CodeBall 2018

Правила

Версия 0.2.2

декабрь 2018 — январь 2019

Оглавление

1 О мире CodeBall 2018 **2**
1.1 Общие положения игры и правила проведения турнира . . . 2
1.2 Описание игрового мира . . . 4
1.3 Форма арены . . . 5
1.4 Мяч и роботы . . . 7
1.5 Нитро . . . 7
1.6 Управление . . . 8

2 Симулятор **9**
2.1 Определение ближайшей точки арены . . . 11
2.2 Константы . . . 16

3 Описание API **18**
3.1 MyStrategy . . . 18
3.2 Action . . . 18
3.3 Arena . . . 19
3.4 Ball . . . 19
3.5 Game . . . 19
3.6 NitroPack . . . 19
3.7 Player . . . 20
3.8 Robot . . . 20
3.9 Rules . . . 20

Глава 1

О мире CodeBall 2018

1.1 Общие положения игры и правила проведения турнира

Данное соревнование предоставляет вам возможность проверить свои навыки программирования, создав искусственный интеллект (стратегию), управляющий командой роботов в специальном игровом мире (подробнее об особенностях мира CodeBall 2018 можно узнать в следующих пунктах этой главы). В зависимости от этапа соревнования у вас в команде будет от 2 до 3 роботов, а также может быть доступно либо недоступно использование нитро. Все роботы одинаковы по параметрам и являются шарами, исходное расположение гарантированно симметричное. Помимо самих роботов в игре также есть мяч.

В каждой игре вам будет противостоять стратегия другого игрока. Цель вышей команды — забрасывать мяч в ворота противника и мешать попаданию мяча в свои ворота. Команда, забившая больше голов, объявляется победителем. Игра может закончиться и ничьей, если обе команды забили одинаковое количество голов.

Турнир проводится в несколько этапов, которым предшествует квалификация в Песочнице. Песочница — соревнование, которое проходит на протяжении всего чемпионата. В рамках каждого этапа игроку соответствует некоторое значение рейтинга — показателя того, насколько успешно его стратегия участвует в играх.

Начальное значение рейтинга в Песочнице равно 1200. По итогам игры это значение может как увеличиться, так и уменьшиться. При этом победа над слабым (с низким рейтингом) противником даёт небольшой прирост, также и поражение от сильного соперника незначительно уменьшает ваш рейтинг. Со временем рейтинг в Песочнице становится всё более инертным, что позволяет уменьшить влияние случайных длинных серий побед или поражений на место участника, однако вместе с тем и затрудняет изменение его положения при существенном улучшении стратегии. Для отмены данного эффекта участник может сбросить изменчивость рейтинга до начального состояния при отправке новой стратегии, включив соответствующую опцию. В случае принятия новой стратегии системой рейтинг участника сильно упадёт после следующей игры в Песочнице, однако по мере дальнейшего участия в играх быстро восстановится и даже станет выше, если ваша стратегия действительно стала эффективнее. Не рекомендуется использовать данную опцию при незначительных, инкрементальных улучшениях вашей стратегии, а также в случаях, когда новая стратегия недостаточно протестирована и эффект от изменений в ней достоверно не известен.

Начальное значение рейтинга на каждом основном этапе турнира равно 0. За каждую игру участник получает определённое количество единиц рейтинга в зависимости от занятого места Если два или более участников делят какое-то место, то суммарное количество единиц рейтинга за это место и за следующие `количество_таких_участников` − 1 мест делится поровну между этими участниками. Например, если два участника делят первое место, то каждый из них получит половину от суммы

единиц рейтинга за первое и второе места. При делении округление всегда совершается в меньшую сторону. Более подробная информация об этапах турнира будет предоставлена в анонсах на сайте проекта.

Сначала все участники могут участвовать только в играх, проходящих в Песочнице. Игроки могут отправлять в Песочницу свои стратегии, и последняя принятая из них берётся системой для участия в квалификационных играх. Каждый игрок участвует примерно в одной квалификационной игре за час. Жюри оставляет за собой право изменить этот интервал, исходя из пропускной способности тестирующей системы, однако для большинства участников он остаётся постоянной величиной. Существует ряд критериев, по которым интервал участия в квалификационных играх может быть увеличен для конкретного игрока. За каждую N-ю полную неделю, прошедшую с момента отправки игроком последней стратегии, интервал участия для этого игрока увеличивается на N базовых интервалов тестирования. Учитываются только принятые системой стратегии. За каждое «падение» стратегии в 10 последних играх в Песочнице начисляется дополнительный штраф, равный 20% от базового интервала тестирования. Подробнее о причинах «падения» стратегии можно узнать в следующих разделах. Интервал участия игрока в Песочнице не может стать больше суток.

