演習２：集合の知性を設計する

(05) 05/20　(06) 05/27

A｜Unity環境の整備・簡単なルールの設計

(07) 06/10　(08) 06/17

B｜ボイドルール１・２・３の実装

(09) 06/24　(10) 07/01

C1｜集合知の解析

(12) 07/08　(13) 07/15

C２｜マイルール（ルール４）の実装・視点の操作

(14-15) 07/22

C２｜発表（One-Minute Movie）

演習２-B

ボイドルールの設計（ルール１）

親近行動 を設計します.

BoidManager.cs

```
//ルールの係数
public float c1 = 0.1f;
public float c2 = 5.0f;
public float c3 = 0.01f;
```

Update()

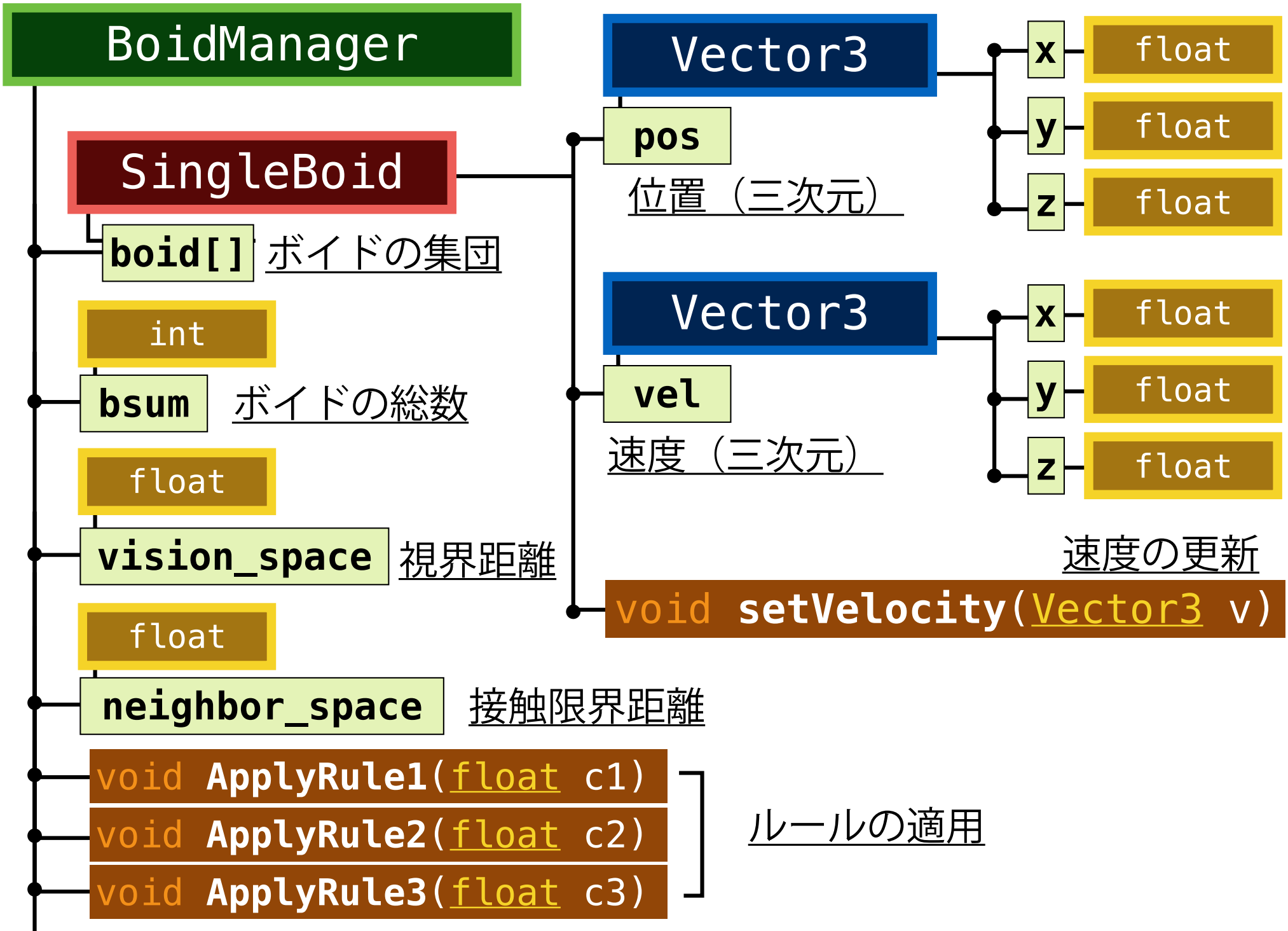
```
if (Input.GetMouseButton(0))
{
    ApplyRuleAttractor(new Vector3(0f, 100f, 0f));
}
```

