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) 06/03　(08) 06/10

B｜ボイドルール１・２・３の実装

(09) 06/17　(10) 6/24　(11) 07/01

C1｜集合知の解析

(12) 07/08　(13) 07/15

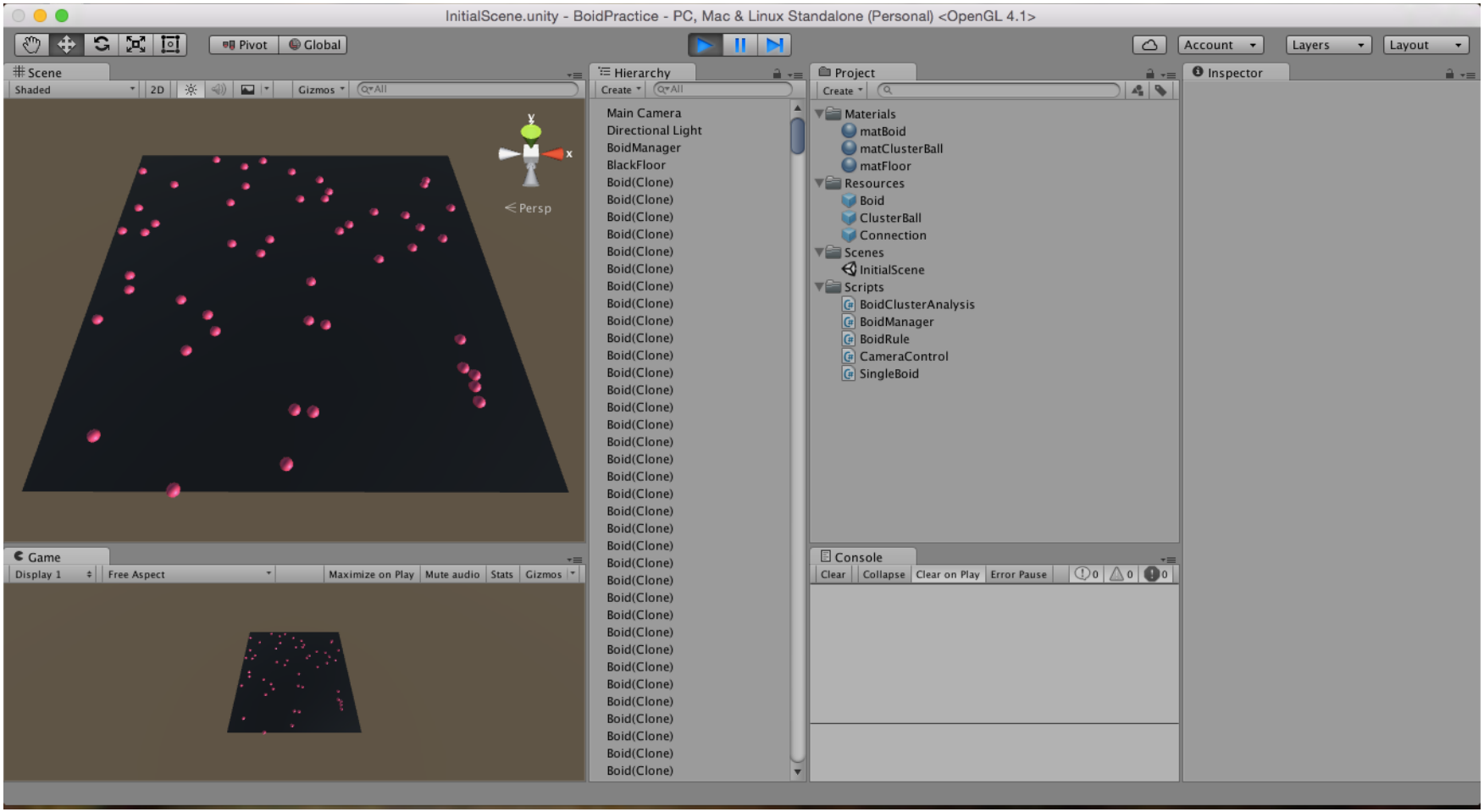
C２｜マイルール（ルール４）の実装・視点の操作

(14-15) 07/22

C２｜発表（One-Minute Movie）

演習２-Ａ

Unity環境の整備・簡単なルールの実装



プロジェクトの構造

BoidManager

BoidManager

描画の実行クラス

BoidClusterAnalysis

ボイドのクラスター解析を行うためのクラス

Main Camera

Directional Light

