Конечный автомат. Написание ИИ для хоккея.

Существуют различные способы создать какую-нибудь особенную игру. Чаще всего разработчик для получения лучшего результата выбирает такую игру, которую он уже в состоянии написать.

Введение

Хоккей – одна из самых популярных спортивных игр. О нем написано множество статей, охватывающих и тактику игры, и поведение игроков в атаке и в защите, и такую тонкую вещь, как работа в команде. И, разумеется, искусственный интеллект. Реализация хоккея отлично подходит для демонстрации сочетания некоторых полезных приемов и методов.

Хоккей – динамичная игра. Если движения игроков будут предопределены, то играть в нее станет совсем скучно и неинтересно. Но как же нам создать динамичную игру, да при том, чтобы игроки вели себя адекватно и осознанно? Ответ прост – с помощью механики рулевого поведения.

Использование рулевого поведения направлено на создание реалистичных моделей передвижения объектов. Они основаны на простых силах, вследствие чего – чрезвычайно динамичны по своей природе. Это делает их идеальным выбором для реализации сложных и реалистичных движений, которые встречаются в футболе или в том же хоккее.

Обзор предстоящей работы

Как уже говорилось выше, хоккей – очень сложная игра. В ней присутствует огромное количество правил, нарушений и т.д. Для сокращения времени обучения, мы несколько упростим игру и сохраним лишь небольшой набор оригинальных правил этого вида спорта: у нас не будет вратарей (все игроки на катке будут двигаться), а также мы не будем учитывать всевозможные штрафы.

Так будет выглядеть наша игра

Ворота в нашей игре тоже будут своеобразные – сетка будет отсутствовать, а для того, чтобы забить гол, достаточно шайбе коснуться «ворот» с любой стороны. После забитого гола все игроки встают на свои позиции, шайба перемещается в центр, и через несколько секунд игра начинается заново.

Касаемо обработки шайбы: если игрок *A* находится с шайбой, а игрок *B* сталкивается с ним, то шайба переходит к игроку *B*, который становится недвижимым на некоторое время.

Для вывода графики будем использовать графический движок *Flixel*. Однако в прилагаемом коде будет опущено все связанное с графикой и максимально обращать ваше внимание на механику игры.

Базовые классы

Давайте начнем с основ – катка, который представляет собой прямоугольник, игроков и двух ворот. Каток имеет физические границы, поэтому ничего не выйдет за пределы поля. Хоккеист будет описываться классом ***Athlete***:

```
public class Athlete
{
    private var mBoid :Boid; // контролирует физическое тело хоккеиста
    private var mId   :int;  // уникальный идентификатор хоккеиста

    public function Athlete(thePosX :Number, thePosY :Number, theTotalMass
:Number) {
        mBoid = new Boid(thePosX, thePosY, theTotalMass);
    }

    public function update():void {
        // очистка всех действующих сил
        mBoid.steering = null;

        // метод блуждания по катку
        wanderInTheRink();

        // главный метод обновления физического тела
        mBoid.update();
    }

    private function wanderInTheRink() :void {
        var aRinkCenter :Vector3D = getRinkCenter();

        // Если расстояние до центра катка больше 80
        // вернуться в центр, иначе бродить
        if (Utils.distance(this, aRinkCenter) >= 80) {
```

```
            mBoid.steering = mBoid.steering + mBoid.seek(aRinkCenter);
        } else {
            mBoid.steering = mBoid.steering + mBoid.wander();
        }
    }
     }
```

Поле *mBoid* является объектом класса *Boid*. Он имеет, среди прочих элементов, вектор направления, вектор силы, а также текущее положение игрока.

Метод *update()* будет вызываться каждый раз, пока запущена игра. Сейчас в этом методе очищается любое активное усилие в рулевом поведении, добавляется эффект блуждания игрока, а также вызывается метод *mBoid.update()*.

Класс, ответственный за саму игру, называется *PlayState*. Среди его полей есть каток, две группы хоккеистов, а также двое ворот.

```
public class PlayState
{
    private var mAthletes  :FlxGroup;
    private var mRightGoal :Goal;
    private var mLeftGoal  :Goal;

    public function create():void {
        // здесь будут создаваться все игровые элементы
    }

    override public function update():void {
        // включить коллизию всех хоккеистов с бортиками катка
        collide(mRink, mAthletes);

        // проверка того, что все хоккеисты в пределах катка
        applyRinkContraints();
    }

    private function applyRinkContraints() :void {

    }
     }
```

Если мы добавим одного хоккеиста на поле, то увидим такой результат:

Следуй за мышью

Мышь имеет координаты на экране, а потому мы можем использовать их в качестве пункта назначения для игрока. Мы можем использовать метод *arrival* объекта *mBoid*. Он задает цель, которую будет преследовать игрок. Движение будет плавным, а по мере приближения скорость хоккеиста будет падать и, в конце концов, станет равна 0.

Давайте заменим блуждающий метод в классе *Athlete* на движение к курсору мыши:

```
public class Athlete
{
    // (...)

    public function update():void {
        // очистка всех действующих сил
        mBoid.steering = null;

        // игрок управляет хоккеистом,
        // поэтому он будет следовать за курсором мыши
        followMouseCursor();

        // главный метод обновления физического тела
        mBoid.update();
    }

    private function followMouseCursor() :void {
        var aMouse :Vector3D = getMouseCursorPosition();
        mBoid.steering = mBoid.steering + mBoid.arrive(aMouse, 50);
    }
     }
```

В результате мы получим возможность управлять нашим игроком мышью, а движение получится плавным и реалистичным.

Добавим шайбу

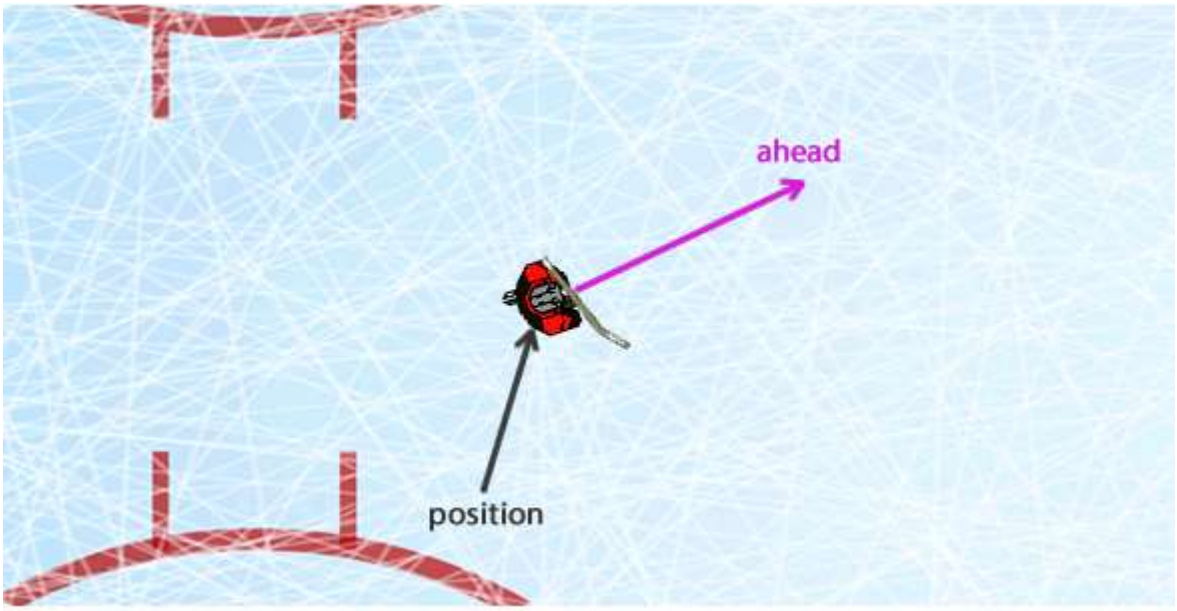
Шайба будет описываться классом *Puck*. Самое важное здесь — метод *update()* и поле *mOwner*.

```
public class Puck
{
    public var velocity :Vector3D;
    public var position :Vector3D;
    private var mOwner :Athlete;    // хоккеист, который владеет шайбой

    public function setOwner(theOwner :Athlete) :void {
        if (mOwner != theOwner) {
            mOwner = theOwner;
```

```
            velocity = null;
        }
    }

    public function update():void {
    }

    public function get owner() :Athlete { return mOwner; }
    }
```

Следуя той же логике, что и выше, метод *update()* будет выполняться каждый раз, пока запущена игра. Поле *mOwner* будет хранить того хоккеиста, который владеет шайбой. Если же *mOwner* равно *null*, то шайба никому не принадлежит, и она свободно скользит по катку.

