

## 演習 2 : 集合の知性を設計する

(05) 05/17

**A | Unity環境の整備・簡単なルール設計**

(06) 05/24 (07) 05/31

**B | ボイドルール 1・2・3の実装**

(08) 06/07

**C | 課題 1 : 集合知の解析**

(09) 06/14 (10) 06/21

**D 1 | SIR (感染モデル)**

(11) 06/28 (12) 07/05 (13) 07/12

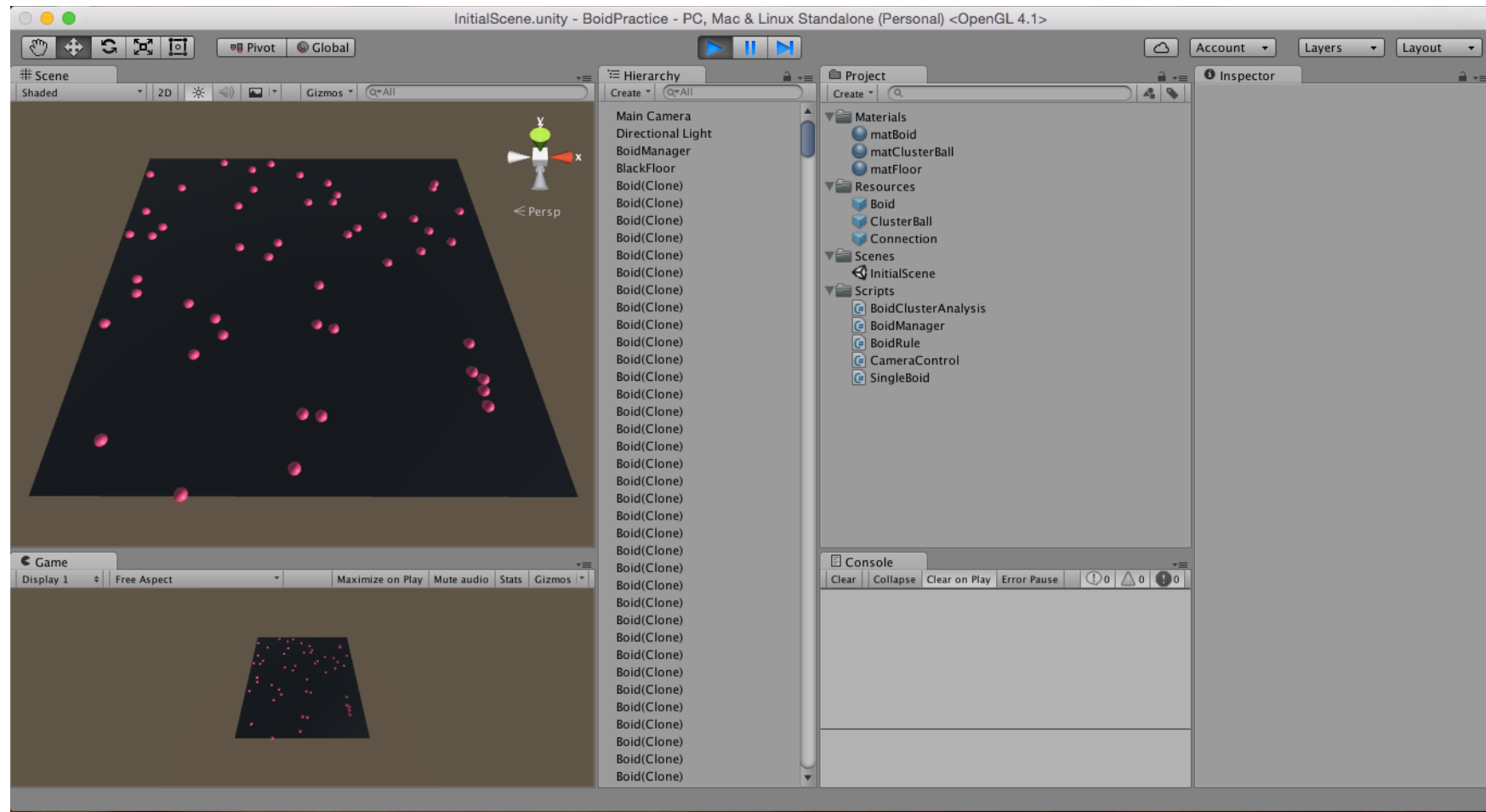
**D 2 | 課題 2 : マイルール・感染ルール・視点操作**

(14-15) 07/19

**D 3 | 発表 (One-Minute Movie)**

## 演習 2 - A

# Unity環境の整備・簡単なルールの実装



# プロジェクト全体の構造

Main Camera

Directional Light

PipeWall

matWall

BlackFloor

matFloor

BoidManager

BoidManager

描画の実行クラス

BoidRuleManager

ボイドのルールの記述用のクラス

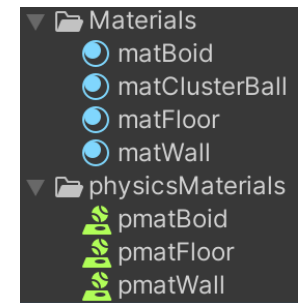
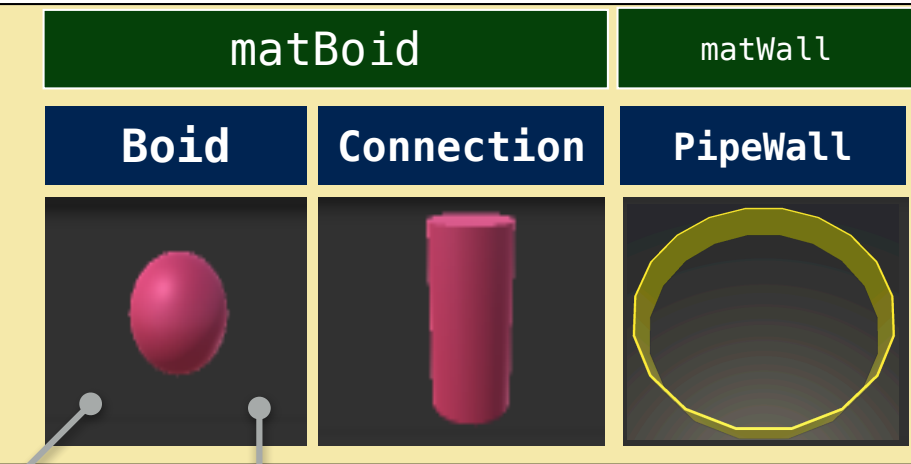
BoidClusterAnalysis

