

# **SFML Collision**

Jack Featherstone  
Version 1.0  
07/15/2019

# 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 vertices 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 providing a vector of pre assembled vertices, 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 vertices from an image file with a varying level of detail to optimize the number of vertices (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 vertices 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 vertices.

The next step of identify vertices 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 vertices 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 vertices 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 vertices 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 vertices (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 vertices.

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 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 vertices, 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 vertices found inside the other shape. This is a highly theoretical process and I 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:

- Line.....7
- Shape
  - Polygon.....11
- VectorMath.....26

# Class Index

## Class List

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

**Line** (The line class is integral to detecting collisions between polygon classes, and thus is used internally in the intersection method, and Polygon object. 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 ) .....7

**Polygon** (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 (sf::CircleShape, sf::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 sf::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 parameters 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 ) ..... 11

**VectorMath** (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 ) .....26

# Class Documentation

## Line Class Reference

The line class is integral to detecting collisions between polygon classes, and thus is used internally in the intersection method, and **Polygon** object. 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** (sf::Vector2f p1, sf::Vector2f p2)

*Construct a new **Line** object which spans between two points (that is, it is a line segment). The order of the points provided is irrelevant for the most part (at least externally).*

**Line** ()

*Construct a new **Line** object with no information, used for comparing line objects.*

float **y** (float x)

*Returns the y value along the line at any given x value.*

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

*Detects whether this line is intersecting with another (within the bounds of both)*

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

*Detects whether this line is intersecting with another, with a few extra options.*

float **getAngle** ()

*Get the Angle object.*

float **getIntercept** ()

*Get the Intercept object.*

float **getSlope** ()

*Get the Slope object.*

sf::Vector2f **getStart** ()

*Get the Start object.*

sf::Vector2f **getEnd** ()

*Get the End object.*

sf::Vector2f **getPerpendicular** ()

*Get the unit normal vector for the line.*

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

*Get a drawable instance (RectangleShape) of the line, such that it can be drawn on a window.*

---

### Detailed Description

The line class is integral to detecting collisions between polygon classes, and thus is used internally in the intersection method, and **Polygon** object. 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> <cmath.h>

---

### Constructor & Destructor Documentation

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

Construct a new **Line** object which spans between two points (that is, it is a line segment). The order of the points provided is irrelevant for the most part (at least externally).

#### Parameters:

<i>p1</i>	The start point
<i>p2</i>	The end point

---

### Member Function Documentation

**float Line::getAngle ()**

Get the Angle object.

#### Returns:

float The angle of the line with respect to the x-axis, in degrees

**sf::RectangleShape \* Line::getDrawable (sf::Color color = sf::Color::Cyan)**

Get a drawable instance (RectangleShape) of the line, such that it can be drawn on a window.

#### Parameters:

<i>color</i>	The color in which the line will be drawn in, default is sf::Color::Cyan
--------------	--

#### Returns:

sf::RectangleShape\* A pointer to the object to be drawn



### **sf::Vector2f Line::getEnd ()**

Get the End object.

#### **Returns:**

sf::Vector2f The second point passed to the constructor

### **float Line::getIntercept ()**

Get the Intercept object.

#### **Returns:**

float The intercept of the line

### **sf::Vector2f Line::getPerpendicular ()**

Get the unit normal vector for the line.

#### **Returns:**

sf::Vector2f A vector that is perpendicular to the line, with a magnitude of 1

### **float Line::getSlope ()**

Get the Slope object.

#### **Returns:**

float The slope of the line (dy/dx)

### **sf::Vector2f Line::getStart ()**

Get the Start object.

#### **Returns:**

sf::Vector2f The first point passed to the constructor

### **bool Line::intersects (Line *line*)**

Detects whether this line is intersecting with another (within the bounds of both)

#### **Parameters:**

<i>line</i>	The line that may be intersecting with this one
-------------	---

#### **Returns:**

true The two lines are intersecting

false The two lines are not intersecting

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

Detects whether this line is intersecting with another, with a few extra options.

**Parameters:**

<i>line</i>	The line that may be intersecting with this one
<i>intersectionPoint</i>	A vector that will be assigned to the point in which the two lines intersect. If they do not intersect, it will remain unchanged.
<i>extendLine</i>	If this parameter is set to true, it will extend this line to a full line instead of just a segment, which can be useful in determining whether the line is inside a set of lines (see Polygon::contains)

**Returns:**

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

**float Line::y (float *x*)**

Returns the y value along the line at any given x value.

