Strategic Outcomes

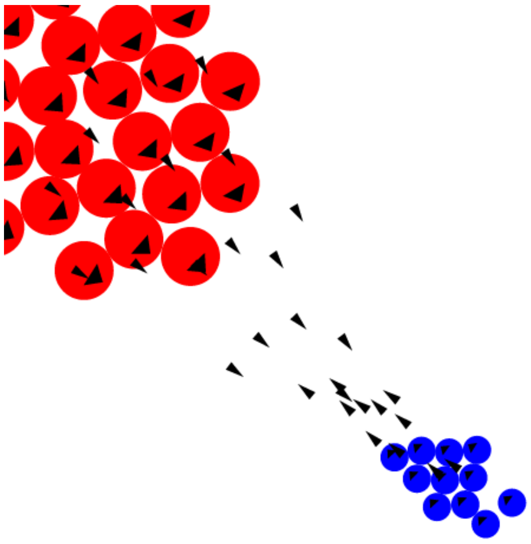
Justen Rickert

May 12, 2017

Contents

1 Purpose

This is an image of a simulation, where the goal could be a number of things, like simulating bacterial infections inside of an open wound, or soldiers on a battlefield, or players competing against one another in a strategy game. In any of the cases, there is a notion of strategy involved at some point. A bacterial infection plays the game of trying to outsmart the natural defense mechanisms of the organism that it is invading. Soldiers wish to succeed in capturing strategic points on the battlefield to keep casualties during conflicts lower than opposing forces. In a strategy game all that exists in the game is strategy; a player must have great knowledge of the rules of the game so he can use his/her time playing to make moves which would place him/her as the victor after the game has been decided.



Surely, any of these things would be at least interesting to look at. However it is not always clear to whom the outcome will be in favor. One could ask good, pertinent questions as: How bad does an infection need to be so that the host of the infection does not have any hope of fighting it off; what amount of soldiers, with what munitions, and with what positional advantage does an army need in order that victory is ensured; or, generally speaking, what strategic decisions need to be made so that one side meets the conditions of success while its opposing side does not? Furthermore what considerations need to be made to give any of these examples determinable outcomes? Perhaps a patient with some ailment is never fully rid of it; or a war is waged on a battlefield indefinitely without a side ever succeeding over the other.

In the simulations described in this paper, success or failure can be verified in a straightforward manner. Run the simulation, then wait until the conditions of success or failure are met, thus halting the simulation. If the simulation does not halt, then in many cases it may be reasonable to assume that the simulation will never halt. It is the purpose then to consider what sorts of techniques can be employed to better determine how the constraints of a simulation lead to a simulation concluding.

2 Nomenclature and convention

The simulation could be implemented in a number of different ways. For reasons of portability and simplicity, this simulation machine is run on an `HTML Canvas Element` with the JavaScript programming language. That is, a canvas capable of running graphical operations on a web page capable of being loaded by any ordinary web browser. The objects of the simulations are circles, and they move around on the canvas by the laws of vector calculus in 2-dimensional Euclidean Space $\mathbb{R}^2$. There are directional vectors, or velocity vectors, in the direction that the circles face themselves, and positional vectors defining the placements of the circles on the canvas.

A circle c may take damage from an opposing circle $\bar{c}$ when the opposing circle damages it. A circle might be capable of damaging another circle when its attack is available to use and if it reaches the circle it wants to damage such that the two circles are touching. A circle might also be able to damage another circle by firing something at it, like a bullet, or maybe even the circle does not even need to fire anything because it is simply capable of damaging another circle from a distance. A circle is eliminated on the canvas when it turns gray, or letting θ to be the predicate function determining death, if $\theta(c) = T$, then the circle c is gray and dead. In contrast $\neg\theta$ or τ denotes alive, where the $\tau(c) = T$ implies that the circle c will be colored either red or blue, depending upon which color it is.

