Unity* による魚群行動: AI を利用して動くオブジェクト動作のシミュレーション

この記事は、インテル® デベロッパー・ゾーンに公開されている「Fish Flocking with Unity*: Simulating the Behavior of Object Moving」の日本語参考訳です。

はじめに

この記事では、AI 行動に似せるため群行動を使用する方法と、それを魚の群れに実装する方法について説明します。この重要な AI 特性には次の段階的なプロセスが含まれます。

1. Unity* 統合開発環境 (IDE) で作業を開始する
2. Fish アセットを使用して環境の構築を開始する
3. 群行動を再現する

動作環境

この記事では、次のシステム構成を使用します。

- ASUS* ROG* ラップトップ
- 第 4 世代インテル® Core™ i7 プロセッサー
- 8GB RAM
- Windows® 10 Enterprise エディション

群行動

群行動とは、オブジェクトがグループとして移動または連携する動作を指します。この行動は奥深く、魚群行動と群泳行動、昆虫の群行動、陸上動物の群行動に類似しています。群行動のシミュレーションは、オブジェクトをまとめて群れを生成する AI ロジックのシミュレーションとして実装されます。これには、「群行動としての群衆シミュレーションと Windows* Mixed Reality: パート 1」(英語) で説明されている分離、調整、凝集が含まれます。