BlackFloor

matFloor

Boid | **Connection** | **ClusterBall**

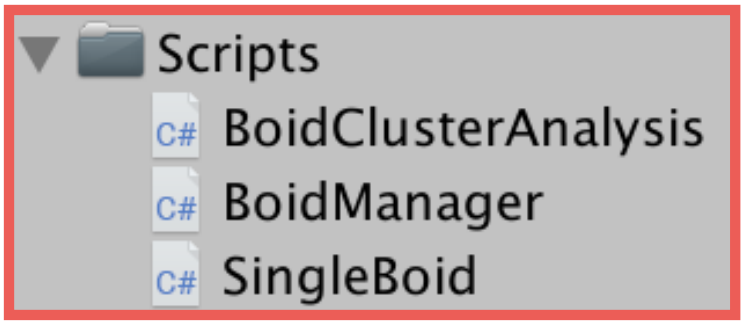
matBoid | matClusterBall

プレハブ

SingleBoid

ボイド単体の振る舞いを記述したクラス

スクリプト



ParentBoid

ParentConnection

ParentClusterBall

プレハブを収納するためのゲームオブジェクト

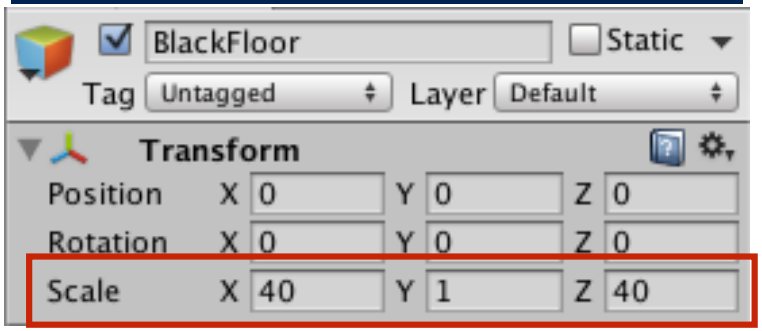
マテリアル

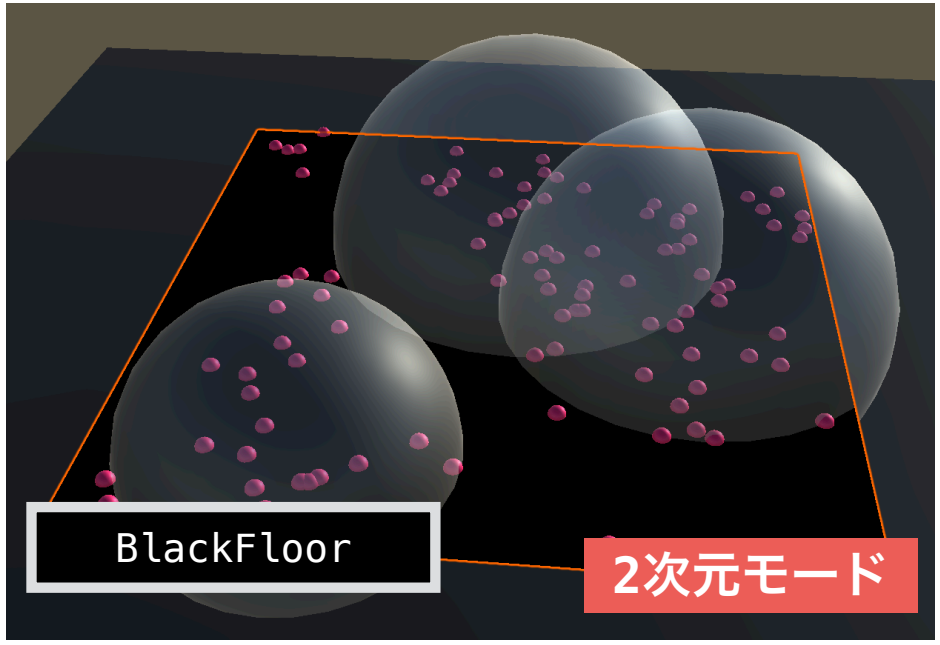
二次元モード・三次元モード

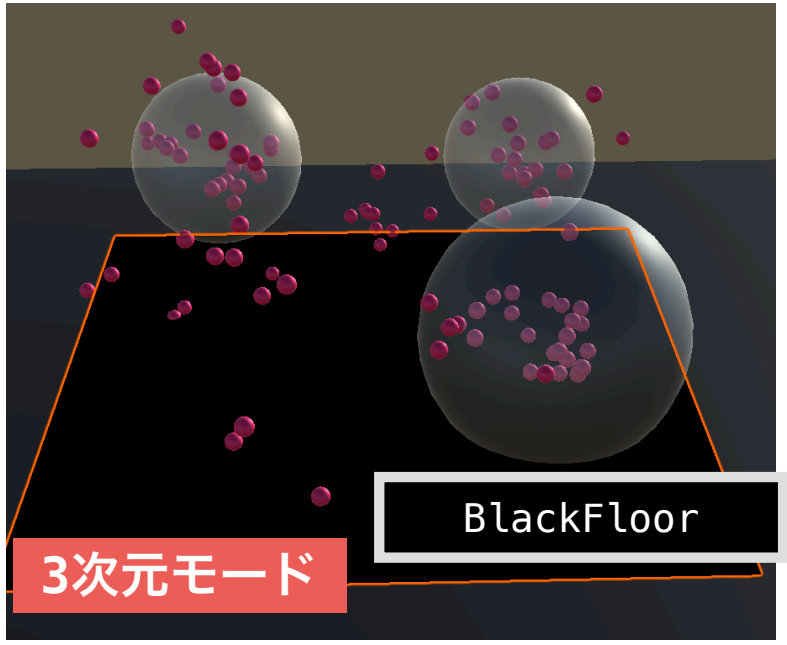
- デフォルトでは, 400 x 400（xz）の空間をボイドが動き回りますが,「3」ボタンを押すと, 三次元モードとなり, 400 x 400 x 200（xzy）の空間を使うことができるようになります.「2」を押すと、元の二次元モードに戻ります。

- 「I」ボタンを押すと, すべてのボイドの位置と速度が初期化されます.

床（Plane）も初期設定では,
400x400のサイズとなっています.







2 二次元モード

3 三次元モード

i ボイドの位置・速度の初期化

BoidManagerクラスのpublicなインスタンス変数

□ BoidManagerのいくつかのフィールドについては, インスペクタビューから設定可能な状態となっています. 初期状態では, すべてのルールは未設計のため, 各ボイドは相互作用をせずに, 初期速度を維持したまま空間内を（ビリヤードのように）ただただ動き回ります.