Игры в Песочнице проходят по набору правил, соответствующему правилам случайного прошедшего этапа турнира или же правилам следующего (текущего) этапа. При этом чем ближе значение рейтинга двух игроков в рамках Песочницы, тем больше вероятность того, что они окажутся в одной игре. Песочница стартует до начала первого этапа турнира и завершается через некоторое время после финального (смотрите расписание этапов для уточнения подробностей). Помимо этого Песочница замораживается на время проведения этапов турнира. По итогам игр в Песочнице происходит отбор для участия в Раунде 1, в который попадут 1080 участников с наибольшим рейтингом на момент начала этого этапа турнира (при равенстве рейтинга приоритет отдаётся игроку, раньше отправившему последнюю версию своей стратегии), а также дополнительный набор в следующие этапы турнира, включая Финал.

Этапы турнира:

- В **Раунде 1** вам предстоит изучить правила игры и освоить базовое управление роботами. В начале игры вам даётся 2 робота, как и вашему оппоненту. Задача — забить больше голов! Всё просто. Раунд 1, как и все последующие этапы, состоит из двух частей, между которыми будет небольшой перерыв (с возобновлением работы Песочницы), который позволит улучшить свою стратегию. Для игр в каждой части выбирается последняя стратегия, отправленная игроком до начала этой части. Игры проводятся волнами. В каждой волне каждый игрок участвует ровно в одной игре. Количество волн в каждой части определяется возможностями тестирующей системы, но гарантируется, что оно не будет меньше десяти. 300 участников с наиболее высоким рейтингом пройдут в Раунд 2. Также в Раунд 2 будет проведён дополнительный набор 60 участников с наибольшим рейтингом в Песочнице (на момент начала Раунда 2) из числа тех, кто не прошёл по итогам Раунда 1.
- В **Раунде 2** вам предстоит улучшить свои навыки управления управления роботами. Теперь ваши роботы смогут использовать нитро. Также на карте появятся рюкзаки, пополняющие запас нитро у роботов. Дополнительно усложняет задачу то, что после подведения итогов Раунда 1 часть слабых стратегий будет отсеяна, и вам придётся противостоять более сильным соперникам. По итогам Раунда 2 лучшие 50 стратегий попадут в Финал. Также в Финал будет проведен дополнительный набор 10 участников с наибольшим рейтингом в Песочнице (на момент начала Финала) из числа тех, кто не прошёл в рамках основного турнира.
- **Финал** является самым серьёзным этапом. После отбора, проведённого по итогам двух первых этапов, останутся сильнейшие. В Финале у каждой команды уже будет не по 2 робота, а по 3. Система проведения Финала имеет свои особенности. Этап по-прежнему делится на две части, однако они уже не будут состоять из волн. В каждой части этапа будут проведены игры между всеми парами участников Финала. Если позволит время и возможности тестирующей системы, операция будет повторена.

После окончания Финала все финалисты упорядочиваются по невозрастанию рейтинга. При

равенстве рейтингов более высокое место занимает тот финалист, чья участвовавшая в Финале стратегия была отослана раньше. Призы за Финал распределяются на основании занятого места после этого упорядочивания.

После окончания Песочницы все её участники, кроме призёров Финала, упорядочиваются по невозрастанию рейтинга. При равенстве рейтингов более высокое место занимает тот участник, который раньше отослал последнюю версию своей стратегии. Призы за Песочницу распределяются на основании занятого места после этого упорядочивания.

1.2 Описание игрового мира

Игровой мир является трёхмерным, и ограничен ареной, являющейся коробкой с закругленными краями и воротами.

Все роботы, а также мяч имеют форму шара. Ни роботы, ни мяч не могут полностью или частично находиться вне арены. Ось X направлена горизонтально, парралельно середине арены, ось Y — вертикально, снизу вверх, ось Z — горизонтально, от ваших ворот к воротам противника.

В начале каждой игры, а также после каждого забитого гола, роботы размещаются на своей половине поля. При этом координаты для вашей стратегии преобразуются таким образом, что ваша стратегия всегда «думает», что ваша половина поля находиться в полупространстве с координатой $z < 0$. роботы расположены случайным образом на полу арены, на одинаковом расстоянии от мяча. При этом расположение ваших роботов симметрично расположению роботов противника относительно центра арены (точки $(0, 0)$). Мяч в этот момент находится в точке $(0, y, 0)$, где y — равновероятно выбирается от `BALL_RADIUS` до $4 \times$ `BALL_RADIUS`.

Время в игре дискретное и измеряется в «тиках». В начале каждого тика симулятор игры передаёт стратегиям участников данные о состоянии мира, получает от них управляющие сигналы и обновляет состояние мира в соответствии с этими сигналами и ограничениями мира. Затем происходит расчёт изменения объектов в мире за этот тик, и процесс повторяется снова с обновлёнными данными. Максимальная длительность любой игры равна 20000 тиков, однако игра может быть прекращена досрочно, если все стратегии «упали».