プロジェクトの作成

Unity* IDE で Fish Flocking AI パッケージを開くには、最初に**新規プロジェクト**を作成します。

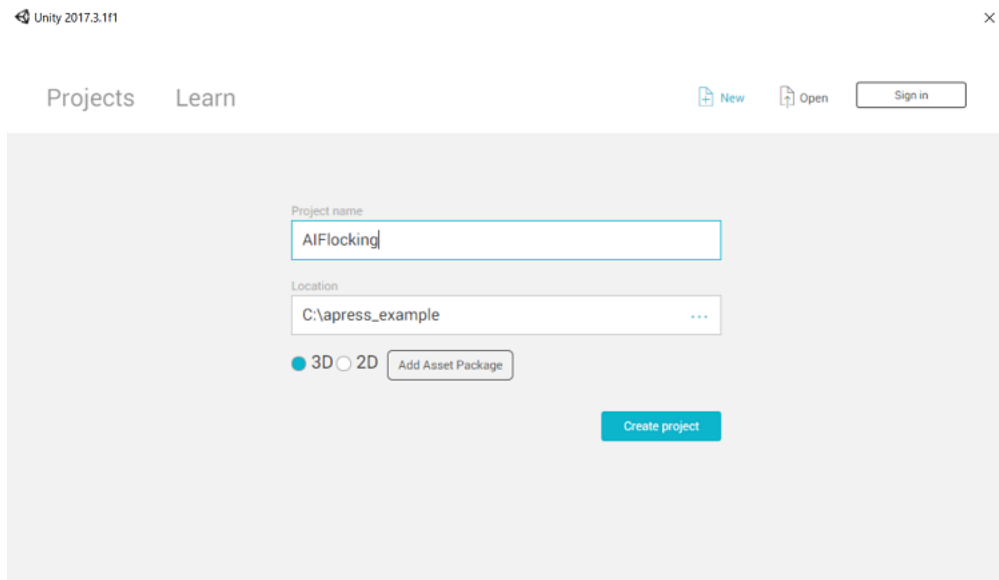

図 1. Unity* で新規プロジェクトを作成

次に、[Assets] > [Import Package] > [Custom Package] を選択して**パッケージ**をインポートします。

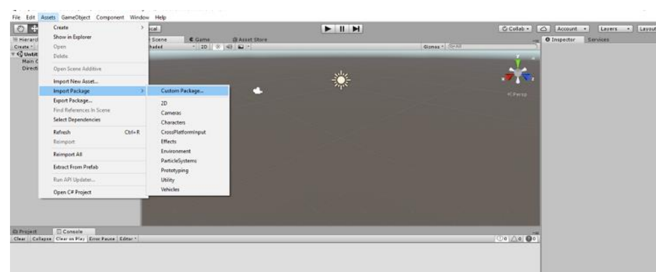

図 2. Unity* でパッケージをインポート

Fish パッケージ全体をインポートします。

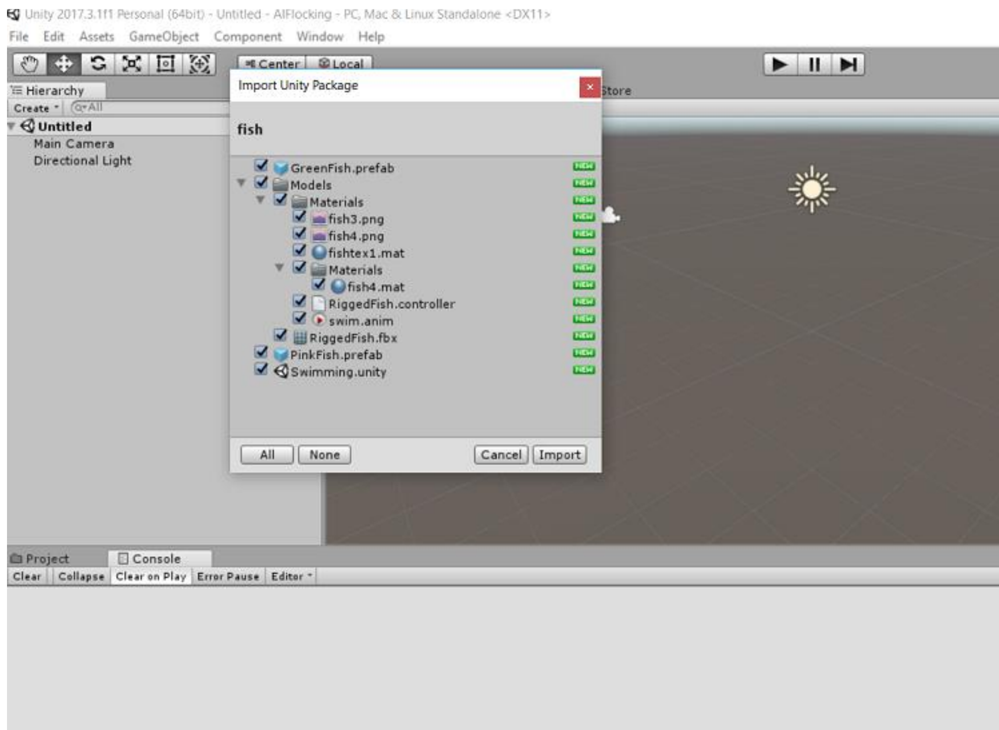

図 3. Fish パッケージ全体をインポート

Asset パッケージとシーンを表示します。

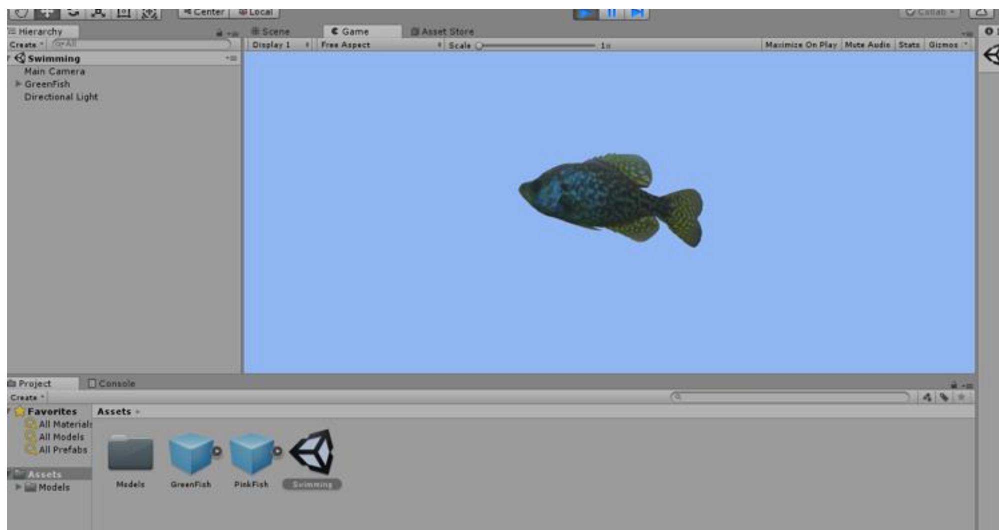

図 4. Asset パッケージとシーンを表示

Unity* でアセットを開いて、AI 行動パターンから作業を開始します。PinkFish と GreenFish が表示されます。Unity* で泳ぐシーンを開くと、GreenFish には泳ぎのアニメーションが設定されています。

