Санкт – Петербургский государственный университет информационных технологий, механики и оптики.

Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Южаков Евгений Михайлович

Построение автономного виртуального робота на основе автоматного подхода

(на примере игры «CodeRally», предложенной на Java Challenge туре чемпионата мира по программированию по версии ACM 2003 г.).

БАКАЛАВРСКАЯ РАБОТА

Научный руководитель - докт. техн. наук, профессор Шалыто А.А.

Санкт – Петербург

2003

Введение

На командном чемпионате мира по программированию по версии *ACM* (*Association for Computing Machinery*) [1] в качестве тренировочного тура участникам традиционно предлагается разработать программу управления роботом в некоторой виртуальной среде. При этом ежегодно изменяется как сам робот, так и «среда его обитания». Язык *Java* [2] является единственным допустимым языком написания программы. После написания программы, созданные роботы соревнуются друг с другом. Поэтому тур называется «*Java Challenge*».

Каждой команде предоставляется эмулятор, осуществляющий обсчет игровой ситуации и визуализацию текущего состояния «игрового мира». Так же в состав этого эмулятора входят элементарные функции для управления роботом. Существующий набор функций дает участникам полный контроль над роботом в рамках установленных правил и предоставляет возможность наблюдать за поведением остальных объектов на поле.

При этом решаемой задачей является написание наиболее совершенной системы управления роботом (создание его искусственного интеллекта). Робот является виртуальным и автономным. Под автономностью понимается, что в процессе соревнований роботом управляет **только** его искусственный интеллект (после начала соревнований автор робота не имеет возможности воздействовать на него).

Для алгоритмизации и программирования верхнего уровня программы управления роботом представляется целесообразным использовать подход, основанный на SWITCH-технологии [3]. Эта технология базируется на применении конечных автоматов и может быть эффективно использована

для решения данной задачи, так как в ней естественным образом выделяются состояния объекта управления (робота).

Данная работа содержит анализ задачи *CodeRally* (*Java Challenge 2003*) [4] и одного из ее решений. Его особенностями, в отличие от предложенного нашей командой (SPb IFMO) на чемпионате мира, являются использование автоматного подхода и учет внутренних свойств среды обитания робота (виртуального мира).

Эффективность работы предложенного робота продемонстрирована в ходе соревнований с роботами, входящими в комплект поставки эмулятора, и с роботом, созданным нами на чемпионате мира.

Работа выполнена в рамках «Движения за открытую проектную документацию» [5], что хотя бы частично решает проблему, сформулированную на форуме официального сайта задачи *CodeRally*: отсутствие примеров решения задачи, а тем более проектной документации на эти роботы.

Для запуска робота необходимо иметь платформу *Eclipse* [6] и эмулятор игрового мира. Процесс их установки подробно описан в соответствующей документации.

1. Постановка задачи «CodeRally»

1.1. Формальная постановка задачи

Задача *CodeRally* состоит в написании программы управления роботом (автомобилем, который в дальнейшем будем называть «машиной») в виртуальном мире на языке программирования *Java*.

Эмулятор виртуального мира поставляется как расширение (plugin) платформы *Eclipse* [6]. После создания робот помещается на трассу эмулятора, на которой ему предстоит соревноваться с другими, ему подобными, роботами. В ходе соревнований роботы «противодействуют» друг другу.

Возможности робота ограничены интерфейсом, предоставляемым эмулятором. Тот же эмулятор обеспечивает возможность для наблюдения за процессом соревнований. Этот процесс представляет собой непрерывную генерацию управляющих действий для робота, основанную на анализе текущих, прошедших и предсказываемых игровых ситуаций.

Более подробно характеристики эмулятора описаны в следующем разделе.

1.2. Характеристики трассы

Трасса – это двумерный мир с началом координат в верхнем левом углу, имеющий 1010 единиц длины по оси X и 580 единиц по оси Y. По периметру трасса огорожена стеной, поэтому машины не могут покинуть трассу. Внутри трассы стен нет. Машины могут свободно передвигаться внутри трассы до тех пор, пока не столкнутся с другой машиной. Объекты передвигаются в направлении, называемом «курсом», который измеряется в целых градусах.

Ноль градусов – это «прямо вверх». Все курсы – положительные числа из промежутка [0, 359] и увеличиваются в направлении по часовой стрелке.

На рис. 1 приведена система координат, используемая в эмуляторе.

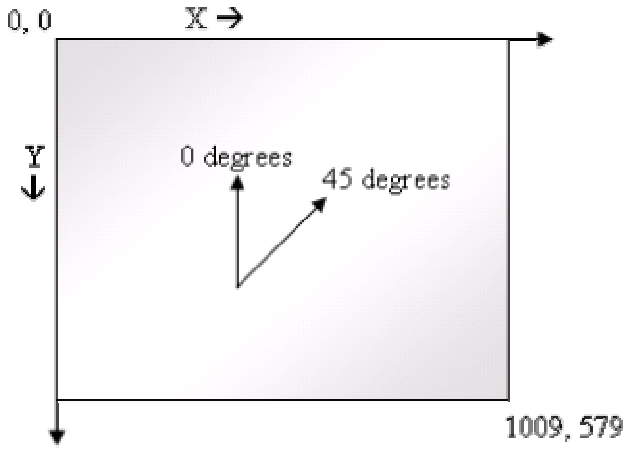

Рис. 1. Система координат, используемая в эмуляторе

Трасса (рис. 2)

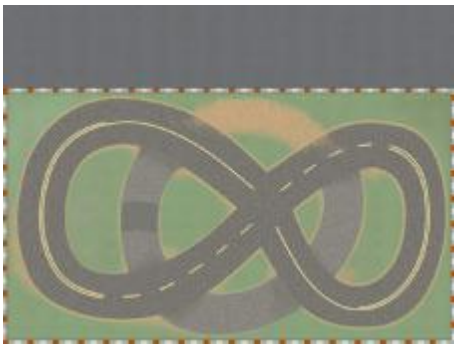

Рис. 2. Внешний вид трассы

имеет следующие характеристики:

- существуют часы, значение которых может быть получено с помощью метода **getClockTicks()**;
- каждая машина (рис. 3) начинает матч, имея 100 единиц топлива и три пули;

Рис. 3. Внешний вид машины

- установка угла поворота и ускорения заставляет машину двигаться продолжительно с этими параметрами до тех пор, пока не будет установлено новое значение соответствующего параметра. Столкновение со стеной или другой машиной не меняет установленные угол поворота и ускорение;
- машины могут изменять значения угла поворота и ускорения мгновенно. Скорость и направление движения меняются постепенно из-за инерции;
- минимальное значение ускорения машины (**MIN_THROTTLE**) – минус 50 единиц и максимальное ускорение (**MAX_THROTTLE**) - 100 единиц;
- максимальное изменение скорости с места (исключая столкновения) - 8 единиц за временной тик;
- минимальный угол поворота (**MIN_STEER_LEFT**) – минус 10 единиц, максимальный угол поворота (**MAX_STEER_RIGHT**) - 10 единиц. Угловое изменение курса зависит от текущей скорости и может быть получено с помощью метода **getChangeInHeading()**;
- длина машины – 60 единиц, ширина – 40 единиц. Положение машины на трассе – точка, находящаяся в центре этого прямоугольника;
- для простоты, круглая пуля считается попавшей в прямоугольную машину, когда расстояние от центра пули до центра машины менее 40 единиц;
- пуля, выпущенная машиной, движется с постоянной скоростью в 12 единиц за один временной тик до тех пор, пока не столкнется со стеной или другой машиной. В момент столкновения она исчезнет. Контрольные точки, заправочные станции и источники пуль не влияют на движение выпущенных пуль. Выпущенные пули не сталкиваются друг с другом;
- максимальное количество топлива, которое может иметь машина - 100 единиц;

- максимальное количество пуль, которые может иметь машина – 5;
- каждый раз, когда машина оказывается на расстоянии не более 25 единиц от заправочной станции (рис. 4), количество топлива увеличивается с постоянной скоростью – одна единица за временной тик. На трассе случайным образом расположены три заправочные станции;

Рис. 4. Заправочная станция

- если машина имеет меньше пяти пуль и оказывается на расстоянии не более 25 единиц от источника пуль (рис. 5), то каждые пять временных тиков количество пуль в машине увеличивается на одну (выполняется загрузка машины пулями). На трассе случайным образом расположены три источника пуль. Пуля и источник пуль в эмуляторе изображаются в виде шины;