For the red team, the set of all red circles $\mathcal{R}$, victory is determined when it is that the set of all blue circles, denoted $\mathcal{B}$, are colored gray and at least one of the red circles is not: $[\forall \bar{c} \in \mathcal{B}, \theta(\bar{c})] \wedge [\exists c \in \tau(c)]$. In such a case where $[\forall \bar{c} \in \mathcal{B}, \theta(\bar{c})] \wedge [\forall c \in \theta(c)]$, victory is given to $\mathcal{R}$ to rule out the possibility of a tie.

2.1 Positioning circles on the canvas with the vector abstraction

In the code block below is a vector abstraction[1] used to program the movement of the circles.

```
class Vector {
  constructor(public x: number, public y: number) { }

  static times = (k: number, v: Vector): Vector =>
    new Vector(k * v.x, k * v.y);

  static minus = (v1: Vector, v2: Vector): Vector =>
    new Vector(v1.x - v2.x, v1.y - v2.y);

  static plus = (v1: Vector, v2: Vector): Vector =>
    new Vector(v1.x + v2.x, v1.y + v2.y);

  static dot = (v1: Vector, v2: Vector): number =>
    v1.x * v2.x + v1.y * v2.y;

  static mag = (v: Vector): number => Math.sqrt(v.x * v.x + v.y * v.y);

  static unit = (v: Vector): Vector =>
    Vector.times(1 / Vector.mag(v), v);

  static distance = (v1: Vector, v2: Vector): number =>
```

[1]Slightly condensed

```
    Vector.mag(Vector.minus(v2, v1));

  static cross = (v1: Vector, v2: Vector): number =>
    v1.x * v2.y - v1.y * v2.x;

  static angleBetween = (v1: Vector, v2: Vector): number =>
    Math.atan2(Vector.cross(v1, v2), Vector.dot(v1, v2));
}
```

The code examples use the TypeScript programming language. The choice was made to use TypeScript because TypeScript is more easily readable language than ordinary JavaScript. Because function definitions are statically typed, their purpose to the rest of the code base is a lot more apparent. A brief explanation is as follows: A class can be named as evident in the above source code block. The class can only be created when passed the parameters as specified in the `constructor` function. Member function parameters are within parentheses, separated by commas as `(parameterName: ParameterType, secondParameterName: SecondParameterType, ...lastParameterName: lastParameterType)`, where to the left of the `:` inside of the parentheses is the parameter name, and to the right is the typing assigned to the parameter. The return object type is listed after these parameters and a `:`. Finally, the function return value is the returned value of whatever is on the right hand side (or next line) after the arrow operator `=>`.

The `Vector.times` function (that is, the `times` member function in the `Vector` class) is a scalar multiplication of a vector value by a constant. `Vector.times` therefore takes a `number` type parameter to be the scalar, and a `Vector` type parameter to be the vector, and returns the `Vector` parameter scaled by the `number` parameter. `Vector.plus` and `Vector.minus` take two `Vector` parameters, and return the vector addition and vector subtraction, respectively, as a single `Vector` type return value. `Vector.dot` and `Vector.distance` take two `Vector` parameters and returns a `number` type dot product of the two vectors and the distance between the two vectors, respectively. `Vector.mag` and `Vector.unit` each take one `Vector` type parameter and return the `number` type vector magnitude and vector unit, respectively. The `Vector.cross` function is somewhat of a misnomer; it takes two `Vector` type parameters and returns a `number` type parameter. Euclidean space $\mathbb{R}^3$ has a meaningful cross product; here, the function is called `cross`, but is not actually a cross product. The `Vector.cross` function is used in the `Vector.angleBetween` function returns which returns a `number` type corresponding to the angle between two `Vector` type parameter velocity vectors.

This vector abstraction is used liberally throughout the game. Dealing with data is much easier when the data is packaged into one element. One need only talk about one vector, instead of the points that make up the vector. An additional example: Circles contain a position vector, a direction vector, and use vectors in many different ways; however, when talking about the circle, it is convenient to say simply the circle's velocity, or the circle's position, or just the circle in general.