ボイドのクラスター解析用のクラス

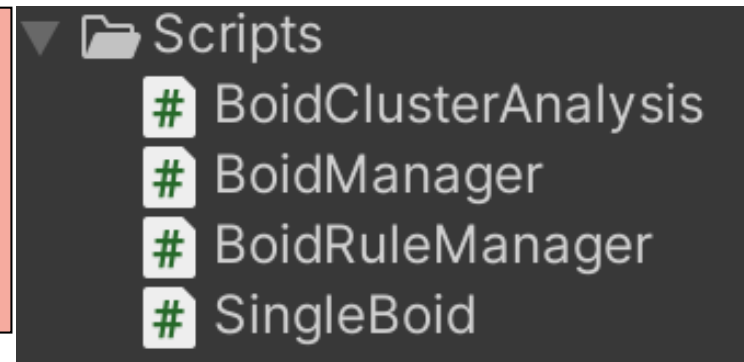
ParentBoid

ParentConnection

ParentClusterBall



マテリアル



スクリプト

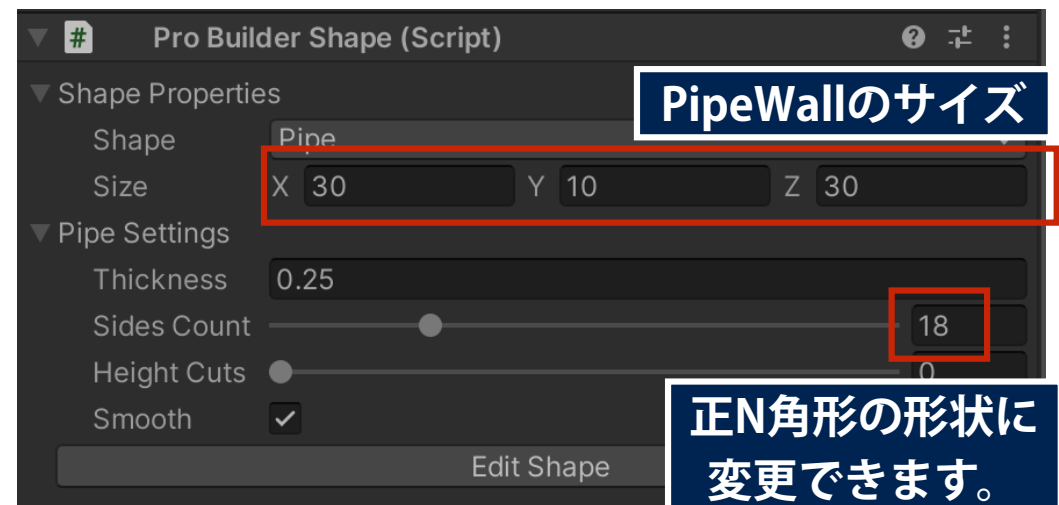
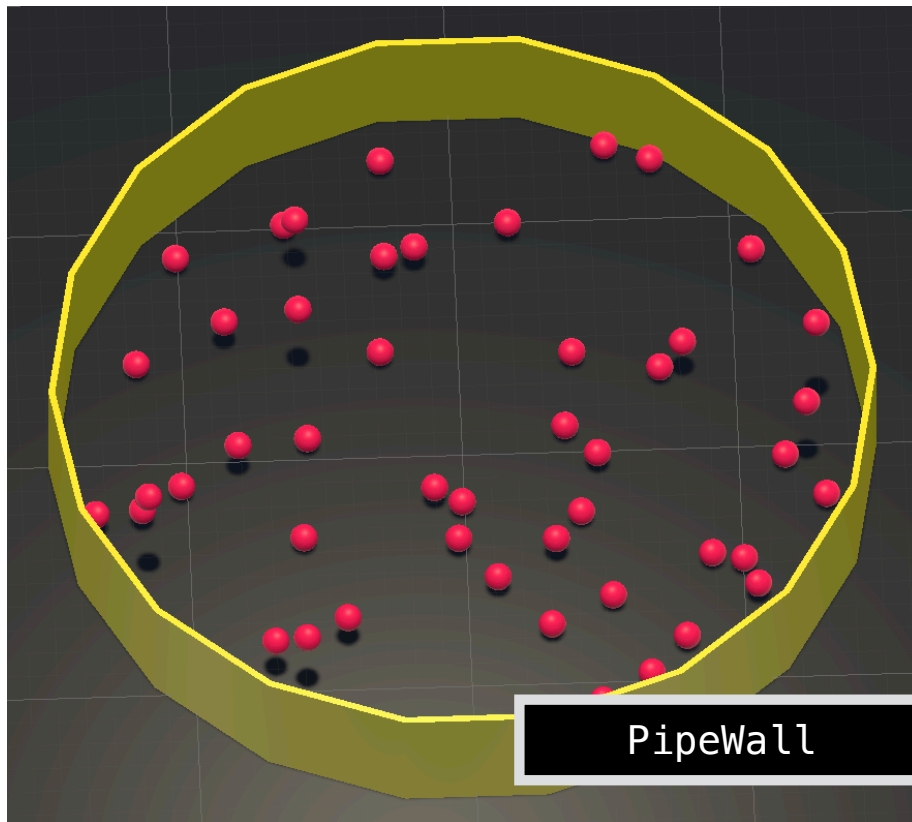
プレハブ



プレハブを収  
納するための  
ゲームオブ  
ジェクト

# 舞台となる空間

- デフォルトでは、半径15mの近似円（正18角形）、高さは5mの筒形の空間をボイドが動き回ります。壁の形状は、インスペクタの「Pro Builder Shape」で変更することが可能です。
- 「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。