Рис. 5. Источник пуль

- машина может защитить себя от столкновения с другими машинами и выпущенными пулями включением защитного режима (рис. 6). Машина в этом режиме потребляет топливо с двойной скоростью. Защитный режим длится 50 тиков;

Рис. 6. Внешний вид машины, находящейся в защитном режиме

- столкновение с другой машиной меняет векторы скорости машин между собой, и обе машины теряют по 10 единиц топлива;
- попадание пулей в другую машину приносит 10 очков. Пораженная машина теряет 10 единиц топлива и ее метод **move()** не вызывается в течение 10 тиков. Машина также приобретает дополнительную скорость в направлении, в котором летела пуля. За попадание пулей в машину, находящуюся в защитном режиме, не дается очков и на нее не оказывается влияния (топливо не уменьшается, скорость не изменяется, метод **move**() продолжает вызываться);
- машина, выпустившая пулю, теряет возможность стрельбы на 25 тиков;
- ограничение времени на выполнение одного хода – 500 миллисекунд. Следует различать тики и миллисекунды. Тик – единица времени, принятая в эмуляторе и символизирующая один ход. Время, затраченное методом **move(),** измеряется в миллисекундах и зависит от кода программы. Если метод **move()** не отработает за 500 миллисекунд, то он не будет вызван вновь в этом заезде, и для данной машины будут использоваться значения угла поворота и ускорения, выставленные в последний успешный (уложившийся в 500 миллисекунд) вызов метода **move()**;
- на трассе расположено некоторое количество контрольных точек (рис.7). Проезд (прохождение на расстоянии не более 25 единиц) через контрольную точку дает два очка. Если текущая контрольная точка имеет номер, следующий за номером последней пройденной точки, то количество полученных очков равняется шести. Проезд второй раз через одну и ту же контрольную точку не приносит очков;

Рис. 7. Контрольная точка

- за каждые 10 единиц топлива, оставшихся в «баке машины» на момент окончания матча, дается одно очко;
- очки присваиваются в соответствии с табл. 1. Суммарное количество очков по результатам матча определяет победителя.

Таблица 1

Действие	**Очки**
Проезд через контрольную точку	2
Проезд через контрольную точку в требуемом порядке	6
Попадание в машину выпущенной пулей	10
За каждые 10 единиц топлива на момент окончания матча	1

1.3. Требования к программе

1.3.1. Ограничения

Для того чтобы программа могла компилироваться и исполняться под эмулятором, она должна удовлетворять следующим ограничениям:

- весь код должен содержаться внутри класса **RallyCar**;
- запрещается определять конструкторы;
- запрещается использовать блок статической инициализации класса;
- запрещается создавать новые потоки, процессы, задания печати, файлы или использовать другие подобные системные функции;
- в классе **RallyCar** необходимо реализовать пять методов интерфейса **ICar**. Методы описаны ниже.

1.3.2. Идентификация машины

В классе **RallyCar** следует реализовать три метода для идентификации машины.

Первый метод – **getSchoolName()** должен вернуть строку, состоящую не более чем из 25 символов и задающую имя школы или университета.

Второй метод – **getName()** должен вернуть название машины – строку, состоящую не более чем из 25 символов.

Третий метод – **getColor()** должен вернуть константу типа байт, выбранную из предложенных в классе **Car**. Этот метод необходим для задания цвета машины. Метод **getColor()** по умолчанию возвращает значение **CAR_BLUE**.

1.3.3. Инициализация

Когда машина помещается на трассу, эмулятор вызывает метод **intialize()**. При необходимости, в этот метод можно поместить код, требуемый для инициализации. В момент выполнения этого метода доступно полное API (API - сокращение от «application programming interface» - программный интерфейс приложения) эмулятора. Необходимо помнить, что эмулятор предоставляет ограниченное количество времени (одну секунду) на выполнение инициализирующего кода, прежде чем он начнет заезд. Если инициализирующий код не завершит свою работу в отведенное время, то машина будет помещена на трассу в неинициализированном состоянии с непредсказуемыми результатами.

1.3.4. Управление машиной

После того, как эмулятор закончит вызовы инициализирующих методов для всех машин, он вызывает методы **move()** этих машин в последовательном порядке. Это происходит каждый временной тик. Программа внутри метода **move()** определяет действия, которые машина совершает в течение матча. Входные параметры метода **move()** дают информацию о ситуации на трассе. Кроме того, программе доступны методы для проверки состояния собственной машины, изменения угла поворота и ускорения, выяснения состояния остальных машин, определения положений объектов на трассе и стрельбы пулями. Например, в качестве объектов на трассе используются заправочные станции для подкачки топлива и источники пуль.

При этом под состоянием машины понимается ее положение на трассе, количество топлива, количество пуль и т.д.

Метод **move()** имеет четыре параметра, предоставляющие некоторую информацию о том, что произошло в игре в течение *предыдущего* цикла передвижения. Эти параметры определяют:

- сколько времени (в миллисекундах) затратил метод **move()** в предыдущем цикле;
- столкнулась ли машина со стеной в предыдущем цикле;
- столкнулась ли машина с другими машинами в предыдущем цикле;
- была ли машина в предыдущем цикле поражена пулей, выпущенной другой машиной.

Первый параметр имеет тип **int**, второй – **boolean**, третий и четвертый параметры возвращают ссылки типа **ICar** на соответствующую (с которой произошло столкновение или которая поразила пулей) машину (или **null**, если столкновения или попадания не было). Первый параметр полезен для определения опасности превышения машиной временного предела, допустимого для выполнения хода.

2. Анализ стратегии и особенностей движения

2.1. Словесное описание стратегии

По мнению автора, основным источником очков должен быть проезд через контрольные точки.

Рассмотрим причины, не позволяющие эффективно использовать такие источники очков как сохранение топлива на конец матча и атака других машин с помощью пуль.

Подсчитаем максимальное количество очков, которые можно заработать, сохраняя топливо к концу заезда. Как отмечалось выше, максимальное количество топлива равняется 100 единицам. За каждые десять единиц можно получить одно очко. Таким образом, получаем десять очков. То же самое количество очков можно заработать, поразив пулей машину противника один раз.

Из изложенного следует, что получение очков за счет накопления топлива к концу заезда неэффективно, так как практика (итоги **Java Challenge 2003**) показывает, что довольно разумные алгоритмы приносят от 100 до 200 очков за заезд, а теоретический предел находится, видимо, не очень далеко от 350 очков.

Если рассмотреть стратегию стрельбы пулями, то можно утверждать, что она не может быть основной стратегией получения очков. Специфика использования пуль становится понятной, если сравнить скорость движения выпущенной пули и среднюю скорость, развиваемую машинами в процессе соревнований. Практическое измерение средней скорости машин дает

примерное значение в 15 единиц. Теоретический предел для скорости машины (как будет показано ниже) равняется 20 единицам (без учета полученных ускорений при столкновении с объектами трассы). Таким образом, пуля, скорость которой постоянна и равняется 12 единицам, не в состоянии догнать движущуюся машину.

Еще одним минусом данной стратегии является необходимость машины быть направленной в сторону стрельбы, так как стрельба идет строго по текущему курсу машины. Следовательно, эффективно атаковать машины, которые двигаются встречным курсом и стоящие машины (машины без топлива или заправляющиеся машины) при условии, что текущий курс разрабатываемой машины совпадает с направлением на стоящую цель. Однако создание ситуаций для поражения пулей при отсутствии стоящих машин представляется довольно трудной задачей. Кроме того, умный соперник всегда может включить защитный режим при обнаружении угрозы, чтобы избежать потери топлива. Стоящая машина при использовании защитного режима вообще не тратит топлива.

Последнее, однако, можно использовать для повышения эффективности стратегии. При заправке топливом требуется стоять на месте продолжительное время, так как топливо добавляется по единице за временной тик. Следовательно, целесообразно использовать защитный режим независимо от того есть угроза или нет.

Помешать проезду контрольных точек другой машине практически невозможно, так как всегда можно объехать искусственное препятствие (машина или движущаяся пуля) или выбрать другую контрольную точку для проезда. Основной проблемой является доступ к топливу, так как одна машина может заблокировать доступ к одной заправочной станции. При

шести машинах на трассе может случиться ситуация, когда все заправочные станции будут заняты.