Если же *mOwner* не равно *null*, то владелец у шайбы имеется. В этом случае шайба насильно ставится перед своим владельцем. Для этого используется вектор скорости хоккеиста, который также соответствует и вектору направления. Вот наглядная иллюстрация:



Ставим шайбу перед ее владельцем

А вот и код:

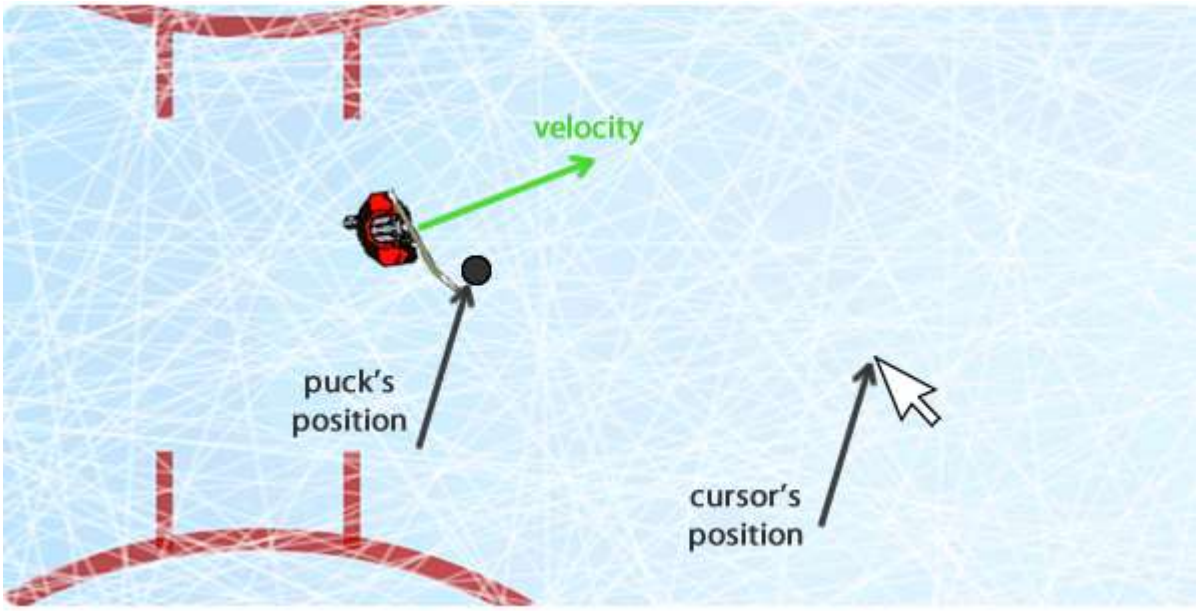
```
public class Puck
{
    // (...)

    private function placeAheadOfOwner() :void {
        var ahead :Vector3D = mOwner.boid.velocity.clone();

        ahead = normalize(ahead) * 30;
```

```
            position = mOwner.boid.position + ahead;
        }

        override public function update():void {
            if (mOwner != null) {
                placeAheadOfOwner();
            }
        }

        // (...)
        }
```

В классе *PlayState* есть проверка на коллизию с шайбой. Если какой-то хоккеист перекроет шайбу, то он становится новым владельцем. Вы можете протестировать это в демо-режиме, представленном ниже:

Удар по шайбе

Реализуем удар по шайбе клюшкой. Независимо от того, кто является владельцем шайбы, нам нужно сымитировать удар по ней и вычислить направление этого удара. Затем отправить шайбу по найденному вектору. Вычислить этот вектор несложно:



Вычисление вектора удара.

А вот и реализация удара по шайбе:

```
public class Puck
{
    // (...)

    public function goFromStickHit(theAthlete :Athlete, theDestination :Vector3D,
theSpeed :Number = 160) :void {
```

```
        // установка шайбы перед хоккеистом во избежании неожиданных
столкновений
        placeAheadOfOwner();

        // очистка владельца шайбы (т.к. был совершен удар)
        setOwner(null);

        // вычисление траектории шайбы
        var new_velocity :Vector3D = theDestination - position;
        velocity = normalize(new_velocity) * theSpeed;
    }
    }
```

В классе *PlayState* метод *goFromStickHit()* выполняется каждый раз, когда игрок нажимаете на экран. Координата клика используется в качестве конечной точки для удара.

Добавление ИИ

До сих пор у нас был только один хоккеист на катке. Поскольку в хоккее по 6 человек на команду, а это уже целая дюжина игроков, то нам стоит задуматься о создании искусственного интеллекта для них. Да причем такого, чтобы игроки вели себя естественно и рационально.

Для этого мы будем использовать конечный автомат (*FSM – finite state machine*). Как было написано ранее, FSM является универсальным и полезным инструментом для реализации ИИ в играх.

Немного изменим наш класс *Athlete*:

```
public class Athlete
{
    // (...)
    private var mBrain :StackFSM; // конечный автомат, контролирующий ИИ

    public function Athlete(thePosX :Number, thePosY :Number, theTotalMass
:Number) {
        // (...)
        mBrain = new StackFSM();
    }

    // (...)
    }
```

Поле mBrain является экземпляром класса *StackFSM*, о котором более подробно вы можете узнать из этой статьи. Он использует стек для управления состоянием ИИ. Каждое состояние описывается методом, и, когда состояние кладется в стек, оно становится активным и вызывается каждый раз при вызове основного метода *update()*.

Все состояния игрока будут выполнять строго определенную функцию: взять шайбу, отобрать шайбу, патрулировать зону и т.д.

Теперь хоккеист может быть, как под нашим контролем, так и под контролем ИИ. Обновим наш класс:

```
public class Athlete
{
    // (...)

    public function update():void {
        // очистка всех действующих сил
        mBoid.steering = null;

        if (mControlledByAI) {
            // хоккеист управляем ИИ
            // обновление логики конечного автомата
            mBrain.update();

        } else {
            // хоккеист управляем игроком,
            // поэтому он будет следовать за курсором мыши
            followMouseCursor();
        }

        // главный метод обновления физического тела
        mBoid.update();
    }
     }
```

Если хоккеист находится под контролем искусственного интеллекта, то мы обновляем ее логику, вызывая *mBrain.update()*. Если же хоккеист под управлением игрока, то логика ИИ игнорируется, а спортсмен следует за мышью.

Что касается самих состояний, посылаемых искусственному интеллекту, то мы реализуем два из них. Первое будет отвечать за подготовку игроков к матчу, т.е. они переместятся к своим стартовым позициям и будут смотреть на шайбу. Второе состояние просто будет заставлять хоккеиста стоять и следить за шайбой.

