Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

June 22, 2023

From Flying Balls to Colliding Polygons

2D Physics Engine: Rigid Body Simulation

Arnaud Fauconnet

Abstract

Physics engines are a fun and interesting way to learn about a lot of different subjects. First the theoretical concepts, such as the equations that dictate the motion of the objects, together with their components, need to be thoroughly understood. Then there is the necessity of finding a way to represent all of those concepts in a given programming language and to make them as efficient as possible so that the simulation runs fluidly. The task to be completed here was to extend an already existing physics engine that only made circles bounce off each other. The extension was focused on having the ability to generate some arbitrary polygons and make them bounce off each other in a physically accurate way. The main issues that rose up during the development of the extension: determining the inertia of a arbitrary polygon, which is important for realistic impacts; having an accurate collision detection system, which allows the engine to know when to make two polygons bounce off each other. Once those aspects were worked on and polished, the rest of the implementation went smoothly.

Advisor
Prof. Antonio Carzaniga

Advisor's approval (Prof. Antonio Carzaniga): Date:

Contents

1 Introduction **2**

1.1 Goal of the project 2

1.2 State of the art 2

2 Technical Background **3**

2.1 Original project 3

2.2 Cairo 3

3 Theoretical Background **4**

3.1 Moment of inertia 4

3.1.1 Rectangle 4

3.1.2 Regular Polygons 5

3.1.3 Arbitrary Polygons 6

3.2 Collision detection 8

3.2.1 Separating Axis Theorem 8

3.2.2 Vertex collisions 9

3.3 Collision resolution 11

4 Implementation **14**

4.1 Structure 14

4.1.1 Polygon generator 14

4.1.2 Collision 15

4.2 Optimization 15

4.3 Known issues 16

5 Conclusion **17**

A Calculations **19**

A.1 Moment of inertia of rectangle 19

A.2 Moment of inertia of sub-triangle of regular polygon 20

A.3 Moment of inertia of sub-triangle of arbitrary polygon 21

A.4 Solving for impulse parameter 22

1 Introduction

1.1 Goal of the project

The goal of the project was to extend an existing physics engine called "flying-balls"[1] by Prof. Antonio Carzaniga. This physics engine simulated the interactions between circles in a two-dimensional space. These circles appear in the window with a random position, together with a random initial velocity vector. The simulation would then just calculate the position of each circle in the following frame and draw it in its new state. If two circles were to collide with each other, the engine would detect it and make those circles bounce off each other. The resulting position and speed would be decided by the physics equations that govern the motion of such objects.

The extension this project was asked to bring is the possibility to have more complex shapes interact with each other, such as polygons. The polygons would have to be arbitrary and bounce off other polygons present in the scene.

1.2 State of the art

There are a lot of 2D physiques engines across the internet. The purpose of this project was not to bring something new to the already existing landscape, but rather learn how to complete every step of the process (polygons generation, collision detection, kinematics resolution) from scratch, simply having a pre-existing way to represent the shapes on the screen.

[1]The state of the project before the extension can be found at `https://github.com/carzaniga/flying-balls/tree/c++-port`

2 Technical Background

The technical background is all the research related to the programming part of this bachelor project. The programming language used in this project is a mixture of C and C++, for this part, the course of Systems Programming taught by Prof. Carzaniga during the third semester. Then came the study of the starting point of the project, which was divided in the logic itself and the framework used to display the state of the simulation on the screen.

2.1 Original project

Before starting to write any code, it was necessary to study carefully the original project. The starting point chosen for this specific project was the last commit on the `c++-port` branch. The reason for this choice is that the project originally started fully in C (which is still the case for the `main` branch) and C++ offers more functionalities that help for a smoother development process.

The life-cycle of the simulation was the typical three-step process:

1. **State initiation:** the state of the application is set with certain starting conditions;
2. **State update:** the state, at each frame, gets applied a set of rules that govern the behaviour of the application;
3. **Termination:** when the user stops the application, it actuates a number of `cleaning` up operations.

Just like any C/C++ project, the modules were split into different files, and those modules where themselves split into header files and implementation files. The header files expose the public interface which other modules can call to execute a determine function, whereas the implementation files, as the name suggests, offer the concrete implementation of the aforementioned functions. The implementation files can use some static[2] functions that it can use as auxiliary or utility functions. The header files usually expose the fields and methods of the class (or `struct`) the module is using, if any, together with one function for each of three steps of the life-cycle mentioned above.

2.2 Cairo

Cairo is a 2D graphics library with support for multiple output devices. Cairo is designed to produce consistent output on all output media while taking advantage of display hardware acceleration when available. The cairo API provides operations similar to the drawing operators of PostScript and PDF. Operations in cairo including stroking and filling cubic Bézier splines, transforming and compositing translucent images, and antialiased text rendering. All drawing operations can be transformed by any affine transformation (scale, rotation, shear, etc.). Reading the documentation[3], and more specifically the practical tutorial[4] was useful to understand how the library works.

The Cairo drawing model relies on a three-layer model, any drawing process takes place in three steps:

1. first a mask is created, which includes one or more vector primitives or forms, i.e., circles, squares, TrueType fonts, Bézier curves, etc;
2. then source must be defined, which may be a color, a color gradient, a bitmap or some vector graphics, and from the painted parts of this source a die cut is made with the help of the above defined mask;
3. finally the result is transferred to the destination or surface, which is provided by the back-end for the output.

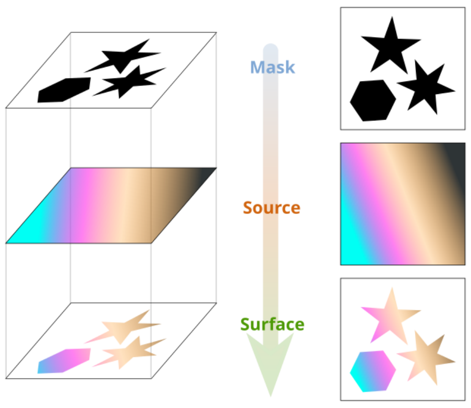


Figure 2.1. Cairo's drawing model[5]

[2]static in the sense of C, i.e. visible only to the file it is declared in
[3]`https://www.cairographics.org/documentation/`
[4]`https://www.cairographics.org/tutorial/`
[5]Image taken from Wikipedia

3 Theoretical Background

The theoretical background is everything related to the physics part of the project. It covers the calculating the inertia of different types of polygons; different algorithms to detect whether there is a collision between two polygons; the resolution of the collision, i.e. finding the final velocity vectors and angular speed of those polygons.

3.1 Moment of inertia

The inertia of an object refers to the tendency of an object to resist a change of its state of motion or rest, it describes how the object behaves when forces are applied to it. An object with a lot of inertia requires more force to change its motion, either to make it move if it's at rest or to stop it if it's already moving. On the other hand, an object with less inertia is easier to set in motion or bring to a halt.

The moment of inertia is similar but is used in a slightly different context, it specifically refers to the rotational inertia of an object. It measures an object's resistance to changes in its rotational motion and how its mass is distributed with respect to is axis of rotation.

In the case of this project the axis of rotation is the one along the z-axis (perpendicular to the plane of the simulation) and placed at the barycenter of the polygon.

The general formula for the moment of inertia is

$$I_Q = \int \vec{r}^2 \rho(\vec{r})\,\mathrm{d}\mathcal{A} \tag{3.1}$$

where ρ is the density of object Q in the point $\vec{r}$ across the small pieces of area $\mathcal{A}$ of the object.