2.2 Designing interesting circle movement on the canvas with the Behavior interface

The main component of each circle's behavior is the `Behavior` interface. Each behavior is built as a class implementing two member functions, `condition`, and `consequence`. The circle is placed onto the canvas on the screen as a `vertex` (as though the canvas is a graph), and it runs in the context of a game loop eventually provided by an object of the class `Game`.

```
interface Behavior {
  condition(v: Vertex, game: Game): boolean;
  consequence(v: Vertex, game: Game): any;
}
```

In the declaration of the `behave` function, it becomes clear how the behaviors are utilized. At any given state, the circle will only be able to use one of the available behaviors. The behaviors transition the circle into the next position on the canvas by greedily choosing the first behavior found to be available.[2]

```
behave = (v: Vertex, game: Game): void => {
  if (v.circle.isDead())
    return
  for (let b of this.behaviors) {
    if (b.condition(v, game)) {
      b.consequence(v, game);
      return
    }
  }
}
```

By the way that the circles are created, it has been made easy to develop new behaviors. Once a behavior is defined, it can simply be added to the list of behaviors of the circle. Then, the behavior should operate correctly given that the consequence and condition functions are defined in the way specified. The following code fragments provide a simple example. `SimpleAimShootBehavior`, and `SimpleAttackBehavior` do nothing but search for an opposing circle to attack, then attack that circle by either shooting a bullet at them or charging towards them.

```
let circle = new RedCircle(i); // for some index i, some unique id.
circle.behaviors = [new SimpleAimShootBehavior(), new SimpleAttackBehavior()];
```

[2]`let` is the keyword for creation of a variable in only the closest namespace. `of` is a keyword specific to the `for` loop constructions in TypeScript. It specifies that at each iteration of the for loop, the `let` variable is assigned to the current element of the array. By contrast, the `in` keyword specifies in the `for` loop constructions that the `let` variable is assigned each of the indexes of the array instead of the element of the array.

The condition of both of these functions looks something like the code figure below. Each circle on the canvas will have to first find an enemy. If there is an attack available for the circle to use and the enemy is in range to be attacked, then the condition is met and so the consequence will then run so that no other behaviors can be chosen for that circle for that game state.

```
condition = (v: Vertex, game: Game): boolean => {
  // find enemy circle to attack
  this.enemyTarget = this.findEnemy(game);
  return this.canAttack();
}
```

In the consequence, the circle makes the decision to attack the enemy target circle. At this point if the circle is not facing the enemy target circle or if the enemy target circle is too far away, then it is impossible to attack the enemy target circle properly. Thus the circle first tries to turn to the appropriate direction of the enemy target circle, and also move towards the position of the enemy target circle. When the attack can be made properly, then the attack is made.

In the case of the `SimpleAttackBehavior`, the circle simply lunges at the enemy target circle. In the case of the `SimpleAimShootBehavior`, the circle simply shoots a bullet in its current direction. However, in the case for `SimpleAimShootBehavior`, the bullet needs to be able to appear in the game, therefore the `shootBullet` function needs the parameter of the circle's vertex passed to it, as well as a parameter of type `Game`. The bullet needs the former for current direction of the circle, and needs the latter so that bullet becomes initialized in a place that it can exist.

```
consequence = (vertex: Vertex, game: Game): boolean => {
  // this.enemyTarget is initialized in the condition function. Also, the
  // condition verified that this.enemyTarget is in range to attack.
  if (Vector.angleBetween(
    vertex.circle.vel,
    Vector.minus(this.enemyTarget.circle.position, v.circle.pos)) === 0
      && this.enemyInRange()) {
    this.attack(this.enemyTarget) // or this.shootBullet(v, game);
  } else {
    v.turnToPosition(this.enemyTarget.circle.position);
    v.moveToPosition(this.enemyTarget.circle.position);
  }
}
```

3 Mathematical foundation

The simulations are to compare themselves to Finite Automatons. A simulation is considered to be determinable if the simulation can be defined as an automaton.