**Parameters:**

<i>x</i>	The x value in which the y value is to be known
----------	---

**Returns:**

float The y value at the given x value

---

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

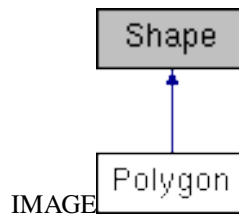
include/Line.hpp  
src/Line.cpp

## 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` (`sf::CircleShape`, `sf::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 `sf::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 parameters 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:



### Public Member Functions

**Polygon** (`sf::Texture *texture`, `Detail detail=Detail::Optimal`, `std::vector< sf::Color > ignoredColors={ }`)  
*Construct a new **Polygon** object from a given texture (image).*

**Polygon** (`std::vector< sf::Vector2f > points`)  
*Construct a new **Polygon** object from a `std::vector` of points.*

**Polygon** (`sf::CircleShape shape`)  
*Construct a new **Polygon** object from a `sf::CircleShape` object.*

**Polygon** (`sf::RectangleShape shape`)  
*Construct a new **Polygon** object from a `sf::RectangleShape` object.*

**Polygon** (`sf::ConvexShape shape`)  
*Construct a new **Polygon** object from a `sf::ConvexShape` object.*

`virtual size_t` **getPointCount** () `const`  
*Get the number of vertices on our polygon. Overridden from `sf::Shape`.*

`virtual sf::Vector2f` **getPoint** (`size_t index`) `const`  
*Get the vertex at index in the `std::vector m_points`. Overridden from `sf::Shape`.*

`std::vector< sf::Vector2f >` **getPoints** ()  
*Returns the entire `std::vector` of points that represent the shape, without any modifications from transformations (rotate, move, scale)*

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

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

sf::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 **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.*

sf::Vector2f **getCenterOfMass** ()

*Get the Center Of Mass object.*

float **getYoungsModulus** ()

*This will return the parameter that describes how the polygon responds to forces and how it bends and distorts.*

void **setYoungsModulus** (float youngsModulus)

*Change the parameter that describes how the polygon responds to forces and how it bends and distorts.*

float **getGamma** ()

*Get the Gamma object.*

void **setGamma** (float gamma)  
*Set the Gamma physical property.*

float **getArea** ()  
*Get the Area object.*

sf::Vector2f **getForce** ()  
*Get the Force object WIP.*

void **setForce** (sf::Vector2f force)  
void **addForce** (sf::Vector2f force)  
void **setVelocity** (sf::Vector2f newVelocity)  
*Changes the linear velocity of the polygon to the paramter provided.*

sf::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 **adjustFromForce** (sf::Vector2f resultant, sf::Vector2f collisionPoint, **Polygon** &shape)  
*Currently WIP!!*

void **setScale** (const sf::Vector2f &scale)  
*An overridden 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 overridden method from sf::Shape that changes the scale like its super-counterpart and also recreates the lines that represent the shape.*

void **scale** (const sf::Vector2f &scale)  
*An overridden 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 overridden 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 overridden 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 overridden method from sf::Shape that changes the rotation like its super-counterpart and also recreates the lines that represent the shape.*

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

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

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

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

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

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

std::vector< sf::Vector2u > **intersects** (**Polygon** shape)

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

bool **intersects** (sf::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** (sf::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** (sf::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, sf::Vector2f &resultant)

*Currently WIP!!*

```
bool intersects (sf::RectangleShape shape, sf::Vector2f &resultant)
bool intersects (sf::CircleShape shape, sf::Vector2f &resultant)
bool intersects (sf::ConvexShape shape, sf::Vector2f &resultant)
bool contains (Polygon shape)
    Not implemented!
```

```
bool contains (sf::RectangleShape shape)
    Not implemented!
```

```
bool contains (sf::CircleShape shape)
    Not implemented!
```

```
bool contains (sf::ConvexShape shape)
    Not implemented!
```