In our case, since we are implementing a 2D engine we can use the $\mathbb{R}^2$ coordinate systems, thus the formula becomes

$$I_Q = \iint \rho(x,y)\vec{r}^2\,\mathrm{d}x\,\mathrm{d}y$$

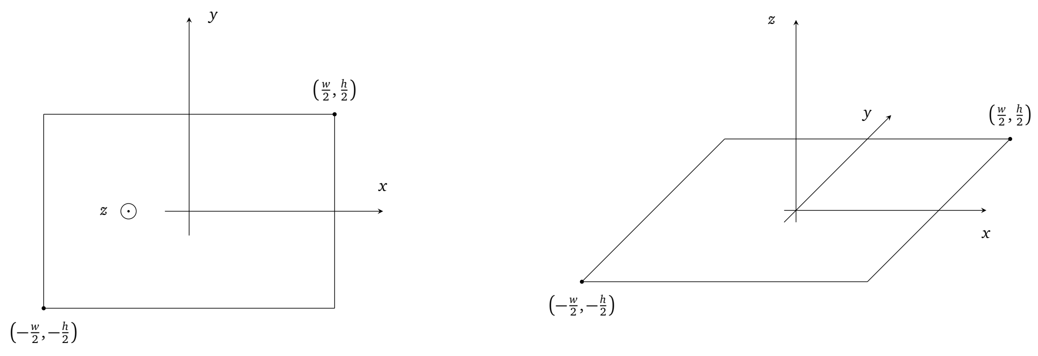
and since the requirements express that the mass of the polygons is spread uniformly across its surface, the formula finally becomes

$$I_Q = \rho \iint x^2 + y^2\,\mathrm{d}x\,\mathrm{d}y \tag{3.2}$$

The bounds of the integral depend on the shape of the polygon. In the following sections, we will describe how to compute those bounds, then we will show a different technique to compute the moment of inertia of arbitrary polygons.

3.1.1 Rectangle

The moment of inertia of a rectangle of width w and height h with respect to the axis of rotation that passes through its barycenter can be visualized in the Figure 3.1.


y
$(\frac{w}{2},\frac{h}{2})$
x
z ⊙
$(-\frac{w}{2},-\frac{h}{2})$
z
y
$(\frac{w}{2},\frac{h}{2})$
x
$(-\frac{w}{2},-\frac{h}{2})$

(a) 2d view of rectangle with axis of rotation

(b) 3d view of rectangle with axis of rotation

Figure 3.1. Representation of rectangle with respect to axis of rotation z

As figure Figure 3.1a implies, the bounds of equation 3.2 are trivial to derive:

$$I_{\text{rect}} = \rho \int_{-\frac{h}{2}}^{\frac{h}{2}} \int_{-\frac{w}{2}}^{\frac{w}{2}} x^2 + y^2 \, dx \, dy = \frac{\rho w h}{12} \left(w^2 + h^2 \right) \tag{3.3}$$

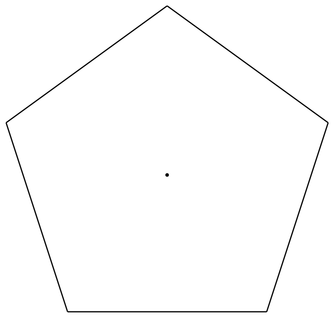
and since $\rho w h$ is the density of the rectangle multiplied by its area, we can replace this term by its mass m, thus

$$I_{\text{rect}} = \frac{1}{12} m \left(w^2 + h^2 \right) \tag{3.4}$$

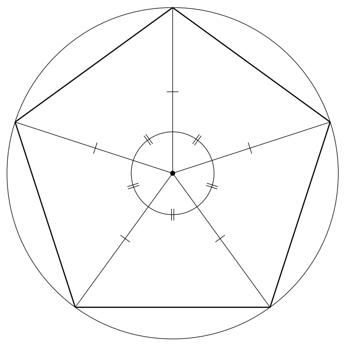
All the steps to compute equation 3.3 can be found in equation A.1 in Appendix A.

3.1.2 Regular Polygons

A regular polygon is a shape that has sides of equal length and angles between those sides of equal measure. A polygon of n sides can be subdivided in n congruent (and isosceles since they are all the radius of the circumscribing circle) triangles that all meet in the polygon's barycenter, as demonstrated in Figure 3.2b with a pentagon.



(a) Regular polygon of 5 sides with its barycenter



(b) Pentagon divided in 5 congruent triangles

Figure 3.2. Subdivision of regular polygons into congruent triangles

If we define one of the sub-triangle of the regular polygon as T, then we can find the moment of inertia I_T when it is rotating about the barycenter. To find the bounds of the integral in equation 3.2, we can take the triangle T and place it along the x-axis so that it is symmetric likes shown in figure. Assuming the side length of the polygon is l, the height of the triangle T is h and the angle of the triangle on the barycenter of the polygon to be θ, then

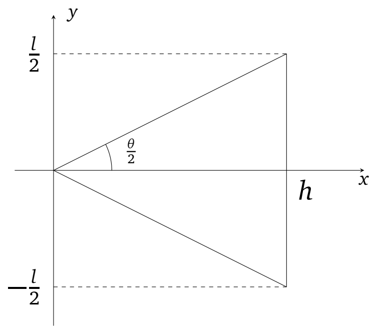


Figure 3.3. Sub-triangle T of regular polygon

we can see the bounds for the integral

$$I_T = \rho \int_0^h \int_{-\frac{lx}{2h}}^{\frac{lx}{2h}} x^2 + y^2 \, dy \, dx = \frac{m_T l^2}{24} \left(1 + 3 \cot^2 \left(\frac{\theta}{2} \right) \right) \tag{3.5}$$

All the steps to compute equation 3.5 can be found in equation A.2 in Appendix A.
Now that we have the moment of inertia of the sub-triangle, we can make the link to the overall polygon. Since

$$\theta = \frac{2\pi}{n} \implies \frac{\theta}{2} = \frac{\pi}{n}$$

and the moment of inertia are additive (as long they are as they are about the same axis) we can get the moment of inertia with

$$I_{\text{regular}} = nI_T$$

and since the mass of the regular polygon m is the sum of the masses of the sub-triangle

$$m = nm_T$$

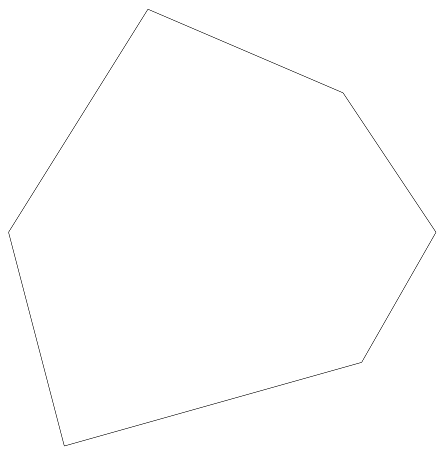
we have that

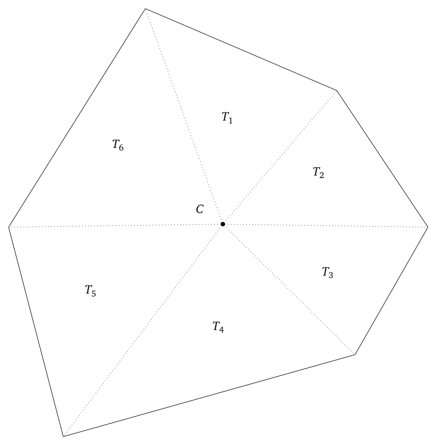
$$I_{\text{regular}} = \frac{ml^2}{24}\left(1 + 3\cot^2\left(\frac{\pi}{n}\right)\right) \tag{3.6}$$

3.1.3 Arbitrary Polygons

For arbitrary polygons, we are taking a slightly different approach. Using the Cartesian coordinate system to solve the equation 3.2 revealed to be more cumbersome than useful. But similarly to regular polygons (c.f. Section 3.1.2), we can use the additive property of the moment inertia to divide our arbitrary polygon into sub-triangles. As opposed to regular polygons, these triangles won't be congruent, so we can't just get the moment of inertia of one of them and multiply it by the number of sides, but we need to calculate them individually. So given a polygon of n sides, we can construct n sub-triangles T_i, for $i = 1, \ldots, n$. So the moment of inertia I of the polygon will be

$$I = \sum_i I_{T_i} \tag{3.7}$$


(a) An arbitrary 6-sided polygon



(b) Arbitrary polygon divided into 6 sub-triangles

To calculate the moment of inertia I_{T_i}, instead of using the classical x- and y-axis as we did before, we decided to use the edges of the triangle as axis and therefore express what we need to integrate in function of those as can be seen in Figure 3.5.