Итак, можно сделать вывод, что основной стратегией является езда через контрольные точки. Использование стрельбы пулями допустимо лишь как дополнительная часть стратегии. При этом даже самая эффективная стрельба может быть легко нейтрализована противником. Накопление топлива к концу заезда (как показано выше) представляется практически бесполезным. Если выбирать между ездой через последовательные контрольные точки и ездой через случайные контрольные точки, то полезнее для максимизации заработанных очков - первый вариант. Этому способствует специфика расположения контрольных точек на трассе.

Рассматривая и анализируя возможные варианты расположения контрольных точек с помощью текста программы эмулятора, приходим к выводу, что в подавляющем большинстве случаев следующая контрольная точка (по номеру) является оптимальной даже без дополнительных очков за последовательность. Создатели эмулятора сделали упор на построение трасс без необходимости крутых поворотов, а также на близкое расположение контрольных точек на трассе. Несмотря на указанные достоинства расстановки контрольных точек, могут иметь место скопления машин, как на заправочных станциях, так и в районе контрольных точек. При этом следует использовать незанятые контрольные точки.

Подводя итог, получим, что основным элементом базовой стратегией является езда через контрольные точки в порядке возрастания номеров. Исключения возможны при блокировке части трассы машинами (одной или более). Стрельба пулями следует использовать не в ущерб основному движению.

Однако заслуживает внимания альтернативная стратегия, относящаяся к атаке пулями. Зачастую, к концу заезда образуется группа машин без топлива, пассивно стоящих на трассе. При наличии в игре «некомпетентных» противников (не реагирующих защитным режимом на угрозу поражения пулей) возможно получение большого количества очков при постоянной атаке пулями. Для этого требуется остановиться на источнике пуль, имея курс на противника. Далее следует постоянно выпускать пули. Скорость получения очков в данном случае превышает скорость получения очков при езде через контрольные точки.

В дальнейшем будем различать понятия тактики и стратегии. Стратегией будем называть все, что относится к выбору целей, ситуаций для стрельбы и ситуаций, благоприятных для пополнения топливом и пулями. Тактикой будем называть все, что относится к способу проезда между двумя выбранными целями, определению момента включения защитного режима, выбору конкретного момента для стрельбы, дозаправки и т.п.

2.2. Движение

Данный раздел целиком посвящен проблемам оптимального движения по трассе. Это объясняется характером выбранной стратегии поведения.

2.2.1. Езда через выбранные контрольные точки

После того, как машина достигла очередной контрольной точки, и уже выбрана следующая контрольная точка, возникает задача выбора оптимальной траектории. Рассмотрим следующую ситуацию (рис. 8).

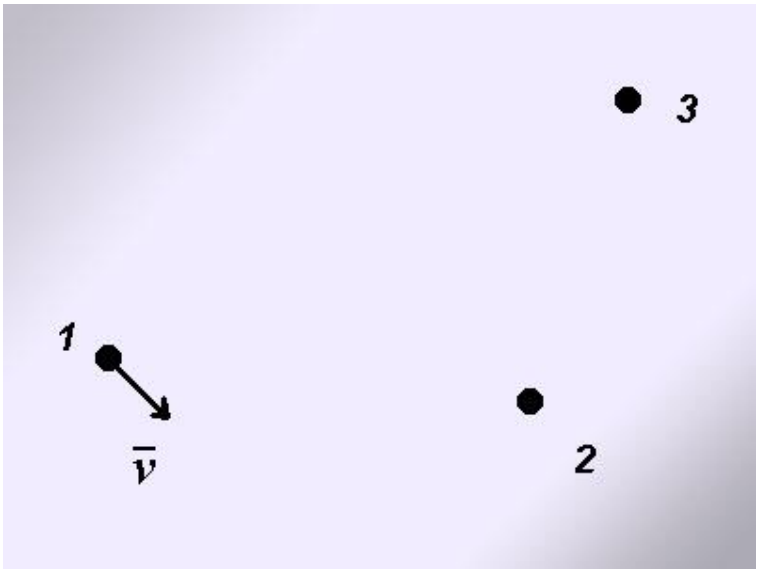

Рис. 8. Проезд контрольных точек

На этом рисунке точкой 1 обозначено текущее положение машины. Точка 2 – следующая контрольная точка. Точка 3 – цель, которая должна быть достигнута после точки 2. Вектор $\boldsymbol{v}$ – текущая скорость машины.

Известно, что кратчайший путь, соединяющий две точки на плоскости – прямая, проходящая через них. Однако машина обладает инерцией, и не может сразу изменить направление в сторону точки 2. Она будет продолжать двигаться по вектору $\boldsymbol{v}$ некоторое время. Попытаемся представить, что произойдет, если машина начнет постепенно поворачиваться в нужную сторону. В начале пути машина пройдет по отрезку окружности. Далее, выровняв курс на точку 2, она пойдет примерно по прямой. Маршрут, пройденный машиной в данной ситуации, показан на рис. 9.

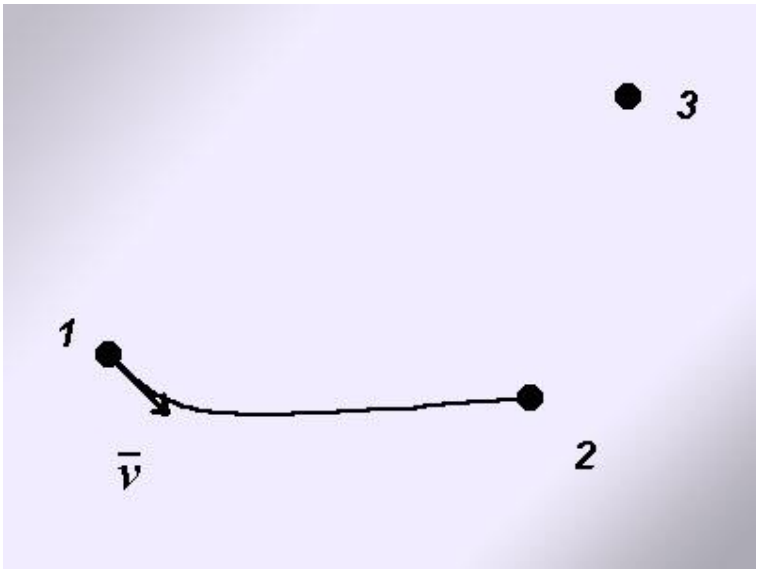

Рис. 9. Локально оптимальный поворот

При описанной тактике поворота достигнута некоторая оптимальность движения на отрезке 1-2. Однако при этом упущено то, что в момент, когда будет достигнута точка 2, придется выравнивать курс на точку 3. Таким образом, достигнута только локальная оптимальность. Кроме того, как показывает практика и будет показано ниже, крутые повороты снижают скорость машины. На практике оказывается более выгодным начальную часть поворота (часть траектории, являющаяся отрезком окружности) растянуть до точки 2, превратив ее в отрезок окружности с большим радиусом.

Вернемся к локальной оптимальности. Если двигаться подобным образом от точки 1 до точки 2, то по достижению последней, можно обнаружить, что придется довольно сильно изменить курс на достаточно коротком отрезке дистанции до точки 3. Следовательно, чтобы избежать подобного, необходимо обеспечить влияние положения отрезка 2-3 на поведение машины на первой части траектории. Жертвуя локальной оптимальностью на обеих частях траектории, получим оптимальность при проезде всей трассы (рис. 10).

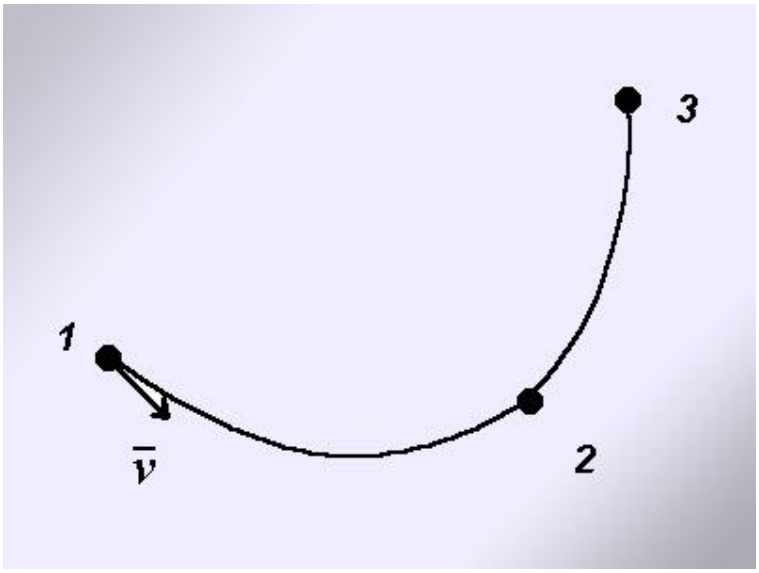

Рис. 10. Оптимальный проезд через контрольную точку

Возвращаясь к определению проезда через контрольную точку (разд. 1.2), можно обнаружить, что имеется некоторая свобода проезда в районе точки 2. Зона, через которую надо проехать – окружность с теоретическим радиусом (фактический радиус несколько меньше) в 25 единиц длины, центр которой расположен в точке 2. Для того чтобы получить очки нет необходимости проезжать строго через точку 2, так как теоретически достаточно лишь коснуться траекторией движения зоны проезда. На рис. 11 приведен вариант выбора точки проезда в допустимой зоне (точка проезда – 4, зона проезда – серый круг). Расположив точку 4 внутри угла 1-2-3, сокращается как длина траектории, так и ее кривизна.