群行動はグループ・アクティビティーであるため、実装するにはさらに多くの魚を作成する必要があります。**群行動マネージャー**を作成するには、階層で右クリックして空のオブジェクトを作成し、その名前を **FlockManager** に変更します。

コーディング・プロセスでは、機能を実装するため**さまざまな C# スクリプト**を作成します。最初に作成してアタッチする C# スクリプトは FlockManager です。

次のようなスクリプトが生成されます。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FlockManager : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

魚オブジェクトのインスタンス化の配列を作成し、群行動マネージャー周辺のランダムな場所に配置するコードを **FlockManager** に追加します。

fishPrefab という名前の**パブリック変数**を作成します。

```
public GameObject fishPrefab;
```

fishPrefab は、PinkFish または GreenFish のどちらでも構いません。**numfish** を **20** に設定します。これは適宜変更できます。

```
public int numFish = 20;
```

Unity* IDE の Inspector ウィンドウで魚の数を変更します。次に、**allFish** 配列を作成します。インスタンス化されると、魚の数が **allFish** 配列に格納されます。

```
public GameObject[] allFish;
```

Swim Limits を設ける (泳げる範囲を制限する) ことで、魚が移動できる範囲が設定され、群行動マネージャーを囲むボックスができます。Swim Limits を使用して、ボックス内の魚の位置を生成します。このロジックは、開始関数内で処理されます。

```
public Vector3 swimLimits = new Vector3(5,5,5);
```

最初に、すべての魚を作成します。すべての魚を格納するのに十分な大きさの配列を作成します。

```
allFish = new GameObject[numFish];
```

設定された回数だけ反復する for ループで魚を配置します。

```
Vector3 pos;
```

この位置は、群行動マネージャーの位置と、正と負の Swim Limits 範囲内のランダムな **Vector3** 制限に基づいています。より大きなボックスを使用する場合、次のように変更します。

```
public Vector3 swimLimits = Vector3 (5,5,5);
```

上記のパラメーターを下記のように変更します。

```
public Vector3 swimLimits = Vector3 (10, 10,10);
```

次のように位置を更新します。

```
Vector3 pos = this.transform.position + new Vector3(Random.Range(-
swimLimits.x,swimLimits.x),

Random.Range(-swimLimits.y,swimLimits.y),

Random.Range(-swimLimits.z,swimLimits.z));
```

魚は、ニュートラルな回転を使用して計算された位置で Prefab からインスタンス化され、配列に格納されます。

```
allFish[i] = (GameObject) Instantiate(fishPrefab, pos, Quaternion.identity);
```

ここまでのすべてのパラメーターの追加と変数の宣言をまとめると次のようになります。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FlockManager : MonoBehaviour {
    public GameObject fishPrefab;
    public int numFish = 20;
    public GameObject[] allFish;
    public Vector3 swimLimits = new Vector3(5,5,5);

    // Use this for initialization
    void Start () {
        allFish = new GameObject[numFish];
        for(int i = 0; i < numFish; i++)
        {
            Vector3 pos = this.transform.position +
                              new Vector3(Random.Range(-swimLimits.x,swimLimits.x),
                                          Random.Range(-swimLimits.y,swimLimits.y),
                                          Random.Range(-swimLimits.z,swimLimits.z));
            allFish[i] = (GameObject) Instantiate(fishPrefab, pos, Quaternion.identity);
            //allFish[i].GetComponent<Flock>().myManager = this;
        }