Состояние покоя

```
public class Athlete
{
    // (...)
```

```
    public function Athlete(thePosX :Number, thePosY :Number, theTotalMass
:Number, theTeam :FlxGroup) {
        // (...)

        // указание ИИ состояния 'idle'
        mBrain.pushState(idle);
    }

    private function idle() :void {
        var aPuck :Puck = getPuck();
        stopAndlookAt(aPuck.position);
    }

    private function stopAndlookAt(thePoint :Vector3D) :void {
        mBoid.velocity = thePoint - mBoid.position;
        mBoid.velocity = normalize(mBoid.velocity) * 0.01;
    }
     }
```

На данный момент это состояние будет активным всегда. Но в дальнейшем, оно заменит собой другие состояния, например, *attack*.

Метод *stopAndlookAt()* высчитывает нужное направление по тому же алгоритму, по которому мы вычисляли направление удара. Вектор, начинающийся с позиции хоккеиста и заканчивающийся позицией шайбы, измеряется по формуле *thePoint – mBoid.position* и используется для указания направления взгляда спортсмена.

Если применить полученный вектор к хоккеисту, то он устремиться к шайбе. Для того, чтобы он оставался на месте, мы умножаем вектор на число, близкое к нулю, т.е. на 0.01. Это удерживает спортсмена на месте, однако, смотреть он будет на шайбу.

Подготовка к матчу

Это состояние ответственно за то, чтобы игроки возвращались на свои позиции и останавливались там. Обновим наш класс *Athlete*:

```
public class Athlete
{
    // (...)
    private var mInitialPosition :Vector3D; // стартовая позиция хоккеиста

    public function Athlete(thePosX :Number, thePosY :Number, theTotalMass
:Number, theTeam :FlxGroup) {
        // (...)
```

```
        mInitialPosition = new Vector3D(thePosX, thePosY);

        // указание ИИ состояния 'idle'
        mBrain.pushState(idle);
    }

    private function prepareForMatch() :void {
        mBoid.steering = mBoid.steering + mBoid.arrive(mInitialPosition, 80);

        // нахожусь ли я в своей начальной позиции?
        if (distance(mBoid.position, mInitialPosition) <= 5) {
            // я в своей стартовой позиции
            mBrain.popState();
            mBrain.pushState(idle);
        }
    }

    // (...)
    }
```

Ниже вы сможете увидеть результат добавления ИИ в игру. Нажмите клавишу G и игроки переместятся в случайные позиции. Затем они встанут на нужные места:

Замечание 1.

Используя конечный автомат с двумя состояниями мы научили игроков готовиться к матчу и занимать положенные места. Далее разберем, как организовать нападение и научить игроков забивать голы!

Несколько слов об атаке

Скоординировать и грамотно провести атаку в любой командной игре достаточно сложная задача. В реальной игре хоккеисты частенько проводят сложнейшие комбинации.

Однако все действия игроков возможны лишь при анализе текущей ситуации на катке. Хоккеисты постоянно просчитывают возможные ходы на игровом поле. Человек в большинстве случаев может объяснить действия одного игрока относительно другого. Например, «этот хоккеист занял более выгодное положение, проанализировав ситуацию на катке». Хоть для нас это и очевидно, объяснить такую тактику компьютеру совсем нетривиальная задача.

Как следствие, если мы попытаемся научить искусственный интеллект думать также, как и человек, то в итоге получим огромную и ужасную кучу кода. К тому же такой код будет трудночитаемым и уж точно сложно модифицируемым.

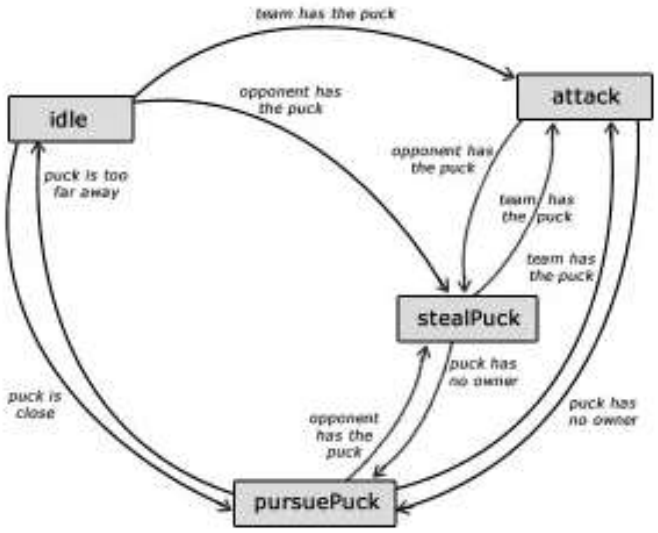
Все вышеперечисленное становится причиной тому, почему мы должны имитировать командную игру людей-игроков, а не научить виртуальных игроков тому, что умеет реальный. Такой

подход будет менее реалистичным, однако код станет проще, а его количество — меньше. Но несмотря на это, результат будет приемлемым в большинстве случаев.

Организация атаки с помощью состояний

Мы разобьем всю атаку на несколько мелких частей, каждая из которых будет выполнять какую-то свою специфическую задачу. Все эти куски будут являться состояниями конечного автомата, основанного на стеке. Как было сказано ранее, каждое состояние будет прикладывать силу в «рулевом управлении» (*steering force*), вследствие чего хоккеист будет вести себя соответственно.

Управление этими состояниями, а также переключение между ними и определяет такое сложное явление, как атака. Изображение ниже иллюстрирует те самые состояния, на которые была разобрана вся атака:



Все эти состояния будут переключаться в зависимости от расстояния до шайбы и от того, кто является ее владельцем. Например, мы будем включать состояние *attack*, если выполняется условие *team has the puck* (наша команда владеет шайбой).

Как видно из изображения выше, атака делится на 4 стадии: i*dle, attack, stealPuck*, и *pursuePuck.* Одно из них, состояние *idle*, уже было реализовано в прошлой статье. Оно является отправной точкой, с которой и начинается процесс атаки. Затем спортсмены переходят к:

- *attack*, если команда уже владеет шайбой;
- *stealPuck*, если соперники владеют шайбой
- *pursuePuck*, если шайба никому не принадлежит и попросту скользит по льду

Давайте более подробно ознакомимся с этими состояниями.

attack описывает наступательные действия. Игрок, владеющий шайбой и именуемый лидером (*leader*), пытается продвинуться к вражеским воротам. Партнеры по команде тоже не стоят на месте. Они продвигаются вперед, чтобы оказать поддержку своему лидеру.

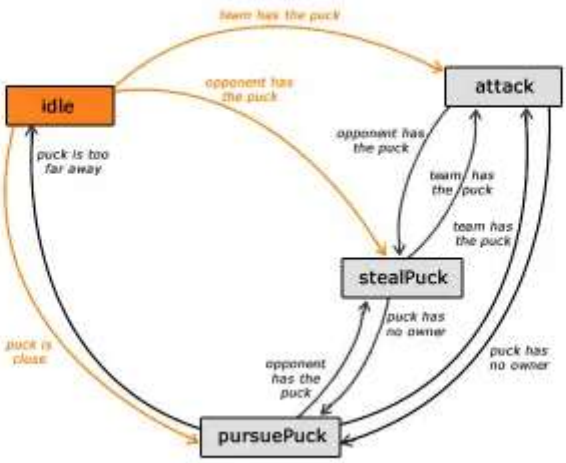
Состояние *stealPuck* является чем-то средним между атакой и защитой. Заключается оно в том, что игрок акцентируется на сопернике, владеющим шайбой и пытается отобрать ее. В случае успеха команда переходит к атакующим действиям.

Ну и наконец, *pursuePuck*. Это состояние не относится ни к атаке, ни к защите и помогает игрокам следовать за шайбой в тех случаях, когда оно никому не принадлежит.