В текущей реализации программы на выбор конкретной точки внутри этого угла влияют – величина угла 1-2-3, длины векторов 1-2 и 2-3, направление и длина вектора ***v***. Гладкость движения обеспечивается методом **getDesiredSteeringTo()**, который ответственен за выбор угла поворота в зависимости от положения цели движения относительно текущего положения машины. Данный метод (в отличие от метода выбора точки 4)

вызывается каждый временной тик, что обеспечивает непрерывный контроль машины на траектории.

Кроме того, в программе для определения проезда через точку 4 используется следующий алгоритм: проезд через эту точку считается совершенным, если корректированная дистанция (разд. 2.3.3) от текущего положения машины до точки 4 менее некоторой константы. Данная дистанция будет пройдена в любом случае благодаря инерции автомобиля, но можно начать поворот на новую выбранную цель раньше. При рассмотрении вопроса о выборе точки 4 следует отметить, что выбор данной точки близко к границе зоны проезда нежелателен, потому что машина движется по траектории дискретно. Следовательно, если пересечение траектории движения и зоны проезда будет иметь малую протяженность, то, возможно, проезд через зону не будет зафиксирован эмулятором.

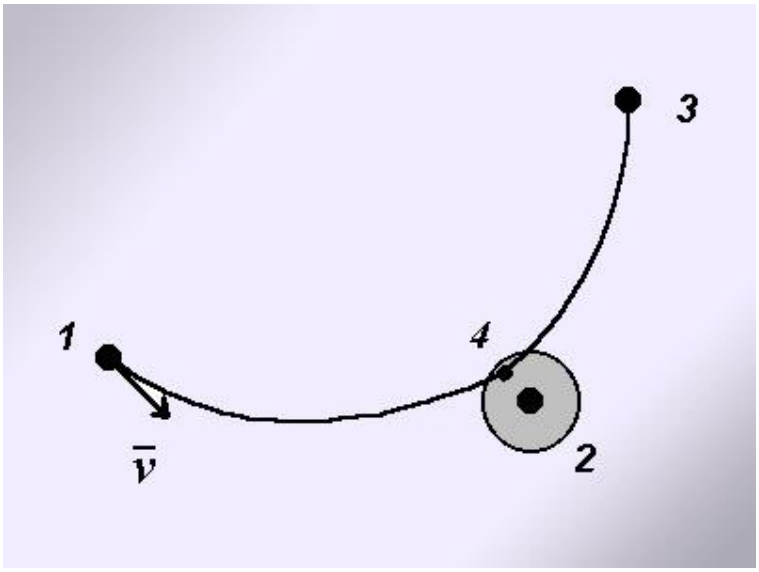

Рис. 11. Оптимальный проезд через зону контрольной точки

2.2.2. О крутых поворотах

Из практики использования эмулятора замечено, что машины на крутых поворотах теряет скорость. Для выяснения причин этого необходимо определить физику движения автомобиля. Обратимся к исходным текстам эмулятора. Они были получены в результате декомпиляции его class-файлов. Рассмотрим класс **Car**, осуществляющего пересчет положения машины в плоскости трассы. При этом можно вычленить следующие основные фрагменты кода, обрабатывающие движение машины при отсутствии столкновений со стенами, другими машинами и отсутствии поражений пулями:

```
public double getSpeed()
{
     int h = (int)heading;
     return FastMath.sin(h) * mx - FastMath.cos(h) * my;
}
...
public int getChangeInHeading()
{
     return (int)(((double)steering * getSpeed()) / 5D);
}
...
car.mx *= ROLLING_FRICTION;
car.my *= ROLLING_FRICTION;
...
int h = (int)car.heading;
car.mx +=FastMath.sin(h)*(double)car.throttle * THROTTLE_EFFECT;
car.my -=FastMath.cos(h)*(double)car.throttle * THROTTLE_EFFECT;
...
double nx = super.x + mx;
double ny = super.y + my;
...
super.x = nx;
```

```
super.y = ny;
...
private static final double ROLLING_FRICTION = 0.75D;
private static final double THROTTLE_EFFECT =
                0.050000000000000003D;
```

Переменные **mx** и **my** – проекции текущей скорости на оси X и Y. Переменная **heading** – курс машины и **throttle** – ускорение. Каждый временной тик текущая скорость умножается на 0.75 - аналог трения в частях машины. После этого к вектору скорости добавляется вектор ускорения - вектор, имеющий направление по курсу машины и длину равную модулю ускорения (**throttle**), умноженному на коэффициент **THROTTLE_EFFECT** = 0.05.

Используя приведенный фрагмент кода, рассчитаем максимальную скорость машины. Для этого потери на трение должны компенсироваться ускорением:

$$MAX\_THROTTLE*THROTTLE\_EFFECT = V_{max}*(1–ROLLING\_EFFECT)$$

$$=>$$

$$V_{max} = 20$$

Зная максимальную скорость, можно рассчитать максимальный угол, на который возможно изменение курса машины (**heading**) за один временной тик. Воспользуемся кодом метода **getChangeInHeading()** и получим следующее значение:

$$MAX\_STEER\_RIGHT*V_{max}/5 = 10*20/5 = 40.$$

Таким образом, при максимальной скорости машина может изменить курс на 40 градусов за временной тик. Покажем, откуда берутся потери скорости.

Если на полной скорости включить максимальный угол поворота, то эффект ускорения будет направлен к вектору текущей скорости под углом в 40 градусов (рис. 12).

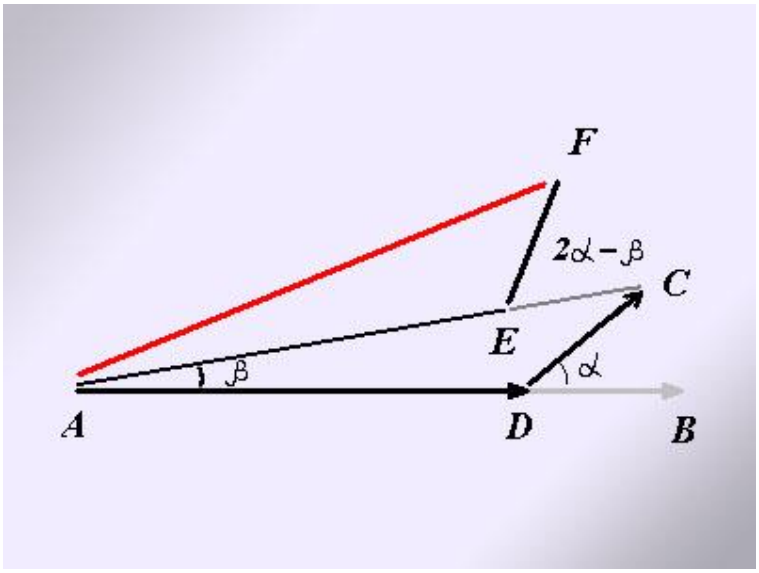

Рис. 12. Потери скорости при быстром повороте

На этом рисунке **AB** – вектор скорости на начало временного тика, **AD** – вектор скорости с учетом трения, **DC** – ускорение, **AC** – результирующая скорость.

