ネットワーク・ゲーム開発

1. ゲームでのネットワーク運用形態

一般的にゲームのネットワークでは TCP/IP を利用しており、現在個別認証には IPv4 が利用されている。

その IPv4 を利用したゲームで利用するネットワークシステムの運用形態をまず考えてみる。運用形態には大きく分けて下記の二つがある。

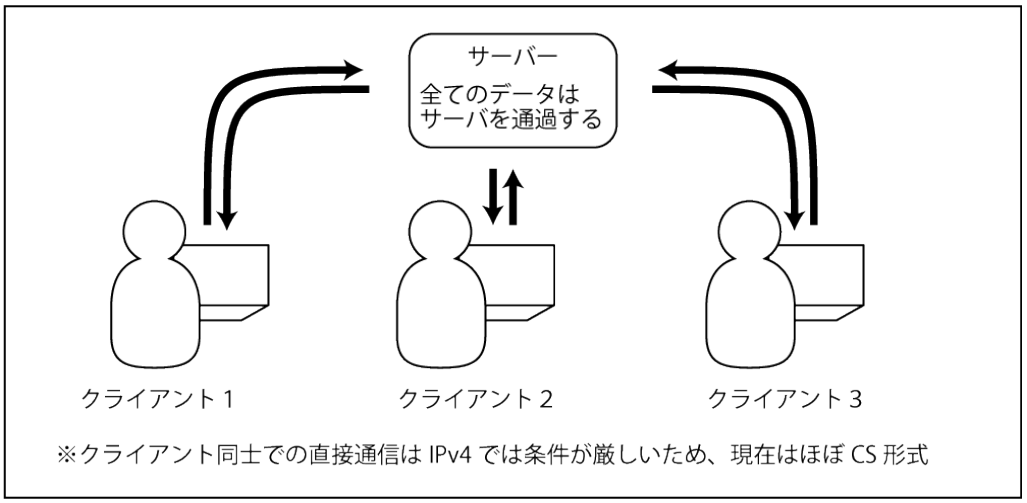
- ○ CS(クライアント・サーバシステム)システム
- ○ P2P(ピア・トゥー・ピア)システム

それぞれの特徴は下記の通り

1.1. CS システムの特徴

システム全体を統括しサービスを提供する “サーバ” と、サービスの提供を受ける “クライアント”に分けて構成する。

イメージ図



運用環境として、“クライアント” だけでなく “サーバ” を同時に開発する必要があるため、開発の難易度は多少高い。

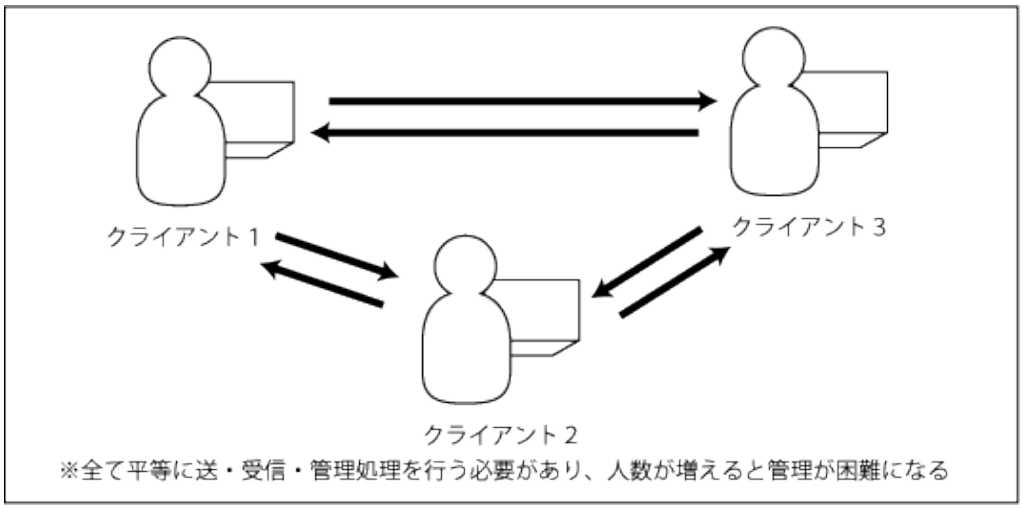
マルチユーザで運用する場合、ユーザ管理など DB などを利用してサーバで管理できるため、クライアント側にあまり負荷をかけずに大規模運用が可能になる。