«Упавшая» стратегия больше не может управлять роботами. Стратегия считается «упавшей» в следующих случаях:

- Процесс, в котором запущена стратегия, непредвиденно завершился, либо произошла ошибка в протоколе взаимодействия между стратегией и игровым сервером.
- Стратегия превысила одно (любое) из отведённых ей ограничений по времени. Стратегии на один тик выделяется не более 20 секунд реального времени. Но в сумме на всю игру процессу стратегии выделяется

$$20 \times <\textit{длительность\_игры\_в\_тиках}> + 20000 \tag{1.1}$$

  миллисекунд реального времени. В формуле учитывается максимальная длительность игры. Ограничение по времени остаётся прежним, даже если реальная длительность игры отличается от этого значения. Все ограничения по времени распространяются не только на код участника, но и на взаимодействие клиента-оболочки стратегии с игровым симулятором.
- Стратегия превысила ограничение по памяти. В любой момент времени процесс стратегии не должен потреблять более 256 Мб оперативной памяти.

1.3 Форма арены

Арена является коробкой с закругленными углами, и воротами, внутри которых также могут находиться мяч и роботы.

Размеры арены доступны в качестве объекта `Arena`, находящегося в поле `arena` объекта `Rules` (подробнее в 3).

Схематичное изображение арены:

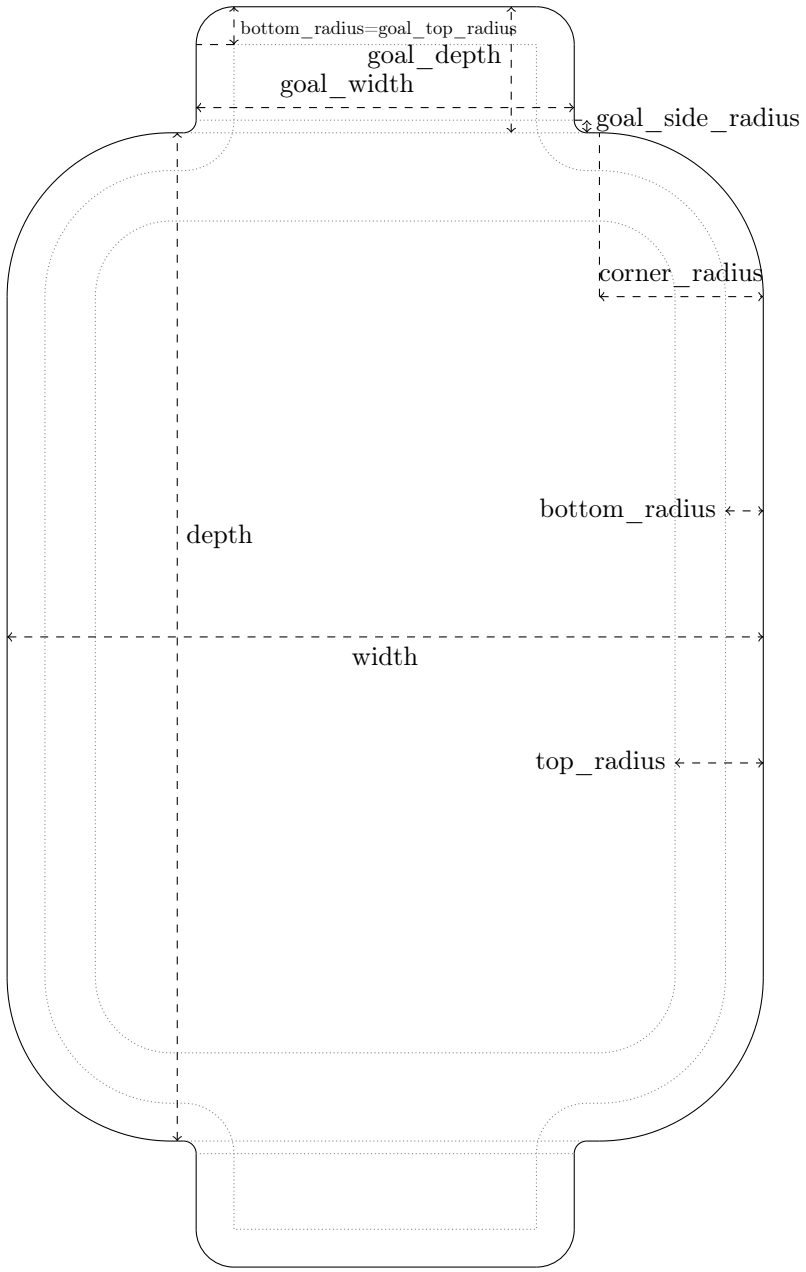

Рис. 1.1: Вид сверху

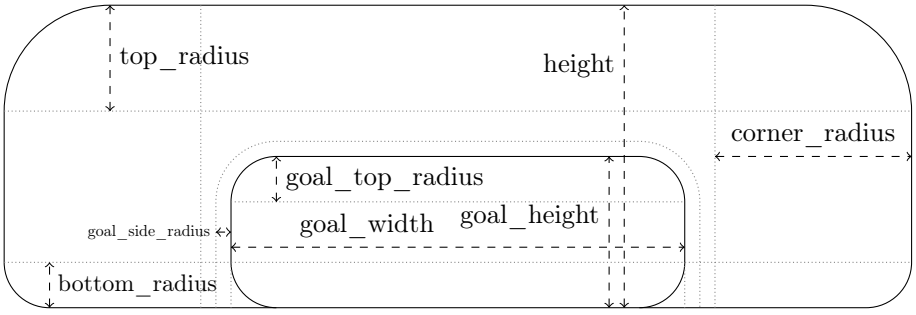

Рис. 1.2: Вид на ворота

1.4 Мяч и роботы

Мяч является шаром с радиусом BALL_RADIUS и массой BALL_MASS. Коэффициент восстановления[1] при столкновении мяча с ареной равен BALL_ARENA_E, таким образом мяч теряет часть своей скорости при столкновении.

В начале симуляции мяч находится в центре арены на случайной высоте в отрезке [BALL_RADIUS..4 × BALL_RADIUS].

Мяч считается забитым, если полностью находится за линией ворот:

$$abs(ball.z) > arena.depth/2 + ball.radius \tag{1.2}$$

После забитого гола симуляция начинается заново через RESET_TICKS тиков.

Роботы также являются шарами (массой ROBOT_MASS), но их радиус может меняться от ROBOT_MIN_RADIUS до ROBOT_MAX_RADIUS, и изменяется при использовании прыжка. Чем выше установлена скорость прыжка робота, тем больше его радиус. Точная формула:

$$\begin{aligned} &radius = \texttt{ROBOT\_MIN\_RADIUS} + \\ &(\texttt{ROBOT\_MAX\_RADIUS} - \texttt{ROBOT\_MIN\_RADIUS}) \times \frac{jump\_speed}{\texttt{ROBOT\_MAX\_JUMP\_SPEED}} \end{aligned} \tag{1.3}$$

Небольшое увеличение радиуса при прыжке дает роботам возможность отпрыгивать от стен арены.

Также, при прыжке, физический симулятор считает, что радиус робота изменяется с заданной скоростью прыжка (несмотря на то, что радиус на самом деле вычисляется по формуле, представленной выше). За счет этого и происходит сам прыжок. Таким образом, помимо возможности подпрыгивать, скорость прыжка также влияет на силу удара по мячу.

Коэффициент восстановления при столкновении робота с ареной равен 0, то есть этот вид столкновения является абсолютно неупругим. При столкновении робота с мячом, или с другим роботом, коэффициент восстановления выбирается равновероятно от MIN_HIT_E до MAX_HIT_E.

1.5 Нитро

Начиная со второго раунда, роботам будет доступно нитро. Нитро позволяет придать ускорения роботу, в любом направлении. Максимальный запас нитро равен MAX_NITRO_AMOUNT, одна единица нитро меняет скорость на NITRO_POINT_VELOCITY_CHANGE. Ускорение при использовании нитро не может превышать ROBOT_NITRO_ACCELERATION.

Пополнять запас нитро можно подбирая рюкзаки с нитро. Рюкзаки являются шарами радиуса NITRO_PACK_RADIUS и считаются подобранными, если расстояние между центром робота и центром рюкзака меньше суммы их радиусов. Каждый рюкзак с нитро восстанавливает запас полностью (до 100). После подбирания, рюкзак снова появится в том же месте через NITRO_RESPAWN_TICKS тиков. Всего на арене появляется 4 рюкзака с нитро, в точках с координатами $(\pm\ x, 1, \pm z)$.

В начале игры и после каждого гола запас нитро всех роботов равен START_NITRO_AMOUNT.

В первом раунде запас нитро всегда равен 0, и рюкзаки с нитро на арене не появляются.

---

[1] Коэффициент восстановления — коэффициент, задающий долю относительной скорости объектов при столкновении, которая восстанавливается после удара

1.6 Управление

Управление роботом состоит в задании желаемого вектора скорости, скорости прыжка, и желании использовать нитро.

Не считая нитро, если робот соприкасается с ареной, то его скорость будет стремится к желаемой скорости, спроецированной на плоскость соприкосновения. Ускорение не может превышать `ROBOT_ACCELERATION`. При этом, чем более вертикальная поверхность, тем меньше ускорение в желаемую сторону (также, если робот касается арены своей верхней половиной, ускорение равно нулю). То есть, при соприкосновении с полом, ускорение максимальное, а при соприкосновении со стеной или потолком, нулевое.

При использовании нитро, скорость также стремится к желаемой, однако уже независимо от соприкосновения с ареной.

Скорость прыжка можно задать любую от 0 (не прыгать) до `ROBOT_MAX_JUMP_SPEED` (прыжок с максимальной скоростью). При прыжке радиус робота немного увеличивается, позволяя ему отпрыгивать от стен. Также при прыжке физический симулятор считает, что радиус робота изменяется со скоростью, равной заданной скорости прыжка. То есть, прыжок осуществляется за счет того, что при столкновении с полом относительная скорость в точке соприкосновения равна скорости прыжка. Таким образом, также можно контролировать силу удара по мячу.

