getAngularVelocity ()

Returns the current angular velocity of the polygon to the paramter provided.

Returns:

float Current angular velocity of the polygon

void Polygon::getArea (vector< Vector2f > *points*, float & *value*) `[static]`

This is a static method that finds the area of any given shape (vector of points) Ngl, I don't remember where I found this method for finding the area of a polygon, but will post when I find it.

Parameters:

points	A Vector of points the represent our shape. See **Polygon::getPoints()**
value	A referenced float that our area will be stored in

float Polygon::getArea ()

Return the area of the polygon.

Returns:

float The area of the polygon

Vector2f Polygon::getCentroid ()

Returns the centroid of the shape (does not recalculate it)

Returns:
Vector2f The centroid of the shape

float Polygon::getDensity ()

Get the relative density of the polygon.

Returns:
float The density of the polygon

float Polygon::getFarthestVertex ()

Returns the distance of the farthest vertex from the centroid. Calculated in findCentroid()

Returns:
float The farthest distance of the shape

vector< Line > Polygon::getLines ()

Return the lines that represent the polygon's outline/border.

Returns:
vector<Line> A vector of lines that represent the outline

float Polygon::getMass ()

Return the mass of the polygon, using the density and area to calculate.

Returns:
float The mass of the shape

float Polygon::getMomentOfInertia ()

Return the moment of inertia of the polygon, using the density and vertex distribution.

Returns:
float The moment of inertia of the shape

Vector2f Polygon::getPoint (size_t *index*) const `[virtual]`

Get the vertex at index in the vector m_points.

Parameters:

index	The index of the point we are looking for

Returns:

Vector2f The point at index in m_points

size_t Polygon::getPointCount () const `[virtual]`

Get the number of verticies on our polygon.

Returns:

size_t The number of verticies

vector< Vector2f > Polygon::getPoints ()

Returns the entire vector of points that represent the shape, without any modifications from transformations (rotate, move, scale)

Returns:

vector<Vector2f> Our shape's vector of verticies

float Polygon::getRigidity ()

Get how much energy is conserved when this object collides with another. 0 for no energy conserved (completely inelastic collision) and 1 for completely elastic (all energy conserved)

Returns:

float The rigidity, 0 for complete inelastic, 1 for complete elastic

Vector2f Polygon::getVelocity ()

Returns the current linear velocity of the shape.

Returns:

Vector2f Current linear velocity of the shape

bool Polygon::intersects (Polygon *shape*)

Check the intersection between two polygon shapes. Has three levels of detection, to reduce unecessary calculations and resource usage. Returns no information about collision after-effects or resultants. For that, see **intersects(Polygon shape, Vector2f& resultant)**.

Parameters:

shape	The shape we are checking to be colliding with the current one

Returns:

true The two shapes are colliding
false The two shapes aren't colliding

bool Polygon::intersects (RectangleShape *shape*)

A wrapper method to check the intersection between a **Polygon** shape and a RectangleShape by converting it to a polygon and then using our full intersection method See **intersects(Polygon shape)** for the full intersection method.

Parameters:

shape	The shape we are checking to be colliding with the current one

Returns:

true The two shapes are colliding
false The two shapes aren't colliding

bool Polygon::intersects (CircleShape *shape*)

A wrapper method to check the intersection between a **Polygon** shape and a CircleShape by converting it to a polygon and then using our full intersection method See **intersects(Polygon shape)** for the full intersection method.

Parameters:

shape	The shape we are checking to be colliding with the current one

Returns:

true The two shapes are colliding
false The two shapes aren't colliding

bool Polygon::intersects (ConvexShape *shape*)

A wrapper method to check the intersection between a **Polygon** shape and a ConvexShape by converting it to a polygon and then using our full intersection method See **intersects(Polygon shape)** for the full intersection method.

Parameters:

shape	The shape we are checking to be colliding with the current one

Returns:

true The two shapes are colliding
false The two shapes aren't colliding

bool Polygon::intersects (Polygon *shape*, Vector2f & *resultant*)

Currently WIP!!

Parameters:

shape	The shape we are checking to be colliding with the current one
resultant	The resultant of the collision (i.e. how the object moves afterwards)

Returns:

true The two shapes are colliding
false The two shapes aren't colliding

bool Polygon::isMovableByCollision ()

Get whether the shape can be moved by being collided with by another object.

Returns:

true The shape can be moved
false The shape cannot be moved

bool Polygon::isSolid ()

Check whether or not the shape can collide with other shapes.

Returns:

true Can collide
false Cannot collide

void Polygon::move (const Vector2f & *d*)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

d	The amount to change x and y by

void Polygon::move (float *dx*, float *dy*)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

dx	Amount to change the x coordinate by
dy	Amount to change the y coordinate by

void Polygon::rotate (float *angle*)

An overriden method from sf::Shape that changes the rotation like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

angle	The angle we are rotating the shape by

void Polygon::scale (const Vector2f & *scale*)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

scale	The scaling factors for our polygon

void Polygon::scale (float *xFactor*, float *yFactor*)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

xFactor	The x scaling factor
yFactor	The y scaling factor

void Polygon::setAngularVelocity (float *newAngularVelocity*)

Changes the angular velocity of the polygon to the paramter provided.

Parameters:

newAngularVelocity	The new angular velocity of the polygon

void Polygon::setDensity (float *newDensity*)

Set the density of the object, used in calculate its mass and moment of inertia (default is 1) and recalculate both values.

Parameters:

newDensity	The density of the object (default is 1)

void Polygon::setMovableByCollision (bool *value*)

Set whether the shape can be moved by being collided with by another object.