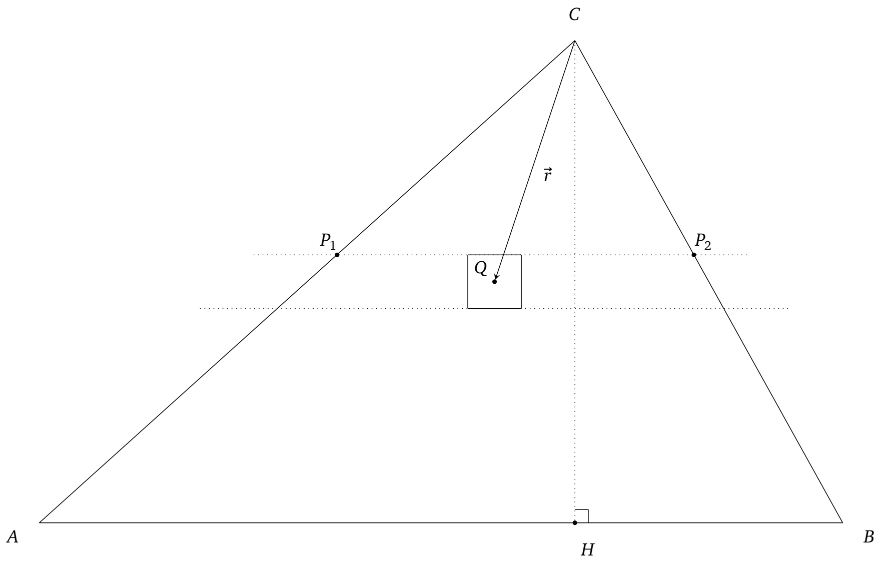


Figure 3.5. Sub-triangle of arbitrary polygon

In Figure 3.5, C represent the barycenter of the polygon (as is shown in Figure 3.4b). The axis we are going to integrate on are $\overrightarrow{CA}$ and $\overrightarrow{AB}$. We can now define

$$\overrightarrow{CP_1} = \alpha\overrightarrow{CA}, \qquad \overrightarrow{CP_2} = \alpha\overrightarrow{CB}, \qquad \forall\alpha \in [0,1] \tag{3.8}$$

and

$$\overrightarrow{P_1Q} = \beta\overrightarrow{P_1P_2}, \qquad \forall\beta \in [0,1]$$

From 3.8, it quickly follows that

$$\overrightarrow{P_1P_2} = \alpha\overrightarrow{AB}$$

therefore

$$\overrightarrow{P_1Q} = \beta\alpha\overrightarrow{AB} \tag{3.9}$$

Finally, if we put together equations 3.8 and 3.9, we have that

$$\vec{r} = \overrightarrow{CP_1} + \overrightarrow{P_1Q} = \alpha\overrightarrow{CA} + \beta\alpha\overrightarrow{AB} \tag{3.10}$$

Now we got the first part equation 3.1. To find the $\mathrm{d}\mathcal{A}$, we just need to get the area of the square that contains Q in Figure 3.5. Since $||\overrightarrow{AB}||$ represents the base of the triangle T_i, we can define

$$b = ||\overrightarrow{AB}||$$

we consequently have that

$$\mathrm{d}\mathcal{A} = b\alpha\,\mathrm{d}\beta h\,\mathrm{d}\alpha \tag{3.11}$$

where $h = ||\overrightarrow{CH}||$ is the height of triangle. We can now assemble 3.10 and 3.11

$$I_{T_i} = \rho\int_0^1\int_0^1 \vec{r}^2 hb\alpha\,\mathrm{d}\alpha\,\mathrm{d}\beta = \frac{\rho hb}{4}\left(\frac{1}{3}\overrightarrow{AB}^2 + \overrightarrow{AB}\cdot\overrightarrow{CA} + \overrightarrow{CA}^2\right) \tag{3.12}$$

Since $\frac{\rho hb}{2}$ is the mass of the triangle we can write the result as

$$I_{T_i} = \frac{m_{T_i}}{2}\left(\frac{1}{3}\overrightarrow{AB}^2 + \overrightarrow{AB}\cdot\overrightarrow{CA} + \overrightarrow{CA}^2\right) \tag{3.13}$$

All the steps to compute equation 3.12 can be found in equation A.3 in Appendix A.

Now that we have the moment of inertia of the sub-triangle, we can make the link to the overall polygon.

$$I_{\text{arbitrary}} = \sum_i I_{T_i} = \sum_{i=1}^{n}\frac{m_{T_i}}{2}\left(\frac{1}{3}\overrightarrow{P_iP_{i+1}}^2 + \overrightarrow{CP_i}\cdot\overrightarrow{P_iP_{i+1}} + \overrightarrow{CP_i}^2\right) \tag{3.14}$$

where, $P_{n+1} = P_1$ in the case of $i = n$.

3.2 Collision detection

Collision detection, as the name suggests, are the algorithms used to detect whether two polygons are colliding. The result of this procedure must be an impact point and a normal vector, that will then be used for the collision resolution 3.3.

3.2.1 Separating Axis Theorem

This algorithm was the first one studied for this project and was inspired by the works of David Eberly [1]. The separating axis theorem (SAT) states that if you can draw a line between two convex objects, they do not overlap. We will call this line a *separating line*. More technically, two convex shapes do not overlap if there exists an axis onto which the two objects' projections do not overlap. We'll call this axis a *separating axis*. This concept can be visualized in Figure 3.6.