Глава 2

Симулятор

Симуляция в CodeBall основана на простой физической модели, использующей метод импульсов. В системе отсутствует какое либо трение, все столкновения разрешаются последовательно в случайном порядке.

Псевдо-код симулятора:

```
function collide_entities(a: Entity, b: Entity):
    let delta_position = b.position - a.position
    let distance = length(delta_position)
    let penetration = a.radius + b.radius - distance
    if penetration > 0:
        let k_a = (1 / a.mass) / ((1 / a.mass) + (1 / b.mass))
        let k_b = (1 / b.mass) / ((1 / a.mass) + (1 / b.mass))
        let normal = normalize(delta_position)
        a.position -= normal * penetration * k_a
        b.position += normal * penetration * k_b
        let delta_velocity = dot(b.velocity - a.velocity, normal)
            + b.radius_change_speed - a.radius_change_speed
        if delta_velocity < 0:
            let impulse = (1 + random(MIN_HIT_E, MAX_HIT_E)) * delta_velocity * normal
            a.velocity += impulse * k_a
            b.velocity -= impulse * k_b

function collide_with_arena(e: Entity):
    let distance, normal = dan_to_arena(e.position)
    let penetration = e.radius - distance
    if penetration > 0:
        e.position += penetration * normal
        let velocity = dot(e.velocity, normal) - e.radius_change_speed
        if velocity < 0:
            e.velocity -= (1 + e.arena_e) * velocity * normal
            return normal
    return None

function move(e: Entity):
    e.velocity = clamp(e.velocity, MAX_ENTITY_SPEED)
    e.position += e.velocity * delta_time
    e.position.y -= GRAVITY * delta_time * delta_time / 2
    e.velocity.y -= GRAVITY * delta_time
```

```
function update(delta_time: float):
    shuffle(robots)

    for robot in robots:
        if robot.touch:
            let target_velocity = clamp(
                robot.action.target_velocity,
                ROBOT_MAX_GROUND_SPEED)
            target_velocity -= robot.touch_normal
                * dot(robot.touch_normal, target_velocity)
            let target_velocity_change = target_velocity - robot.velocity
            if length(target_velocity_change) > 0:
                let acceleration = ROBOT_ACCELERATION * max(0, robot.touch_normal.y)
                robot.velocity += clamp(
                    normalize(target_velocity_change) * acceleration * delta_time,
                    length(target_velocity_change))

        if robot.action.use_nitro:
            let target_velocity_change = clamp(
                robot.action.target_velocity - robot.velocity,
                robot.nitro * NITRO_POINT_VELOCITY_CHANGE)
            if length(target_velocity_change) > 0:
                let acceleration = normalize(target_velocity_change)
                    * ROBOT_NITRO_ACCELERATION
                let velocity_change = clamp(
                    acceleration * delta_time,
                    length(target_velocity_change))
                robot.velocity += velocity_change
                robot.nitro -= length(velocity_change)
                    / NITRO_POINT_VELOCITY_CHANGE

        move(robot)
        robot.radius = ROBOT_MIN_RADIUS + (ROBOT_MAX_RADIUS - ROBOT_MIN_RADIUS)
                    * robot.action.jump_speed / ROBOT_MAX_JUMP_SPEED
        robot.radius_change_speed = robot.action.jump_speed

    move(ball)

    for i in 0 .. length(robots) - 1:
        for j in 0 .. i - 1:
            collide_entities(robots[i], robots[j])

    for robot in robots:
        collide_entities(robot, ball)
        collision_normal = collide_with_arena(robot)
        if collision_normal is None:
            robot.touch = false
        else:
            robot.touch = true
            robot.touch_normal = collision_normal
    collide_with_arena(ball)

    if abs(ball.position.z) > arena.depth / 2 + ball.radius:
        goal_scored()
```

```
    for robot in robots:
        if robot.nitro == MAX_NITRO_AMOUNT:
            continue
        for pack in nitro_packs:
            if not pack.alive:
                continue
            if length(robot.position - pack.position) <= robot.radius + pack.radius:
                robot.nitro = MAX_NITRO_AMOUNT
                pack.alive = false
                pack.respawn_ticks = NITRO_PACK_RESPAWN_TICKS

function tick():
    let delta_time = 1 / TICKS_PER_SECOND
    for _ in 0 .. MICROTICKS_PER_TICK - 1:
        update(delta_time / MICROTICKS_PER_TICK)

    for pack in nitro_packs:
        if pack.alive:
            continue
        pack.respawn_ticks -= 1
        if pack.respawn_ticks == 0:
            pack.alive = true
```