Так как ускорение не совпадает по направлению с первоначальной скоростью, результирующая скорость будет меньше (|**AC**| < |**AB**|). И чем больше будет угол между ускорением и текущей скоростью, тем больше будут потери. При первой итерации (первом временном тике) угол α между скоростью и ускорением равняется 40 градусам. Однако перед второй итерацией уже будет накоплена разница между курсом и скоростью (физический аналог – занос, когда машина движется не туда, куда «смотрит»). Эта разница = α - β. Добавив изменение курса за один шаг, получим, что угол между скоростью и ускорением будет равняться 2α – β. Строго говоря, разница будет несколько меньше, из-за того, что изменение курса пропорционально скорости, а скорость на начало второй итерации уже

частично потеряна. Однако при небольших потерях скорости на ранних итерациях это несущественно.

Несложный расчет дает значение для угла β менее 10 градусов. Следовательно, угол разницы между ускорением и скоростью во второй итерации составит уже примерно 70 градусов. Возрастут и потери (|**AC**| - |**AF**| > |**AB**| - |**AC**|).

Моделируя поворот машины (с помощью специально написанной программы), можно получить следующую таблицу, связывающую скорость и потери скорости с текущей итерацией (табл. 2).

Таблица 2

Итерация	Скорость (единицы)	Угол между скоростью и ускорением (градусы)	Суммарное изменение угла скорости (градусы)	Отношение скорости к максимальной (проценты)
0	20.0000	0.00	0.00	100.00
1	19.1025	30.3	19.69	95.51
2	16.8144	52.46	25.75	84.07
3	13.8796	65.02	46.81	69.40
4	11.3276	66.62	72.97	56.64
5	9.9122	58.98	103.26	49.56

Уточним названия четвертого и пятого столбцов. Четвертый столбец – суммарное изменение направления скорости в результате действия всех предыдущих итераций, а пятый – значение, находящееся во втором столбце, деленное на максимальную скорость машины ($\mathbf{V}_{max}$) и выраженное в процентах. Для того чтобы получить изменение курса необходимо сложить второй и третий столбцы.

Из этой таблицы следует, что продолжительные по времени повороты резко снижают текущую скорость машины. Например, при изменении курса на 110 градусов потери скорости составят 30 процентов (направление скорости при этом изменяется всего лишь на 45 градусов). Однако, если стараться сохранять примерно одно направление курса и скорости, то можно за одну итерацию повернуть вектор скорости на 20 градусов, потеряв при этом лишь пять процентов скорости. Приведенные расчеты подтверждают эффективность тактики поворота, описанной в предыдущем разделе, по сравнению с тактикой быстрого поворота. Таким образом, если не требуется быстрый поворот, ввиду близости цели, то следует использовать менее крутые траектории поворота.

2.2.3. Корректированная дистанция

Ввиду того, что машина обладает инерцией, во многих случаях использование обычной (эвклидовой) дистанции на плоскости оказывается некорректным. Рассмотрим выбор ближайшей заправочной станции. В ситуации, показанной на рис. 13, выбор станции 1 оказывается неверным.

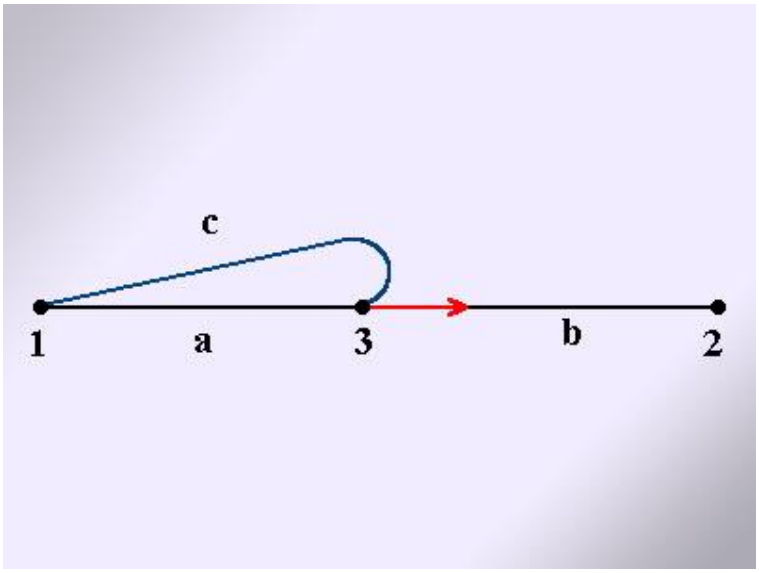

Рис. 13. Необходимость использования корректированной дистанции (**a = b – ε, ε << b**, а 1 и 2 – точки заправочных станций).

Объясним отмеченное выше. Пусть машина находится в точке 3 и движется в сторону точки 2. Если сравнивать дистанции до точек 1 и 2, то может показаться, что выгоднее двигаться к точке 1. Однако благодаря тому, что машина имеет инерцию, разворот на месте невозможен и фактическая траектория будет иметь вид кривой **c**. Временные и дистанционные затраты на разворот и движение к точке 1 окажутся выше, чем при движении к точке 2. Для того чтобы избежать подобных ошибок при расчетах, основанных на измерениях дистанции, в программе используется *корректированная дистанция*.

Корректированная дистанция – это расстояние на плоскости, сложенное с некоторой монотонной возрастающей штрафной функцией от угла поворота. В текущей версии программы штрафная функция равняется абсолютной величине относительного угла (угла относительно курса машины) на объект. При этом угол берется с некоторым коэффициентом (порядка единицы). Значение корректированной дистанции можно получить с помощью метода **getApproximateDistanceTo(IObject obj),** приведенного в тексте программы в приложении.

3. Класс «RallyCar»

3.1. Словесное описание

Класс инкапсулирует логику программы, управляющей машиной. Он унаследован от класса **Car** и реализует интерфейс **ICar**. В этом классе реализованы также все методы, перечисленные в разд. 1.3.1 - **move(), getSchoolName(), getName(), getColor()** и **initialize()**. Кроме того, в классе реализован подкласс **FieldPoint**, необходимый для унификации работы с объектами, находящимися на трассе, и виртуальными точками, вводимыми самой программой.

В классе, кроме того, реализован **автомат**, осуществляющий принятие стратегических решений (управляющий автомат). Автомат реализован одним методом, а действия, выполняемые в состояниях – другими методами. Применение автомата позволяет централизовать логику управления машиной и сделать ее более понятной.

В результате получения событий автомат формирует две переменные, содержащие необходимую информацию. Первая из них (**target)** имеет тип **IObject**, а вторая (**flags)** - тип **int**. Биты второй переменной представляют собой набор флагов различного назначения. Описание этих переменных приведено в разд. 3.2.5.

3.2. Метод «move()»

3.2.1. Описание

Тело этого метода можно условно разбить на три части. Первая – определение необходимости стрельбы пулями и включения защитного режима. Вторая – генерация событий для управляющего автомата. Третья –

движение к текущей цели. Выбранная цель хранится в переменной **target**. Изменение цели осуществляется выходными воздействиями автомата.

3.2.2. Стрельба

При стрельбе пулями сначала проверяется возможность выстрелить (имеется пуля и прошло более 25 тиков с последнего выстрела). После этого, при наличии незащищенного соперника (не находящегося в защитном режиме) на расстоянии не более 80 единиц от позиции машины и относительном угле не более 20 градусов, производится выстрел. Выстрел возможен, даже если топливо закончилось.

3.2.3. Защитный режим

Защитный режим включается в четырех случаях.

Первый – закончилось топливо. Это означает, что машина уже выбыла из игры и представляет собой практически статический объект. Включение защитного режима предохраняет от поражения пулями и не дает противникам зарабатывать очки.

Второй и третий случаи – защитный режим включается при наличии машин соперника или пуль, выпущенных соперниками, на расстоянии менее 80 единиц от позиции машины.

Четвертый случай – «бесконечное топливо» (раз. 3.2.4).

3.2.4. Генерация событий

В программе предусмотрена генерация управляющих событий для управляющего автомата. Всего существует шесть типов событий (табл. 3).

Таблица 3

Событие	Комментарий
1	Осталось мало топлива
2	Количество топлива близко к максимуму
3	Расстояние до цели менее или равно 25 единицам
4	Расстояние до цели более 25 единиц, но менее или равно 40 единицам
5	Расстояние до цели более 40 единиц
6	«Бесконечное топливо»

Опишем более подробно событие «бесконечное топливо». Из исходных текстов эмулятора можно вычленить следующую информацию:

```
...
double d = Math.abs((double)car.throttle / 400D);
if(protectMode > 0)
    d *= 2D;
car.fuel -= d;
if(car.fuel < 0.0D)
car.fuel = 0.0D;
...
```

Из этого фрагмента кода следует, что максимальная скорость потребления топлива равняется (MAX_THROTTLE/400)*2 = 0.5 (максимальное ускорение и включенный защитный режим).