マウスボタンを押すと, ApplyRuleAttractor が作動

```
//ルール1の適用
if (rule1) {
    ApplyRule1 ();
}
```

rule1 が true のときに, ApplyRule1を適用

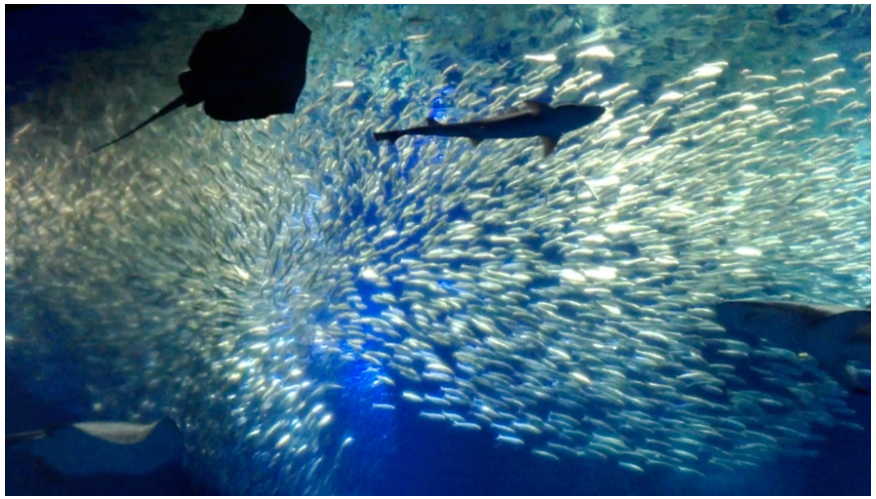
BoidManager / SingleBoidの関係（復習）

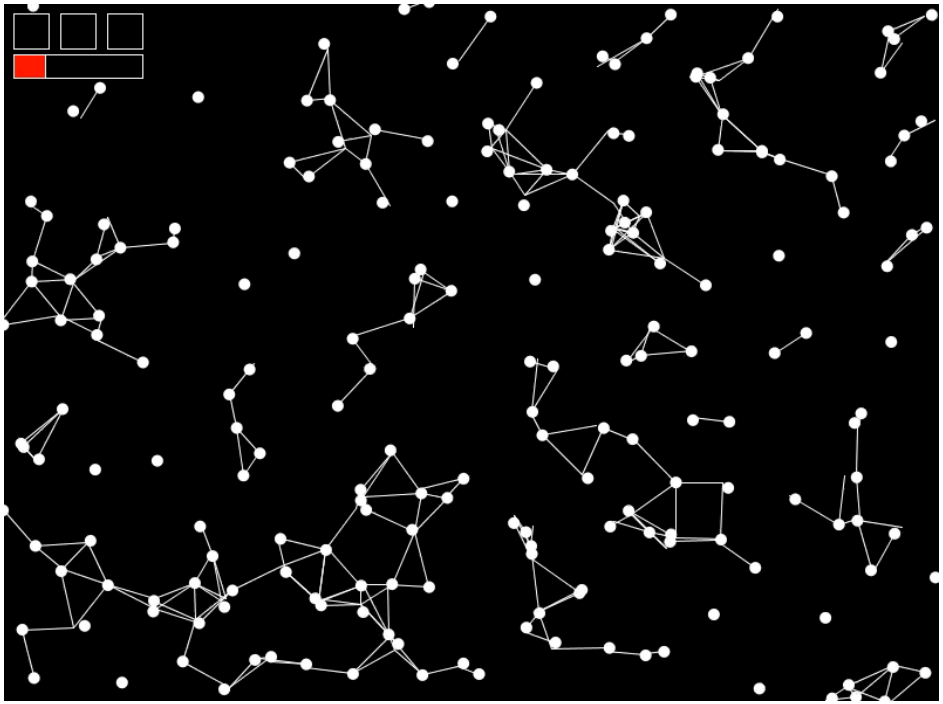