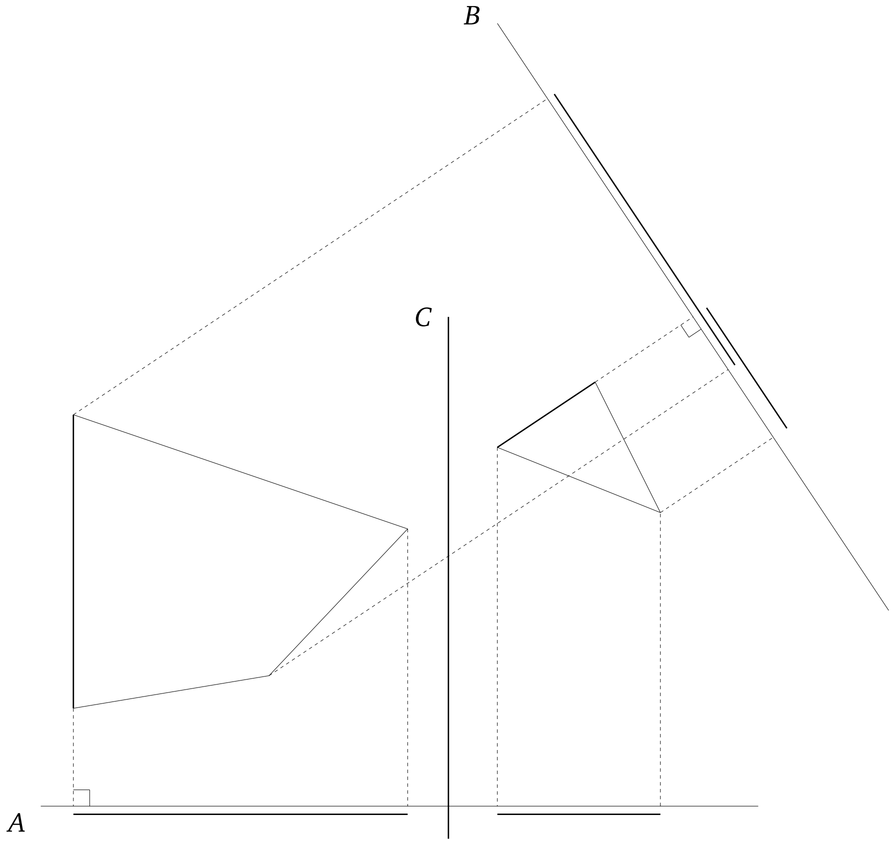


Figure 3.6. SAT: Separating axis (*A*) vs non-separating axis (*B*), with separating line (*C*)

As we can see in Figure 3.6, the axis *B* show that the projections of the both polygons overlap, but we were able to find an axis *A* where this is not the case. As soon as we find an axis for which the projections do not overlap, it means that the polygons are not colliding. For 2D objects, we only need to consider the axes that are orthogonal to each edge. In Figure 3.6, only two of those axes are shown for better readability, but they would be 7, one for each edge.

To move (or push) one polygon away from the other, we also need to find a vector that, when added to the polygons position, will make the shapes not overlap. We want the minimum displacement possible, We'll call this vector the minimum push vector (MPV). For 2-dimensional polygons, this vector will lie in some of the orthogonal axes.

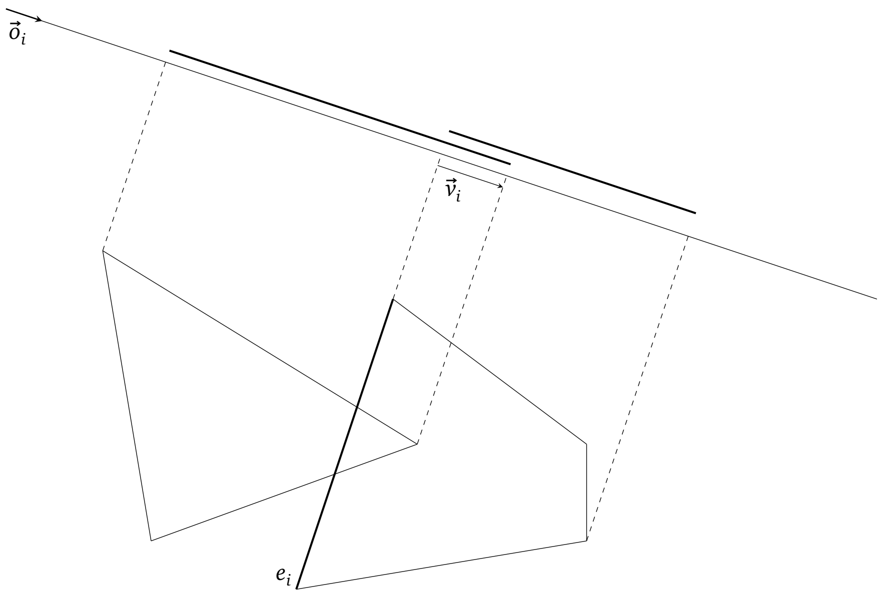


Figure 3.7. SAT: Minimum push vector $\vec{v}_i$ on axis defined by $\vec{o}_i$, orthogonal to edge e_i

The candidate MPVs $\vec{v}_i$ are the vectors that define the axis $\vec{o}_i$ (orthogonal to edge e_i), with $\|\vec{o}_i\| = 1$, multiplied by the minimum overlap between the two polygons, as shown in 3.7. The final MPV is simply the $\vec{v}_i$ with the smallest norm.

Pitfalls of SAT The issue with the SAT algorithm is that although it is good to find whether two polygons are colliding and the MPV, it isn't trivial to gather the point of impact, i.e. the vertex that is penetrating the other polygon. It is doable, but during the implementation, it came with some caveats that introduced some bugs, so we decided to switch strategy and go with an algorithm of our own. Moreover, SAT only supports convex polygons, which limits the original objective of the project, which was to have any arbitrary polygon.

3.2.2 Vertex collisions

The solution that was adopted for the project, after trying SAT, was a more intuitive one, developed by Prof. Carzaniga. The idea is simple: check if a vertex of a polygon is colliding with an edge of another polygon.

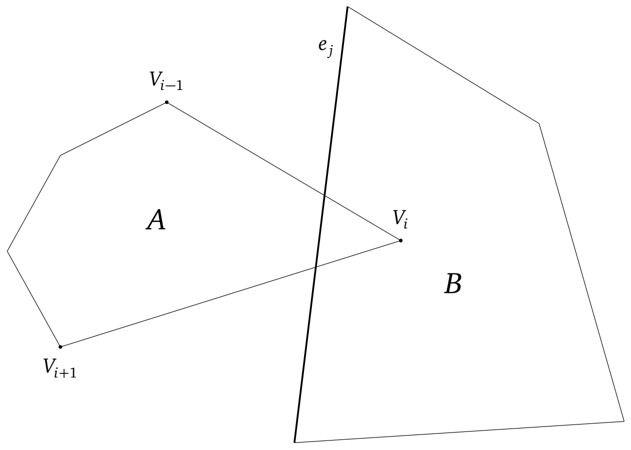


Figure 3.8. Vertex-edge collision between polygons A and B

If we have a polygon defined as a set of points $P \subset \mathbb{R}^2$, we define a vertex as a pair of segments $\left(\overline{V_{i-1}V_i}, \overline{V_{i+1}V_i}\right)$. To check if vertex V_i of polygon A is inside polygon B, we just check if the both segments $\overline{V_{i-1}V_i}$ and $\overline{V_{i+1}V_i}$ intersect edge e_j. If such is the case, we know that V_i is inside, and we can use it as impact point. We can now take the vector perpendicular to the edge e_j and normalize it, which is the normal we need for the collision resolution.

Edge cases With this approach (and many other collision detection algorithms), it is easy to see that the case described in the Figure 3.8 is only a general one. It will ultimately happen most often, but there are especially two edge cases that are note-worthy and occurred during the simulation a greater number of times than expected.

Parallel collision Parallel collision occur when two edge collide with each other, an example can be seen in Figure 3.9. This collision does not get detected by the vertex-edge detection method because the edge of A parallel to the edge of B do not collide, since they are parallel.