Обновим состояние idle

Когда мы писали функцию *idle*, у нас еще не было состояний, в которые мы могли перевести хоккеиста. Теперь ситуация изменилась, и, поскольку *idle* является отправной точкой почти для всех состояний, давайте обновим ее.

На изображении ниже вы можете наблюдать переходные состояния, в которые будут попадать игроки из *idle*:



Если команда спортсмена имеет при себе шайбу, то нужно переходить к атаке. Если же шайба у противников, то нужно переходить к состоянию *stealPuck*. В случае, когда шайба бесхозная и игрок находиться достаточно близко к ней, он переходит к состоянию *pursuePuck*. Вот и вся логика. А теперь приведем код:

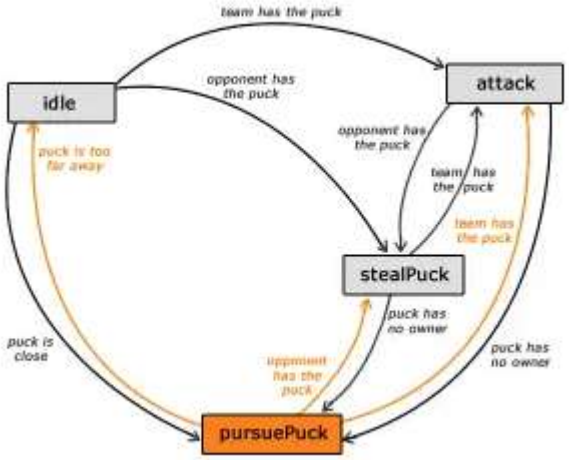
```
class Athlete {
    // (...)
    private function idle() :void {
        var aPuck :Puck = getPuck();

        stopAndlookAt(aPuck);

        // это простой хак, нужен для тестирования ИИ
```

```
        if (mStandStill) return;

        // есть ли у шайбы владелец?
        if (getPuckOwner() != null) {
            // да, есть!
            mBrain.popState();

            if (doesMyTeamHaveThePuck()) {
                // шайба у моей команды, время атаковать!
                mBrain.pushState(attack);
            } else {
                // шайба у соперников, надо ее отобрать
                mBrain.pushState(stealPuck);
            }
        } else if (distance(this, aPuck) < 150) {
            // шайба попросту катится по катку, надо подобрать ее
            mBrain.popState();
            mBrain.pushState(pursuePuck);
        }
    }

    private function attack() :void {
    }

    private function stealPuck() :void {
    }

    private function pursuePuck() :void {
    }
     }
```

Как вы могли заметить, методы *attack()*, *stealPuck()* и *pursuePuck()* объявлены, но не реализованы. Давайте реализуем их.

Следование за шайбой – pursuePuck

Это состояние будет активно, когда шайба не будет принадлежать никому (начало игры тоже попадает под это описание, поскольку шайба без владельца будет находиться в центре поля).



Хоть мы и должны бежать к никому не принадлежащей шайбе, нельзя забывать об имитации реальных действий. В настоящем хоккее игроки, увидев такую шайбу, не погонятся всей командой за ней. Из стратегических соображений, погнаться за ней должен тот, кто ближе всех к ней. Все остальные игроки должны помогать ему.

Если же шайбу кто-то уже взял, то нам следует перейти либо к атаке, либо к отбору. Все это выполняется в зависимости от того, кто же умудрился захватить шайбу – игрок нашей команды или соперник.

А вот код состояния *pursuePuck*:

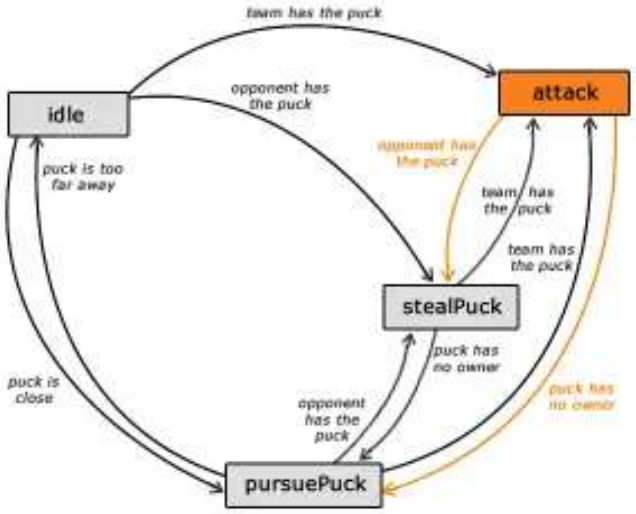
```
class Athlete {
    // (...)
    private function pursuePuck() :void {
        var aPuck :Puck = getPuck();

        mBoid.steering = mBoid.steering + mBoid.separation();

        if (distance(this, aPuck) > 150) {
            // шайба слишком далеко, быть может,
            // кто-то из моей команды ближе и он ее подберет
            mBrain.popState();
            mBrain.pushState(idle);
        } else {
            // шайба рядом, надо попытаться подобрать ее
            if (aPuck.owner == null) {
                // никто не подобрал шайбу, это наш шанс
                mBoid.steering = mBoid.steering + mBoid.seek(aPuck.position);

            } else {
                // кто-то уже подобрал шайбу;
                // если шайба принадлежит нашей команде, надо переходить к атаке,
```

```
                // иначе пытаемся отобрать ее у соперника
                mBrain.popState();
                mBrain.pushState(doesMyTeamHaveThePuck() ? attack : stealPuck);
            }
        }
    }
    }
```

Обратите внимание на 6 строчку из приведенного выше кода. Она отвечает за то, чтобы игроки не оставались слишком близко друг к другу во время активного состояния *pursuePuck*, поскольку это будет выглядеть не естественно.

Проведение атаки

После того, как игрок получил шайбу, он должен стремиться к воротам соперника и забить гол. Это и будет целью состояния *attack*.