3.1 Finite Automaton

Def 3.1. Finite Automaton A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. δ: $Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

In relation to the project at hand, it will be shown that some collections of behaviors given to circles to produce simulations of finite state automatons. Each of the parameters of the 5-tuple are related to the simulation as follows:

1. The set of states, Q is a finite set, where each circle has a personal set of states $Q(c) \in Q$. Each state consists of a 2-dimensional vector position for every circle as well as its quality of aliveness. Each position corresponds to the position of the circle where the number of possible positions is finite by the number of possible pixel locations on the canvas.[3]

2. The alphabet is a finite set of transition names corresponding to a behavior function, σ, that determines what transition a circle will take. That is, each transition corresponds to the behavior that the circle takes to move, denoted $\sigma_i(c)$ for the particular state $q_i(c)$ at the i^{th} turn of the simulation, where $i < N \in \mathbb{N}$ for some N. Since it would be a waste of time for a circle to not be behaving, then it makes sense that a transition should always be taken by the circle every time there is a transition to be taken. The alphabet being read then corresponds to the sequence of behaviors taken by each of the circles during the duration of the simulation.

3. As just stated, the transition function corresponds to the name of the behavior the circle reads, so $\delta_i(q_i, \Sigma_i) = q_{i+1}$ for every state $i < N \in \mathbb{N}$. The transition function, or behavior, determines where the next position that each circle will be, and therefore what its next state will be. δ is also denoted to act upon individual circles as $\delta_i((q_i(c), \sigma(c))) = q_{i+1}(c)$.

[3]It should be noted that is not exactly precise. Any position could not be finite in the sense that there is not a limit in the number of pixels that a circle can occupy on the canvas of the web browser. JavaScript, (and therefore TypeScript, as TypeScript transpiles to JavaScript) uses only rational numbers in its computations—there is no integer primitive. So the point that I'm making here, is wrong in a sense, but it's a point that could be worked out better. The ECMAScript language specification states:

> According to the ECMAScript standard, there is only one number type: the double-precision 64-bit binary format IEEE 754 value (number between $-(2^{53} - 1)$ and $2^{53} - 1$). There is no specific type for integers. In addition to being able to represent floating-point numbers, the number type has three symbolic values: `+Infinity`, `-Infinity`, and `NaN` (not-a-number).

4. The start state can be set to any valid state, where all circles are given a position, and a behavior or a list of behaviors.

5. The ***set of accept states*** is any state where the red team has at least one of its surviving circles, and the blue team doesn't have any of its surviving members. This also suggests that the set of non-accept states is any in which there are no surviving circles of the red team, but at least one surviving circle on the blue team.

At each state of the game q_i, every circle c and every opposing circle $\overline{c}$ chooses its behavior to run its corresponding transition function, that is $[\forall c \in \mathcal{R}, q_{i+1}(c) = \delta((q_i(c), \sigma_i(c)))] \wedge [\forall \overline{c} \in \mathcal{B}, q_{i+1}(\overline{c}) = \delta((q_i, \sigma_i(\overline{c})))]$ at every state $i < N \in \mathbb{N}$ for some N. The simulation halts when $q_{i+1} = f \in F$ for some accepting state f.

3.2 Non-determinism

Def 3.2. Nondeterministic Finite Automaton A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set called the ***states***,

2. Σ is a finite set called the ***alphabet***,

3. δ: $Q \times \Sigma \longrightarrow \mathscr{P}(Q)$ is the ***transition function***,

4. $q_0 \in Q$ is the ***start state***, and

5. $F \subseteq Q$ is the ***set of accept states***.

The only difference between this definition and the other definition of a finite automaton is in the definition of the transition function δ. The state q following a transition δ is any $q' \in \mathscr{P}(Q)$ instead of some specific $q' = \delta(q)$. It is also that a transition can move the circle to any valid position on the canvas. This implies a massive set of possibilities. If a game has a canvas size of $640px \times 640px$, that means that a circle at some position on the canvas has 639^2 different possible next transitions to take, and that is just between the current state and the following state. If each game is expected to run at 60 frames per second, then it would be unfeasible on an average computer to make considerations about all of these moves. As an example, 50 circles would need to make 1,224,963,000 ($50\times$ 639^2 $\times$ 60) computations about their next positions every second.