基本的に全てのデータが、サーバを経由するため、ユーザ・アイテムの管理、チートなどの監視など様々な機能を実装できる。

1.2. P2P(ピア・トゥー・ピア)システムの特徴

ホストとピアのプログラムを同一のプロジェクトで開発できるので、比較的楽に実装できる。

イメージ図



全体管理ができないため、機能の追加時の配布などが困難であり、また人数が増えると処理負荷の増加や通信管理が困難になる。また IP 変換を利用している環境では、直接通信ができない。

また、個別のクライアントで不正が行われても防ぐことが非常に困難になる。

1.3. 通信プロトコルの選択

TCP/IP で通信を行う場合、通信品質と運用環境を考慮し、TCP/UDP の選択が必要になる。それぞれ特徴があるため、適切なプロトコルの選択が必要になる。

○TCP

プロトコルに再送などのエラー処理が組み込まれているため、受信データの品質が高いが、思わぬ遅延が発生しやすいためリアルタイムでの処理には適さない。リアルタイム性が必要のない、ボードゲーム、チャットなどに適している。

○UDP

動画や音楽のストリーミングなど、リアルタイム性が必要な用途に利用される。エラー処理などクライアント側での処理が必要になるが、通信のタイミングをコントロールしやすい。

1.4. 通信のシリアライズ

TCP/IP 通信のパケットは、全て Byte データでやりとりされる。そのため送受信データの形式を Byte データに変換及び、復号する必要がある。これをシリアライズという。

また、通信する Byte データの並びは CPU のアーキテクチャによって異なる場合があるため「ネットワークバイトオーダ」のルールがあり、ビッグエンディアンが用いられる。

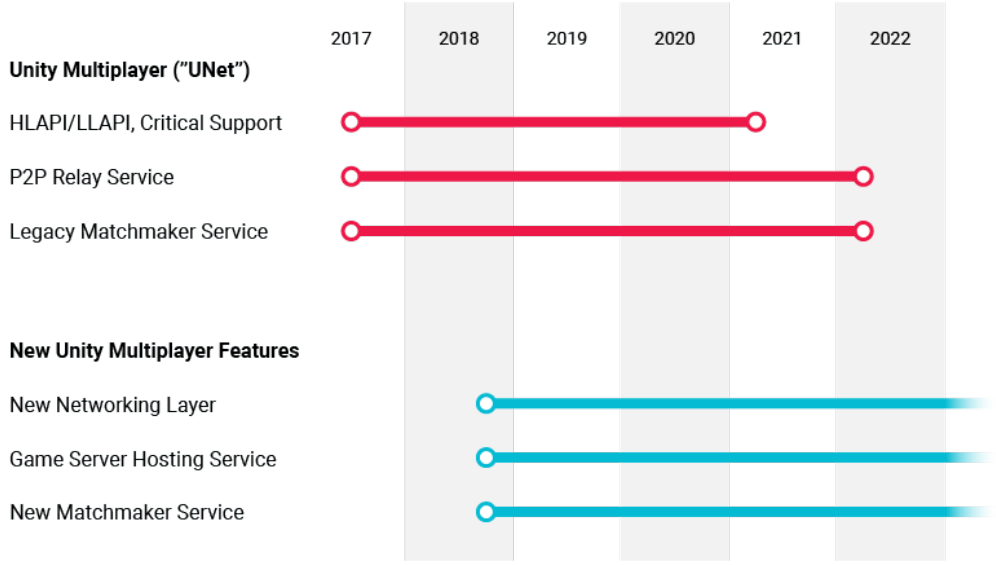
2. Unity のネットワークゲームフレームワーク UNET

前節の通り、ネットワークを利用するために、多くの実装項目がある。基本的な内容はゲームによって同様の内容も多いが、ゲームの安定運用やネットワークパフォーマンスの追及を行うため独自の実装を行う必要があり、ハードルが高い。