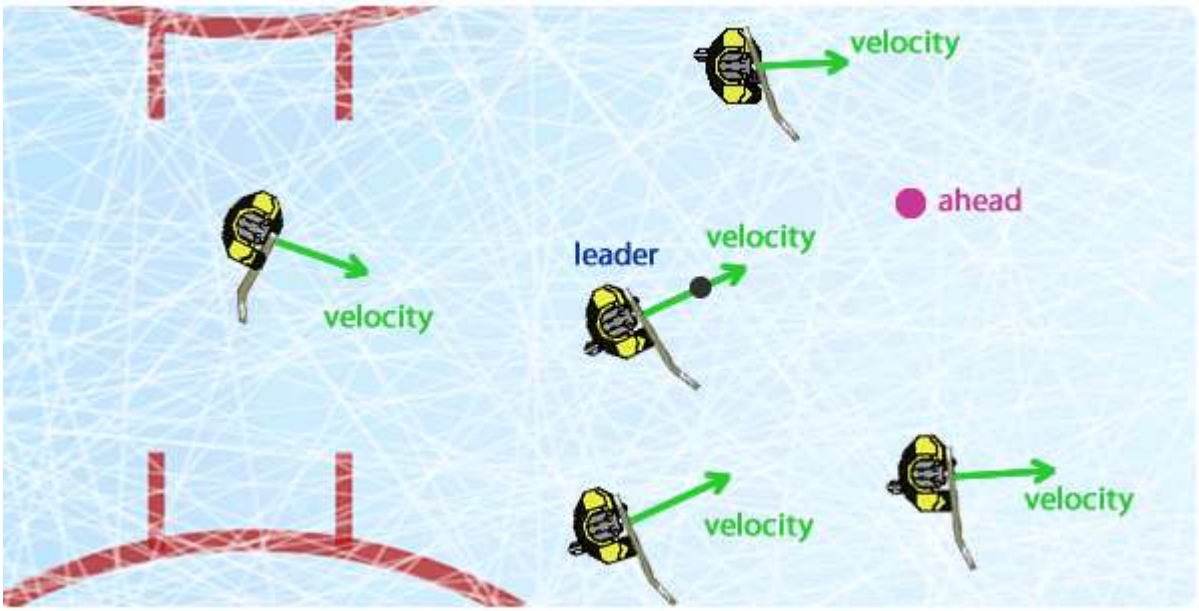


Атака имеет два переходных состояния: *pursuePuck* и *stealPuck*. Игроки, будучи в состоянии *attack*, должны будут бежать к воротам противников. Давайте реализуем это:

```
class Athlete {
    // (...)
    private function attack() :void {
        var aPuckOwner :Athlete = getPuckOwner();

        // есть ли у шайбы владелец?
        if (aPuckOwner != null) {
            // да, есть. давайте проверим, кто же это
            if (doesMyTeamHaveThePuck()) {
                if (amIThePuckOwner()) {
                    // шайба у моей команды, более того - она у меня
```

```
                    // пытаемся прорваться к вражеским воротам
                    mBoid.steering = mBoid.steering +
mBoid.seek(getOpponentGoalPosition());

                } else {
                    // шайба у моей команды, но не у меня;
                    // надо бежать за лидером и помогать ему
                    mBoid.steering = mBoid.steering +
mBoid.followLeader(aPuckOwner.boid);
                    mBoid.steering = mBoid.steering + mBoid.separation();
                }
            } else {
                // шайба у противников, останавливаем атаку
                // и пытаемся отобрать шайбу
                mBrain.popState();
                mBrain.pushState(stealPuck);
            }
        } else {
            // шайба никому не принадлежит;
            // надо подобрать ее
            mBrain.popState();
            mBrain.pushState(pursuePuck);
        }
    }
     }
```

Разберем приведенный код. Если у шайбы есть владелец, причем это игрок сам, то он несется к воротам противника (строка 13). Если же шайба принадлежит команде игрока, но он не является лидером, то следует бежать за нашим форвардом и помогать ему (строка 18). Обратите внимание на метод *mBoid.separation()* в строке 19, который мы использовали чуть ранее. На изображении ниже вы можете увидеть, как игроки помогают своему лидеру.



Обратите внимание также на то, что игроки без шайбы следуют не точно за лидером, а в точку перед ним. Это позволит избежать образования толпы, а также даст лидеру возможность совершить пас, если на его пути возникнуть какие-либо препятствия.

В случае, когда шайбой владеют противники, мы переводим игрока в состояние *stealPuck*. А если она никому не принадлежит, то игрок должен ее подобрать. Это делается переводом в состояние *pursuePuck*.

Улучшение поддержки атаки

Текущая реализация атаки находится на достаточно хорошем уровне исполнения. Однако недостаток все же имеется. Проявляется он в том случае, когда лидер оказывается дальше от ворот соперников, чем кто-либо другой из его команды.

Присмотритесь на последний наш результат. В случае, когда какой-нибудь игрок *A* находится к чужим воротам ближе, чем лидер, то игрок *A* начинает вести себя немного неестественно, пока не окажется позади форварда.

Этот недостаток с легкостью можно исправить путем проверки положения игрока – находится он позади лидера или опережает его:

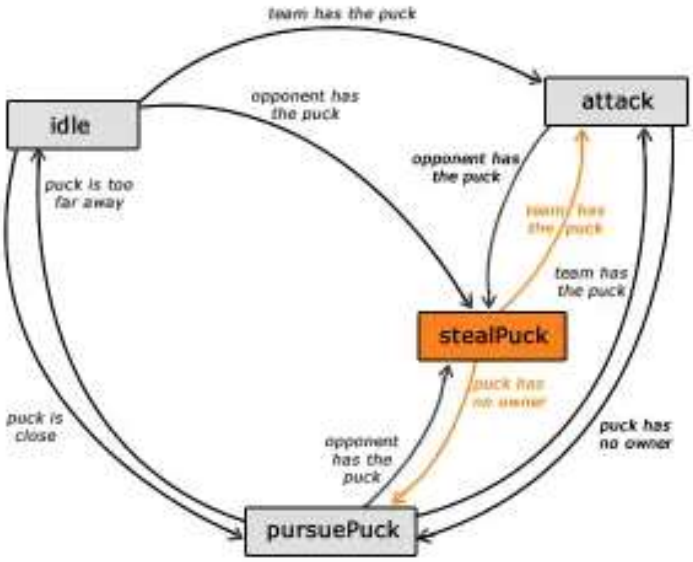
```
class Athlete {
    // (...)
    private function isAheadOfMe(theBoid :Boid) :Boolean {
        var aTargetDistance :Number = distance(getOpponentGoalPosition(), theBoid);
        var aMyDistance :Number = distance(getOpponentGoalPosition(), mBoid.position);

        return aTargetDistance <= aMyDistance;
    }

    private function attack() :void {
```

```
        var aPuckOwner :Athlete = getPuckOwner();

        // есть ли у шайбы владелец?
        if (aPuckOwner != null) {
            // да, есть
            if (doesMyTeamHaveThePuck()) {
                if (amIThePuckOwner()) {
                    // шайба у моей команды, более того - она у меня
                    // пытаемся прорваться к вражеским воротам
                    mBoid.steering = mBoid.steering +
mBoid.seek(getOpponentGoalPosition());

                } else {
                    // шайба у моей команды, но не у меня; а лидер впереди меня?
                    if (isAheadOfMe(aPuckOwner.boid)) {
                        // да, он впереди; побегу за ним,
                        // чтобы помочь ему
                        mBoid.steering = mBoid.steering +
mBoid.followLeader(aPuckOwner.boid);
                        mBoid.steering = mBoid.steering + mBoid.separation();
                    } else {
                        // нет, лидер позади меня; давайте добавим "разделение"
(separation),
                        // чтобы предотвратить давку
                        mBoid.steering = mBoid.steering + mBoid.separation();
                    }
                }
            } else {
                // шайба у противников, останавливаем атаку
                // и пытаемся отобрать шайбу
                mBrain.popState();
                mBrain.pushState(stealPuck);
            }
        } else {
            // шайба никому не принадлежит;
            // надо подобрать ее
            mBrain.popState();
            mBrain.pushState(pursuePuck);
        }
    }
     }
```

А теперь взглянем на полученный результат. Игроки ведут себя на порядок реалистичнее!

Отбор шайбы

Последним состоянием атаки является *stealPuck*, который становится активным, когда шайба принадлежит соперникам. Цель – отобрать шайбу для проведения собственной атаки.



А теперь перейдем к реализации:

```
class Athlete {
    // (...)
    private function stealPuck() :void {
        // есть ли у шайбы владелец?
        if (getPuckOwner() != null) {
            // да, есть
            if (doesMyTeamHaveThePuck()) {
                // шайба у моей команды
                // время перейти в атаку
                mBrain.popState();
                mBrain.pushState(attack);
            } else {
                // шайба у противников
                var aOpponentLeader :Athlete = getPuckOwner();

                // попробуем отобрать шайбу у врага,
                // но попытаемся предсказать его положение
                // и перехватим его там
                mBoid.steering = mBoid.steering + mBoid.pursuit(aOpponentLeader.boid);
                mBoid.steering = mBoid.steering + mBoid.separation();
            }
        } else {
            // шайба никому не принадлежит;
            // надо подобрать ее
```

```
            mBrain.popState();
            mBrain.pushState(pursuePuck);
        }
    }
     }
```

Алгоритм предельно прост. Если шайба принадлежит команде игрока, то мы переходим к состоянию *attack* (строка 11). Если же она у команды противника, то игрок попытается перехватить ее. Однако, обратите внимание. Мы не можем передавать нашему игроку координаты противника, поскольку это приведет к тому, что наш хоккеист будет просто его преследовать. Именно поэтому игрок будет предсказывать то положение, в котором окажется противник в ближайшее время, и будет стремиться перехватить своего соперника. Все это реализовано при помощи «поведения преследования» (*pursue behavior*) в строке 19.