### Static Public Member Functions

```
static void calculateArea (std::vector< sf::Vector2f > points, float &value)
    This is a static method that finds the area of any given shape (std::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 (sf::CircleShape, sf::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 sf::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 parameters 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>

---

### Constructor & Destructor Documentation

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

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

#### Parameters:

<i>texture</i>	The texture for the shape/sprite we want to model
----------------	---

<i>detail</i>	The level of detail to keep in the shape, from least to most: Less, More, Optimal, Exact
<i>ignoredsf::Colors</i>	By default, all pixels that arent (0, 0, 0, 0) will be included, any colors specified here will also be ignored

### **Polygon::Polygon (std::vector< sf::Vector2f > *points*)**

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

#### **Parameters:**

<i>points</i>	The points that constitute our shape
---------------	--------------------------------------

### **Polygon::Polygon (sf::CircleShape *shape*)**

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

#### **Parameters:**

<i>shape</i>	The CircleShape object whose points will be used
--------------	--

### **Polygon::Polygon (sf::RectangleShape *shape*)**

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

#### **Parameters:**

<i>shape</i>	The RectangleShape object whose points will be used
--------------	---

### **Polygon::Polygon (sf::ConvexShape *shape*)**

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

#### **Parameters:**

<i>shape</i>	The ConvexShape object who points will be used
--------------	--

---

## **Member Function Documentation**

**void Polygon::adjustFromForce (sf::Vector2f *resultant*, sf::Vector2f *collisionPoint*, Polygon & *shape*)**



Currently WIP!!

**Parameters:**

<i>resultant</i>	The unit vector in the direction of the new motion
<i>shape</i>	The other colliding shape

**void Polygon::calculateArea (std::vector< sf::Vector2f > *points*, float & *value*)**  
**[static]**

This is a static method that finds the area of any given shape (std::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:**

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

**bool Polygon::contains (Polygon *shape*)**

Not implemented!

**Parameters:**

<i>shape</i>	
--------------	--

**Returns:**

true  
false

**bool Polygon::contains (sf::RectangleShape *shape*)**

Not implemented!

**Parameters:**

<i>shape</i>	
--------------	--

**Returns:**

true  
false

**bool Polygon::contains (sf::CircleShape *shape*)**

Not implemented!

**Parameters:**

<i>shape</i>	
--------------	--

**Returns:**

true  
false

**bool Polygon::contains (sf::ConvexShape *shape*)**

Not implemented!

**Parameters:**

<i>shape</i>	
--------------	--

**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

**float Polygon::getArea ()**

Get the Area object.

**Returns:**

float The area of the polygon

**sf::Vector2f Polygon::getCenterOfMass ()**

Get the Center Of Mass object.

**Returns:**

sf::Vector2f The local coordinates of the center of mass of the object

**sf::Vector2f Polygon::getCentroid ()**

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

**Returns:**

sf::Vector2f The centroid of the shape

**float Polygon::getDensity ()**

Get the relative density of the polygon.

**Returns:**

float The density of the polygon

**sf::Vector2f Polygon::getForce ()**

Get the Force object WIP.

**Returns:**

sf::Vector2f The current force on the object

**float Polygon::getGamma ()**

Get the Gamma object.

**Returns:**

float Return the gamma physical property of the object

**std::vector< Line > Polygon::getLines ()**

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

**Returns:**

std::vector<Line> A std::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

**sf::Vector2f Polygon::getPoint (size\_t index) const [virtual]**

Get the vertex at index in the std::vector m\_points. Overridden from sf::Shape.

**Parameters:**

<i>index</i>	The index of the point we are looking for
--------------	---

**Returns:**

sf::Vector2f The point at index in m\_points

**size\_t Polygon::getPointCount () const [virtual]**

Get the number of vertices on our polygon. Overridden from sf::Shape.

**Returns:**

size\_t The number of vertices

**std::vector< sf::Vector2f > Polygon::getPoints ()**

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

**Returns:**

std::vector<sf::Vector2f> Our shape's std::vector of vertices

**sf::Vector2f Polygon::getVelocity ()**

Returns the current linear velocity of the shape.

**Returns:**

sf::Vector2f Current linear velocity of the shape

**float Polygon::getYoungsModulus ()**

This will return the parameter that describes how the polygon responds to forces and how it bends and distorts.

**Returns:**

float The Young's Modulus of the shape, default is 1

**std::vector< sf::Vector2u > Polygon::intersects (Polygon shape)**

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

**Parameters:**

<i>shape</i>	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 (sf::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:**

<i>shape</i>	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 (sf::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:**

<i>shape</i>	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 (sf::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:**

<i>shape</i>	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*, sf::Vector2f & *resultant*)**

Currently WIP!!

#### **Parameters:**

<i>shape</i>	The shape we are checking to be colliding with the current one
<i>resultant</i>	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 sf::Vector2f & d)**

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

**Parameters:**

<i>d</i>	The amount to change x and y by
----------	---------------------------------

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