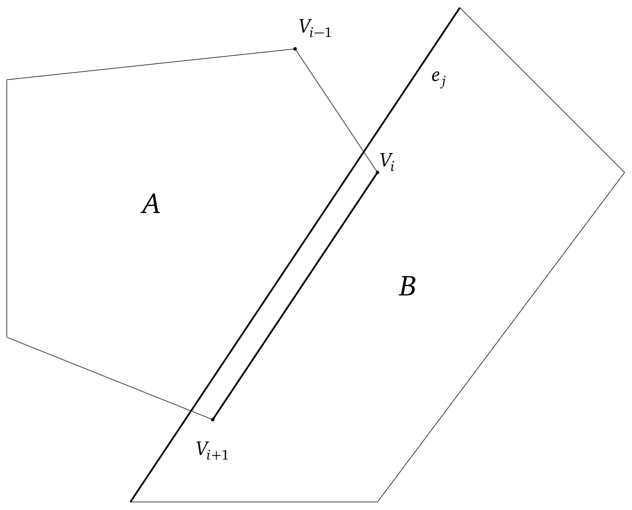
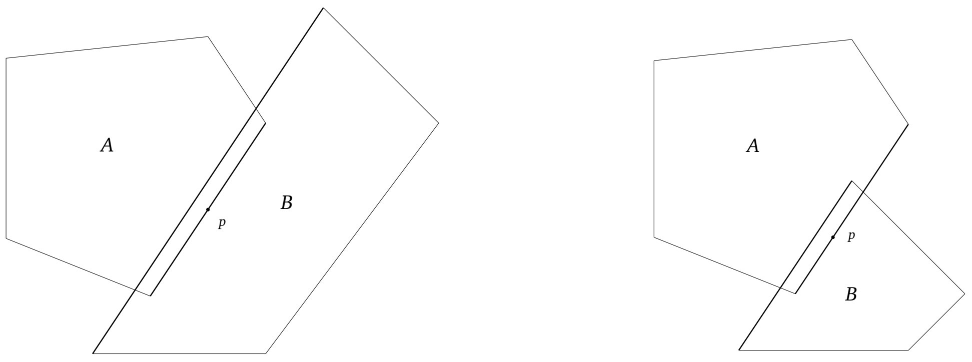


Figure 3.9. Parallel edges collision between polygons A and B

To determine if polygon A is having a parallel collision with polygon B we check if only the segment $\overline{V_{i-1}V_i}$ intersects with edge e_j and if the segment $\overline{V_{i+1}V_i}$ is parallel to edge e_j.

To find the impact point, we first need to find the minimum overlap between the parallel edges. We can calculate it by projecting the points that make the parallel edges of A and B on the axis generated by the edge of A (c.f. Figure 3.10).Finally, the impact point, is the midpoint of the said overlap.



(a) Edge fully contained by other edge

(b) Edge partly contained by other edge

Figure 3.10. Parallel collision, finding the impact point

The normal vector is given in the same way as the vertex-edge collision, i.e. taking the perpendicular vector to e_j and normalizing it.

Vertex on vertex collision These collision happen when, at the moment of the frame, two polygons are "inside each other", i.e. they both have have a vertex present inside the area of the other polygon, as shown in Figure 3.11.

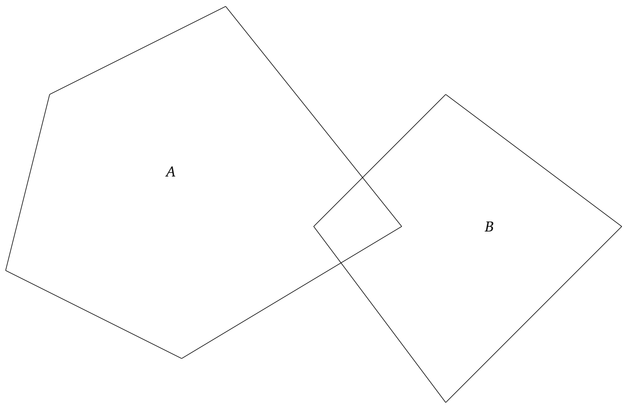


Figure 3.11. Vertex on vertex collision between polygons *A* and *B*

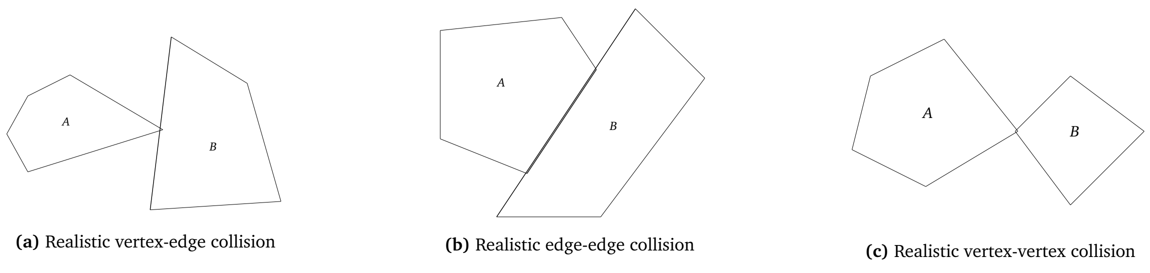
Two problems arise when trying to deal with this edge case:

1. determining which of the two vertices we chose as the impact point;

2. determining what normal vector.

For point 1, it actually doesn't really matter. What is represented in Figures 3.8-3.11 are an over exaggeration of what happens in the engine. Since the time delta between two frames is so small, the collisions look more like what is in Figure 3.12c, where the concerned vertices are located practically in the same spot, and the difference is negligible. So for convenience with the general vertex-edge case, we will take the vertex of *A* being inside *B* as the impact point.

For point 2, that's where the real problem of the edge case occur. We are unable to determine what vectors is supposed to be the normal vector. One might say "just take the vector that goes from the vertex of *A* (that is inside *B*) to the one of *B* (that is inside *A*)", but it doesn't work out in general. It works (kind of) when we have non-rotating objects going straight at each other, but as soon as you have rotational motion, this reasoning collapses. The solution that was decided (together with advisor Prof. Carzaniga) was to treat the collision as a vertex-edge collision, and choosing whatever the first edge of *B* comes up first in the calculations as the edge to find the normal. The results look realistic enough to be accepted.



(a) Realistic vertex-edge collision **(b)** Realistic edge-edge collision **(c)** Realistic vertex-vertex collision

Figure 3.12. Realistic collisions

3.3 Collision resolution

The collision resolution is the last step in the processing of the collision. Algorithmically, it is much less heavy than collision detection, since, once the simulation has two colliding polygons, a point of impact and a normal vector, it's just a case of applying the rigid body physics formulas to the polygons that are colliding. This part has been helped a lot by the works of Erik Neumann [2] and Chris Hecker [3].

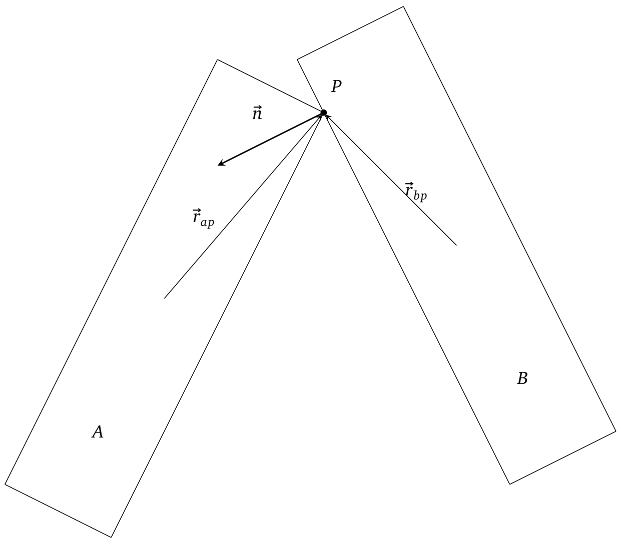


Figure 3.13. Collision resolution between polygons A and B

Variable definition Before getting into any maths, let's define some variables that we are going to use