Улучшение отбора шайбы

Мы написали довольно неплохой метод отбора шайбы. Но в настоящей игре хоккеисты не поступали бы так, как они ведут себя у нас. Отобрать шайбу они пытаются всей командой, но это совсем неестественно и может привести к образованию толпы. Мы можем исправить этот недочет путем проверки расстояния от игрока до лидера соперников.

```
class Athlete {
    // (...)
    private function stealPuck() :void {
        // есть ли у шайбы владелец?
        if (getPuckOwner() != null) {
            // да, есть
            if (doesMyTeamHaveThePuck()) {
                // шайба у моей команды
                // время перейти в атаку
                mBrain.popState();
                mBrain.pushState(attack);
            } else {
                // шайба у противников
                var aOpponentLeader :Athlete = getPuckOwner();

                // лидер соперников близок ко мне?
                if (distance(aOpponentLeader, this) < 150) {
                    // да, он рядом; надо предугадать его положение и
                    // перехватить его
                    mBoid.steering =
mBoid.steering.add(mBoid.pursuit(aOpponentLeader.boid));
                    mBoid.steering = mBoid.steering.add(mBoid.separation(50));
```

```
                } else {
                    // нет, он слишком далеко; в скором времени на этом
                    // месте мы будем переходить в защиту
                    // TODO: mBrain.popState();
                    // TODO: mBrain.pushState(defend);
                }
            }
        } else {
            // шайба никому не принадлежит;
            // надо подобрать ее
            mBrain.popState();
            mBrain.pushState(pursuePuck);
        }
    }
}
```

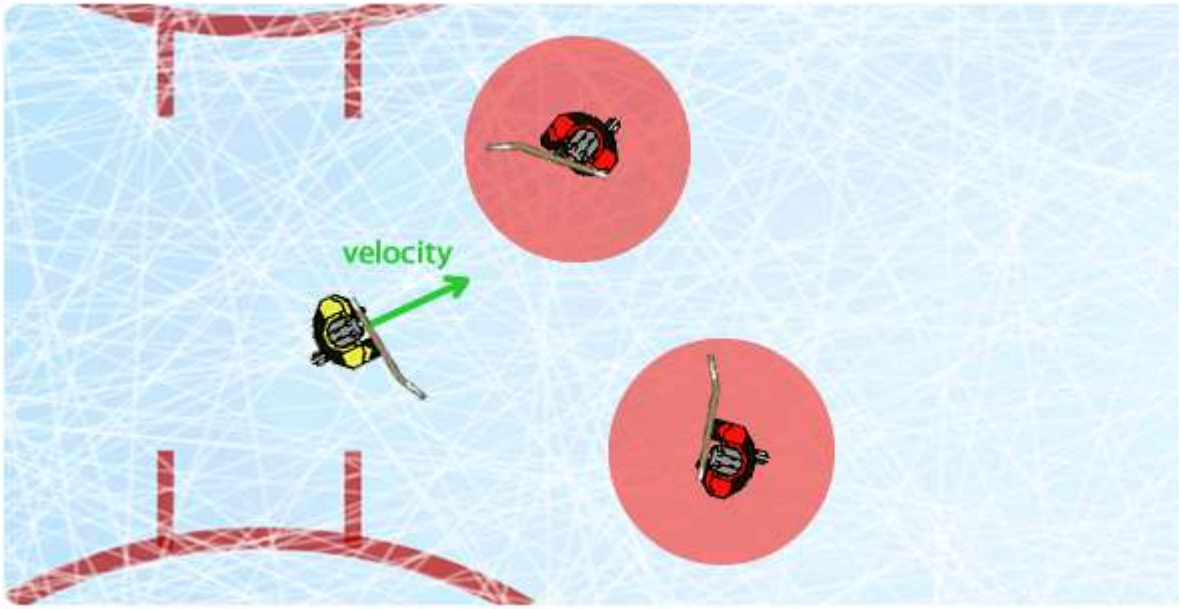
Теперь наши игроки не будут бежать сломя голову к противнику, пытаясь отобрать у него шайбу. Побегут только те, расстояние от которых до соперника не превышает 150.

Все оставшиеся игроки останутся на своих местах, поскольку они слишком далеко. Целесообразно их перевести в состояние защиты, однако, мы пока его не реализовали.

Избежание вражеских защитников

В нашей игре не хватает еще одного трюка. В настоящей игре хоккеисты при получении шайбы не несутся сломя голову к воротам противника. Они пытаются каким-то образом маневрировать между игроками вражеской команды. Давайте реализуем это и в нашей игре.

Мы можем использовать «избежание коллизий» (*collision avoidance*) для того, чтобы игроки могли уклоняться от противников. Вот, как это будет выглядеть:



Для реализации уклонений нам следует добавить лишь одну строку (14 строка) в наш код:

```
class Athlete {
    // (...)
    private function attack() :void {
        var aPuckOwner :Athlete = getPuckOwner();

        // есть ли у шайбы владелец?
        if (aPuckOwner != null) {
            // да, есть
            if (doesMyTeamHaveThePuck()) {
                if (amIThePuckOwner()) {
                    // шайба у моей команды, более того - она у меня
                    // пытаемся прорваться к вражеским воротам и уклоняемся от
врагов
                    mBoid.steering = mBoid.steering +
mBoid.seek(getOpponentGoalPosition());
                    mBoid.steering = mBoid.steering +
mBoid.collisionAvoidance(getOpponentTeam().members);

                } else {
                    // шайба у моей команды, но не у меня; а лидер впереди меня?
                    if (isAheadOfMe(aPuckOwner.boid)) {
                        // да, он впереди; побегу за ним,
                        // чтобы помочь ему
                        mBoid.steering = mBoid.steering +
mBoid.followLeader(aPuckOwner.boid);
                        mBoid.steering = mBoid.steering + mBoid.separation();
                    } else {
```

```
                    // нет, лидер позади меня; давайте добавим "разделение"
(separation),
                    // чтобы предотвратить давку
                    mBoid.steering = mBoid.steering + mBoid.separation();
                }
            }
        } else {
            // шайба у противников, останавливаем атаку
            // и пытаемся отобрать шайбу
            mBrain.popState();
            mBrain.pushState(stealPuck);
        }
    } else {
        // шайба никому не принадлежит;
        // надо подобрать ее
        mBrain.popState();
        mBrain.pushState(pursuePuck);
    }
  }
 }
```

Несколько слов о защите

Организация защиты в хоккее – довольно сложный процесс, который заключается не в простом столпотворении возле ворот, дабы не позволить противнику закатить шайбу в ворота. Не дать соперникам забить гол – лишь одна из задач защитной линии.

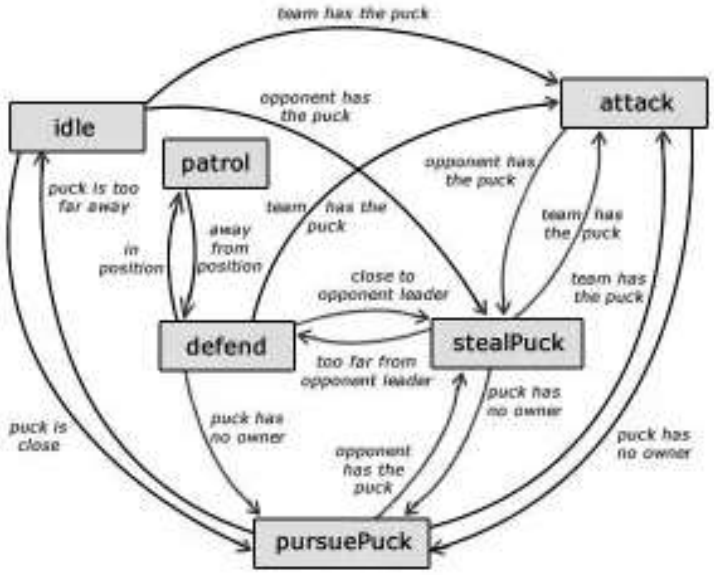
Стоить помнить, что если мы уйдем в глухую оборону, то все наши игроки станут всего лишь препятствиями своим соперникам и не более. Противник попытается прорваться сквозь нашу оборону, и забитый ими гол в наши ворота станет лишь вопросом времени.