При включенном защитном режиме машина не подвержена дополнительным потерям топлива вследствие попадания в нее пулей или столкновения с другой машиной. Если количество временных тиков до конца матча меньше, чем количество топлива, умноженное на два, то можно гарантировать, что у машины достаточный запас топлива, чтобы оставаться в движении до конца

матча при любых событиях на трассе. Отпадает необходимость заправки, а защитный режим поддерживается включенным все время.

3.2.5. Движение к текущей цели

Информация о текущей цели содержится в переменной **target**. Способ проезда цели содержится в двух младших битах старшего байта переменной **flags**. Назовем их условно bit_0 и bit_1. и поясним их назначение.

Для того чтобы заправиться топливом необходимо подъехать к заправочной станции и остановиться, а контрольная точка не требует остановки. Поведение машины должно различаться от типа цели, к которой она направляется. Для этого и используются указанные биты. «Выключенный» бит bit_0 сигнализирует о необходимости проезда сквозь цель (остановка не требуется), а «включенный» бит bit_0 – о необходимости остановки. «Включенный» бит bit_1 имеет более высокий приоритет по сравнению с битом bit_0 и сигнализирует о полном выключении двигателя (цель достигнута и требуется оставаться некоторое время в данной позиции).

Итак, в зависимости от состояния битов выполняются разные действия (табл. 4)

Таблица 4

bit_0	bit_1	Угол	Ускорение
0	0	getDesiredSteeringTo(target));	MAX_THROTTLE
1	0	getDesiredSteeringTo(target));	getDesiredThrottle(target)
0	1	0	0
1	1	0	0

Метод **getDesiredSteeringTo(IObject obj)** возвращает угол поворота колес в зависимости от положения объекта **obj** относительно машины.

Метод **getDesiredThrottle(IObject obj)** возвращает ускорение в зависимости от положения объекта **obj** относительно машины.

3.3. Автомат стратегии

3.3.1. Словесное описание

Автомат стратегии – это автомат, ответственный за управление машиной. Он выбирает точку, к которой необходимо двигаться. При этом на основании событий и его текущего состояния выбирается следующая точка и устанавливается способ ее проезда. Автомат определяет эффективность действий робота. Как бы хорошо не были реализованы методы движения, определения угрозы, определения моментов для стрельбы, без хорошо спроектированного управляющего автомата (или его аналога без явного выделения состояния) невозможно создать качественную программу управления роботом.

3.3.2. События

Автомат является событийным - использует события, описанные в разд. 3.2.4.

3.3.3. Состояния автомата. Действия в состояниях автомата

Автомат имеет шесть состояний. С каждым состоянием ассоциировано действие. Состояния и соответствующие действия имеют одинаковые номера. В программе методы, реализующие действия, имеют названия **state_action_X()**, где X принимает значения от 0 до 5 и соответствует номеру действия.

Перечислим выполняемые действия:

z0 – выбор контрольной точки для проезда;

z1 – выбор оптимальной траектории движения к контрольной точке и обеспечение движения к этой точке;

z2 – выбор заправочной станции;

z3 – движение к заправке;

z4 – заправка топливом;

z5 – расчет траектории проезда через следующую контрольную точку.

3.3.4. Граф переходов

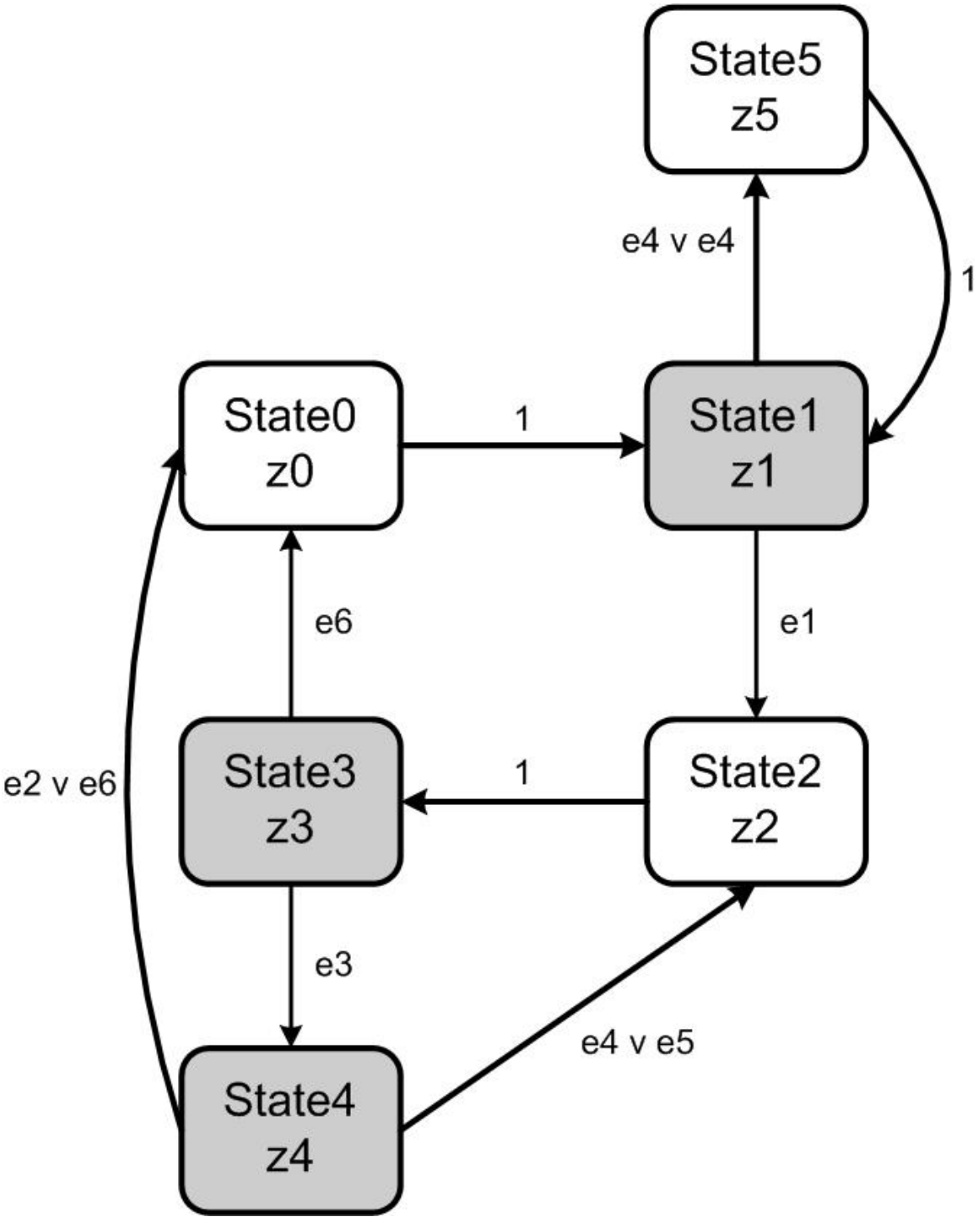

Рис. 14. Граф переходов

В этом графе StateN – состояние N, eN – событие N, zN – действие N. Стрелки с единицами – безусловный переход. Состояния, закрашенные

серым цветом, являются конечными состояниями, и соответствуют трем типам движения, реализованным в методе **move()**.

4. Заключение

Результатом выполнения бакалаврской работы явилась работоспособная, эффективная программа управления роботом.

Анализ характеристик трассы, включая исследование исходных текстов эмулятора, предшествующий написанию кода, позволил уже на ранних этапах проектирования программы отказаться от неэффективных стратегий.

Формализация физики виртуального мира стала ключом к «послушности» (высокая вероятность выполнения полученной команды) робота.

Построен управляющий автомат, который определяет эффективность действий робота – он знает, что делать и в какой последовательности. Как бы хорошо не были реализованы методы движения, определения угрозы, определения моментов для стрельбы, без управляющего автомата (или некоторого его аналога без явного выделения состояний) невозможно создать эффективного робота.

Эффективность работы предложенного робота продемонстрирована в ходе соревнований с роботами, входящими в комплект поставки эмулятора, и с роботом, созданным нами на чемпионате мира. При этом разработанная программа набирает до 350 очков, что как отмечалось в разд. 2.1, близко к теоретическому пределу.