    }

    // Update is called once per frame
    void Update () {

    }
}
```

このコードとパラメーターを Unity* に追加して実行します。最初に、作成した空の **FlockManager object** に **FlockManager スクリプト**を追加します。**GreenFish Prefab** を削除します。FlockManager で **Fish Prefab を追加**します。

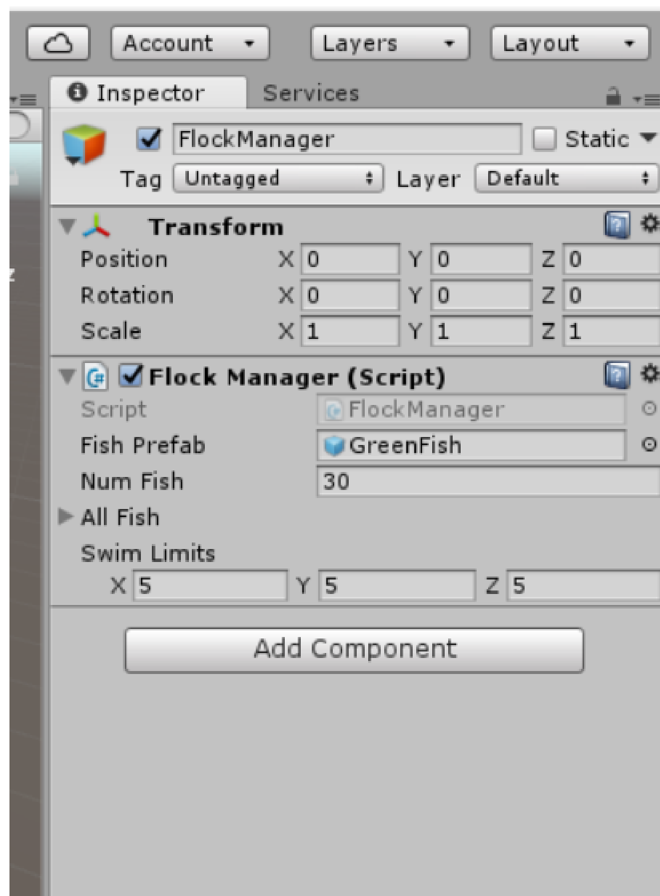

図 5. FlockManager パラメーター

シーンを**実行**します。

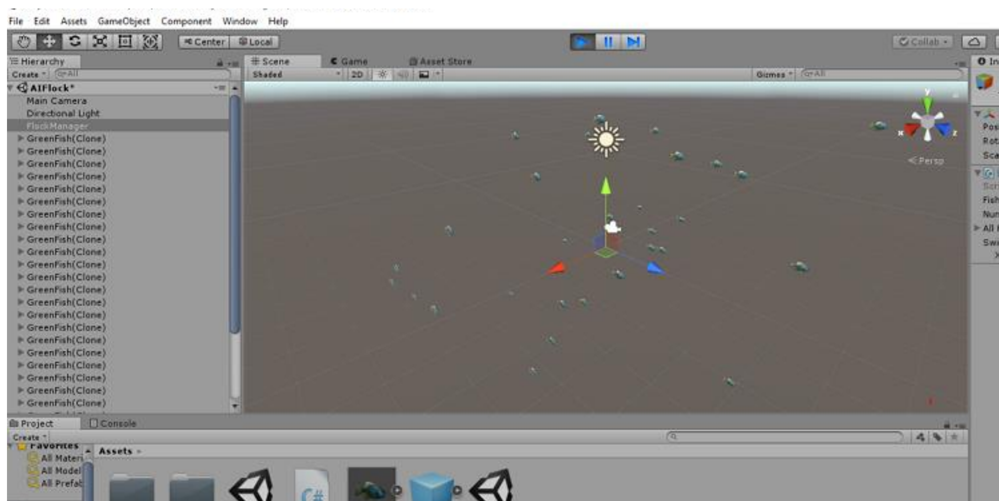
図 6. Unity* でシーンを実行

すべての魚が階層および Inspector ウィンドウ内に表示されます。すべての魚が **allFish** 配列内にあるため、それぞれの魚は群れの中のほかの魚を見つけることができます。

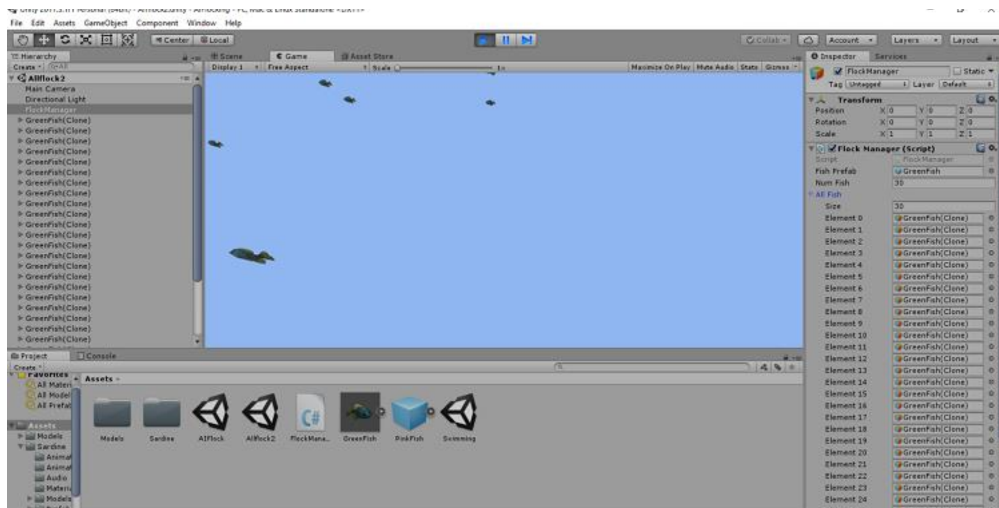

図 7. allFish 配列のビュー

魚の動き

魚を動かすには、魚に速度値を追加します。**FlockManager** で各群れの特定の魚に異なる変数を設定し、複数の群れを作成します。

FlockManager を使用してパブリック値を追加し、範囲を指定してスライダーで設定できるようにします。まず、ヘッダー設定を行い、**public minspeed** と **public maxspeed** の 2 つの変数を宣言します。