BoidManager

<int> vision_space

各々のボイドの視界距離

<int> neighbor_space

各々のボイドの接触限界距離

<int> bsum

ボイドの総数（開始後は変更不可）

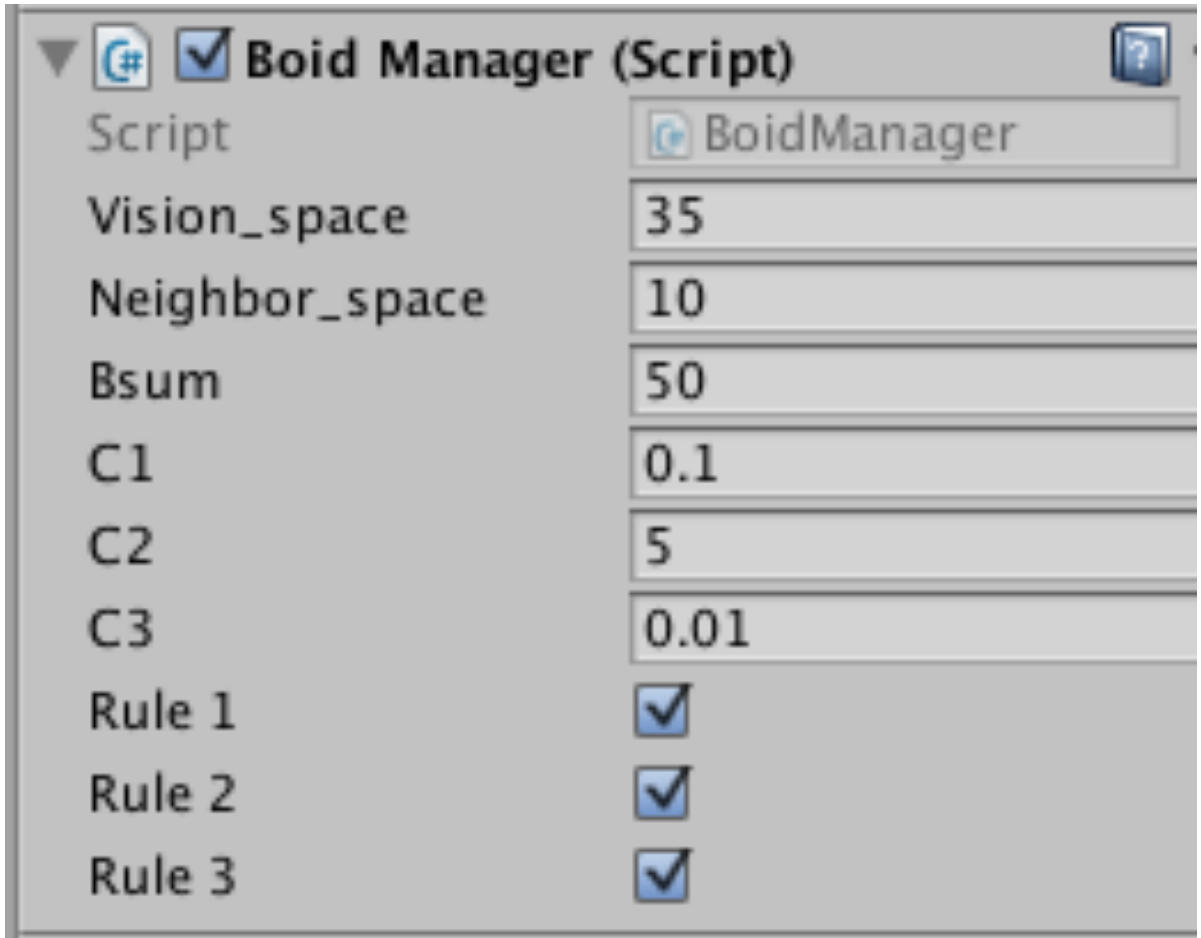
<bool> rule1, rule2, rule3

ルール1・2・3の適用の有無

<float> c1, c2, c3

各ルールの影響度（係数）

123 ルール１・２・３の切り替え



BoidManagerにおけるSingleBoidオブジェクトの呼び出し

クラス変数

```
//ボイドの配列（インスペクタには非表示）
[HideInInspector]
public SingleBoid[] boid;
```

BoidManager.cs

Start()