Работа выполнена в рамках «Движения за открытую проектную документацию» [5]. Она является одним из решений проблемы, сформулированной на форуме сайта задачи *CodeRally* [4], состоящей в отсутствии достаточно сложных примеров ее решения, а тем более в отсутствии проектной документации на эти решения.

Разработанная программа робота, а также программа, написанная нами на Java Challenge 2003, выложены на указанном форуме.

В заключение приведем скриншот игры (рис. 15).

Рис. 15. Скриншот игры

5. Ссылки

1. http://acm.baylor.edu – сайт чемпионата мира по программированию по версии ACM.
2. http://www.sun.com – *Sun Technologies.*
3. http://is.ifmo.ru – сайт, посвященный SWITCH-технологии.
4. http://alphaworks.ibm.com/tech/coderally - сайт задачи *CodeRally.*
5. *Шалыто А. А.* Новая инициатива в программировании. Движение за открытую проектную документацию //Мир ПК. 2003. №10 (http://is.ifmo.ru).
6. http://www.eclipse.org – сайт платформы *Eclipse.*

Приложение. Текст программы

```
import com.ibm.rally.Car;
import com.ibm.rally.ICar;
import com.ibm.rally.*;
import java.awt.geom.AffineTransform;
import java.awt.Shape;


/**
 * This is the class that you must implement to enable your car within
 * the CodeRally track. Adding code to these methods will give your car
 * it's personality and allow it to compete.
 */
public class RallyCar extends Car {

  protected class FieldPoint implements IObject {
    public FieldPoint(double x, double y) {
      this.x = x;
      this.y = y;
    }

    public FieldPoint(IObject obj) {
      this.x = obj.getX();
      this.y = obj.getY();
    }

    public double getX() {
      return x;
    }

    public double getY() {
      return y;
    }

    public double getDistanceTo(IObject obj)
    {
      return getDistanceTo(obj.getX(), obj.getY());
    }

    public double getDistanceTo(double xx, double yy)
    {
      double dx = xx - x;
      double dy = yy - y;
      return Math.sqrt(dx * dx + dy * dy);
    }

    public int getBearing(double dx, double dy)
    {
      double d = Math.atan2(dx, dy);
      return 180 - (int)(d * 57.295779512999999D);
    }

    public int getHeadingTo(IObject obj)
    {
      return getBearing(obj.getX() - x, obj.getY() - y);
    }

    public int getHeadingTo(double xx, double yy)
    {
      return getBearing(xx - x, yy - y);
    }
```

```
  public double x, y;
}
FieldPoint points[] = new FieldPoint [120];
double [][] route_rating = null;

boolean infinite_fuel;
int checknum, checkpointsNum, first_used_defined_point, desired_point;
int flags, iter1, bad_target_cnt = 0;
IObject target;
int state, strategy_state;
double speedsum = 0;
int speedn = 0;
double x1, y1;

//utilitary variables
int curCheckpoint, curGasDepot;

/**
 * @see com.ibm.rally.Car#getName()
 */
public String getName() {
  return "AI";
}

/**
 * @see com.ibm.rally.Car#getSchoolName()
 */
public String getSchoolName() {
  return "IFMO #1";
}

/**
 * @see com.ibm.rally.Car#getColor()
 */
public byte getColor() {
  return CAR_BLUE;
}

public double getDistToSegment(IObject car, IObject p1, IObject p2)  {
  double a = p1.getDistanceTo(p2);
  double b = p1.getDistanceTo(car);
  double c = p2.getDistanceTo(car);
  double dist;
  dist = (b<c)?b:c;
  if (b > 0.01 && a > 0.01)
  {
    double cs = (a*a + b*b - c*c)/2/a/b;
    if (cs > 0.01 && cs*b<a)
      dist = b * Math.sqrt(1 - cs*cs);
  }
  return dist;
}

public double vecLength(FieldPoint p) {
  return Math.sqrt(p.x * p.x + p.y * p.y);
}

public void vecMul(FieldPoint p, double m) {
  p.x *= m;
  p.y *= m;
}

public void normalize(FieldPoint p) {
  double l = vecLength(p);
```

```
  if (l > 0.001) {
    p.x /= l;
    p.y /= l;
  }
}

public int correctAngle(int angle)
{
  while (angle <= -180) angle += 360;
  while (angle > 180) angle -= 360;
  return angle;
}

public int getRelativeAngle(double x, double y){
  int h = getHeadingTo(x, y);
  int m = getHeading();
  return correctAngle(h - m);
}

public int getRelativeAngle(IObject obj) {
  return getRelativeAngle(obj.getX(), obj.getY());
}

public double getApproximateDistTo(IObject obj) {
  return getDistanceTo(obj) + Math.abs(getRelativeAngle(obj));
}

public int getDesiredSteeringTo(int angle) {
  int value = Math.abs(angle);
  int res;

  if (value >= 40)
    res = MAX_STEER_RIGHT;
  else
  if (value >= 15)
    res = MAX_STEER_RIGHT*2/3;
  else
  if (value >= 8)
    res = MAX_STEER_RIGHT/2;
  else
    res = MAX_STEER_RIGHT/3;
  if (angle < 0) res = -res;
  return res;
}

public int getDesiredSteeringTo(IObject obj) {
  return getDesiredSteeringTo(getRelativeAngle(obj));
}

public int getDesiredThrottle(double dist, int angle) {
  int res;

  //new formula
  double sp = getSpeed();
  if (dist < 25)
    res = 0;
  else {
    res = (int)Math.round((dist/4 - sp)*20*3/4);
  }
  if (res > MAX_THROTTLE) res = MAX_THROTTLE;
  if (res < MIN_THROTTLE) res = MIN_THROTTLE;

  return res;
}
```

```
  public int getDesiredThrottle(IObject obj) {
    return getDesiredThrottle(getDistanceTo(obj), getRelativeAngle(obj));
  }

  public void calcPoint(ICar p1, IObject p2, IObject p3, int index, boolean
skip) {
    double l;
    FieldPoint v1 = new FieldPoint(p1.getX() - p2.getX(), p1.getY() -
p2.getY());
    FieldPoint v2 = new FieldPoint(p3.getX() - p2.getX(), p3.getY() -
p2.getY());

    FieldPoint v = new FieldPoint(v1.x + v2.x, v1.y + v2.y);

    int angle = Math.abs(correctAngle(p2.getHeadingTo(p1) -
p2.getHeadingTo(p3)));
    int mul = 0;
    if (angle < 160)
    {
      mul = (160 - angle)*20/(160-130);
      if (mul > 20)
        mul = 20;
    }

    l = vecLength(v);
    if (Math.abs(l) < 0.001)
      vecMul(v, 0);
    else
    if (skip)
      vecMul(v, 80/l);
    else {
      vecMul(v, mul/l);
      FieldPoint head = new FieldPoint(
        Math.sin(p1.getHeading()) * 20,
        -Math.cos(p1.getHeading()) * 20
      );
      v.x -= head.x;
      v.y -= head.y;
      if ((l = vecLength(v)) > 20)
        vecMul(v, 20/l);
    }



    points[first_used_defined_point + index].x = p2.getX() + v.x;
    points[first_used_defined_point + index].y = p2.getY() + v.y;
  }

  public boolean mayHit(double x, double y) {
    double dist = 3200D;
    return x*x + y*y < dist;
  }

  public boolean checkForCarHit(double x, double y, double h) {
    java.awt.geom.Rectangle2D rect = new java.awt.geom.Rectangle2D.Double(-
20D - 7.5D, -30D - 10D, 40D + 15D, 60D + 20D);
    java.awt.geom.Rectangle2D rect2 = new java.awt.geom.Rectangle2D.Double(-
20D - 7.5D, -30D - 10D, 40D + 15D, 60D + 20D);
    double hh = h / (360 / 2 / Math.PI);
    AffineTransform transform = AffineTransform.getTranslateInstance(x, y);
    transform.concatenate(AffineTransform.getRotateInstance(hh));
    Shape shape2 = transform.createTransformedShape(rect2);
    return shape2.intersects(rect);
```

```
  }

  FieldPoint getTargetFromNum(int num) {
    num = num & 0xff;
    if (num == 0) return null;
    return points[num - 1];
  }

  /**
   * @see com.ibm.rally.Car#initialize()
   */
  public void initialize() {
    infinite_fuel = false;
    strategy_state = 0;
    desired_point = -1;
    checkpointsNum = getCheckpoints().length;

    IObject objs[] = getCheckpoints();
    for (int i=0; i<checkpointsNum; i++)
      points[first_used_defined_point + i] = new FieldPoint(objs[i]);
    first_used_defined_point += checkpointsNum;

    objs = getFuelDepots();
    for (int i=0; i<3; i++)
      points[first_used_defined_point + i] = new FieldPoint(objs[i]);
    first_used_defined_point += 3;

    objs = getSpareTireDepot();
    for (int i=0; i<3; i++)
      points[first_used_defined_point + i] = new FieldPoint(objs[i]);
    first_used_defined_point += 3;