そこで Unity には、ネットワークを利用したゲーム開発を行うための環境が用意されている。それが UNET で Unity2017 より実装されている。

2.1. 次世代の Unity ネットワークゲームフレームワーク

Unet は Version5 から実装され、現在、次世代の環境への移行を計画している。



2018 より徐々にその機能の提供が切り替わることになるが、ネットワークゲームの基本機能を学習する上で Unet で主たる機能の学習は可能であり、今回 Unet の環境を使いネットワークゲームへの実装制作を通して、ネットワークゲームで必要な技術を学習する。

3. Unet を用いたネットワークゲーム開発

Unet を利用したネットワーク対戦シューティングゲームの制作を行い必要な機能について学習する。

3.1. Netwok ゲームの制作

本来 Network ゲームの開発では、Network に関する要素を考慮して作成する必要があるが、Unet では、最初から高機能なライブラリが用意されているため、比較的容易に Network ゲームの基礎を体験できる。

今回通常のクライアント作成から、Unet による Netowrk 要素を徐々に付加する形式でゲーム作成を行いながら、Network ゲームで必要な技術の習得を目指す。

3.2. NetworkManager と NetworkManagerHUD 及び NetworkIdentity

Unet を用いてネットワークゲームを作成する際、キーになるコンポーネントがある。順を追ってそのコンポーネントの説明を行う。

3.2.1. NetworkManager

Unet の NetworkManager コンポーネントは、ゲームのネットワークの状態を管理する。このコンポーネントにはサーバ機能も含まれるため CS システムが容易に実装できる。

サーバ機能については、クライアント間でのデータの通信管理だけでは無く、ゲーム状態の管理、オブジェクト生成（Spawning）の管理、シーン管理、デバッグ情報、マッチメイキングなど実装されている機能は多岐にわたる。今回、簡易な CS システム実装を行うが、ネットワーク内の 1 台のみサーバ兼クライアントの役割を担うことになる。

参考資料 NetworkManager

3.2.2. NetworkManagerHUD

NetworkManagerHUD を用いることで、ソフトウェア起動時に、上記 NetworkManager の動作環境を、下記のように設定するための UI を提供する。

○NetworkManager.StartClient ：クライアント機能を開始する。
○NetworkManager.StartServer ：サーバ機能を開始する。
○NetworkManager.StartHost ：同一アプリ内でクライアント、サーバ両機能を開始する。

参考資料 NetworkManagerHUD

3.2.3. NetworkIdentity

Network ゲームは、各クライアントで同一のクライアントソフトが動作することになる。そのため、各クライアントで自分が利用しているクライアントで制御するオブジェクトと、別クライアントからの情報により動作するオブジェクトの判別を行う必要がある。それを識別するために NetworkIdentity コンポーネントが準備されている。

NetworkIdentity を Local Player Authority に設定すると、自分が利用しているクライアントで制御している、オブジェクトの動きを制御できるようになる。

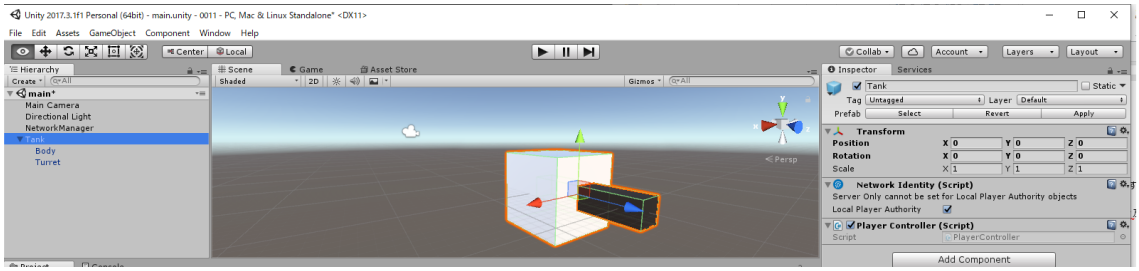
参考資料 NetworkIdentity

4. ゲームの実装

4.1. Player Prefab の作成

NetworkManager によって、各シーン上に Player を生成できる。まずは、そのための prefab オブジェクトを準備する。各自適当な Prefab を準備しておく。ここでは例として簡易な下記のモデルを使用することとする。