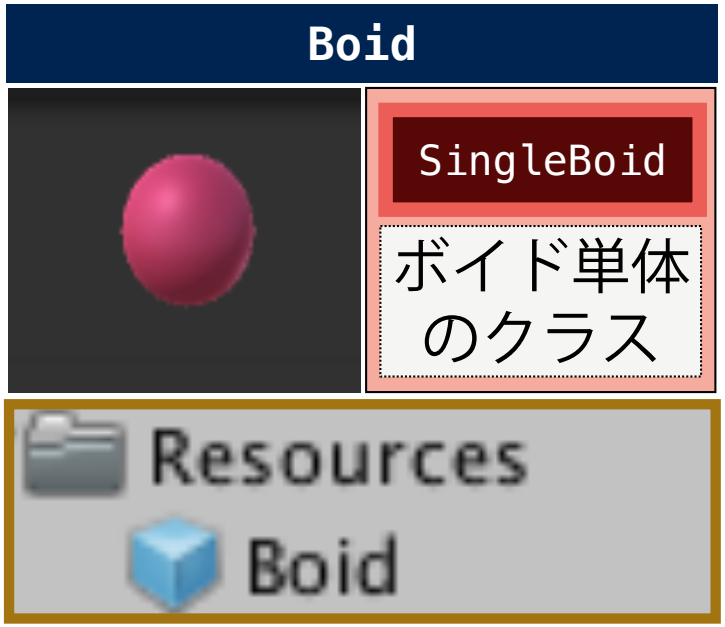
```
/* ボイドオブジェクト（SingleBoidクラス｜スクリプト） */

boid = new SingleBoid[bsum];
GameObject bpar = GameObject.Find("ParentBoid"); //親のゲームオブジェクトを探索

for (int i = 0; i < bsum; i++) {
    //GameObject bobj = Instantiate ((GameObject)Resources.Load ("Boid"));
    GameObject bobj = Instantiate((GameObject)Resources.Load("Boid"),bpar.transform);

    boid[i] = bobj.GetComponent<SingleBoid> ();
}
```

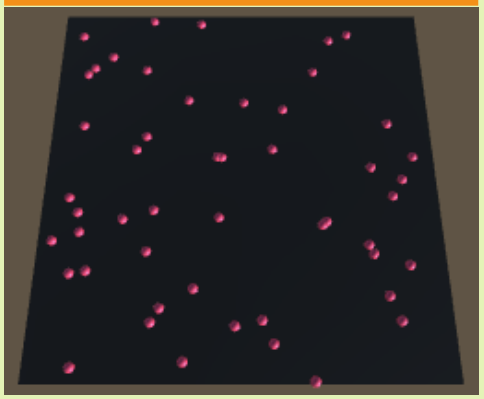
- ボイド単体の基本的な振る舞いは, Boidプレハブのコンポーネントである SingleBoid.cs の中で記述されています.
- BoidManagerは, **start**関数のなかで, まずBoidプレハブのクローンを作成 (**Instantiate**関数) したのちに, Boidのコンポーネントとして, SingleBoid オブジェクトを取り出し, 配列を構成します.



BoidManager・SingleBoidクラスの関係

演習で主に追記するのはここ！！

BoidManager



ボイド集団に対するルールの適用

相互作用に関するパラメータの変更

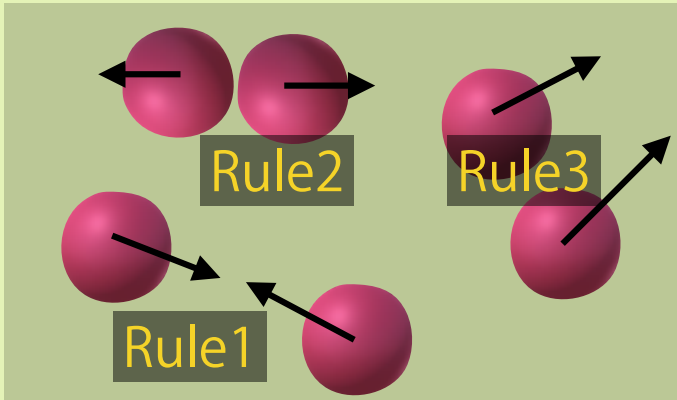
Vision_space	40
Neighbor_space	10
Bsum	50
C1	0.1
C2	5
C3	0.01
Rule 1	☑
Rule 2	☐
Rule 3	☑

```
void applyRule1()
void applyRule2()
void applyRule3()
```

具体的なルールの記述

ルールの適用の有無

```
bool rule1
bool rule2
bool rule3
```



BoidManager.cs

BoidManagerクラスは, ボイドの集団をフィールドとして管理するとともに, 各種のボイド間の相互作用（ルール）の適用の有無（bool rule1, rule2, rule3）、そしてルールの具体的な内容（void applyRuleX）を管理します.