```
[Header("Fish Settings")]
[Range(0.0f, 5.0f)]
public float minSpeed;
[Range(0.0f, 5.0f)]
public float maxSpeed;
```

Inspector ウィンドウに変更が反映されます。

図 8. Inspector ウィンドウの FlockManager の設定

Flock という名前の**新しい C# ファイル**を作成して、**GreenFish と PinkFish** の両方にスクリプトを追加します。

デフォルトのスクリプトを開いてみましょう。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Flock : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

最初に必要なのは、群行動マネージャーへのリンクです。

```
public FlockManager myManager;
```

次は float speed です。

```
float speed;
```

start メソッドで、マネージャーの Min Speed (最小速度) から Max Speed (最大速度) 設定まで、Random.range に速度を宣言します。

```
speed = Random.Range(myManager.minSpeed,myManager.maxSpeed);
```

update メソッドで、魚を前方または z 軸に沿って移動する変換文を宣言します。魚の速度値は z 軸の内側になります。

```
transform.Translate(0, 0, Time.deltaTime * speed);
```

魚が宣言された直後に FlockManager.cs スクリプトとその for ループ内にある Flock.cs を接続します。

```
allFish[i].GetComponent<Flock>().myManager = this;
```

変更後のコードは次のようになります。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FlockManager : MonoBehaviour {
    public GameObject fishPrefab;
    public int numFish = 20;
    public GameObject[] allFish;
    public Vector3 swimLimits = new Vector3(5,5,5);

    [Header("Fish Settings")]
    [Range(0.0f, 5.0f)]
    public float minSpeed;
    [Range(0.0f, 5.0f)]
    public float maxSpeed;

    // Use this for initialization
    void Start () {
        allFish = new GameObject[numFish];
        for(int i = 0; i < numFish; i++)
        {
            Vector3 pos = this.transform.position +
                                   new Vector3(Random.Range(-swimLimits.x,swimLimits.x),
                                               Random.Range(-swimLimits.y,swimLimits.y),
                                               Random.Range(-swimLimits.z,swimLimits.z));
            allFish[i] = (GameObject) Instantiate(fishPrefab, pos, Quaternion.identity);
            allFish[i].GetComponent<Flock>().myManager = this;
        }

    }

    // Update is called once per frame
    void Update () {

    }
}
```

