IntersectionInstance

object1

object2

__does_intersect = False

__is_infinitesimal = False

_intersection_points = []

---

intersect() -> None
  the entry point for performing the intersection algorithm on its two objects. No return value, the other methods do the job. The results are then taken by calling methods such as does_intersect, is_infinitesimal and so on.

does_intersect() -> bool

cylinder_cube()

line_circle() -> list, float

rectangle_lines() -> list

rotation_matrix_2d()

horizontal_line_segment_intersection()

cube_cube()

cylinder_cylinder()

is_infinitesimal()

get_intersection_point()

World

objects: dict[ObjectId, Object]

__vis_data : dict

__creation_ts: float

__num_evolutions: int

__current_time_ms: int

__vis_output_filename: str

__duration_sec: int

__vis_frame_interval_ms: float

__next_frame_time_msec: float

---

constructor(map: Map, vis_filename)

evolve(delta-t)
  calls evolve() of all objects which have no owner, takes care of offspring objects, and finally kills objects which ask for it

run()
  manages delta-t (how small it should be), manages intersections using intersect(), and calls evolve()

intersect() -> tuple
  instantiates InIn for each pair of objects and calls InIn.intersect() to evaluate the intersection

add_object(new_objects: dict[ObjectId, Object]) -> None

delete_expired_objects() -> None

register_intersections(intersection_result: InInType) -> None

pick_delta_t() -> float

update_visualization_json()

dump_all_shapes_info()

dump_vis_data_to_file()

visualize()

Map

next_available_id

---

parse_map(self, filename: str) -> dict[ObjectId, Object]

get_next_id(self) -> ObjectId

instantiate_object(obj_json, new_id: ObjectId, name: string, owner: Object) -> Object

get_shape(obj_json) -> Shape

get_position(obj_json) -> Shape

get_objects(self, obj_map: dict[ObjectId, Object], parsed, owner: Object) -> dict[ObjectId, Object]

Position

position and orientation

---

x,y,z

phi,theta

Shape

---

bounding_box()
  returns a x-y plane bounding box. Can be done using a generalized algorithm, no implemented only in the parent class.

dump_info()

Cylinder

Cube

Sensor

inherently an Object, so needs shape, position, evolve(), etc.

---

sense()

BumperSensor

detects at a hit upon intersection, boolean result

Object

oid: objectID
  every object needs an ID to be tracable in map and in state

name: string
  merely a name, irrelevant to the simulation

shape: Shape
  which could be empty if the object is an owner

position: Position
  position and orientation of an anchor point of the object

__previous_position: Position
  to keep the previous state so we can revert

owner_object: ObjectID

dependent_objects: dict[ObjectId, Object]

_latest_intersections: list[IntersectionInstance]

_infinitesimal_intersection_occured: bool

---

evolve(delta-t) -> list[Object]: offspring-objects
  changes the state (position, internal attributes, etc) of the object
  trvial evolution: when the object never changes state
  offsprings are the possibly non-physical objects required to accomplish something.

bounding-box() -> Box
  returns a box which contains the whole object. used to optimize intersection evaluation

get-required-delta-t()
  calculates the delta-t it requires to operate. By default 0, meaning no requirement

time-to-die() -> bool
  tells the World if it wants to be eliminated. This might be where Agent Smith cheated the matrix!

add_dependent_object(obj: Object)

set_intersections(intersections: list[in_in.IntersectionInstance]) -> None

update_position(new_position)-> None

revert_position() -> None

visualize()
  returns the information required for visualization

dump_shape_info() - >None

is_evolvable() -> bool

Robot

---

evolve() is the heart of Robot actions.

VaccumCleanerV0

in version 0, a cylinder with no wheels and stuff.

---

evolve()
  overriden

te_required_delta_t()
  overriden

RigidPhysicalObject

The most general form of it, supporting bumps

---

acceleration: float

velocity: float

---

evolve(delta_t)
  overridden to include bump

new_position_upon_bump()
  placeholder, to be overriden by children

RigidPointBall

A ball with relevant calculations for reflection upon bump, but not rotating
  v0: 2D cylinder instead of sphere.

---

new_position_upon_bump()
  overriden to calculate bump reflection thing

calculate_circle_bounce(circle_center, phi, theta, bump_point)

polar_to_cartesian(r, phi_degree, theta_degree)

cartesian_to_polar(x, y, z)

get_required_delta_t() -> float

Box