ボイドの位置・速度の初期化

# 二次元モード・三次元モード

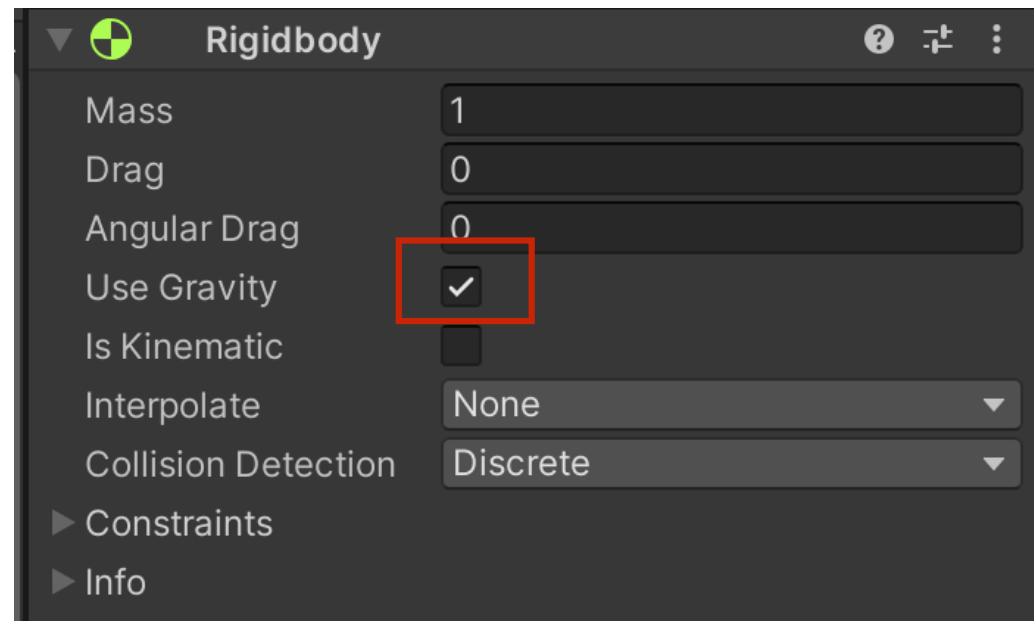
- デフォルトは2次元モードで、各ボイドに重力が作用します。
- 「3」を押すと、3次元モードとなり、重力がキャンセルされます。再び重力を作用させたい場合、「2」を押してください。
- 重力は、プレハブから生成されるボイドにアタッチされた剛体（rigidbody）に対するパラメータで操作しています。

2

二次元モード

3

三次元モード



# BoidManager・BoidRuleManagerクラスのpublic変数

- BoidManager・BoidRuleManager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**