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

**Parameters:**

<i>dx</i>	Amount to change the x coordinate by
<i>dy</i>	Amount to change the y coordinate by

**void Polygon::rotate (float angle)**

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

**Parameters:**

<i>angle</i>	The angle we are rotating the shape by
--------------	--

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

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

#### **Parameters:**

<i>scale</i>	The scaling factors for our polygon
--------------	-------------------------------------

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

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

#### **Parameters:**

<i>xFactor</i>	The x scaling factor
<i>yFactor</i>	The y scaling factor

### **void Polygon::setAngularVelocity (float *newAngularVelocity*)**

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

#### **Parameters:**

<i>newAngularVelocity</i>	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:**

<i>newDensity</i>	The density of the object (default is 1)
-------------------	--

### **void Polygon::setGamma (float *gamma*)**

Set the Gamma physical property.

#### **Parameters:**

<i>gamma</i>	The new gamma physical property value
--------------	---------------------------------------

### **void Polygon::setMovableByCollision (bool *value*)**

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

**Parameters:**

<i>value</i>	Whether or not the shape can be moved by another polygon
--------------	--

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

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

**Parameters:**

<i>position</i>	The new x and y coordinates of the shape
-----------------	--

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

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

**Parameters:**

<i>x</i>	New x coordinate
<i>y</i>	New y coordinate

**void Polygon::setRotation (float *angle*)**

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

**Parameters:**

<i>angle</i>	The angle we are setting the rotation to (default is 0)
--------------	---

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

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

**Parameters:**

<i>scale</i>	The scaling factors for our polygon
--------------	-------------------------------------

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

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

**Parameters:**

<i>xFactor</i>	The x scaling factor
<i>yFactor</i>	The y scaling factor



### **void Polygon::setSolid (bool *state*)**

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

#### **Parameters:**

<i>state</i>	Whether or not the shape is solid
--------------	-----------------------------------

### **void Polygon::setVelocity (sf::Vector2f *newVelocity*)**

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

#### **Parameters:**

<i>newVelocity</i>	The new velocity of the polygon
--------------------	---------------------------------

### **void Polygon::setYoungsModulus (float *youngsModulus*)**

Change the parameter that describes how the polygon responds to forces and how it bends and distorts.

#### **Parameters:**

<i>youngsModulus</i>	The new value for the young's modulus, default is 1
----------------------	---

### **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:**

<i>elapsedTime</i>	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 float **dot** (sf::Vector2f v1, sf::Vector2f v2)

*Compute the dot product between two vectors (sf::Vector2f). The order of the vectors given doesn't matter.*

static float **cross** (sf::Vector2f v1, sf::Vector2f v2)

*Gives the magnitude of the crossed vector between two 2d vectors (sf::Vector2f). The order of the vectors given doesn't matter, as we are only looking at magnitude.*

static float **angleBetween** (sf::Vector2f v1, sf::Vector2f v2)

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

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

*Apply a rotation transformation to a point about some origin.*

static float **mag** (sf::Vector2f v)

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

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

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

---

### 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> (sf::Vector2f) <tgmath.h> (cos and sin)

---

### Member Function Documentation

**float VectorMath::angleBetween (sf::Vector2f v1, sf::Vector2f v2) [static]**

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

**Parameters:**

<i>v1</i>	The first vector
<i>v2</i>	The second vector
<i>angle</i>	The variable in which the angle will be stored in

**float VectorMath::cross (sf::Vector2f v1, sf::Vector2f v2)[static]**

Gives the magnitude of the crossed vector between two 2d vectors (sf::Vector2f). The order of the vectors given doesn't matter, as we are only looking at magnitude.

**Parameters:**

<i>v1</i>	The first vector
<i>v2</i>	The second vector

**Returns:**

float

**float VectorMath::dot (sf::Vector2f v1, sf::Vector2f v2)[static]**

Compute the dot product between two vectors (sf::Vector2f). The order of the vectors given doesn't matter.

**Parameters:**

<i>v1</i>	The first vector
<i>v2</i>	The second vector
<i>value</i>	The variable in which the value will be stored in

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

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

**Parameters:**

<i>v</i>	The vector
----------	------------

**Returns:**

float The magnitude of the given vector

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

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

**Parameters:**

<i>v</i>	The vector. The normalized value will overwrite the old value
<i>magnitude</i>	The magnitude we want the vector to have, default is 1

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

Apply a rotation transformation to a point about some origin.

**Parameters:**

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

---

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

include/VectorMath.hpp  
src/VectorMath.cpp