2.1 Определение ближайшей точки арены

```
function dan_to_plane(point: Vec3D, point_on_plane: Vec3D, plane_normal: Vec3D):
    return {
        distance: dot(point - point_on_plane, plane_normal)
        normal: plane_normal
    }

function dan_to_sphere_inner(point: Vec3D, sphere_center: Vec3D, sphere_radius: Float):
    return {
        distance: sphere_radius - length(point - sphere_center)
        normal: normalize(sphere_center - point)
    }

function dan_to_sphere_outer(point: Vec3D, sphere_center: Vec3D, sphere_radius: Float):
    return {
        distance: length(point - sphere_center) - sphere_radius
        normal: normalize(point - sphere_center)
    }
}

function dan_to_arena_quarter(point: Vec3D):
    // Ground
    let dan = dan_to_plane(point, (0, 0, 0), (0, 1, 0))

    // Ceiling
    dan = min(dan, dan_to_plane(point, (0, arena.height, 0), (0, -1, 0)))

    // Side x
    dan = min(dan, dan_to_plane(point, (arena.width / 2, 0, 0), (-1, 0, 0)))
```

```
// Side z (goal)
dan = min(dan, dan_to_plane(
        point,
        (0, 0, (arena.depth / 2) + arena.goal_depth),
        (0, 0, -1)))

// Side z
let v = (point.x, point.y) - (
        (arena.goal_width / 2) - arena.goal_top_radius,
        arena.goal_height - arena.goal_top_radius)
if point.x >= (arena.goal_width / 2) + arena.goal_side_radius
        or point.y >= arena.goal_height + arena.goal_side_radius
        or (
            v.x > 0
            and v.y > 0
            and length(v) >= arena.goal_top_radius + arena.goal_side_radius):
    dan = min(dan, dan_to_plane(point, (0, 0, arena.depth / 2), (0, 0, -1)))

// Side x & ceiling (goal)
if point.z >= (arena.depth / 2) + arena.goal_side_radius:
    // x
    dan = min(dan, dan_to_plane(
            point,
            (arena.goal_width / 2, 0, 0),
            (-1, 0, 0)))
    // y
    dan = min(dan, dan_to_plane(point, (0, arena.goal_height, 0), (0, -1, 0)))

// Goal back corners
assert arena.bottom_radius == arena.goal_top_radius
if point.z > (arena.depth / 2) + arena.goal_depth - arena.bottom_radius:
    dan = min(dan, dan_to_sphere_inner(
            point,
            (
                clamp(
                    point.x,
                    arena.bottom_radius - (arena.goal_width / 2),
                    (arena.goal_width / 2) - arena.bottom_radius,
                ),
                clamp(
                    point.y,
                    arena.bottom_radius,
                    arena.goal_height - arena.goal_top_radius,
                ),
                (arena.depth / 2) + arena.goal_depth - arena.bottom_radius),
            arena.bottom_radius))

// Corner
if point.x > (arena.width / 2) - arena.corner_radius
        and point.z > (arena.depth / 2) - arena.corner_radius:
    dan = min(dan, dan_to_sphere_inner(
            point,
            (
                (arena.width / 2) - arena.corner_radius,
                point.y,
                (arena.depth / 2) - arena.corner_radius
```

```
            ),
            arena.corner_radius))

    // Goal outer corner
    if point.z < (arena.depth / 2) + arena.goal_side_radius:
        // Side x
        if point.x < (arena.goal_width / 2) + arena.goal_side_radius:
            dan = min(dan, dan_to_sphere_outer(
                    point,
                    (
                        (arena.goal_width / 2) + arena.goal_side_radius,
                        point.y,
                        (arena.depth / 2) + arena.goal_side_radius
                    ),
                    arena.goal_side_radius))
        // Ceiling
        if point.y < arena.goal_height + arena.goal_side_radius:
            dan = min(dan, dan_to_sphere_outer(
                    point,
                    (
                        point.x,
                        arena.goal_height + arena.goal_side_radius,
                        (arena.depth / 2) + arena.goal_side_radius
                    ),
                    arena.goal_side_radius))
        // Top corner
        let o = (
            (arena.goal_width / 2) - arena.goal_top_radius,
            arena.goal_height - arena.goal_top_radius
        )
        let v = (point.x, point.y) - o
        if v.x > 0 and v.y > 0:
            let o = o + normalize(v) * (arena.goal_top_radius + arena.goal_side_radius)
            dan = min(dan, dan_to_sphere_outer(
                    point,
                    (o.x, o.y, (arena.depth / 2) + arena.goal_side_radius),
                    arena.goal_side_radius))

    // Goal inside top corners
    if point.z > (arena.depth / 2) + arena.goal_side_radius
            and point.y > arena.goal_height - arena.goal_top_radius:
        // Side x
        if point.x > (arena.goal_width / 2) - arena.goal_top_radius:
            dan = min(dan, dan_to_sphere_inner(
                    point,
                    (
                        (arena.goal_width / 2) - arena.goal_top_radius,
                        arena.goal_height - arena.goal_top_radius,
                        point.z