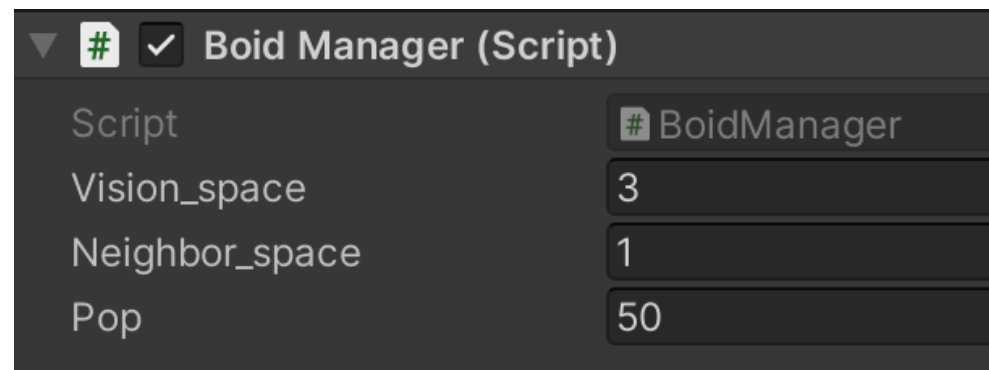
各々のボイドの視界距離

**<int> neighbor\_space**

各々のボイドの接触限界距離

**<int> pop**

ボイドの総数（開始後は変更不可）



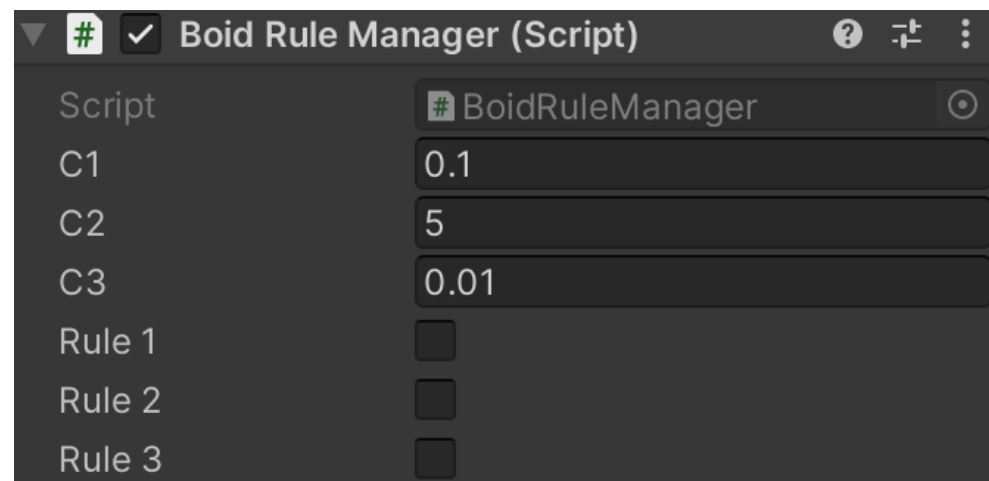
## BoidRuleManager

**<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）



# BoidManagerにおけるSingleBoidオブジェクトの呼び出し

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

クラス変数

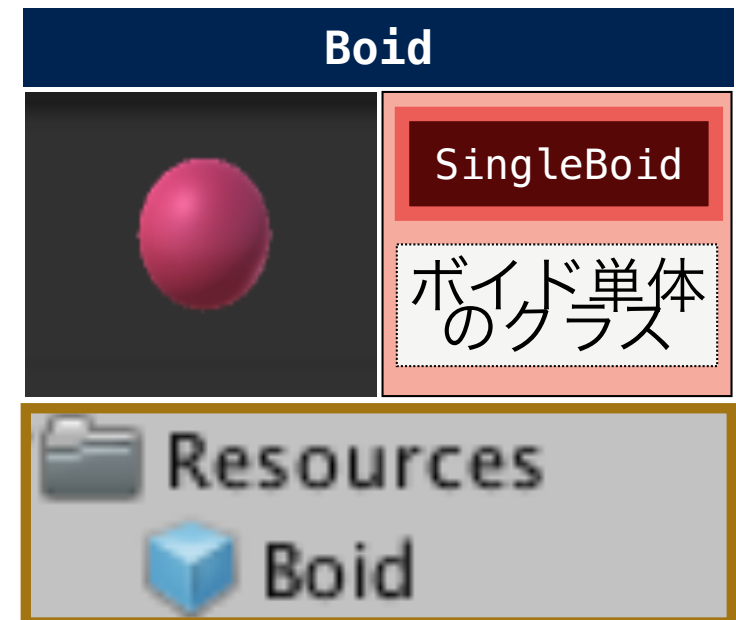
BoidManager.cs

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

Start()

```