- m_a, m_b = mass of the bodies A and B
- $\vec{r}_{ap}$ = distance vector from center of mass of body A to point P
- $\vec{r}_{bp}$ = distance vector from center of mass of body A to point P
- ω_{a1}, ω_{b1} = initial angular velocity of bodies A, B
- ω_{a2}, ω_{b2} = final angular velocity of bodies A, B
- $\vec{v}_{a1}, \vec{v}_{b1}$ = initial velocities of center of mass bodies A, B
- $\vec{v}_{a2}, \vec{v}_{b2}$ = final velocities of center of mass bodies A, B
- $\vec{v}_{ap1}$ = initial velocity of impact point P on body A
- $\vec{v}_{bp1}$ = initial velocity of impact point P on body B
- $\vec{v}_{p1}$ = initial relative velocity of impact points on body A, B
- $\vec{v}_{p2}$ = final relative velocity of impact points on body A, B
- $\vec{n}$ = normal vector
- e = elastic coefficient (0 = inelastic, 1 = perfectly elastic)

Some of those variables, like the ones in the left column, are already given by the simulation, some of them have been computed thanks to the algorithms in section 3.2 (normal vector and position of point P), and some have still to be defined mathematically, such as $\vec{v}_{ap1}, \vec{v}_{bp1}, \vec{v}_{p1}$ and $\vec{v}_{p2}$. So the velocity vectors of impact point P on both bodies, before the collision, are

$$\begin{aligned} \vec{v}_{ap1} &= \vec{v}_{a1} + \omega_{a1} \times \vec{r}_{ap} \\ \vec{v}_{bp1} &= \vec{v}_{b1} + \omega_{b1} \times \vec{r}_{bp} \end{aligned} \tag{3.15}$$

Similarly, the final velocities are

$$\begin{aligned} \vec{v}_{ap2} &= \vec{v}_{a2} + \omega_{a2} \times \vec{r}_{ap} \\ \vec{v}_{bp2} &= \vec{v}_{b2} + \omega_{b2} \times \vec{r}_{bp} \end{aligned} \tag{3.16}$$

Here we are regarding the angular velocity as a 3-dimensional vector perpendicular to the plane, so that the cross product is calculated as

$$\omega \times \vec{r} = \begin{pmatrix} 0 \\ 0 \\ \omega \end{pmatrix} \times \begin{pmatrix} r_x \\ r_y \\ 0 \end{pmatrix} = \begin{pmatrix} -\omega r_y \\ \omega r_x \\ 0 \end{pmatrix}$$

We these variables, we can finally define the relative velocities $\vec{v}_{p1}$ and $\vec{v}_{p2}$

$$\begin{aligned} \vec{v}_{p1} &= \vec{v}_{ap1} - \vec{v}_{bp2} \\ \vec{v}_{p2} &= \vec{v}_{ap2} - \vec{v}_{bp2} \end{aligned} \tag{3.17}$$

If we expand by using 3.15 and 3.16, we get

$$\begin{aligned}\vec{v}_{p1} &= \vec{v}_{a1} + \omega_{a1} \times \vec{r}_{ap} - \vec{v}_{b1} + \omega_{b1} \times \vec{r}_{bp} \\ \vec{v}_{p2} &= \vec{v}_{a2} + \omega_{a2} \times \vec{r}_{ap} - \vec{v}_{b2} + \omega_{b2} \times \vec{r}_{bp}\end{aligned} \tag{3.18}$$

The relative velocity of point P along the normal vector $\vec{n}$ is

$$\vec{v}_{p1} \cdot \vec{n}$$

Note that for a collision to occur this relative normal velocity must be negative (that is, the objects must be approaching each other). Let e be the elasticity of the collision, having a value between 0 (inelastic) and 1 (perfectly elastic). We now make an important assumption in the form of the following relation

$$\vec{v}_{p2} \cdot \vec{n} = -e\vec{v}_{p1} \cdot \vec{n} \tag{3.19}$$

This says that the velocity at which the objects fly apart is proportional to the velocity with which they were coming together. The proportionality factor is the elasticity e.

Collision Impulse In simple terms, collision impulse refers to the change in momentum experienced by an object during a collision. It is a measure of the force applied to an object over a short period of time. We imagine that during the collision there is a very large force acting for a very brief period of time. If you integrate (sum) that force over that brief time, you get the impulse.

In our simulation, we are assuming that no friction is happening during the collision, so that the only force we have to consider is the one of along the normal vector $\vec{n}$. The friction would create a force perpendicular to the normal, and it would make things a little too complicated for the scope of this project.

Since the only force we consider is the normal one, we can consider the net impulse to be $j\vec{n}$, where j is the impulse parameter. The body A will experience the net impulse $j\vec{n}$ and body B will experience it's negation $-j\vec{n}$ since the force that B experience is equal an opposite to the one experienced by A. The impulse is a change in momentum. Momentum has units of velocity times mass, so if we divide the impulse by the mass we get the change in velocity. We can relate initial and final velocities as

$$\vec{v}_{a2} = \vec{v}_{a1} + \frac{j\vec{n}}{m_a} \tag{3.20}$$

$$\vec{v}_{b2} = \vec{v}_{b1} - \frac{j\vec{n}}{m_b} \tag{3.21}$$

For the final angular speed, it's change from the impulse $j\vec{n}$ is given by $\vec{r}_{ap} \times j\vec{n}$. We can divide the result by the moment of inertia, which was calculated in section 3.1, to get convert the change in angular momentum into change in angular speed. Similarly as above, we can relate the initial and final angular velocity as

$$\omega_{a2} = \omega_{a1} + \frac{\vec{r}_{ap} \times j\vec{n}}{I_a} \tag{3.22}$$

$$\omega_{b2} = \omega_{b1} - \frac{\vec{r}_{bp} \times j\vec{n}}{I_b} \tag{3.23}$$

Solving for the impulse parameter Now we have everything we need to solve for the impulse parameter j. The bulk of the calculations can be found in section A.4 of the appendix A. But basically we start with equation 3.19 and we end up with

$$j = \frac{-(1+e) \cdot \vec{v}_{ap1} \cdot \vec{n}}{\frac{1}{m_a} + \frac{1}{m_b} + \frac{\left(\vec{r}_{ap} \times \vec{n}\right)^2}{I_a} + \frac{\left(\vec{r}_{bp} \times \vec{n}\right)^2}{I_b}} \tag{3.24}$$

4 Implementation

This section will dive into the actual implementing of the project, trying to describe how the theoretical concepts in section 3.

4.1 Structure

The added code to the project can be divided into three parts

- polygon generator;
- collision detection;
- collision resolution.

Each part will be explained in the following sub-sections.

4.1.1 Polygon generator

Before talking about how we generate polygons, let's first talk about how the polygons are defined in this project.

In the file `polygons.h` we define a polygon as

Listing 1. Polygon class (simplified)

```
class polygon {
    std::vector<vec2d> points;

    vec2d center;
    double angle;

    double inertia;
    double mass;

    std::vector<vec2d> global_points();

    vec2d speed;
    double angular_speed;
}
```

Polygons possesses multiple fields. Firstly we have `std::vector<vec2d> points`, a collection of `vec2d` objects, which represent the set of ordered points that compose the polygon. Those points are expressed in local coordinates, and the center of mass of the polygon is placed the origin.

Now that we know how the polygon is composed, we want to move it around the space it lives in, that is the purpose of the `vec2d center` field. It represents where the center of mass of the polygon is located in simulation. Since the shapes also rotate, we need to keep track of the rotation of the polygon, hence the use of `double angle`. All of these fields are used when computing the result of the method `std::vector<vec2d> global_points()`.

Finally, we have `double inertia` and `double mass` which, as their name suggest, store the value for the polygon's inertia and mass, that are used to calculate the polygons final speed and angular speed, who are represented by `vec2d speed` and `double angular_speed`.

Polygon generator Now that we know how polygons are represented in our simulation, we can generate them. In `polygon_generator.h` (and its implementation file `polygon_generator.cc`), there are some functions for that. In Listing 2 we can see the signature of the functions that generate

- a rectangle of a certain width and height;
- a square of a certain side length;
- a triangle given two side lengths and an angle between them;
- a regular polygon;
- an arbitrary polygon.

