

AGH, WEAIiB	<u>Modelling of Physical Systems</u>	Date 15.06.2018
Systems modelling and Data analysis	Project topic: Multibody collision	
	Authors: Michał Morawiec, Jakub Bieńkowski	

1. Aim:

The goal of our project was to prepare physical model presenting collision between few objects. Task has been divided into two problems, physical calculations and form of collisions presentation.

2. Theoretical introduction:

2.1 Elastic collision:

An elastic collision is a collision in which kinetic energy is conserved. That means no energy is lost as heat or sound during the collision. In the real world, there are no perfectly elastic collisions on an everyday scale of size. In an elastic collision, both kinetic energy and momentum are conserved (the total before and after the collision remains the same).

Conservation of momentum: $m_1v_1 + m_2v_2 = m_1v'_2 + m_2v'_2$

Conservation of kinetic energy: $\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1v_1'^2 + \frac{1}{2}m_2v_2'^2$

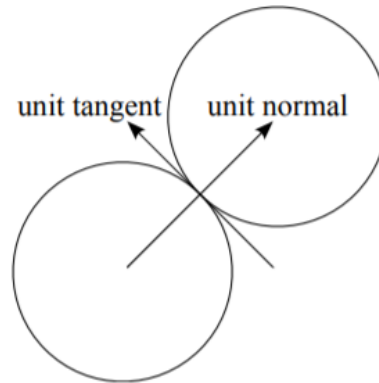
3. Methodology description:

During collision implementation we have facing with two kinds of possible collisions: collisions between two circles and collision between circle and border. Each methodology will be briefly described below:

3.1 Collision between two circles:

To simplify model we have decided to treat out phenomena as simple elastic collision in two dimensions. We assumed that in our model we will have only one specific types of body – circle. Each body have velocity, mass, radius and coordinates. Used algorithm has been divided into 6 stages.

In stage one our goal was to find normal and unit tangent vector. The unit normal vector is a vector which has a magnitude of 1 and a direction that is normal to the surfaces of the objects at the point of collision. Unit tangent vector is vector with a magnitude of 1 which is tangent to the circles' surfaces at the point of collision.



First we have to find unit normal vector which is calculated with following formula:

$$\vec{un} = \frac{\vec{n}}{\sqrt{n_x^2 + n_y^2}}, \text{ where } \vec{n} = \langle x_2 - x_1, y_2 - y_1 \rangle$$

When we have obtained normal vector we can easily calculate unit tangent vector:

$$\vec{ut} = \langle -n_y, n_x \rangle$$

Stage two was based on resolving of velocity vectors for each body into both components – tangential and normal, to perform this computations we have to calculate dot product, it is calculated according to following formulas:

$$v_{1n} = \vec{un} \circ \vec{v}_1, \quad v_{1t} = \vec{ut} \circ \vec{v}_1, \quad v_{2n} = \vec{un} \circ \vec{v}_2, \quad v_{2t} = \vec{ut} \circ \vec{v}_2$$

In stage three we have computed new tangential velocities. The tangential part of velocity have not changed during collision because there is no force between the circles in the tangential direction. So we will simply assign values before collision to values after collision.

$$v'_{1t} = v_{1t}, \quad v'_{2t} = v_{2t}$$

Stage four is most complicated part, in this step we have to combine conservation of kinetic energy and conservation of momentum in order to obtain new normal velocities. After performing few algebraic operations we have obtained proper formulas (Important thing to note is fact that in this step we treat collision as typical one – dimensional collision)

$$v'_{1n} = \frac{v_{1n}(m_1 - m_2) + 2(m_2 - v_{2n})}{m_1 + m_2}, \quad v'_{2n} = \frac{v_{2n}(m_2 - m_1) + 2(m_1 - v_{1n})}{m_1 + m_2}$$

Now we have to convert scalar normal and tangential velocities into vector. To do that we have to perform following computations:

$$\overrightarrow{v'_{1n}} = v'_{1n} \cdot \overrightarrow{un}, \quad \overrightarrow{v'_{1t}} = v'_{1t} \cdot \overrightarrow{ut}, \quad \overrightarrow{v'_{2n}} = v'_{2n} \cdot \overrightarrow{un}, \quad \overrightarrow{v'_{2t}} = v'_{2t} \cdot \overrightarrow{ut}$$

In the last stage we want to obtain final velocity vectors, there is no complicated computations, in this order it is enough to add normal and tangential vector for each body:

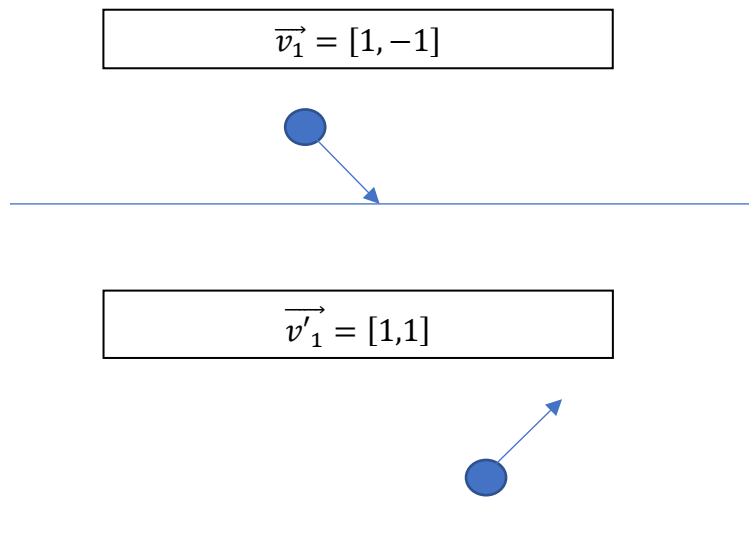
$$\overrightarrow{v'_1} = \overrightarrow{v'_{1n}} + \overrightarrow{v'_{1t}}, \quad \overrightarrow{v'_2} = \overrightarrow{v'_{2n}} + \overrightarrow{v'_{2t}}$$

3.2 Collision between border and circles:

Second implemented type of collision is collision between border and circle, same as in previous example we have considered only elastic collision. This model is significantly simpler than model above, we don't perform any complicated computations. It is sufficient to multiply proper coordinate by -1. In this collision model we distinct collisions with two kinds of border: vertical and horizontal. Depending on border type our computations are a little bit different:

Collision with horizontal border:

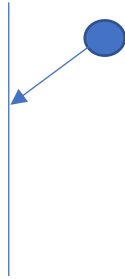
$$\overrightarrow{v'} = [v_x, -1 \cdot v_y]$$



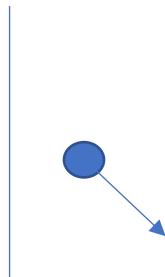
Collision with vertical border:

$$\vec{v'} = [-1 \cdot v_x, v_y]$$

$$\vec{v_1} = [-1, 1]$$



$$\vec{v'_1} = [1, 1]$$



4. Documentation

4.1 Technologies

- Python 3.6
 - NumPy – to simplify calculation on vectors
 - websocket - server
- JavaScript (ECMAScript 2015)
 - webpack - to build and serve user interface
- WebSockets - for frontend-backend communication
- Docker with docker-compose

4.2 Project Assumption

- Fast backend - able to handle at least 60 frames per second
- Flexible physics implementation - easy to switch with more complex model
- User interface - with menu, so the project can be managed easily
- Web browser's window is treated as a world - the boundaries of the browser's window are the borders

4.3 Project Structure

- backend - backend server with world and collision logic
- frontend - user interface with presentation and management menu of multibody collision world

4.4 Backend

Backend code is extracted to separate backend directory, The executable file is main.py which uses the code that is stored inside src directory. Inside one can find:

- Abstracts - contains an interface to use by all collision implementations
- Controllers - contains CircleController, which takes care of all circles (objects) inside the world and processes their collisions (if there are any)
- Factories - factory classes that create objects - Circle (collidable objects) and Rectangle (world)
- Models - contains all world models' classes: Circle, Rectangle and its Borders
- Physics - implements collision interface with use of Strategy Pattern
- Utils - helper classes, the most interesting one might be CircleWrapper which is used to store information about nearby objects that might collide with each other
- WebSockets - contains Websockets Server that initializes the world on user connection
- World - contains the World logic

main.py

To analyze how the backend server works let's start with main.py. The file is pretty self-explanatory. All it does is initialization of world object, which is then handled by server object. WebSockets server is started and is ready to handle incoming connections from frontend. Finally the infinite while world.running(): loop is started. The world.running() method always returns True, but it is covered with simple and readable API. Last but not least, the world is processed - checked for collisions which are handled - and objects' positions are send to frontend (to any client connected to the server). time.sleep is used to limit number of frames per second to stable (non-oscillating) number.

main.py uses two classes src.WebSockets.Server and src.World.World. Let's analyze the WebSockets server first.

src.WebSockets.Server

The WebSockets server is started in separate thread. This is because the server itself runs in infinite loop as well, which would block the remaining part of main.py's code. The server is handling world, because messages coming from frontend are configuration of backend's world. The server listens to three events:

- new client connected
- client disconnected
- message received

The `_message_received` method handles the most important part of Websockets server - it listens for the changes from the frontend and initializes the world with the received data. MessageHandler is used to parse JSON message to Python dictionary.

src.World.World

The next important part of code is world.handle. It accepts three commands:

- init
- start
- pause

start and pause just starts or pauses already initialized world, so there is no point in further documentation. init however creates new world with passed arguments:

- x and y - world's width and height
- objects - circles (objects) created by CircleFactory using received data by WebSockets server

What's more, the boundary and controller are initialized as well. The CircleController is the class that is responsible for physics' logic and will be covered in Physics chapter.