コードを保存して結果をシーンで確認します。

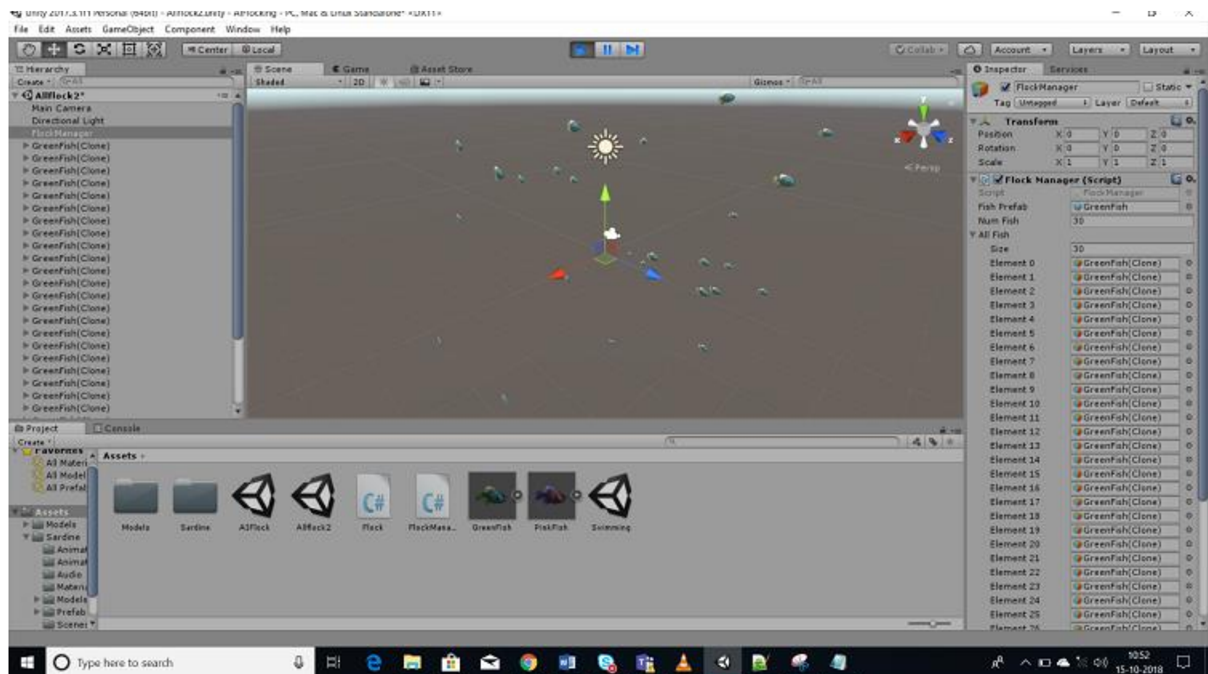

図 9. 結果シーンを表示

群行動ルール

魚の群れを作成する際の基本ルールは、次のとおりです。

グループの平均的な位置に向かって移動する

つまり、グループ内のすべての位置の合計÷グループ内の魚の数です。個々のメンバーの位置を加算して、グループ内のメンバーの数で除算することで、それぞれの位置とどのようにグループへ向かっているのかを特定するのに役立ちます。

グループの平均的な進行方向に合わせる

グループ内の個々の魚の進行方向を加算して、グループのサイズで除算することで、進行方向の調整に使用する平均ベクトルを算出できます。

ほかのグループメンバーとの衝突を回避する

ほかのメンバーとの衝突を回避するため、個々の魚は周囲にいるほかの魚の位置と方向転換するタイミングを把握する必要があります。個々の魚の新しい進行方向を計算するには、グループの中心へ向かって移動するベクトル、グループの進行方向に合わせるためのベクトル、および周囲にいるメンバーから方向転換するためのベクトルを加算します。

群行動の効果とルール

FlockManager の次の値は、魚の動作に対して設定する必要があります。

```
[Range(1.0f, 10.0f)]
public float neighbourDistance;
[Range(0.0f, 5.0f)]
public float rotationSpeed;
```

neighborDistance (周囲の魚との距離) を上げると、群れがなくなります。neighborDistance を元に戻しても、すべての魚が群れになるわけではありません。これには回転速度も影響します。回転速度を上げると、いくつかの新しい動作が見られるようになります。

コードに戻っていくつかの変更を加えます。群行動の効果を得るには、flocking.cs コードの **update** メソッドで新しい **ApplyRules()** メソッドを記述します。

```
void ApplyRules()
```

現在の群行動内のすべての魚を保持する **gos** ホルダーを作成します。

```
GameObject[] gos;
gos = myManager.allFish;
```

vcenter に格納される平均中心を計算します。**vavoid** に格納される平均回避ベクトルも計算します、0 に設定されている **avoidance vector** を計算します。

```
Vector3 vcentre = Vector3.zero;
Vector3 vavoid = Vector3.zero;
```

float **gspeed** はグループのグローバル速度です。これは、平均的なグループが移動する速度で、計算に使用されます。

```
float gSpeed = 0.01f;
```