Listing 2. Polygon Generator header file

```
namespace poly_generate {
    polygon rectangle(double width, double height);

    inline polygon square(double width) {
        assert(width > 0);
        return rectangle(width, width, label);
    };

    polygon triangle(double side1, double side2, double angle);

    polygon regular(double radius, uint n_sides);

    polygon general(std::vector<vec2d> points);
};
```

The implementation of those functions are fairly straight forward, they generate the appropriate number of points in the correct place. After what they calculate the mass and subsequent inertia of the polygon as shown in section 3.1.

4.1.2 Collision

The algorithm described in section 3.2.2 is implemented in `collision.cc` and exposed to other modules with `collision.h`.

The module exposes a collision structure and a collides function.

Listing 3. Collision header file

```
struct collision {
    bool collides = false;
    vec2d impact_point;
    vec2d n;
    vec2d overlap;
};

extern collision collides(polygon& p, polygon& q);
```

The `collides` function takes in two polygons and checks with vertex-collision algorithm whether they collide or not. The result is return through an instance of `struct collision`. The `bool collides` and `vec2d impact_point` fields is self-explanatory. The `vec2d n` is the normal vector that pushes `polygon& p` away from `polygon& q`. Finally, `vec2d overlap` is a scalar multiplication of `vec2d n`. Where the latter is a normalized vector, the former represents how deep `p` is in `q`. If we push `p` the exact amount of `overlap`, then `p` would just be touching `q` with point `impact_point`, and no further overlap would occur.

In reality, we do not simply push `p` by `overlap`, but we push `p` and `q` away from each other proportionally to their mass.

Collision resolution In `polygons.cc`, among other functions, we apply the collision resolution, which simply uses the physics results found in 3.3.

4.2 Optimization

In order to optimise the collision detection and to avoid having to do a check whether each vertex of every polygon was colliding with each edge of every other polygon we decided to apply the bounding box acceleration structure.

The bounding box of an object is, in two dimensions, the rectangle that contains the object and which the sides are aligned with axis of the coordinate system.

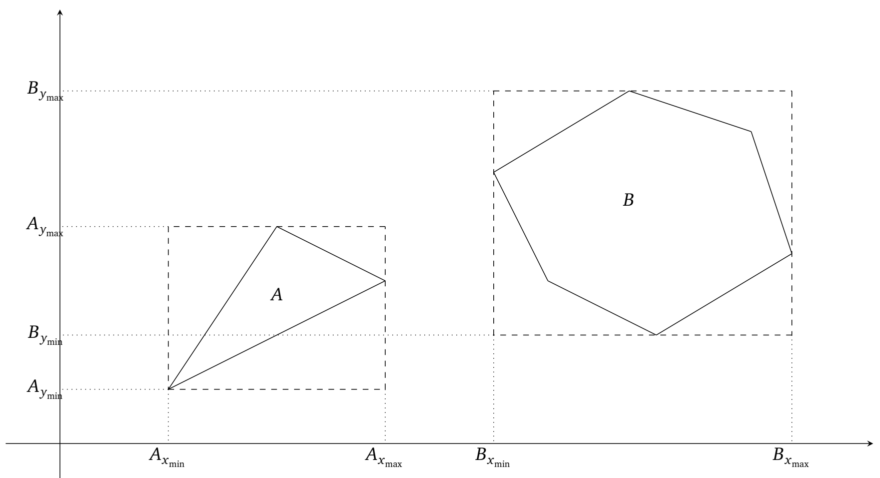


Figure 4.1. Bounding box example

To get the four coordinates that compose, we just perform a linear scan through all the points that compose a polygon, record the minimum and maximum of both they x and y coordinate.

Once those points have been found, we just perform some basic axis-aligned rectangle collision detection, which is much less computationally expensive than checking each vertex-edge pair.

The complexity of the collision detection algorithm went from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, where n is the total number of vertices across all polygons in the simulation.

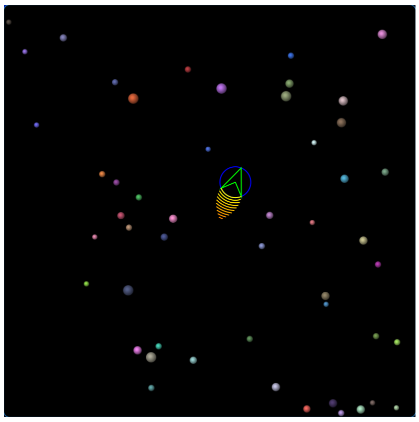
4.3 Known issues

The simulation has one major flaw, and it is inherent to the last part of section 4.1.2 where we talk about the `overlap` vector: we said that this vector allows to displace both polygons so that and the end of the collision resolution, they are not overlapping anymore. This was done to avoid issues where the collision resolution had taken place in one frame, but the polygons were still overlapping in the following frame, which lead to polygons getting stuck together.

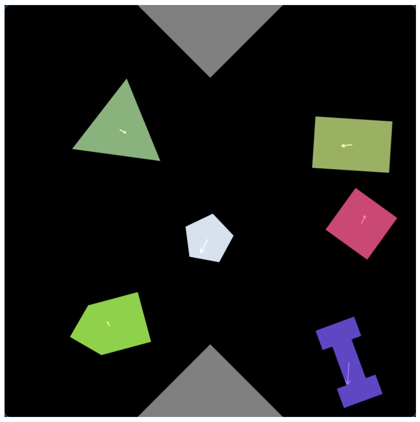
The unfortunate consequence of such a decision is that, when we start playing around with the restitution coefficient and all the shapes start falling to the ground, the shapes are not able to rest still. Since there is gravity, at each frame their speed get updated to simulate its effect, but since they are lying on the floor, the collision detection algorithm kicks in right away, resulting in the polygon getting pushed completely up, since the ground has infinite mass. This results in the polygon bouncing on the floor instead of lying down.

5 Conclusion

The overall development of this extension was very instructive. Getting to discover some theoretical concepts, having to understand them and then being able to apply them in the code is a very rewarding experience. The objective of project is considered accomplished. In figure 5.1, we can see a comparison of the state of the application before and after the extension.



(a) Original state of the simulation



(b) State after the extension

Figure 5.1. Before v. After

References

[1] David Eberly. Intersection of convex objects: The method of separating axes. *WWW page*, pages 2–3, 2001.

[2] Erik Neumann. Rigid body collisions. *WWW page: myPhysicsLab.com*, 2003.

[3] Chris Hecker. Physics part 3: Collision response. *Game Developer Magazine*, 1997.