Процесс защиты есть комбинация оборонительных и наступательных действий. Лучший способ прекратить атаку противника – отобрать шайбу, защищаясь, и сразу же начать свою атаку. Сразу вспоминается выражение «Лучшая защита – это наступление». Все это может быть немного запутанным, однако, в такой тактике есть смысл.

Задача спортсмена – защищать свою команду, не позволяя противнику забить шайбу. Он должен отобрать шайбу у игрока противоположной команды или же перехватить ее в то время, когда противники пасуют друг другу. Это мы и попробуем реализовать – оборонительная тактика с атакующими элементами.

Сочетание нападения и защиты

Для того, чтобы реализовать оборонительные действия, содержащие в себе несколько атакующих элементов, мы добавим еще два состояния для ИИ:



Состояние *defend* будет фундаментом, основой всего процесса обороны. Находясь в этом режиме, игроки будут двигаться к бортику, попутно пытаясь отобрать у противника шайбу.

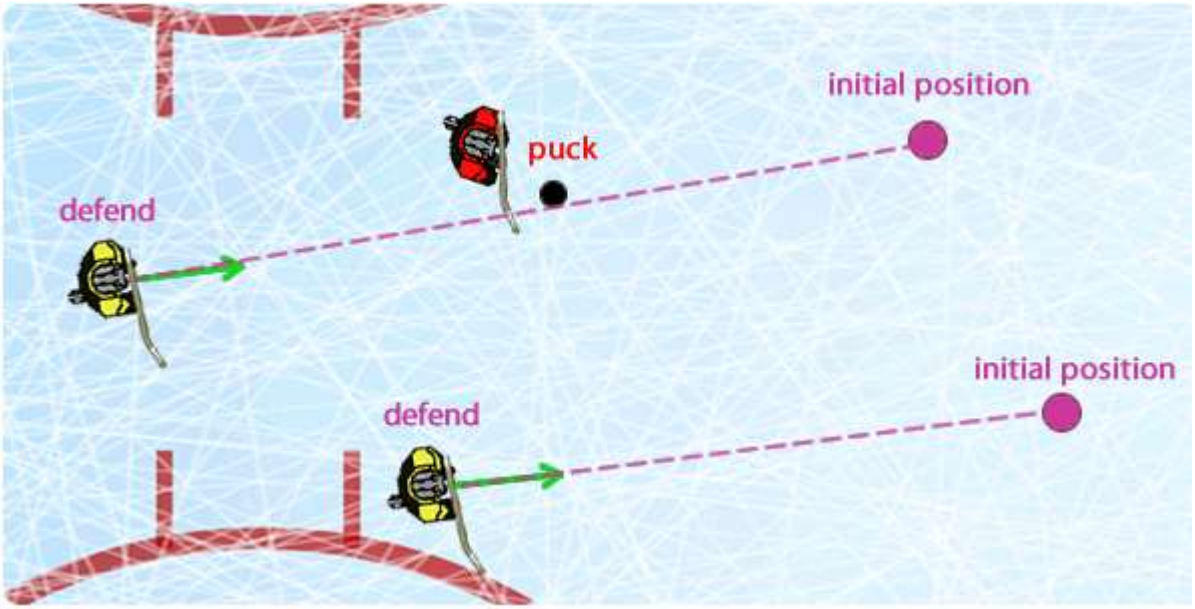
Состояние *patrol* является дополнением к *defend*. Оно будет выводить игроков из состояния покоя, будет держать их в постоянном движении и патрулировании некоторых зон, что приведет к лучшему результату.

Описание состояния defend

Состояние *defend* предельно простое. Когда оно активно, каждый спортсмен будет двигаться к своему изначальному положению на катке. Мы уже использовали эту координату, описанную в поле *mInitialPosition* класса *Athlete*, в методе *prepareForMatch*.

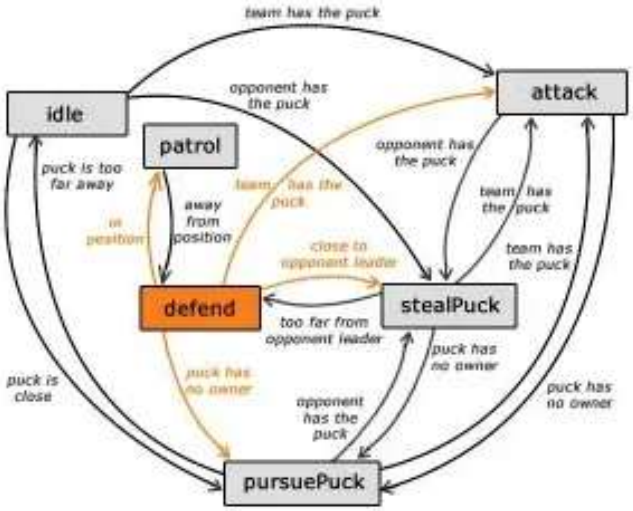
Во время движения к своей стартовой позиции игрок будет будет пытаться выполнить некоторые атакующие действия, если это возможно. Например, как только соперник с шайбой окажется рядом с нашим игроком, состояние *defend* будет заменено на более подходящее, скажем, на *stealPuck*.

Таким образом, находясь в состоянии атаки, игроки перемещаются по всем катку. Переключившись на защиту, спортсмены возвращаются к своей стартовой позиции (одновременно пытаясь отобрать шайбу у противника, если это возможно) и охватывают внушительную территорию, тем самым, обеспечивая неплохую тактику обороны. Более наглядно этот процесс вы сможете наблюдать ниже:



Реализация состояния defend

defend имеет четыре переходных состояния:



Три из них (*team has the puck, close to opponent leader* и *puck has no owner*) отвечают за атакующие действия игроков. Состояние *in position* будет срабатывать тогда, когда игрок вернулся в свою начальную позицию.

Первым делом мы реализуем перемещение нашего игрока к своей начальной позиции. Т.к. хоккеист при достижении конечной точки замедляется, то использование рулевого поведения здесь подойдет идеально:

```
class Athlete {
    // (...)

    private function defend() :void {
        var aPuckOwner :Athlete = getPuckOwner();

        // плавно перемещаемся к стартовой позиции
```

```
        mBoid.steering = mBoid.steering + mBoid.arrive(mInitialPosition);

        // есть ли у шайбы владелец?
        if (aPuckOwner != null) {
            // да, есть
            if (doesMyTeamHasThePuck()) {
                // шайба у моей команды, время перейти в атаку
                mBrain.popState();
                mBrain.pushState(attack);

            } else if (Utils.distance(aPuckOwner, this) < 150) {
                // шайба у противника, он рядом, поэтому
                // пытаемся отобрать шайбу
                mBrain.popState();
                mBrain.pushState(stealPuck);
            }
        } else {
            // шайба никому не принадлежит; нет смысла переходить в защиту
            // надо подобрать шайбу
            mBrain.popState();
            mBrain.pushState(pursuePuck);
        }
    }

    // (...)
     }
```

Пока активно состояние *defend*, метод *arrive* создаст силу, которая будет толкать игрока к его начальной позиции (*mInitialPosition*). Затем мы проверяем, есть ли у кого-нибудь шайба, а если есть, то у кого: у игрока нашей команды или противника. Также мы проверяем расстояние между нашим игроком и противником с шайбой. В зависимости от этих данных мы переводим игрока в подходящее состояние.