生命と生命のリズム（同調）

集団の同調

Cooperativeness / 群知能





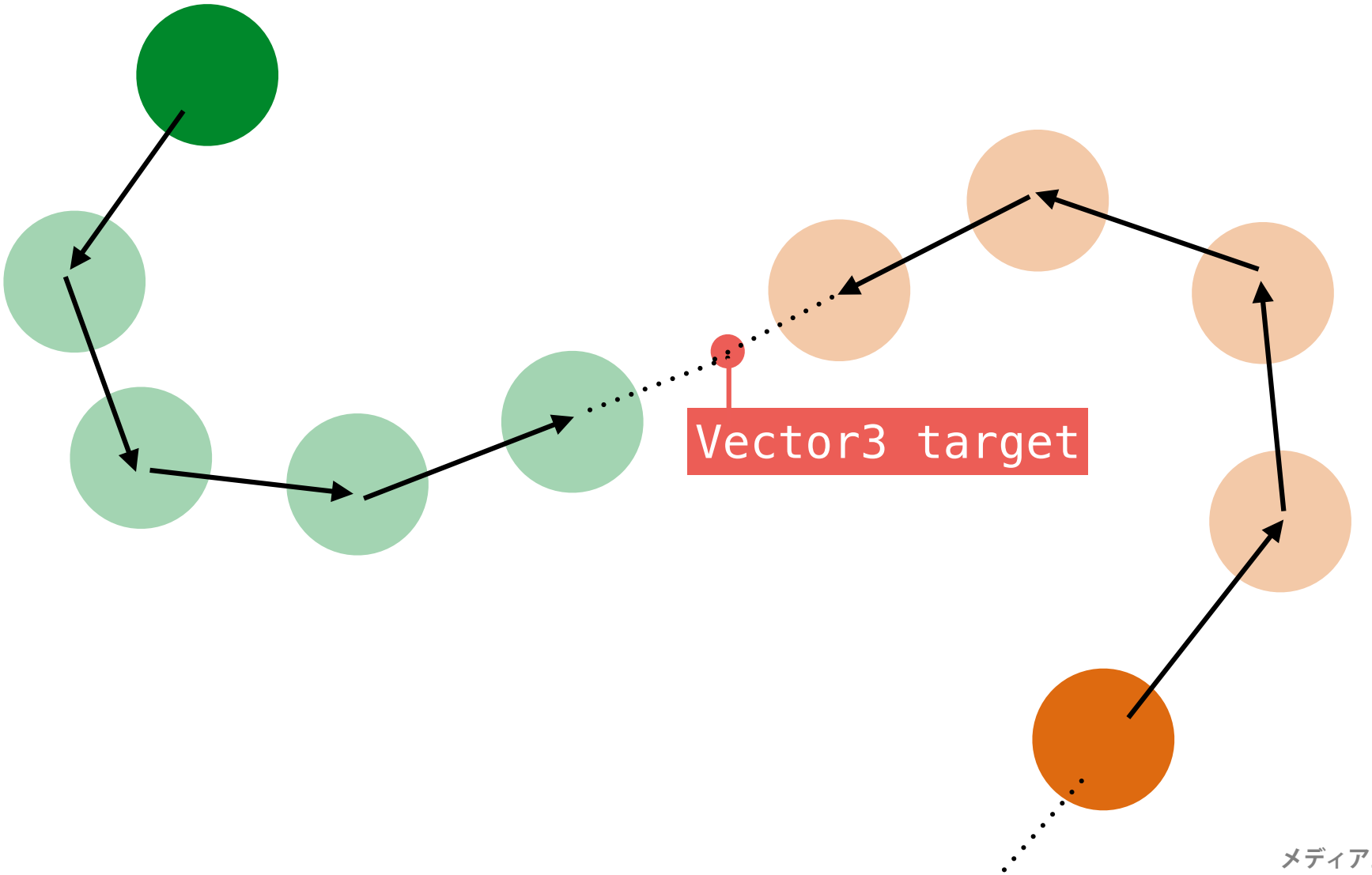
boids（ボイド; bird-like droids）

準備

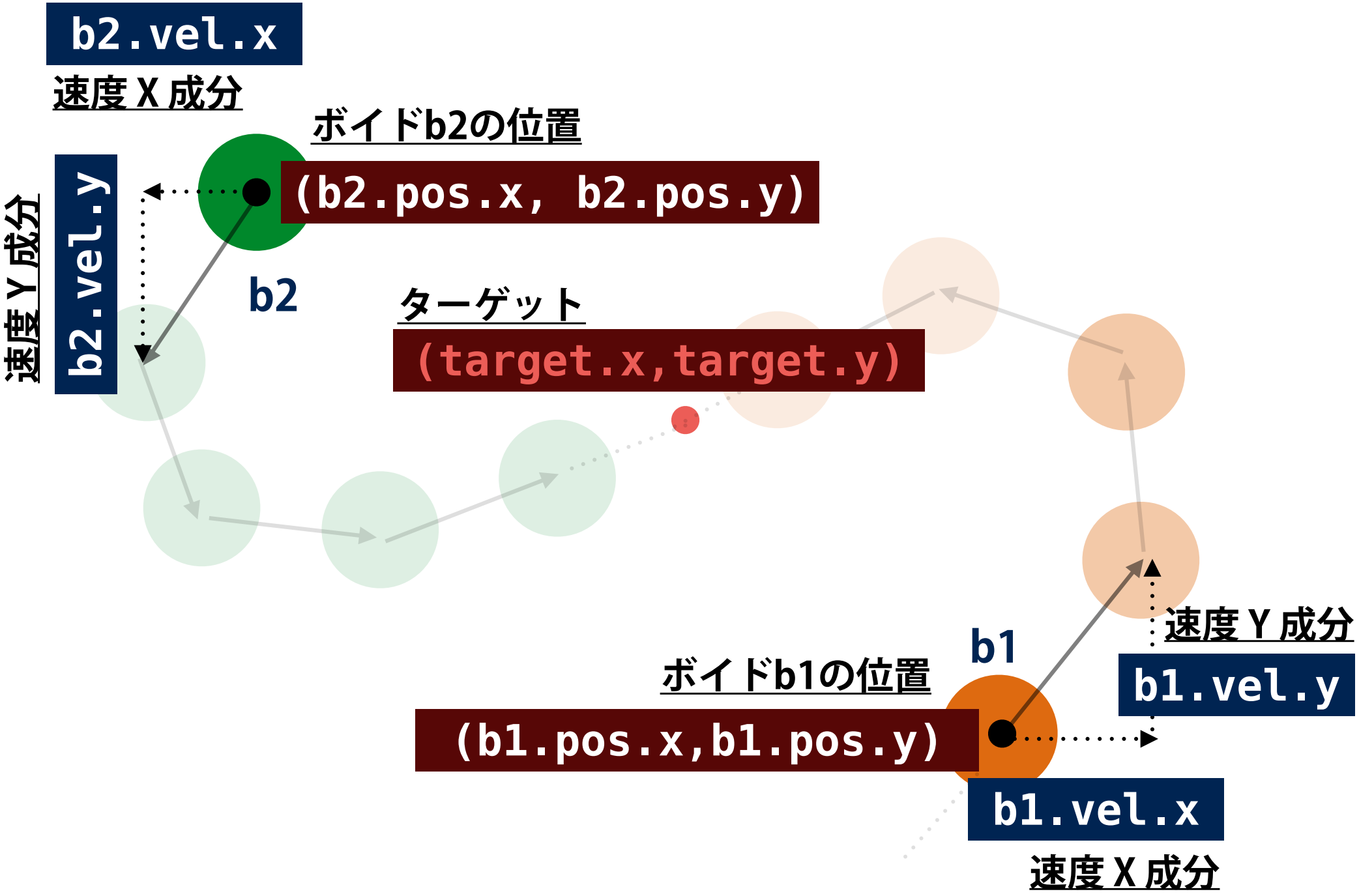
`BoidRule.cs` の関数 `void ApplyRuleAttractor(Vector3 target)`