                    ),
                    arena.goal_top_radius))
        // Side z
        if point.z > (arena.depth / 2) + arena.goal_depth - arena.goal_top_radius:
            dan = min(dan, dan_to_sphere_inner(
                    point,
                    (
```

```
                    point.x,
                    arena.goal_height - arena.goal_top_radius,
                    (arena.depth / 2) + arena.goal_depth - arena.goal_top_radius
                ),
                arena.goal_top_radius))

// Bottom corners
if point.y < arena.bottom_radius:
    // Side x
    if point.x > (arena.width / 2) - arena.bottom_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    (arena.width / 2) - arena.bottom_radius,
                    arena.bottom_radius,
                    point.z
                ),
                arena.bottom_radius))
    // Side z
    if point.z > (arena.depth / 2) - arena.bottom_radius
            and point.x >= (arena.goal_width / 2) + arena.goal_side_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    point.x,
                    arena.bottom_radius,
                    (arena.depth / 2) - arena.bottom_radius
                ),
                arena.bottom_radius))
    // Side z (goal)
    if point.z > (arena.depth / 2) + arena.goal_depth - arena.bottom_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    point.x,
                    arena.bottom_radius,
                    (arena.depth / 2) + arena.goal_depth - arena.bottom_radius
                ),
                arena.bottom_radius))
    // Goal outer corner
    let o = (
        (arena.goal_width / 2) + arena.goal_side_radius,
        (arena.depth / 2) + arena.goal_side_radius
    )
    let v = (point.x, point.z) - o
    if v.x < 0 and v.y < 0
            and length(v) < arena.goal_side_radius + arena.bottom_radius:
        let o = o + normalize(v) * (arena.goal_side_radius + arena.bottom_radius)
        dan = min(dan, dan_to_sphere_inner(
                point,
                (o.x, arena.bottom_radius, o.y),
                arena.bottom_radius))
    // Side x (goal)
    if point.z >= (arena.depth / 2) + arena.goal_side_radius
            and point.x > (arena.goal_width / 2) - arena.bottom_radius:
        dan = min(dan, dan_to_sphere_inner(
```

```
                point,
                (
                    (arena.goal_width / 2) - arena.bottom_radius,
                    arena.bottom_radius,
                    point.z
                ),
                arena.bottom_radius))
    // Corner
    if point.x > (arena.width / 2) - arena.corner_radius
            and point.z > (arena.depth / 2) - arena.corner_radius:
        let corner_o = (
            (arena.width / 2) - arena.corner_radius,
            (arena.depth / 2) - arena.corner_radius
        )
        let n = (point.x, point.z) - corner_o
        let dist = n.len()
        if dist > arena.corner_radius - arena.bottom_radius:
            let n = n / dist
            let o2 = corner_o + n * (arena.corner_radius - arena.bottom_radius)
            dan = min(dan, dan_to_sphere_inner(
                    point,
                    (o2.x, arena.bottom_radius, o2.y),
                    arena.bottom_radius))

// Ceiling corners
if point.y > arena.height - arena.top_radius:
    // Side x
    if point.x > (arena.width / 2) - arena.top_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    (arena.width / 2) - arena.top_radius,
                    arena.height - arena.top_radius,
                    point.z,
                ),
                arena.top_radius))
    // Side z
    if point.z > (arena.depth / 2) - arena.top_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    point.x,
                    arena.height - arena.top_radius,
                    (arena.depth / 2) - arena.top_radius,
                )
                arena.top_radius))

    // Corner
    if point.x > (arena.width / 2) - arena.corner_radius
            and point.z > (arena.depth / 2) - arena.corner_radius:
        let corner_o = (
            (arena.width / 2) - arena.corner_radius,
            (arena.depth / 2) - arena.corner_radius
        )
        let dv = (point.x, point.z) - corner_o
        if length(dv) > arena.corner_radius - arena.top_radius:
```

```
                let n = normalize(dv)
                let o2 = corner_o + n * (arena.corner_radius - arena.top_radius)
                dan = min(dan, dan_to_sphere_inner(
                        point,
                        (o2.x, arena.height - arena.top_radius, o2.y),
                        arena.top_radius))

    return dan
}

function dan_to_arena(point: Vec3D):
    let negate_x = point.x < 0
    let negate_z = point.z < 0
    if negate_x:
        point.x = -point.x
    if negate_z:
        point.z = -point.z
    let result = dan_to_arena_quarter(point)
    if negate_x:
        result.normal.x = -result.normal.x
    if negate_z:
        result.normal.z = -result.normal.z
    return result
```