個々のボイドの位置・速度の更新は, SingleBoidクラスで管理されています.

Single Boid

ボイド単体の定義

SingleBoid.cs

SingleBoidオブジェクトの振る舞い

宣言部

```
/* パブリックなフィールド（別クラスから参照可能） */
public Vector3 pos, vel; //位置・速度

/* プライベートなフィールド */
private Rigidbody rb;                                  //剛体オブジェクト
private float xmax,xmin,ymax,ymin,zmax,zmin; //空間の境界
private float speedmax;                                //速さの最大値
```

SingleBoid

SingleBoid.cs

Start()

```
void Start ()
{
    speedmax = BoidManager.speedmax;
    rb = this.GetComponent<Rigidbody> ();

    this.setBorder ();              //飛翔空間の決定
    this.setRandomPosition ();  //初期位置の決定
    this.setRandomVelocity ();  //初期速度の決定

}
```

Update()

```
void Update ()
{
    //位置と速度の更新
    pos = this.transform.position;
    vel = this.rb.velocity;

    Rebound ();                 //境界判定
    LimitVelocity ();           //速度制限
    ConstrainHeight ();         //高さの制限（2Dモード）

}
```

毎フレームの処理（設計者は意識する必要のない処理です．）

Rebound() ▶ **LimitVelocity()** ▶ **ConstrainHeight()**

境界を越えたら速度を反転

xzの初期値は±200, yは0から200

速さが最大値を越えたら, 抑制する

speedmaxの初期値は100

2Dモードの場合,
Y座標を強制的に0とする

SingleBoidオブジェクトの主なパブリック変数・関数

SingleBoid

<Vector3> pos, vel

ボイドの位置と速度

Update()

```
void Update ()
{
    //位置と速度の更新
    pos = this.transform.position;
    vel = this.rb.velocity;

    Rebound ();              //境界判定
    LimitVelocity ();        //速度制限
    ConstrainHeight ();      //高さの制
```

void setVelocity(Vector3 *vel*)

ボイドの速度を具体的に設定する

void setRandomPosition()

位置をランダムに設定

void setRandomVelocity()

位置をランダムに設定

SingleBoid.cs

```
public void setVelocity(Vector3 v){
    this.rb.velocity = v;
    this.vel = this.rb.velocity;
}
```

```
public void setRandomPosition(){
    float rx = xmin + (xmax - xmin) * Random.value;
    float ry = ymin + (ymax - ymin) * Random.value;
    float rz = zmin + (zmax - zmin) * Random.value;

    this.transform.position = new Vector3 (rx, ry, rz);
    this.pos = this.transform.position;

}
```

```
public void setRandomVelocity(){

    float vx = -speedmax + 2f * speedmax * Random.value;
    float vy = -speedmax + 2f * speedmax * Random.value;
    float vz = -speedmax + 2f * speedmax * Random.value;

    rb.velocity = new Vector3 (vx, vy, vz);
    this.vel = this.rb.velocity;

}
```

SingleBoidクラスのオブジェクトの位置と速度は，*boid*.pos，*boid*.velによって取得できます．また，新たに速度を設定し直す場合は，*boid*.setVelocity（*vel*）の関数を使います（*boid* は，SingleBoidクラスのインスタンスとします）．