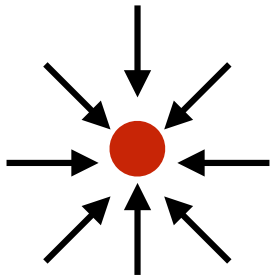
を定義し，

ボイドの速度を，引数で指定された位置へと徐々に修正するルールを追加してください．



準備（二次元で図示します.）



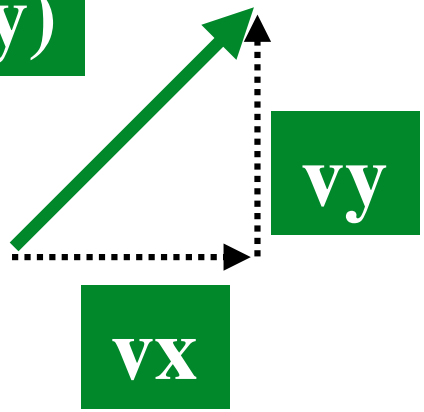


速度を，指定された点に（徐々に）向けるためのベクトル計算

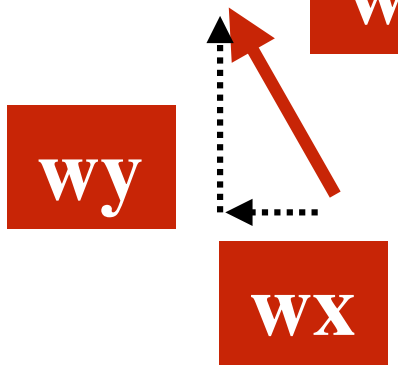
位置による引き込み（引力）

ベクトルの加算

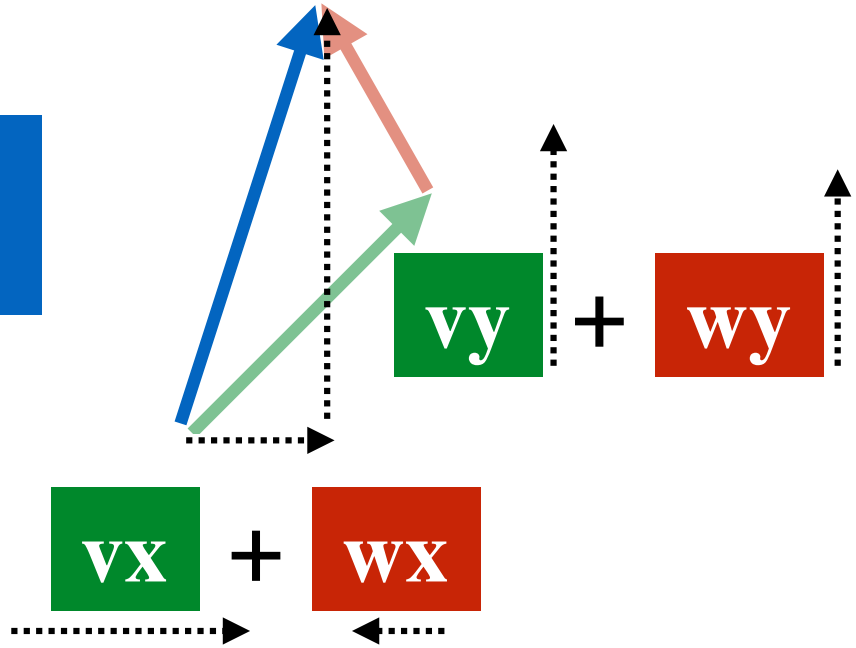
$v' = (vx, vy)$



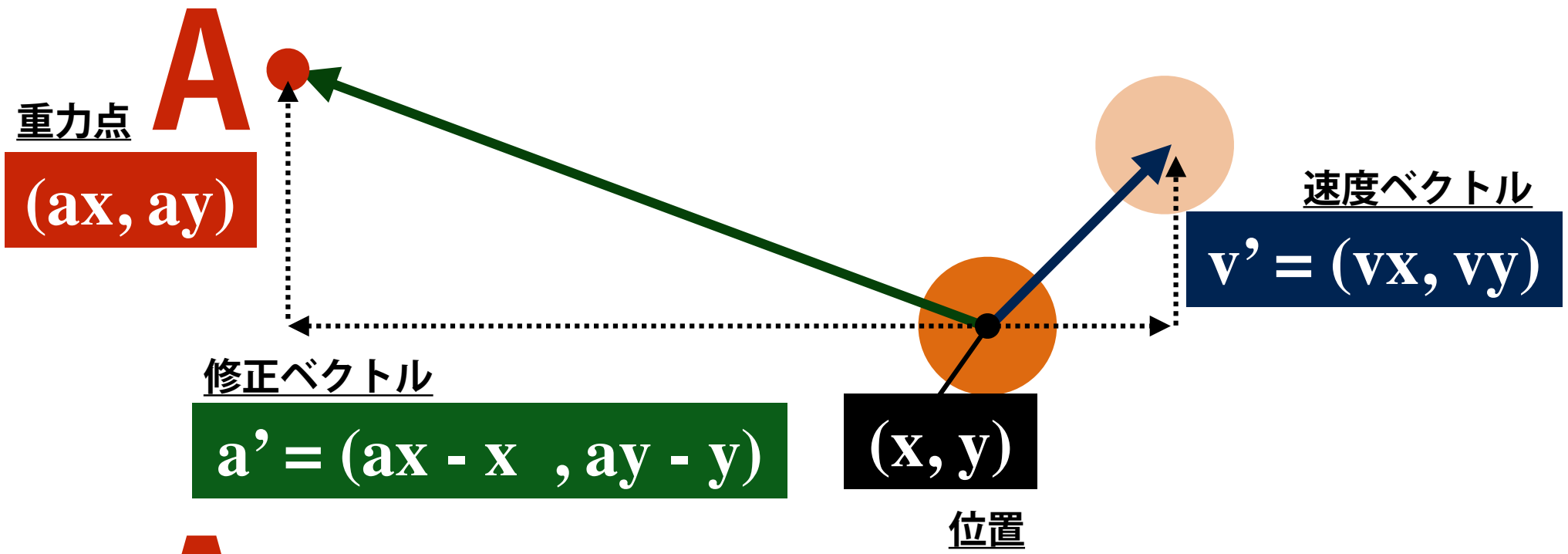
$w' = (wx, wy)$