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.



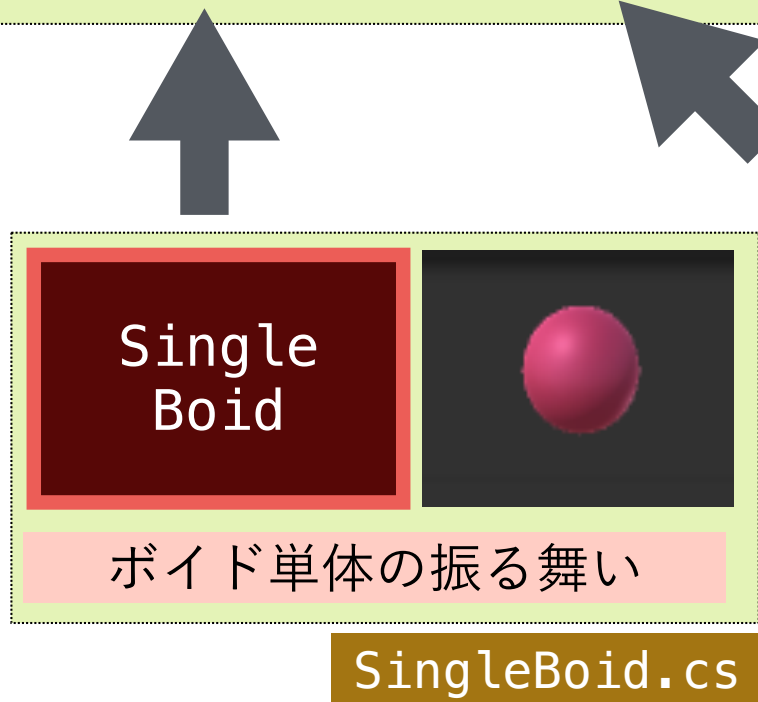
# BoidManager・BoidRuleManager・SingleBoidクラスの関係



BoidManagerクラスは、ボイドの集団をフィールドとして管理します。

BoidRuleManagerは、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。



**BoidRule Manager**

ルールの適用の有無

<bool> rule1

<bool> rule2

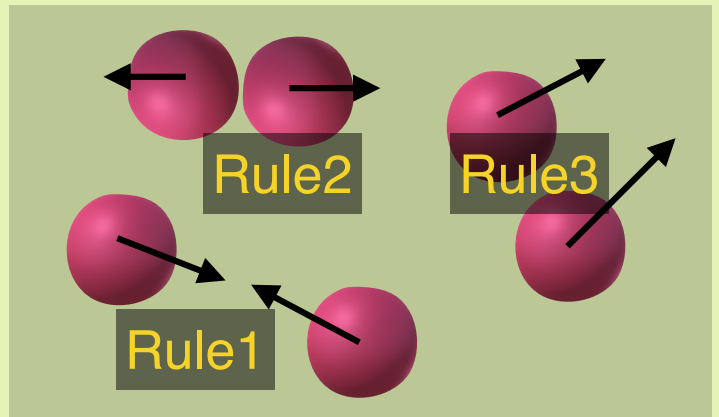
<bool> rule3

void ApplyRule1()

void ApplyRule2()

void ApplyRule3()

具体的なルールの記述（演習の対象）





# SingleBoidオブジェクトの振る舞い