BoidManagerにおけるルールの記述

applyRule1()

```
void applyRule1(){


}
```

applyRule2()

```
void applyRule2(){


}
```

applyRule3()

```
void applyRule3(){


}
```

Update()

```
if (rule1) {
    applyRule1 ();
}
if (rule2) {
    applyRule2 ();
}
if (rule3) {
    applyRule3 ();
}
```

ruleXがtrueの場合,「applyRuleX」を実行する.

```
/* 全てのボイドの位置・速度を初期化（I） */
if (Input.GetKeyDown (KeyCode.I)) {
    initBoidPosition ();
    initBoidVelocity();
}
```

<I>キーが押されたときに,「initBoidPosition」と「initBoidVelocity」を実行する.

initBoidPosition()

```
/* 全てのボイドの位置をランダムに初期化*/
1 reference
void initBoidPosition(){
    for (int i = 0; i < bsum; i++) {
        boid [i].setRandomPosition ();
    }
}
```

initBoidVelocity()

```
/* 全てのボイドの速度をランダムに初期化*/
1 reference
void initBoidVelocity(){
    for(int i=0;i<bsum;i++)
    {
        boid[i].setRandomVelocity();
    }
}
```

ボイドの位置・速度の初期化. SingleBoidクラスのメソッドを呼び出しています.）

BoidManager.cs

例題 1

SingleBoid

`<Vector3> pos, vel`

ボイドの位置と速度

`void setVelocity(Vector3 vel)`

Voidの速度を更新する.

```
if (Input.GetKeyDown(KeyCode.R)) {
    ReverseVelocity();
}
```

Rキー が押されたら, ReverseVelocityを実行する.

Update()

BoidManager.cs

「R」ボタンで, 全てのボイドの速度が反転（Reverse）するように, BoidMangerクラスのクラスメソッド ReverseVelocityに記述しましょう.

```
void ReverseVelocity()
{
    for(int i = 0; i < bsum; i++)
    {
        Vector3 ivel = boid[i].vel;
        Vector3 v = new Vector3(-ivel.x, -ivel.y, -ivel.z);

        boid[i].setVelocity(v);
    }
}
```

全てのボイド（**boid[0], boid[1], …boid[bsum-1]**）に対して

ボイド **i** の速度を取得し, **ivel** とし, 新しい速度ベクトル **v** を, **ivel** の全ての速度成分を反転させたものとして定義する.

ボイド **i** の速度を更新する.

BoidManager.cs

例題２

いずれかのボイドからの距離が10以下となると，停止するルール9を加えてください。

Vector3

```
float *Distance(Vector3 p1, Vector3 p2)
```

p1とp2の距離を返すVector3のクラスメソッド

```
<Vector3> *zero
```

ゼロベクトル (0f, 0f, 0f)

例題２（ヒント）

```
void ApplyRule9(){
    for (int i = 0; i < bsum; i++) {
        Vector3 ipos = boid [i].pos;
        for (int j = i + 1; j < bsum; j++) {
            Vector3 jpos = boid [j].pos;
            if ([redacted]) {
                boid [i].setVelocity [redacted];
                boid [j].setVelocity [redacted];
            }
        }
    }
}
```

i 番目のボイドの位置

j 番目のボイドの位置

ipos と jpos の距離が10以下であれば, それぞれの速度をゼロとする.

BoidRule.cs

例題３

いずれかのボイドからの距離が30以下となると, 速さの大きい方の速度に合わせるルール8を追加してください.

Vector3

```
float *Distance(Vector3 p1, Vector3 p2)
```

p1とp2の距離を返すVector3のクラスメソッド

```
<float> *magnitude
```

ベクトルの大きさ・長さ（|(x,y,z)|）

例題３（ヒント）

```
void ApplyRule8(){
    for (int i = 0; i < bsum; i++) {
        Vector3 ipos = boid [i].pos;
        Vector3 ivel = boid [i].vel;

        for (int j = i + 1; j < bsum; j++) {
            Vector3 jpos = boid [j].pos;
            Vector3 jvel = boid [j].vel;

            [redacted]

        }
    }
}
```

i 番目のボイドの位置・速度

j 番目のボイドの位置・速度

BoidRule.cs