○空のゲームオブジェクトを(0,0,0)に作成し、名前を “Tank” にする。
○その下に Cube を作成し、名前を “body” に変える。
○“Tank”下に Cube を作成し、名前を ”Turret” に変更する。
○“Turret” の位置、スケールを Position(0,0,0.5) , Scale(0.2,0.2,1.5) に変更する。
○新規マテリアル“Turret” を作成し、色を黒(0,0,0) に変更した後、先ほどの ”Turret” に適用する



これを Player として、Player がキーボードで動けるよう Script：PlayerController.cs を作成後、Player にアタッチし動作することを確認し、 動作確認後、Player を Prefab 化しておく。

PlayerController.cs

```
1   using UnityEngine;
2
3   public class PlayerController : MonoBehaviour
4   {
5       public float SpeedRate      = 150.0f;
6       public float RotateRate     = 3.0f;
7       void Update()
8       {
9           var x = Input.GetAxis("Horizontal") * Time.deltaTime * SpeedRate;
10          var z = Input.GetAxis("Vertical") * Time.deltaTime * RotateRate;
11
12          transform.Rotate(0, x, 0);
13          transform.Translate(0, 0, z);
14      }
15  }
```

4.2. Player のネットワーク対応

現在スタンドアロンで、画面上に配置したオブジェクトがキーボードで動くだけのものだが、これをネットワーク化する。そのためには、3.2 にある各コンポーネントを、利用する事になる。

4.2.1. NetworkManager コンポーネントの実装

ゲームシーン内に NetworkManager を配置する。そのためには、空のオブジェクトを Hierarchy ビューに配置し空のオブジェクトの名前を “NetworkController” に変更した後、

Component ＞ Network ＞ NetworkManager
Component ＞ Network ＞ NetworkManagerHUD

にて各コンポーネントをアタッチする。その後、Unity エディタの Play ボタンを押し、実行する。そうすると NetworkManagerHUD により下記のような UI が表示されるので、赤枠で囲んだ “LAN Host” ボタンを押すと、クライアント･サーバの機能を持った状態で起動する。そうすると先ほどと同じように動作するが、まだこのままでは Network に対応できていない。

一旦停止して、Network 上で認識できるように準備した Prefab か、先ほど作成した“Tank” に NetworkIdentity コンポーネントをアタッチする。

Compornent ＞ Network ＞ NetworkIdentity

アタッチ後、Inspector で下記の通り ”Local Player Authority” にチェックを入れる。
これにより、NetworkManager で生成された Player オブジェクトが各クライアントで制御できるようになる。

しかし、実行すると Console に下記のエラー表示がされているはずだ。

```
The PlayerPrefab is empty on the NetworkManager. Please setup a PlayerPrefab object.
UnityEngine.Networking.NetworkIdentity:UNetStaticUpdate()
```