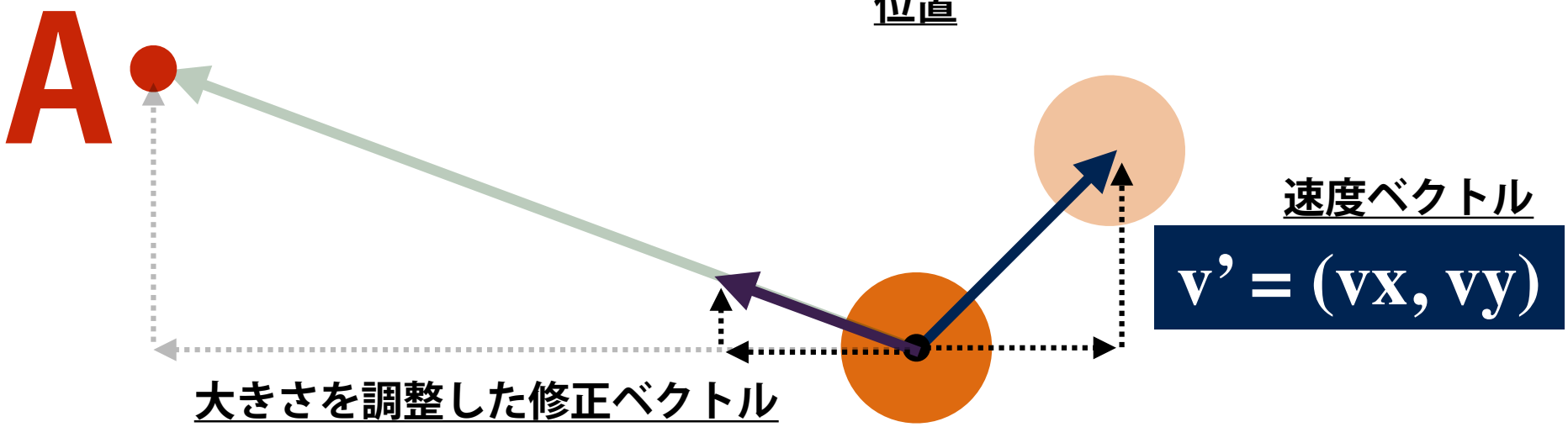


$v' + w' = (vx+wx, vy+wy)$

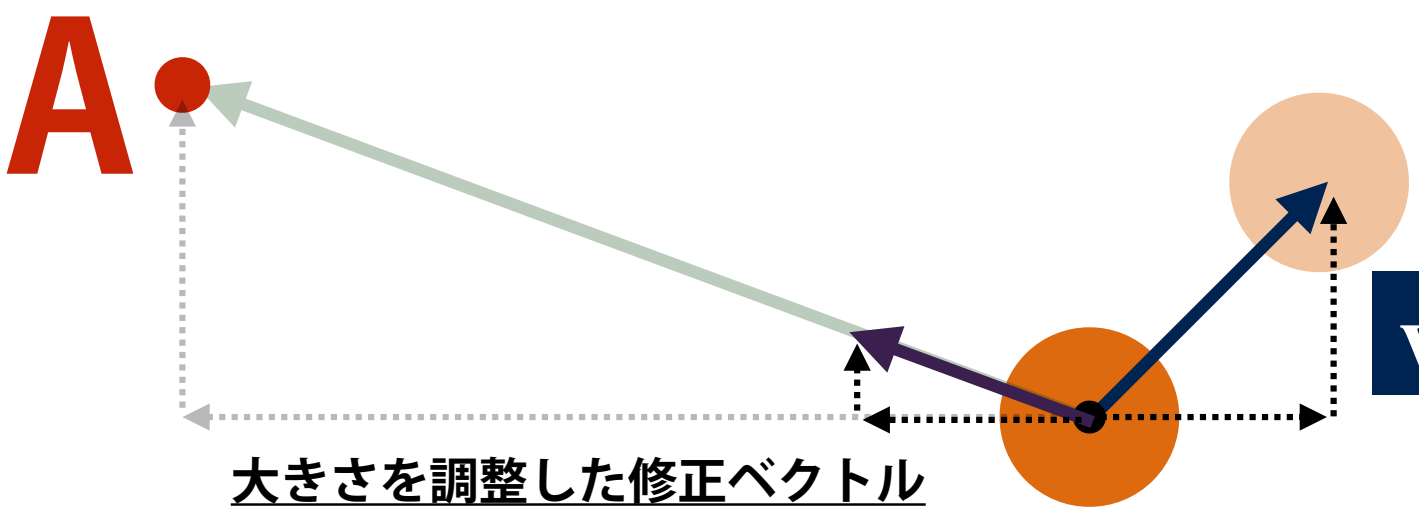


<速度 v'> を<点 A> の方向に修正する





ca' = (c1*(ax - x), c1*(ay - y))

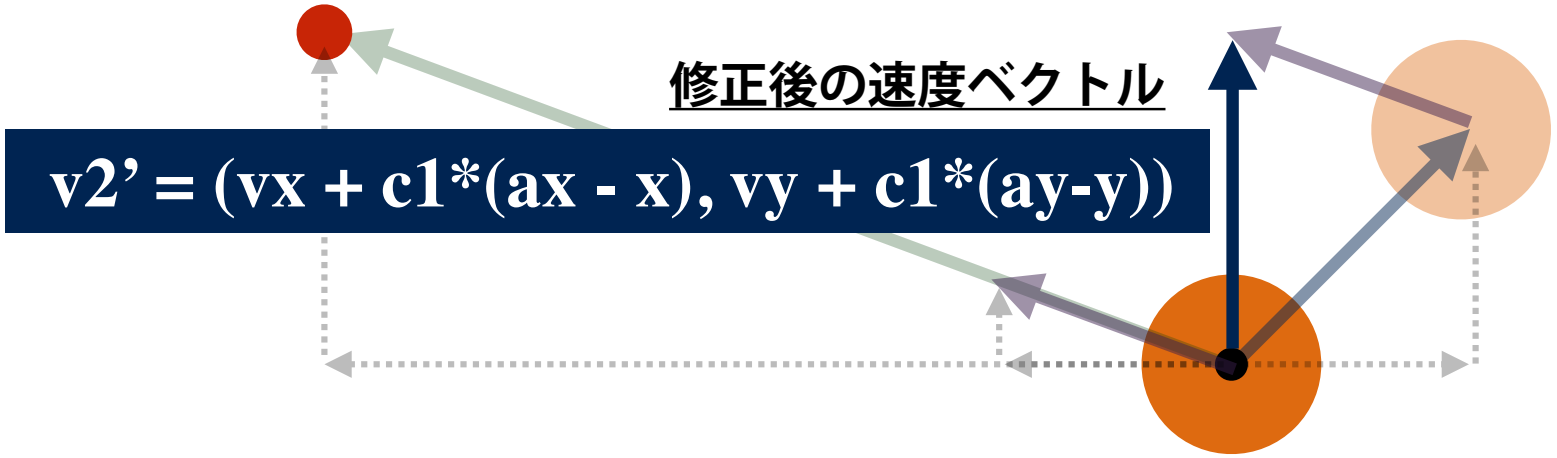


速度ベクトル

$v' = (vx, vy)$

大きさを調整した修正ベクトル

$ca' = (c1*(ax - x), c1*(ay - y))$



修正後の速度ベクトル

$v2' = (vx + c1*(ax - x), vy + c1*(ay-y))$

三次元表現でも同様

$v2' = (vx, vy, vz) + c1*(ax-x, ay-y, az-z)$

ヒント

```
void ApplyRuleAttractor(Vector3 target){

    for (int i = 0; i < bsum; i++) {

        Vector3 ipos = boid [i].pos;
        Vector3 ivel = boid [i].vel;


        float c = 0.1f;

        Vector3 cv = new Vector3 ();

        cv.x
        cv.y
        cv.z

        boid [i].setVelocity (          );
    }

}
```