4.5 Frontend

The frontend directory itself does not contain any interesting files. The main file is `src/index.js` which uses two other JavaScript files:

- `draw.js` - with code responsible for drawing the circles (objects)
- `world.js` - which takes care of world initialization and sending proper data to backend server

Some Cascade Style Sheets (CSS) are included as well, to give the project proper look.

When user enters the `http://localhost:8080` the `index.js` file is loaded by the browser. The connection to the WebSockets server is opened and maintained until the browser's tab is closed. The initial values are parsed and processed by the backend. As the result, the user can see one object bouncing on the screen. Using the Show/hide menu, user can modify the world:

- change the number of the objects
- select whether the objects should have equal mass or not
- pause or start or init the world again

draw.js

The objects are presented using HTML `<canvas>`. How `draw.js` creates object can be found here. Each object's color is representing different mass, therefore objects of the same color have the same mass.

world.js

Retrieves and processes the data about the world. As the world is defined by the browser's window, each resize of the window has to be handled as the world dimensions are changing as well. This is why the `updateWorldSize` has been implemented.

4.6 Physics

Code responsible for physical computations were encapsulated in `src.Physics` directory. In this directory we can find two files:

- `Collision.py` – Include method which allow to execute proper Collision depending passed Tag which describe type of collision
- `CollisionsStrategies.py` – include implementation of particular type of collision

Next important directory in terms of physical computations is `src.Controllers` in this file is we encounter class `CircleController.py` which observe position of circles, and detect collisions.

src.Physics.CollisionsStrategy.py

In this file are located two very important classes which defines behavior of objects involved into collisions.

TwoDimensionCollision – this class inherit from abstract class **AbstractCollisionStrategy**, so it has to implement method **collide**. This method takes two arguments, those arguments are two colliding objects. In method we have implemented a lot of computations according to theory introduced in chapter 3.1

BorderCollision – this class also inherit from class **AbstractCollisionStrategy**, thus here is also implemented method **collide**. In contrary to implementation in class **TwoDimensionCollision**, we expect that second parameter passed to the method is object with type **Border**. If this condition is not satisfied method will rise an **Exception**, however if condition is satisfied it will perform computations related to collision. Detailed description how collision runs is involved in chapter 3.2. Method decide which computation perform using object **_b**'s field called **direction**. This field can have value: “Vertical” or “Horizontal”

src.Controllers.CircleController.py

This is one of the most important class in the project, class contains three filed:

- **boundary** – field describe rectangle in which circles performs collisions
- **circle_list** – list filled with all circles occurring in program
- **memory** – utility field which helps us avoid situation where collision between two same circle occurred twice in the same time

CircleController class contain method **update_position**, in this method looping through previously mentioned field **circle_list**. First, we are trying to determine if currently examined circle is colliding with border if yes, we are performing collision with border according to algorithm mentioned in previous chapter and we are starting to examine next circle. If collision detection with border check failed we are finding two closest circles to our examined circle. Now we are using our class field - **memory** and trying to determine if collision occurred between examined circle and previously found circle. If yes, we are starting new loop iteration and examining new circle. If check failed we have are trying to determine if examined circles and previously found circles are close enough to perform collision.

5. Project Installation

- **Install Docker** – Follow detailed instruction available at this page: <https://docs.docker.com/install/>
- **Install docker - compose** – Follow detailed instruction available at this page: <https://docs.docker.com/compose/install/>
- **Start project locally** - To start project locally type `docker-compose up -d --build` (or use Docker for Windows/Mac). The process of creating and provisioning of the project can take up to few minutes

When the project has started you can visit <http://localhost:8080> to see user interface.

6. Literature

- Source code: <https://github.com/jbienkowski311/multibody-collision>
- *2-Dimensional Elastic Collisions without Trigonometry*, 2009 Chad Berchek: <http://www.vobarian.com/collisions/2dcollisions2.pdf>
- *Elastic collision*, Wikipedia: https://en.wikipedia.org/wiki/Elastic_collision