Using only the transitions written alleviates pretty much all of this computational complexity. At each frame, each circle only needs to make considerations about each of its owned behaviors. In the simple example where each circle's behaviors are `SimpleAimShootBehavior`, and `SimpleAttackBehavior`. 50 circles would need only to make up to 6,000 (50 $\times$ 2 $\times$ 60) computations every second.

The conceptual difference between the simulation being deterministic or nondeterministic lies in the choice of behavior functions σ. At any given state i, multiple different simulations could be run with each state q_i acting as the start state where other simulations could start from. At the moment, however, there have not been very many different behavior functions σ defined. In other words, the derivable nondeterministic δ when considering how the simulation relates to a nondeterministic automaton is not yet quite interesting enough to consider.

3.3 Running loop

Let's consider how the simulation machine itself might actually run. Earlier the win condition for the red team was defined as

$$[\forall \bar{c} \in \mathcal{B}, \theta(\bar{c})] \wedge [\exists c \in \mathcal{R}, \neg\theta(c)].$$

This can easily be computed with the following code snippet.

```
static winCondition = (game: Game): boolean =>
  game.blue.circle.every((c) => game.circleIsDead(c))
  && game.red.circle.some((c) => !game.circleIsDead(c));
```

We can then use the behave function from earlier as well to satisfy the mathematical statement

$$[\forall c \in \mathcal{R}, q_{i+1}(c) = \delta((q_i(c), \sigma_i(c)))] \wedge [\forall \bar{c} \in \mathcal{B}, q_{i+1}(\bar{c}) = \delta((q_i, \sigma_i(\bar{c})))].$$

Since the behave function determined which behavior to use, and then produced the next state using the current state of the circle, one can easily write a computation that satisfies the statement. Note that `game.vertexes` contains all of the blue and red circles, as well as their positions on the canvas, which is required for the `behave` function.

```
static behaviorRun = (game: Game): void =>
  game.vertexes.forEach((v) => v.circle.behave(v, game));
```

4 Qualifying behavior

The purpose of this section is qualify the difference between good behavior and bad behavior in the simulation. Ideally, in the way that we want to determine which simulations can be related to deterministic automatons, we want to be able to say that a simulation can have good behavior. Since the overall behavior of the circles relies on the individual behaviors given to the circles, good and bad behavior globally should be determinable by the individual behaviors given to the circles, or, in more complex situations, the way that individual behaviors interact with each other.

4.1 Good behavior

The first example, using the `SimpleAimShootBehavior` and `SimpleAttackBehavior`, is easily verifiable as a finite state automaton. In any valid consideration of starting states, the two defined behaviors result in the simulation reaching an accepting state.

At the beginning of this example simulation, each circle's `AttackBehavior` condition is met so that it begins to move towards an opposing team's circle in the consequence.