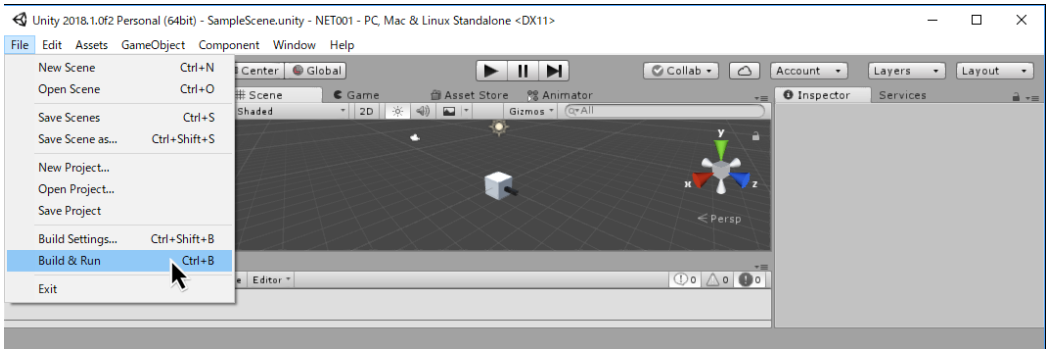
このエラーは、NetworkManager が生成すべき Prefab が設定されていないために起こる。現在、シーン上に Player オブジェクトを配置しているが、あくまでクライアント上に独立しているオブジェクトであり、他のクライアントと接続した場合表示されない。実際確かめてみよう。

4.2.2. Network ゲームの動作確認

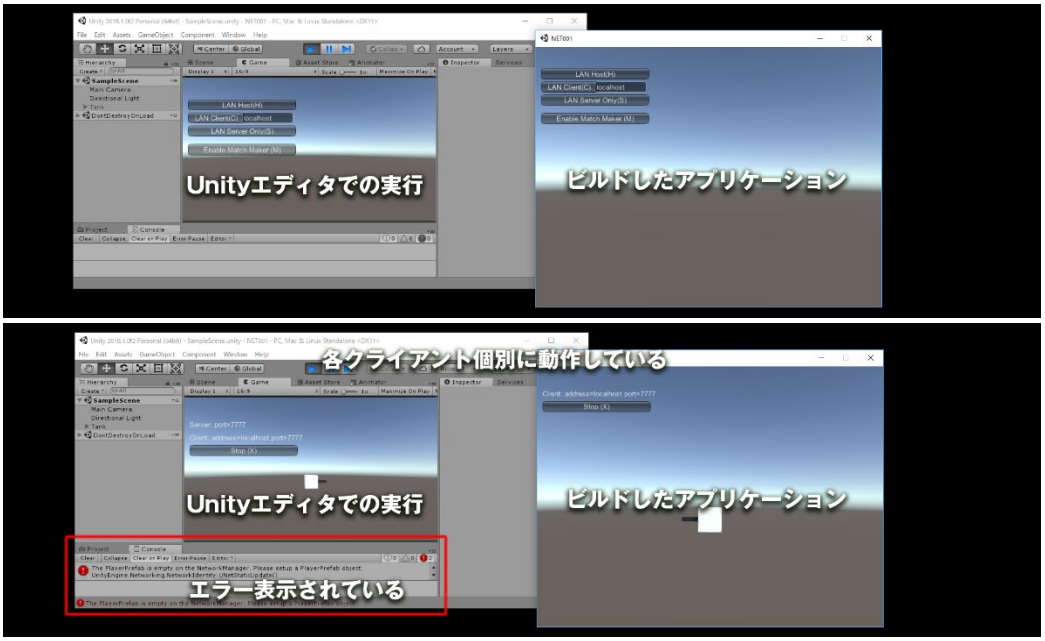
Network ゲームの動作確認を行うためには、複数のクライアントを起動する必要がある。この場合、作業マシンと別に動作環境を準備できればいいのだが、その環境が準備されていない場合も多い。

その場合、1 台の開発マシンで Network ゲームの動作確認するために環境は TCP/IP で準備されている。TCP/IP には特別な IP アドレスとして “localhost(127.0.0.1)” が準備されており、自分自身に向け IP 通信を行い Network 通信が可能である。

ふたつの動作環境を作成するために、Unity エディタより File ＞ Build&Run を行い現在のソフトを単体アプリとして立ち上げる。このアプリと Unity エディタ側の実行環境で通信確認を行うことで、実際の Network 上での動作確認を行うことが可能になる。