Parameters:

value	Whether or not the shape can be moved by another polygon

void Polygon::setPosition (const Vector2f & *position*)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

position	The new x and y coordinates of the shape

void Polygon::setPosition (float *x*, float *y*)

An overriden method from sf::Shape that changes the position like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

x	New x coordinate
y	New y coordinate

void Polygon::setRigidity (float *value*)

Set how much energy is conserved when this object collides with another. 0 for no energy conserved (completely inelastic collision) and 1 for completely elastic (all energy conserved)

Parameters:

value	The new rigidity, 0 for complete inelastic, 1 for complete elastic

void Polygon::setRotation (float *angle*)

An overriden method from sf::Shape that changes the rotation like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

angle	The angle we are setting the rotation to (default is 0)

void Polygon::setScale (const Vector2f & *scale*)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

scale	The scaling factors for our polygon

void Polygon::setScale (float *xFactor*, float *yFactor*)

An overriden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.

Parameters:

xFactor	The x scaling factor
yFactor	The y scaling factor

void Polygon::setSolid (bool *state*)

Set whether the shape is solid (can collide with other shapes)

Parameters:

state	Whether or not the shape is solid

void Polygon::setVelocity (Vector2f *newVelocity*)

Changes the linear velocity of the polygon to the paramter provided.

Parameters:

newVelocity	The new velocity of the polygon

void Polygon::update (float *elapsedTime*)

Updates the shape and applies both linear and angular velocity to update the position and rotation of the polygon.

Parameters:

elapsedTime	The amount of time that has elapsed since the last update

---

The documentation for this class was generated from the following files:

include/Polygon.hpp
src/Intersects.cpp
src/Polygon.cpp

VectorMath Class Reference

This class contains several different mathematical operations and calculations used to apply transformations, find dot/cross products, and several other use cases. All methods defined in this class are static (and any new additions should follow suit), though there is no other connection between the functions. Also, not all functions use vectors, as the name might suggest, but rather that they have to deal with directionality.

`#include <VectorMath.hpp>`

Static Public Member Functions

static void **dot** (Vector2f v1, Vector2f v2, float &value)

Compute the dot product between two vectors (sf::Vector2f)

static void **dot** (Vector3f v1, Vector3f v2, float &value)

Compute the dot product between two vectors (sf::Vector3f)

static Vector2f **cross** (Vector2f, Vector2f)
static Vector3f **cross** (Vector3f, Vector3f)
static void **angleBetween** (Vector2f v1, Vector2f v2, float &angle)

Find the angle between two vectors (sf::Vector2f) in 2D.

static void **rotate** (Vector2f &p, Vector2f origin, float angle)

Apply a rotation transformation to a point about some origin.

static float **mag** (Vector2f v)

Find the magnitude of a vector (sf::Vector2f)

static void **normalize** (Vector2f &v, float magnitude=1)

Adjust a vector (sf::Vector2f) such that its total magnitude is equal to the parameter provided.

static int **quadrant** (Vector2f point, Vector2f origin=Vector2f(0, 0))

Find what quadrant a point is in relative to a origin.

Detailed Description

This class contains several different mathematical operations and calculations used to apply transformations, find dot/cross products, and several other use cases. All methods defined in this class are static (and any new additions should follow suit), though there is no other connection between the functions. Also, not all functions use vectors, as the name might suggest, but rather that they have to deal with directionality.

Dependencies: <SFML/Graphics.hpp> (Vector2f and Vector3f) <tgmath.h> (cos and sin)

Namespaces: sf (SFML)

Member Function Documentation

void VectorMath::angleBetween (Vector2f *v1*, Vector2f *v2*, float & *angle*) [static]

Find the angle between two vectors (sf::Vector2f) in 2D.

Parameters:

v1	The first vector
v2	The second vector
angle	The variable in which the angle will be stored in

void VectorMath::dot (Vector2f *v1*, Vector2f *v2*, float & *value*) [static]

Compute the dot product between two vectors (sf::Vector2f)

Parameters:

v1	The first vector
v2	The second vector
value	The variable in which the value will be stored in

void VectorMath::dot (Vector3f *v1*, Vector3f *v2*, float & *value*) [static]

Compute the dot product between two vectors (sf::Vector3f)

Parameters:

v1	The first vector
v2	The second vector
value	The variable in which the value will be stored in

float VectorMath::mag (Vector2f *v*) [static]

Find the magnitude of a vector (sf::Vector2f)

Parameters:

v	The vector

Returns:

float The magnitude

void VectorMath::normalize (Vector2f & *v*, float *magnitude* = 1) [static]

Adjust a vector (sf::Vector2f) such that its total magnitude is equal to the parameter provided.

Parameters:

v	The vector. The normalized value will overwrite the old value
magnitude	The magnitude we want to the vector to have

int VectorMath::quadrant (Vector2f *point*, Vector2f *origin* = `Vector2f(0, 0)`) `[static]`

Find what quadrant a point is in relative to a origin.

Parameters:

point	The point who quadrant we want to find
origin	The origin

Returns:

int A quadrant number (1-4)

void VectorMath::rotate (Vector2f & *p*, Vector2f *origin*, float *angleInDegrees*)`[static]`

Apply a rotation transformation to a point about some origin.

Parameters:

p	The point we want to rotate. Rotated value will overwrite previous value
origin	The point we are rotating p about
angleInDegrees	The amount we are rotating, in degrees

The documentation for this class was generated from the following files:

include/VectorMath.hpp
src/VectorMath.cpp

Index

INDE