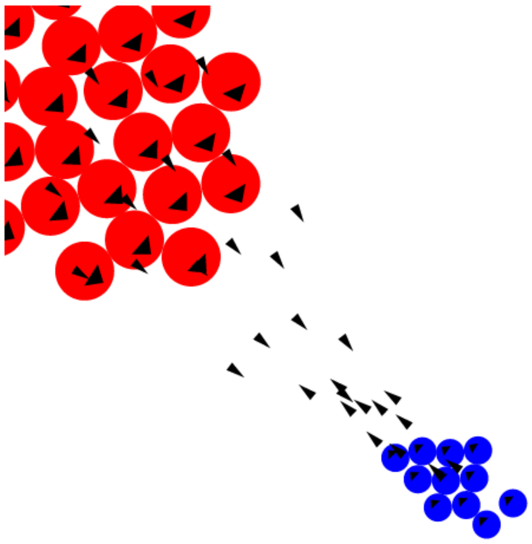


Figure 1: Just after starting the good behavior configuration

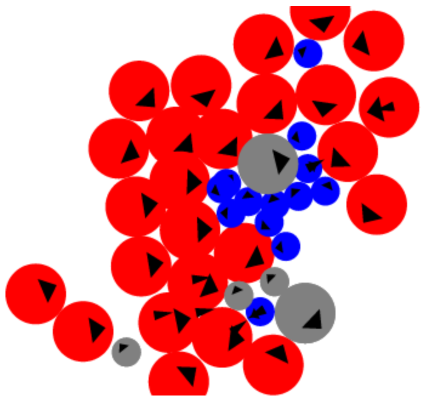


Figure 2: Towards the end of the good behavior simulation

At some point, the circles are close enough to their opposing target circle so that the `SimpleAimShootBehavior` condition is met, wherein the consequence is a bullet being fired. This portion of the simulation could look like this:

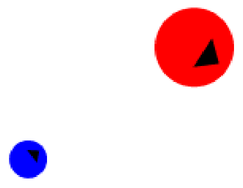
Notably, this construction is well-behaved, as each of the behaviors of the circles have the circles actively searching out opposing circles to eliminate. Towards the end of the simulation the circles must move around the grayed-out, eliminated circles, however their ability to eliminate the opposing circles is still good enough to where the simulation reaches an accepting state.

Unfortunately, certain collections of behaviors may not always yield terminating simulations. The idea is then to decide what kinds of collections of behaviors lead to having simulations with good behavior.

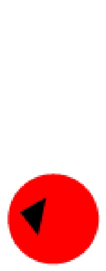
4.2 Simple bad behavior

Consider two simple opposing circles, one given a behavior `EnCircleBehavior`, the other given `SimpleFollowBehavior`. The former circle follows the circumference of an imaginary circular shape on the canvas, and the latter simply follows its opposing circle around. Given this simple setup, it should be clear that the resulting simulation will never find itself in an accepting state.

The simulation at first is something like:

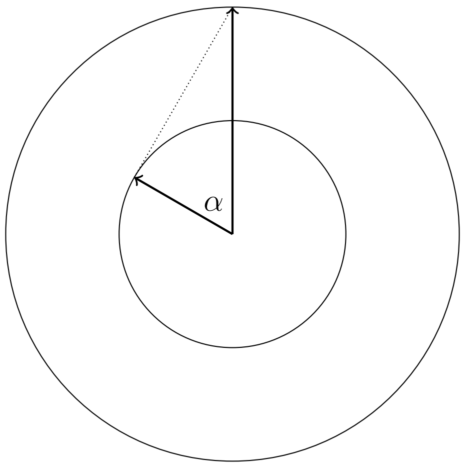


Then, after some time, looking very similarly as before, is something like:



In this particular misbehaving example, the large, red circle has a speed twice as much as the smaller, blue circle. This setup results in the two circles following the circumference of two imaginary, concentric circular shapes different in phase from each other by an angle of $\frac{\pi}{3}$ as detailed in Figure 3.

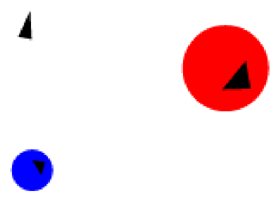
Figure 3: The two imaginary concentric shapes followed by the circles, where $\alpha = \frac{\pi}{3}$. The blue circle is placed at the end of the shorter arrow, and the red is placed at the end of the longer arrow. The intended path of travel of the blue circle at a given instance of time is represented by the dotted line.



Since the two concentric circles, as in the image, have differing radii, their paths never cross. Therefore, the circles moving along the paths of the imaginary circles will never touch, and neither of them will ever receive any damage.

4.3 Complex bad behavior

Given that the circles in the good behavior example had the ability to shoot bullets, one might think that the bad behavior could possibly be rectified by giving the blue circle in the `Simple bad Behavior` section the ability to shoot at the large, red circle that it was chasing. This is a good thought, but the bullets still may not have the speed to satisfy the needs of the blue circle to meet the accepting condition of the game.



Here, the bullets still miss the large red circle completely and so there still needs to be a fashion to determine how good behavior can be reached globally given behaviors that do not interact at all well with each other.