次は float **nDistance** です。それぞれの魚にどれぐらい離れているか確認してから、グループにどれぐらい近いのか特定します。

```
float nDistance;
```

groupSize カウンターは、グループ内の魚の数をカウントします。隣接距離にある群れの小さなサブセクションのように機能します。

```
int groupSize = 0;
```

すべてゼロに初期化されます。

foreach ループで 1 匹ずつ、群れのすべての魚をループします。foreach ループ内の魚の構造です。

```
foreach (GameObject go in gos)

{

}
```

現在の魚 go は、このインスタンスのコード **(if(go != this.gameObject)** で実行している魚とは等しくないため、次のように周囲の魚との距離を計算します。ゲームオブジェクトの配列位置 (**go.transform.position**) とワールド座標位置 (**this.transform.position**) の間の距離 **Vector3.distance** を特定します。

```
nDistance = Vector3.Distance(go.transform.position,this.transform.position);
```

その距離が周囲の魚との距離以下の場合、**closenet** グループの魚と見なされ、その位置が **vcenter** に格納される中心に加算されます。vcenter は、グループの平均位置です。すべての魚をループして、位置の合計をグループサイズで除算します。そして、グループサイズを 1 増やします。すべての魚をループしたら、平均を計算します。

```
if(nDistance <= myManager.neighbourDistance)
                {
                    vcentre += go.transform.position;
                    groupSize++;
                    if(nDistance < 1.0f)
                    {
                        vavoid = vavoid + (this.transform.position - 
go.transform.position);
                    }

                }
```

次に、if 文を使用して **nDistance** が非常に小さな値 (ここではハードコードされている **1.0**) 未満かどうかテストします。これは、魚が衝突を回避する前にほかの魚とどれぐらい接近できるかを定義します。**nDistance** が 1.0f 未満の場合、平均ベクトルはそのベクトルにほかの魚から離れるためのベクトルを加えた値になります (つまり現在の位置からほかの魚を回避する位置を引いた値です)。

```
                    if(nDistance < 1.0f)
                    {
                        vavoid = vavoid + (this.transform.position - 
go.transform.position);
                    }
```

グローバル速度 (**gSpeed**) は、群れの中の特定の魚の速度の合計です。魚の Flock コードをアタッチして、その魚の速度を取得し **gSpeed** に加算します。

```
Flock anotherFlock = go.GetComponent<Flock>();
gSpeed = gSpeed + anotherFlock.speed;
```

変更したコードの全体フローを以下に示します。

```
foreach (GameObject go in gos)
        {
            if(go != this.gameObject)
            {
                nDistance = 
Vector3.Distance(go.transform.position,this.transform.position);
                if(nDistance <= myManager.neighbourDistance)
                {
                    vcentre += go.transform.position;
                    groupSize++;

                    if(nDistance < 1.0f)
                    {
                        vavoid = vavoid + (this.transform.position - 
go.transform.position);
                    }

                    Flock anotherFlock = go.GetComponent<Flock>();
                    gSpeed = gSpeed + anotherFlock.speed;
                }
            }
        }
```

各ループが群れの中のすべての魚を処理し、最も近くにいる魚かどうか検証し終わると、魚の位置と回避ベクトルが判明します。

foreach ループ内に記述されたロジックには、グループサイズがゼロよりも大きいかどうか確認する if 文があります。魚が群れと一緒に動いていて周囲にほかの魚がいない場合、グループサイズがゼロになると、ルールは適用されず魚は直進し続けます。この魚がほかの魚またはほかの群れの影響を受けている場合、計算された平均中心 (**vcentre**) ベクトルを取得し、それらすべての合計値をグループサイズで除算します。

```
vcentre = vcentre/groupSize
```