```
/* 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
```

宣言部

```
/* 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

```
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  
}
```

Start()

```
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)  
}
```

Update()

2023バージョンは、少し異なります。

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。）

高さのみ！！  
Rebound()

LimitVelocity()

現バージョンで  
は未使用！！  
ConstrainHeight()

境界を越えたら速度  
を反転

速さが最大値を越え  
たら、抑制する

2Dモードの場合、  
Y座標を強制的に0とする

xzの初期値は±15, yは0から200

speedmaxの初期値は10

# 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

```
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
```

Update()

**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;  
}
```

形状に合わせて若干の修正あります。

```
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, 0, vz);  
    this.vel = this.rb.velocity;  
}
```

Y方向の速度は0としています。

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)。

# BoidRuleManagerにおけるパブリックメソッド

## BoidRuleManager.cs

### void SetBoid(BoidManager m)

現在のボイドの状態（総数・位置・方向・視界距離・接触距離）を、自クラスの同名の変数にコピーする。ルール計算の前に実行する。

```
public void SetBoid(BoidManager m)
{
    boid = m.boid;
    pop = m.pop;

    if (is_vision_space_global)
    {
        for (int i = 0; i < pop; i++)
        {
            boid[i].SetVisionSpace(m.vision_space);
        }
    }

    if (is_neighbor_space_global)
    {
        for (int i = 0; i < pop; i++)
        {
            boid[i].SetNeighborSpace(m.neighbor_space);
        }
    }
}
```

### void ApplyRules()

ルールを全て実行する。パブリックなbool変数（rule1・rule2・rule3）の符号によって、特定のルールのみを実行することが可能。ルールの内容は、ApplyRuleX()の中に記述する。

```
public void ApplyRules()
{
    if (rule1)
    {
        ApplyRule1();
    }

    if (rule2)
    {
        ApplyRule2();
    }

    if (rule3)
    {
        ApplyRule3();
    }
}
```

```
void ApplyRule1()
```

```
void ApplyRule2()
```

```
void ApplyRule3()
```

```
/* ルール1の実行内容*/
```

```
1 reference
```

```
private void ApplyRule1()
{
```

```
}
```

```
/* ルール2の実行内容 */
```

```
1 reference
```

```
private void ApplyRule2()
{
```

```
}
```

```
/* ルール3の実行内容 */
```

```
1 reference
```

```
private void ApplyRule3()
{
```

```
}
```

# BoidManagerとBoidRuleManagerの関係

## BoidManager.cs

### Start()

```
/* ボイドルールマネージャの生成 */  
rule = this.GetComponent<BoidRuleManager> ();
```

### Update()

```
//ボイドルールマネージャによるルールの適用  
rule.SetBoid(this);  
rule.ApplyRules();
```

自分自身のボイド変数を, BoidRuleMangerオブジェクトにコピーした後に, ルールを実行させる.