4.4 Analysis

In these examples, it is assumed that there is no real intelligence behind the behaviors exhibited by the circles. The behaviors satisfy one particular movement on the canvas, and do not learn how to fix their behavior if it proves not to be very successful. For example, in the simple bad behavior simulation, if the blue, chasing were to turn itself around and place itself, then it would be able to intercept the running, red circle as it moves around its circle. This would result in better behavior globally, however if the red circle were equally as intelligent, then it would simply itself from moving after it sees the blue, chasing circle stopping itself to turn around. Thus having intelligent circle behavior might result in the circles not having good behavior, in the sense that we want, globally.

Without making any assumptions about the circles' abilities to learn from their mistakes, the challenge of structuring the behaviors of the circles becomes much. For example in the simple bad behavior simulation, the blue, chasing circles' chances of success would easily be increased by simply increasing its movement speed. One could determine the lower limit that the chasing circle's speed need be in order to ensure catching up to the red circle running away. This could be solved analytically by way of considering the properties of the imaginary concentric that the circles moved around. The circles shared the same rotational velocity ω when moving around the circle, and their tangential velocities v_t, having could so be related by ω. When the inner circle moves around the circle with $v_t^{(1)} \cdot r_1 = \omega$, the outer circle moves around the circle with $v_t^{(2)} \cdot r_2 = \omega$. With the relationship $v_t^{(1)} \cdot r_1 = \omega = v_t^{(2)} \cdot r_2$, one could determine, given velocity $v_t^{(2)}$ and radius r_2 the required velocity $v_t^{(1)}$ needed to minimize the distance between the outer circle and the inner circle. When the inner circle has radius ρ_1 and the outer circle has radius ρ_2, it must be that radius $r_1 = \sqrt{r_2^2 - (\rho_1 + \rho_2)^2}$ when the distance between ρ_1 and ρ_2 is minimized. This means that given

$$v_t^{(1)} > \frac{v_t^{(2)} \cdot r_2}{\sqrt{r_2^2 - (\rho_1 + \rho_2)^2}},$$

the simple bad behavior can be made into good behavior.

The same kind of analysis can be made in the complex bad behavior example as well if the assumption is made that the blue, chasing circle fires at a random angle to its right. Say that the red circle moves with a velocity, $v_t^{(2)}$ around the imaginary circle having radius r_2. Given a bullet speed b_s fired at random at an angle $\beta \leq \frac{\pi}{4}$ to the right of the direction of the blue circle at time, the bullet has a chance to hit the red circle traveling in the way of the bullet if it has moved at most $\frac{2\pi}{3} \cdot r_2$ distance around the circumference of the imaginary circle in the time in takes the bullet to at most travel $2 \cdot r_2$ distance across the imaginary circle. This circumstance happens at the latest when

$$t = \frac{b_s}{2 \cdot r_2},$$

having the outer, red circle traveling a distance

$$\int_0^t v_t^{(2)} dt = \frac{2\pi}{3} \cdot r_2.$$

Given that $v_t^{(2)}$ is a constant, the previous equations can be put together as

$$v_t^{(2)} \frac{b_s}{2 \cdot r_2} = \frac{2\pi}{3} \cdot r_2$$

so that the bullet speed b_s can be put in terms of $v_t^{(2)}$ and r_2 as

$$b_s = \frac{4\pi}{3} \cdot \frac{r_2^2}{v_t^{(2)}}.$$

Using this bullet speed, the least angle of β possible to hit the red circle; however, this least angle would only be applicable in the this particular case, and probably we would not want a shooting behavior that never fired directly forward.

In any case, it is this sort of analysis which leads one to be able to determine what parameters need to be placed on the given behaviors written in order to ensure the prevalence of good global behavior in the simulation.

5 Bibliography

- ECMA-262 7th edition: ECMAScript 2016 Language Specification (June 2016) by E.C.M.A. International

- Sipser, M. (2006). Introduction to the theory of computation. Boston: Thomson Course Technology.