魚の速度は、**gSpeed** を **groupSize** で除算した値です。

```
speed = gSpeed/groupSize;
```

その情報を利用して、魚が進む方向を決定できます。**Vector3** 方向は、次のように求めることができます。

```
Vector3 direction =(vcentre + vavoid) -transform.position
```

if(direction ==0) の場合、正しい方向に向かっています。

if(direction !=Vector3.zero) の場合、**Slerp** を使用して、回転速度に応じて回転します (**transform.rotate**)。ゆっくりと魚を必要な方向に回転します。

```
if(direction != Vector3.zero)
    transform.rotation = Quaternion.Slerp(transform.rotation,
                         Quaternion.LookRotation(direction),
                         myManager.rotationSpeed * Time.deltaTime);
```

これらの変更を適用した後の flock.cs コードは次のようになります。

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Flock : MonoBehaviour {

    public FlockManager myManager;
    float speed;

    // Use this for initialization
    void Start () {
        speed = Random.Range(myManager.minSpeed,
                             myManager.maxSpeed);
    }

    // Update is called once per frame
    void Update () {
        transform.Translate(0, 0, Time.deltaTime * speed);
        ApplyRules();

    }
    void ApplyRules()
    {
        GameObject[] gos;
        gos = myManager.allFish;

        Vector3 vcentre = Vector3.zero;
        Vector3 vavoid = Vector3.zero;
        float gSpeed = 0.01f;
        float nDistance;
        int groupSize = 0;

        foreach (GameObject go in gos)
        {
            if(go != this.gameObject)
```

```
            {
                nDistance =
Vector3.Distance(go.transform.position,this.transform.position);
                if(nDistance <= myManager.neighbourDistance)
                {
                    vcentre += go.transform.position;
                    groupSize++;

                    if(nDistance < 1.0f)
                    {
                        vavoid = vavoid + (this.transform.position -
go.transform.position);
                    }

                    Flock anotherFlock = go.GetComponent<Flock>();
                    gSpeed = gSpeed + anotherFlock.speed;
                }
            }
        }

        if(groupSize > 0)
        {
            vcentre = vcentre/groupSize;
            speed = gSpeed/groupSize;

            Vector3 direction = (vcentre + vavoid) - transform.position;
            if(direction != Vector3.zero)
                transform.rotation = Quaternion.Slerp(transform.rotation,
                                     Quaternion.LookRotation(direction),
                                     myManager.rotationSpeed * Time.deltaTime);

        }
    }
}
```

Unity* でアプリケーションを実行して、群れの動きを確認します。

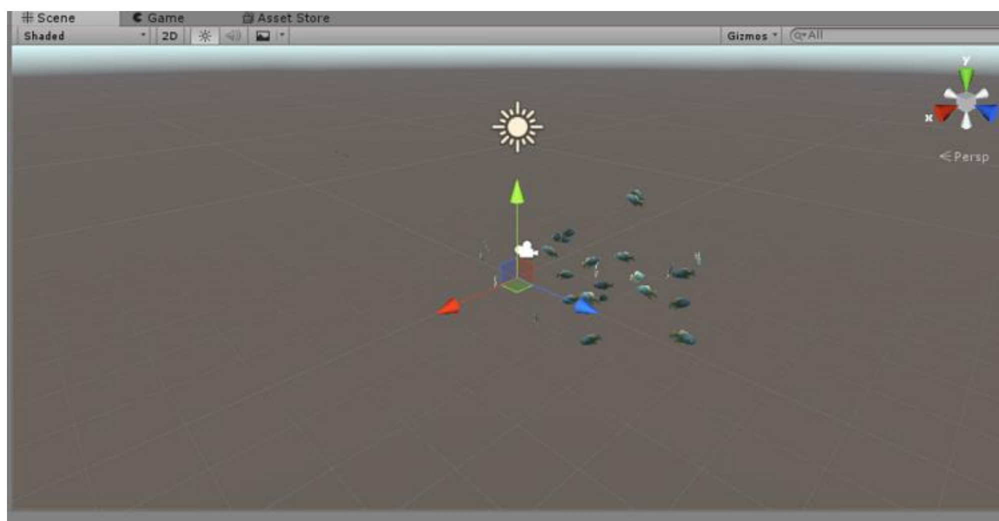

図 10. 結果シーンを表示

まとめ

この記事では、Unity* を使用した新しい AI 動作である、魚の群行動を作成するプロセスを説明しました。群行動を実現するため群行動ルールを適用しました。群衆シミュレーションとしての新しい動作の実装については、「群行動としての群衆シミュレーションと Windows* Mixed Reality: パート 1」(英語) を参照してください。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。