Если у шайбы нет владельца, то вероятнее всего она попросту катится по катку. В таком случае мы посылаем ИИ состояние *pursuePuck*. Если же владельцем шайбы является игрок нашей команды, значит мы можем переходить в атаку (состояние *attack*). Ну и наконец, в случае, если владельцем шайбы является соперник, находящийся на расстоянии атаки, то мы смело переходим к состоянию *stealPuck* и пытаемся отобрать шайбу.

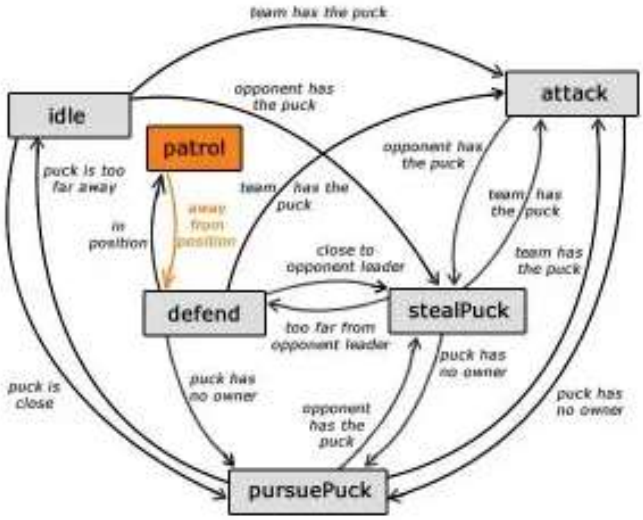
В результате, мы получаем команду, способную защитить свои ворота. К тому же, она способна плавно перейти из защиты в атаку.

Патрулирование

Тот уровень защиты, который демонстрируют игроки на текущий момент, уже является достаточно приемлемым. Однако, если присмотреться к нашему результату повнимательнее, то можно заметить, что игроки, достигая своей начальной позиции, останавливаются и не двигаются.

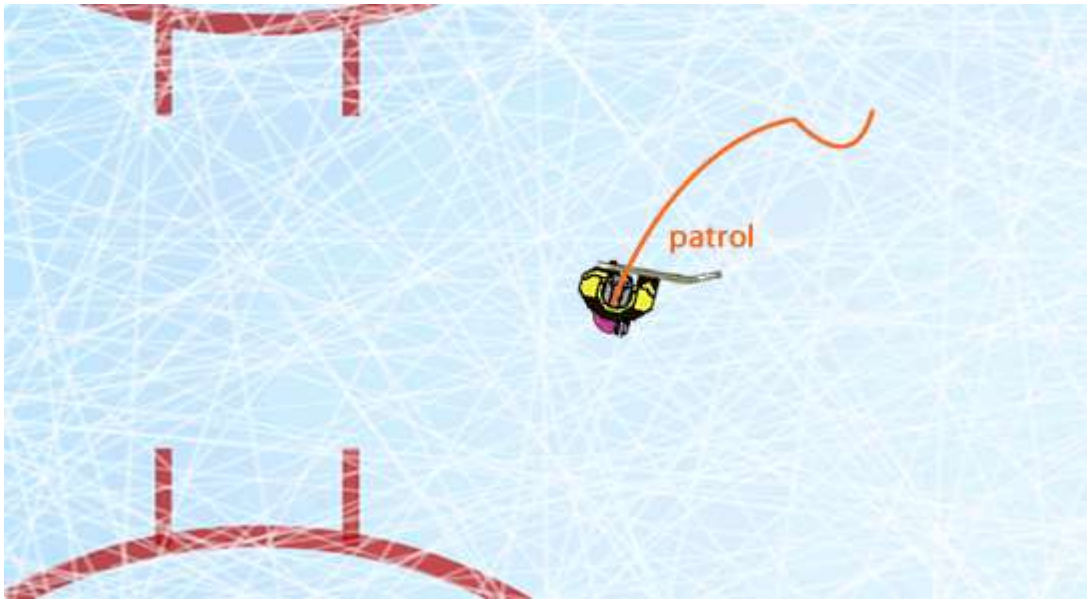
Обратите внимание, если игрок возвращается к своей позиции и по пути не встречает ни одного соперника с шайбой, то при достижении нужной точки он там и останется. Ровно до тех пор, пока мимо не будет проезжать соперник с шайбой или команда не получит шайбу в свое распоряжение.

Мы можем улучшить поведение игрока добавлением очередного состояния *patrol*, которое будет возникать тогда, когда игрок добрался до своей начальной позиции.



Реализация этого состояния предельно проста. Игрок, добравшийся до своей начальной позиции, будет рандомно двигаться в течение короткого времени. Такие действия создадут иллюзию, будто бы хоккеист защищает свою зону.

Когда расстояние между хоккеистом и его стартовой позицией больше 10, например, состояние *patrol* мы убираем и переводим игрока в состояние *defend*. Если же наш игрок в процессе защиты снова вернулся к своей начальной позиции, то мы вновь переводим его в состояние *patrol*. И так всегда. Вот, как должен выглядеть этот процесс:



А теперь реализуем данное состояние:

```
class Athlete {
    // (...)

    private function patrol() :void {
        mBoid.steering = mBoid.steering + mBoid.wander();

        // я далеко ушел от своей стартовой позиции?
        if (Utils.distance(mInitialPosition, this) > 10) {
            // да, надо остановить патрулирование и вернуться
            // к своей стартовой позиции
            mBrain.popState();
            mBrain.pushState(defend);
        }
    }

    // (...)
```

Собираем все вместе

Выше мы реализовали состояние *stealPuck*. Однако там возникала ситуация, когда дальнейшая попытка отобрать шайбу у противника теряла смысл. Нам следовало перейти в защиту, но на тот момент состояние *defend* не было реализовано.

Эта ситуация заключается в следующем: при попытке отобрать шайбу, расстояние до противника может быть слишком большим, и мы, скорее всего, его уже не догоним. В этом случае лучше перейти в защиту и надеяться, что наш товарищ по команде окажется к противнику ближе.

Давайте дополним код:

```
class Athlete {
    // (...)
```

```
    private function stealPuck() :void {
        // есть ли у шайбы владелец?
        if (getPuckOwner() != null) {
            // да, есть
            if (doesMyTeamHasThePuck()) {
                // шайба у моей команды
                // время переходить в атаку
                mBrain.popState();
                mBrain.pushState(attack);
            } else {
                // шайба у противников
                var aOpponentLeader :Athlete = getPuckOwner();

                // соперник с шайбой рядом со мной?
                if (Utils.distance(aOpponentLeader, this) < 150) {
                    // да, он достаточно близок;
                    // пытаемся предугадать его положение
                    // и стремимся перехватить его
                    mBoid.steering = mBoid.steering +
mBoid.pursuit(aOpponentLeader.boid);
                    mBoid.steering = mBoid.steering + mBoid.separation(50);

                } else {
                    // нет, соперник далеко; переходим в защиту
                    // и надеемся, что его перехватит другой игрок
                    mBrain.popState();
                    mBrain.pushState(defend);
                }
            }
        } else {
            // шайба никому не принадлежит; нет смысла ее у кого то отбирать
            // надо подобрать шайбу
            mBrain.popState();
            mBrain.pushState(pursuePuck);
        }
    }

    // (...)
    }
```

После того, как мы обновили состояние *stealPuck*, игроки начинают вести себя естественно, а поведение выглядит более осмысленным.

Задание к лабораторной работе!

Программно реализовать любую спортивную командную игру, пользуясь описанными выше классами.