ボイド i の位置と速度

引き込みの程度

修正ベクトルの計算

i 番目のボイドの速度ベクトルを 更新

BoidRule.cs

ヒント

```
void ApplyRuleAttractor(Vector3 target){

    for (int i = 0; i < bsum; i++) {

        Vector3 ipos = boid [i].pos;
        Vector3 ivel = boid [i].vel;


        float c = 0.1f;

        Vector3 cv = new Vector3 ();

        cv.x = c * (target.x - ipos.x);
        cv.y = c * (target.y - ipos.y);
        cv.z = c * (target.z - ipos.z);

        boid [i].setVelocity (ivel + cv);
    }

}
```

ボイド i の位置と速度

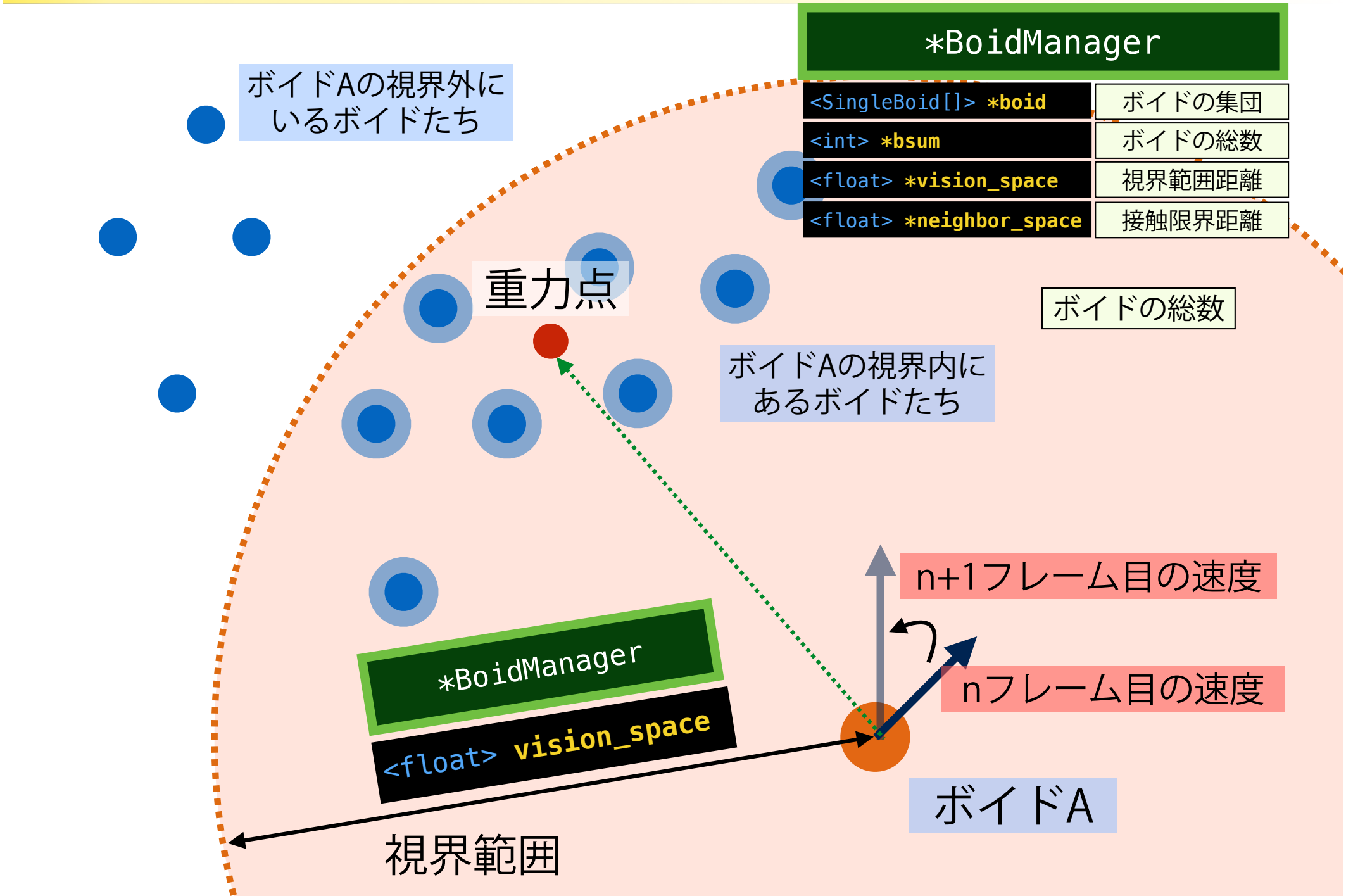
引き込みの程度

修正ベクトルの計算

i 番目のボイドの速度ベクトルを 更新

BoidRule.cs

結合ルールの実装


*BoidManager
<SingleBoid[]> *boid
ボイドの集団
<int> *bsum
ボイドの総数
<float> *vision_space
視界範囲距離
<float> *neighbor_space
接触限界距離
ボイドAの視界外にいるボイドたち
重力点
ボイドの総数
ボイドAの視界内にあるボイドたち
n+1フレーム目の速度
nフレーム目の速度
*BoidManager
<float> vision_space
ボイドA
視界範囲

結合ルールの実装

1. 重力点を視界内にある全てのボイドの位置の重心として求めます.