A Calculations

A.1 Moment of inertia of rectangle

$$
\begin{aligned}
I_{\text{rect}} &= \rho \int_{-\frac{h}{2}}^{\frac{h}{2}} \int_{-\frac{w}{2}}^{\frac{w}{2}} x^2 + y^2 \, \mathrm{d}x \, \mathrm{d}y \\
&= 4\rho \int_{0}^{\frac{h}{2}} \int_{0}^{\frac{w}{2}} x^2 + y^2 \, \mathrm{d}x \, \mathrm{d}y \\
&= 4\rho \int_{0}^{\frac{h}{2}} \left[\frac{1}{3} x^3 + x y^2 \right]_0^{\frac{w}{2}} dy \\
&= 4\rho \int_{0}^{\frac{h}{2}} \frac{1}{3} \frac{w^3}{8} + \frac{w}{2} y^2 dy \\
&= 2\rho \int_{0}^{\frac{h}{2}} \frac{w^3}{12} + w y^2 dy \\
&= 2\rho \left[\frac{w^3}{12} y + \frac{w}{3} y^3 \right]_0^{\frac{h}{2}} \\
&= 2\rho \frac{w}{3} \left[\frac{w^2}{4} y + y^3 \right]_0^{\frac{h}{2}} \\
&= 2\rho \frac{w}{3} \left(\frac{w^2}{4} \frac{h}{2} + \frac{h^3}{8} \right) \\
&= \rho \frac{w}{3} \left(\frac{w^2}{4} h + \frac{h^3}{4} \right) \\
&= \frac{\rho w h}{12} \left(w^2 + h^3 \right)
\end{aligned}
\tag{A.1}
$$

A.2 Moment of inertia of sub-triangle of regular polygon

Before starting the calculations, it is to be noted that according to Figure 3.3, we have that

$$\tan\left(\frac{\theta}{2}\right) = \frac{\frac{l}{2}}{h} = \frac{l}{2h}$$

it will be useful to simplify the result of the integral.

$$\begin{aligned}
I_T &= \rho \int_0^h \int_{-\frac{lx}{2h}}^{\frac{lx}{2h}} x^2 + y^2 \,\mathrm{d}y\,\mathrm{d}x \\
&= 2\rho \int_0^h \int_0^{\frac{lx}{2h}} x^2 + y^2 \,\mathrm{d}y\,\mathrm{d}x \\
&= 2\rho \int_0^h \left[x^2 y + \frac{1}{3} y^3 \right]_0^{\frac{lx}{2h}} \mathrm{d}x \\
&= 2\rho \int_0^h x^2 \frac{lx}{2h} + \frac{1}{3}\frac{l^3 x^3}{8h^3} \,\mathrm{d}x \\
&= 2\rho \left(\frac{l}{2h} + \frac{l^3}{24h^3} \right) \int_0^h x^3 \,\mathrm{d}x \\
&= 2\rho \left(\frac{l}{2h} + \frac{l^3}{24h^3} \right) \left[\frac{1}{4} x^4 \right]_0^h \\
&= \frac{h^4 \rho}{2} \left(\frac{l}{2h} + \frac{l^3}{24h^3} \right) \\
&= \frac{\rho l h^3}{4} \left(1 + \frac{l^2}{12h^2} \right) \\
&= \frac{m_T h^2}{2} \left(1 + \frac{l^2}{12h^2} \right) \\
&= \frac{m_T}{2} \frac{l^2}{4\tan^2\left(\frac{\theta}{2}\right)} \left(1 + \frac{4\tan^2\left(\frac{\theta}{2}\right)}{12} \right) \\
&= \frac{m_T l^2}{24} \left(1 + 3\cot^2\left(\frac{\theta}{2}\right) \right)
\end{aligned} \tag{A.2}$$

A.3 Moment of inertia of sub-triangle of arbitrary polygon

Recall equation 3.10 defines

$$\vec{r} = \alpha\overrightarrow{CA} + \beta\alpha\overrightarrow{AB}$$

$$\begin{aligned}
I_{T_i} &= \rho \int_0^1 \int_0^1 \vec{r}^2 hb\alpha \,\mathrm{d}\alpha\,\mathrm{d}\beta \\
&= \rho hb \int_0^1 \int_0^1 \left(\alpha\overrightarrow{CA} + \beta\alpha\overrightarrow{AB}\right)^2 \alpha \,\mathrm{d}\alpha\,\mathrm{d}\beta \\
&= \rho hb \int_0^1 \int_0^1 \left(\alpha^2\overrightarrow{CA}^2 + 2\alpha^2\beta\overrightarrow{AB}\cdot\overrightarrow{CA} + \alpha^2\beta^2\overrightarrow{AB}^2\right) \alpha \,\mathrm{d}\alpha\,\mathrm{d}\beta \\
&= \rho hb \int_0^1 \int_0^1 \alpha^3 \left(\overrightarrow{CA}^2 + 2\beta\overrightarrow{AB}\cdot\overrightarrow{CA} + \beta^2\overrightarrow{AB}^2\right) \mathrm{d}\alpha\,\mathrm{d}\beta \\
&= \rho hb \int_0^1 \left[\frac{1}{4}\alpha^4 \left(\overrightarrow{CA}^2 + 2\beta\overrightarrow{AB}\cdot\overrightarrow{CA} + \beta^2\overrightarrow{AB}^2\right)\right]_0^1 \mathrm{d}\beta \\
&= \frac{\rho hb}{4} \int_0^1 \beta^2\overrightarrow{AB}^2 + 2\beta\overrightarrow{AB}\cdot\overrightarrow{CA} + \overrightarrow{CA}^2 \,\mathrm{d}\beta \\
&= \frac{\rho hb}{4} \left[\frac{1}{3}\beta^3\overrightarrow{AB}^2 + \beta^2\overrightarrow{AB}\cdot\overrightarrow{CA} + \beta\overrightarrow{CA}^2\right]_0^1 \\
&= \frac{\rho hb}{4} \left(\frac{1}{3}\overrightarrow{AB}^2 + \overrightarrow{AB}\cdot\overrightarrow{CA} + \overrightarrow{CA}^2\right)
\end{aligned} \tag{A.3}$$

A.4 Solving for impulse parameter

We start with equation 3.19:

$$\begin{aligned} \vec{v}_{p2} \cdot \vec{n} &= -e\vec{v}_{p1} \cdot \vec{n} \\ \left(\vec{v}_{ap2} - \vec{v}_{bp2}\right) \cdot \vec{n} &= -e\vec{v}_{p1} \cdot \vec{n} \\ \left(\vec{v}_{a2} + \omega_{a2} \times \vec{r}_{ap} - \vec{v}_{b2} - \omega_{b2} \times \vec{r}_{bp}\right) \cdot \vec{n} &= -e\vec{v}_{p1} \cdot \vec{n} \end{aligned}$$

We now expand the bracket on the left-hand side using the equations 3.20 - 3.23 and then simplify with equation 3.17.

$$\begin{aligned} \left(\vec{v}_{a1} + \frac{j\vec{n}}{m_a} + \omega_{a1} + \frac{\vec{r}_{ap} \times j\vec{n}}{I_a} \times \vec{r}_{ap} - \vec{v}_{b1} + \frac{j\vec{n}}{m_b} - \omega_{b1} + \frac{\vec{r}_{bp} \times j\vec{n}}{I_b} \times \vec{r}_{bp}\right) \cdot \vec{n} &= -e\vec{v}_{p1} \cdot \vec{n} \\ \left(\frac{j\vec{n}}{m_a} + \frac{\vec{r}_{ap} \times j\vec{n}}{I_a} \times \vec{r}_{ap} + \frac{j\vec{n}}{m_b} + \frac{\vec{r}_{bp} \times j\vec{n}}{I_b} \times \vec{r}_{bp}\right) \cdot \vec{n} &= -(1+e)\vec{v}_{p1} \cdot \vec{n} \end{aligned}$$

Using the triple scalar product rule, we can derive that

$$j\left(1/m_a + 1/m_b + \frac{\left((\vec{r}_{ap} \times \vec{n})^2\right)}{I_a} + \frac{\left(\vec{r}_{bp} \times \vec{n}\right)^2}{I_b}\right) = -(1+e)\vec{v}_{p1} \cdot \vec{n}$$

and therefore

$$j = \frac{-(1+e) \cdot \vec{v}_{ap1} \cdot \vec{n}}{\frac{1}{m_a} + \frac{1}{m_b} + \frac{\left(\vec{r}_{ap} \times \vec{n}\right)^2}{I_a} + \frac{\left(\vec{r}_{bp} \times \vec{n}\right)^2}{I_b}} \tag{A.4}$$