    for (int i=first_used_defined_point; i<120; i++)
      points[i] = new FieldPoint(0, 0);

    route_rating = new double [checkpointsNum][checkpointsNum];
    for (int i=0; i<checkpointsNum; i++)
    for (int jj=0; jj<checkpointsNum; jj++)
    if (i != jj) {
      int j = jj;
      if (j < i) j+= checkpointsNum;
      double dist = points[i].getDistanceTo(points[j]);
      for (int k=i; k<j; k++)
      dist +=
points[k%checkpointsNum].getDistanceTo(points[(k+1)%checkpointsNum]);
      int cost = (j-i) * 6 + 2;
      route_rating[i][jj] = cost/dist;
    }


    setHeadlightsOn(false);
    iter1 = 1;

    int tg = getNextTarget(0);
    target = getTargetFromNum(tg);
    flags = tg&0xff00;

    setThrottle(MAX_THROTTLE);
  }

  public void sendEvent(int event) {
    if (event != 0) {
      int tg = getNextTarget(event);
      target = getTargetFromNum(tg);
```

```
      flags = tg&0xff00;
    }
    return;
  }

  boolean isDangerousCheckpoint(int index) {
    ICar op[] = getOpponents();
    for (int i=0; i<op.length; i++)
      if (op[i].getSpeed() < 5)
        if (getCheckpoints()[index].getDistanceTo(op[i]) < 70)
          return true;
    return false;
  }

  public int getNextTarget(int event) {
    int targetnum = -1, flags = 0;
    boolean stop = false;
    while (!stop)
    switch (strategy_state)
    {
      case 0: //looking for checkpoints
        //System.out.println("state 0");
        targetnum = 0;
        int desired_checkpoint = getPreviousCheckpoint(), points1, points2;
        if (desired_checkpoint >= 0)
          desired_checkpoint = (desired_checkpoint + 1)%checkpointsNum;

        for (int i=1; i<getCheckpoints().length; i++) {
          points1 = (desired_checkpoint == i)?3:2;
          points2 = (desired_checkpoint == targetnum)?3:2;
          if (getApproximateDistTo(getCheckpoints()[targetnum])*points1 >
getApproximateDistTo(getCheckpoints()[i])*points2)
            targetnum = i;
        }
        curCheckpoint = targetnum;
        strategy_state = 1;
        stop = true;
        break;
      case 1: //go throught checkpoints
        speedn++;
        double speed = getSpeed();
        speedsum += speed;
        //System.out.println("state 1 "+ getSpeed() + " " +
speedsum/speedn);
        if (event == 1) {
          desired_point = -1;
          strategy_state = 2; //low fuel
        }
        else {
          //checkpoint passed
          if ((event == 3) || (event == 4))
          {
            IObject ch[] = getCheckpoints();
            boolean skip_checkpoint = isDangerousCheckpoint((curCheckpoint
+ 1)%checkpointsNum);
            setHeadlightsOn(skip_checkpoint);
            curCheckpoint = (curCheckpoint + 1)%checkpointsNum;
            calcPoint(
              this,
              ch[curCheckpoint],
              ch[(curCheckpoint + 1)%checkpointsNum],
              0,
              skip_checkpoint
              );
```

```
              desired_point = first_used_defined_point;
            }
            if (desired_point != -1)
              targetnum = desired_point;
            else
              targetnum = curCheckpoint;
            stop = true;
          }
          break;
        case 2: //looking for gas station
          //System.out.println("state 2");
          targetnum = 0;
          if (event == 7) {
            if (targetnum == curGasDepot) targetnum++;
            for (int i=1; i < 3; i++)
              if (i != curGasDepot)
                if (getApproximateDistTo(getFuelDepots()[targetnum]) >
getApproximateDistTo(getFuelDepots()[i]))
                  targetnum = i;
            event = 0;
          } else {
            for (int i=1; i<3; i++)
              if (getApproximateDistTo(getFuelDepots()[targetnum]) >
getApproximateDistTo(getFuelDepots()[i]))
                targetnum = i;
          }
          curGasDepot = targetnum;
          targetnum += checkpointsNum;
          strategy_state = 3;
          flags |= 0x0100;
          stop = true;
          break;
        case 3:
          //go to gas station
          //System.out.println("state 3");
          if (event == 7) {
            strategy_state = 2;
          } else
          if (event == 3)
            strategy_state = 4;
          else
          if (event == 6)
            strategy_state = 0;
          else {
            flags |= 0x0100;
            stop = true;
          }
          targetnum = curGasDepot + checkpointsNum;
          break;
        case 4:
          //refill fuel
          //System.out.println("state 4 distance = " + getDistanceTo(target));
          if ((event == 4) || (event == 5))
            strategy_state = 2;
          else
          if ((event == 2) || (event == 6))
            strategy_state = 0;
          else {
            flags |= 0x0200;
            flags |= 0x0100;
            stop = true;
          }
          targetnum = curGasDepot + checkpointsNum;
          break;
```

```
        default:
          break;
      }
      return (targetnum + 1)|flags;
    }


    /**
     * @see com.ibm.rally.Car#move(int, boolean, ICar, ICar)
     * Put the car in reverse for a few moves if you collide with another car.
     * Go toward the first gas depot.
     */
    public void move(int lastMoveTime, boolean hitWall, ICar collidedWithCar,
ICar hitBySpareTire) {


      if (getFuel() == 0) {
        enterProtectMode();
      }

      ICar op[] = getOpponents();
      if (isReadyToThrowSpareTire())
      {
        for (int i=0; i<op.length; i++)
          if (getDistanceTo(op[i]) < 80 && Math.abs(getRelativeAngle(op[i])) <
20)
            if (!op[i].isInProtectMode())
              throwSpareTire();
      }

      for (int i=0, j=op.length; i< j; i++) {
        ICar o = op[i];
        double s1 = getSpeed(), s2 = o.getSpeed();

        double x1 = getX() + Math.sin(getHeading())*s1;
        double y1 = getY() - Math.cos(getHeading())*s1;
        double x2 = o.getX() + Math.sin(o.getHeading())*s2;
        double y2 = o.getY() - Math.cos(o.getHeading())*s2;

        double x = x2 - x1;
        double y = y2 - y1;
        double h = correctAngle(o.getHeading() - getHeading());

        if (mayHit(x, y))
          if (checkForCarHit(x, y, h))
            enterProtectMode();

        if (getDistanceTo(op[i]) < 80)
          enterProtectMode();
      }

      boolean stop = false;

      //events
      // low fuel             #1
      // hi fuel              #2
      // distancetotarget <= 25    #3
      // 25 < distancetotarget <= 40 #4
      // 40 < distancetotarget     #5
      // "infinite fuel"        #6 (disables #1 #2)
      if (getFuel()*2 >= MAX_CLOCK_TICKS - getClockTicks())
        infinite_fuel = true;
      if (!infinite_fuel) {
```

```
        if (getFuel()<30) sendEvent(1);
        if (getFuel()>=95) sendEvent(2);
      } else
      {
        sendEvent(6);
        enterProtectMode();
      }
      double d = getDistanceTo(target);
      if (d <= 25) {
        sendEvent(3);
      }
      else
      if (d <= 40) {
        sendEvent(4);
      }
      else
        sendEvent(5);

      if (Math.abs(d - 35)<10 && Math.abs(Math.abs(getRelativeAngle(target)) -
40)<15) {
        bad_target_cnt++;
      } else {
        bad_target_cnt = 0;
      }
      if (bad_target_cnt > 10 && strategy_state == 3) {
        sendEvent(7);
        bad_target_cnt = 0;
      }

      if ((flags & 0x0200) > 0) {
        setThrottle(0);
        setSteeringSetting(0);
      } else
      if ((flags & 0x0100) > 0) {
        setThrottle(getDesiredThrottle(target));
        setSteeringSetting(getDesiredSteeringTo(target));
      } else {
        setThrottle(MAX_THROTTLE);
        setSteeringSetting(getDesiredSteeringTo(target));
      }
    }
}
```