```
/* 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.

## BoidRuleManager.cs

### BoidRuleManager

```
void SetBoid(BoidManager m)
```

```
void ApplyRules()
```

### InitBoidPosition()

```
/* 全てのボイドの位置をランダムに初期化*/  
1 reference  
private void InitBoidPosition()  
{  
    for (int i = 0; i < pop; 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  
private void InitBoidVelocity()  
{  
    for (int i = 0; i < pop; 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.)

# 例題

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 < pop ; 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1

いずれかのボイドからの距離が「1m」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1 (ヒント)

```
void ApplyRule9(){  
    for (int i = 0; i < pop ; i++) {  
        Vector3 ipos = boid [i].pos; i 番目のボイドの位置  
        for (int j = i + 1; j < pop ; j++) {  
            Vector3 jpos = boid [j].pos; j 番目のボイドの位置  
            if ( [redacted] ) {  
                boid [i].SetVelocity [redacted] ;  
                boid [j].SetVelocity [redacted] ;  
            }  
            ipos と jpos の距離が1以下であれば, それぞれの速度をゼロとする.  
        }  
    }  
}
```

**BoidRuleManager.cs**



## 例題2

いずれかのボイドからの距離が「2m」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 ( $|x, y, z|$ )



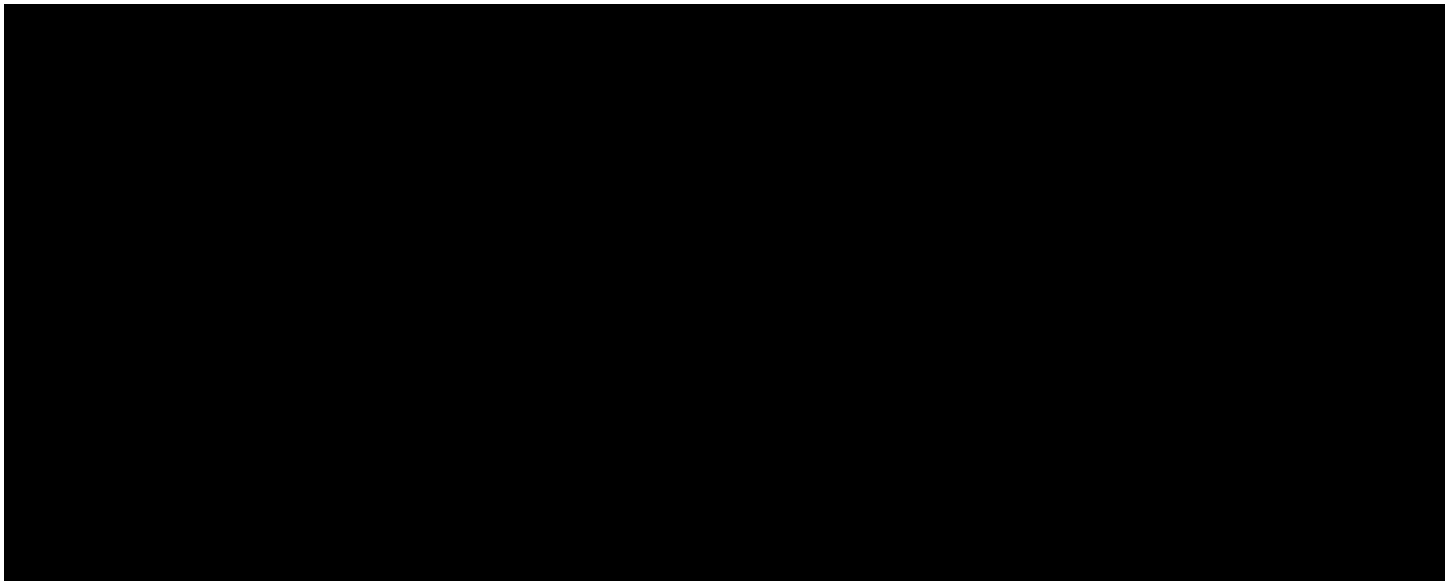
## 例題2 (ヒント)

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

i 番目のボイドの位置・速度

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

j 番目のボイドの位置・速度



```
    }
```

```
}
```

```
}
```

BoidRuleManager.cs