2.2 Константы

Обратите внимание, что константы скоростей и ускорений даны в системе $\frac{\text{единица расстояния}}{\text{секунда}}$ и $\frac{\text{единица расстояния}}{\text{секунда}^2}$.

```
ROBOT_MIN_RADIUS = 1
ROBOT_MAX_RADIUS = 1.05
ROBOT_MAX_JUMP_SPEED = 15
ROBOT_ACCELERATION = 100
ROBOT_NITRO_ACCELERATION = 30
ROBOT_MAX_GROUND_SPEED = 30
ROBOT_ARENA_E = 0
ROBOT_RADIUS = 1
ROBOT_MASS = 2
TICKS_PER_SECOND = 60
MICROTICKS_PER_TICK = 100
RESET_TICKS = 2 * TICKS_PER_SECOND
BALL_ARENA_E = 0.7
BALL_RADIUS = 2
BALL_MASS = 1
MIN_HIT_E = 0.4
MAX_HIT_E = 0.5
MAX_ENTITY_SPEED = 100
MAX_NITRO_AMOUNT = 100
START_NITRO_AMOUNT = 50
NITRO_POINT_VELOCITY_CHANGE = 0.6
NITRO_PACK_X = 20
NITRO_PACK_Y = 1
NITRO_PACK_Z = 30
NITRO_PACK_RADIUS = 0.5
```

```
NITRO_PACK_AMOUNT = 100
NITRO_RESPAWN_TICKS = 10 * TICKS_PER_SECOND
GRAVITY = 30
```

Глава 3

Описание API

3.1 MyStrategy

В языковом пакете для вашего языка программирования находится файл с именем `MyStrategy/my_strategy.<ext>`. В нем находится класс `MyStrategy`, в котором содержится метод `act`, содержащий логику вашей стратегии.

Этод метод будет вызываться каждый тик, для каждого вашего робота отдельно.

```
class MyStrategy:
      method act(me: Robot, rules: Rules, game: Game, action: Action):
            // Your implementation
```

Параметры в методе `act`:

- `me` — ваш робот, действие которого сейчас определяется
- `rules` — объект, содержащий правила игры (не меняется между тиками)
- `game` — объект, содержащий информацию о текущем состоянии игры (меняется между тиками)
- `action` — объект, задающий действие вашего робота. Меняя поля этого объекта, вы задаете действие

3.2 Action

Объект, определяющий действие робота.

```
class Action:
      target_velocity_x: Float      // Координата X желаемого вектора скорости
      target_velocity_y: Float      // Координата Y желаемого вектора скорости
      target_velocity_z: Float      // Координата Z желаемого вектора скорости
      jump_speed: Float             // Желаемая скорость прыжка
      use_nitro: Bool               // Использовать/не использовать нитро
```

3.3 Arena

Объект, определяющий размеры арены (см. 1.3).

```
class Arena:
      width: Float
      height: Float
      depth: Float
      bottom_radius: Float
      top_radius: Float
      corner_radius: Float
      goal_top_radius: Float
      goal_width: Float
      goal_height: Float
      goal_depth: Float
      goal_side_radius: Float
```

3.4 Ball

Объект, определяющий мяч.

```
class Ball:
      x: Float                      // Текущие координаты центра мяча
      y: Float
      z: Float
      velocity_x: Float             // Текущая скорость мяча
      velocity_y: Float
      velocity_z: Float
      radius: Float                 // Радиус мяча
```

3.5 Game

Объект, содержащий себе информацию о текущем состоянии игры.

```
class Game:
      current_tick: Int             // Номер текущего тика
      players: Player[]             // Список игроков
      robots: Robot[]               // Список роботов
      nitro_packs: NitroPack[]      // Список рюкзаков с нитро
      ball: Ball                    // Мяч
```

3.6 NitroPack

Объект, определяющий рюкзак с нитро.

```
class NitroPack:
      id: Int
      x: Float                      // Координаты центра рюкзака
```

```
      y: Float
      z: Float
      radius: Float                // Радиус
      respawn_ticks: Int?          // Кол-во тиков, через которое рюкзак снова появится
                                         (или null, если еще не был подобран)
```

3.7 Player

Объект, определяющий игрока (стратегию).

```
class Player:
      id: Int
      me: Bool                     // true, если это оъект вашего игрока
      strategy_crashed: Bool
      score: Int                   // Текущий счет игрока
```

3.8 Robot

Объект, определяющий робота.

```
class Robot:
      id: Int
      player_id: Int
      is_teammate: Bool            // true, если это робот вашего игрока
      x: Float                     // Текущие координаты центра робота
      y: Float
      z: Float
      velocity_x: Float            // Текущая скорость робота
      velocity_y: Float
      velocity_z: Float
      radius: Float                // Текущий радиус робота
      nitro_amount: Float          // Текущий запас нитро робота
      touch: bool                  // true, если робот сейчас касается арены
      touch_normal_x: Float        // Вектор нормали к точке соприкосновения с ареной
      touch_normal_y: Float              (или null, если нет касания)
      touch_normal_z: Float
```

3.9 Rules

Объект, задающий данные о мире, не меняющиеся между тиками.

```
class Rules:
      max_tick_count: Int          // Максимальная продолжительность игры
      arena: Arena                 // Описание размеров арены
```