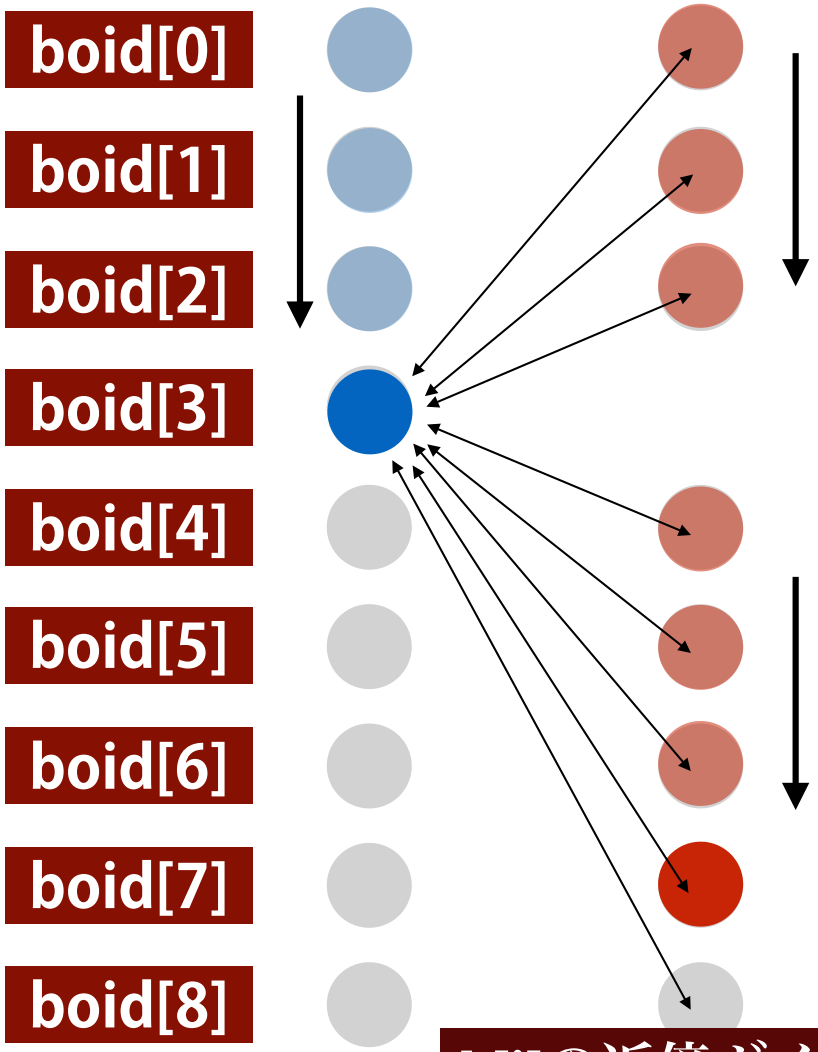
1. 重心の計算

2. 重心に引き込まれるように, ボイドの速度を新しい速度に更新する.

2. 速度の修正

2はapplyRuleAttractorで学習済みですので, ここでは1の解説をします.

b[i]の近傍ボイドの重心計算フロー




```
for(int i=0;i<bsum;i++)
```

初期化

視界内にあるボイドの総数 — `int count = 0;`

視界内にあるボイドの座標の総和 — `Vector3 posTotal= Vector3.zero;`

```
for(int j=0;j<bsum;j++)
```

b[i]とb[j]の距離が, vision_space以内であったときの処理

```
count++;
posTotal += boid[1].pos;
count++;
posTotal += boid[4].pos;
count++;
posTotal += boid[6].pos;
count++;
posTotal += boid[7].pos;
```

b[i]の近傍ボイドの重心座標：Vector3 posMean = posTotal / count

結合ルール関数内部の記述例 BoidManager.cs

全てのボイド（i=0…pop−1）について，
1） 視界内にあるボイドの重心位置 target を求め，
2） 速度を重心に引き込まれるように更新します．

```
void ApplyRule1()
{
    for (int i = 0; i < bsum; i++) {

        Vector3 ipos = boid [i].pos;
        Vector3 ivel = boid [i].vel;

        float count = 0f;
        Vector3 posTotal = new Vector3 ();

        for (int j = 0; j < bsum; j++) {

            Vector3 jpos = boid [j].pos;
            float dis = [redacted]

            if ([redacted]) {
                posTotal += [redacted]
                count++;
            }
        }

        if (count > 0) {
            Vector3 target = [redacted]

            Vector3 cv = [redacted]

            boid [i].setVelocity ([redacted]
        }
    }
}
```

i 番目のボイドの位置・速度をipos, ivelとする.

j番目のボイドの位置をjposとする.

近傍判定

ボイド i とボイド j の距離を disとする.

ボイド i の近傍ボイドの位置の平均値をtargetとする.

1. 重心の計算

係数「c1」を用いて修正ベクトルの計算

ボイド i の速度を更新.

2. 速度の修正