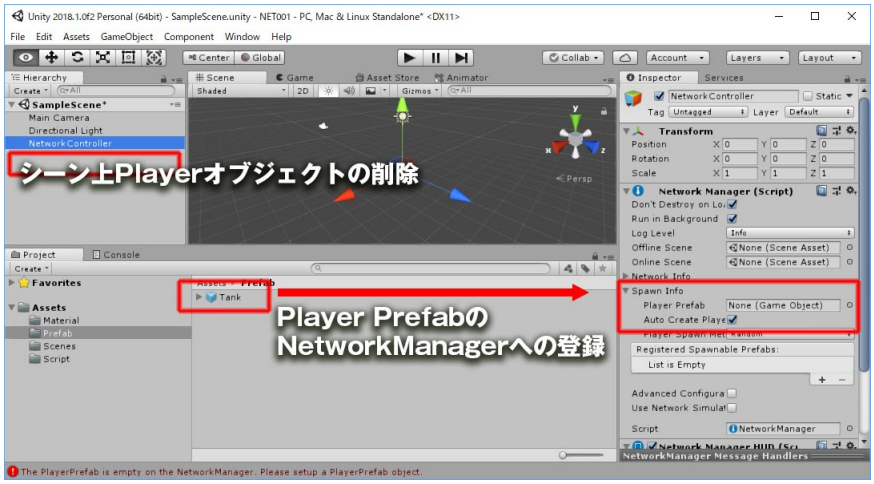


起動した後、一方を “LAN HOST”、もう一方を “LAN Client” で起動することで、2 つのソフトが Network で接続される。しかし、各々の画面に表示されるオブジェクトは 1 つで、各画面でしか動作しない。



4.2.3. Player Prefab の NetworkManager への登録

いったん停止し、今度は先ほど作成した Player Prefab を NetworkManager に登録する。登録することで、NetwrokManager が、各クライアントに対して、Player オブジェクトを生成し各々制御できるようになる。



NetworkManager により Player を生成するため、上図のように、シーン上にある Player オブジェクトを削除し、当該 Player の Prefab を NetworkManager に登録する。

Player オブジェクトは生成されたが、まだ個別に動作している。

次は、このオブジェクトの動きを Network 化していく。そのためには、スクリプトの Network 対応が必要になる。次節でその方法を解説する。

4.2.4. Player のネットワーク化

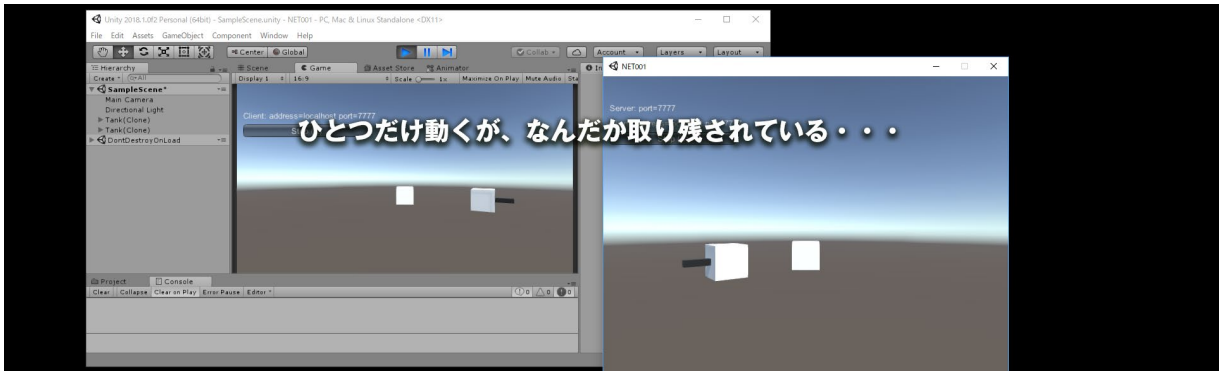
プレイヤーの動きをネットワークと連携させ、ローカルプレイヤーのみがローカルプレイヤーゲームオブジェクトを制御できるように、PlayerController スクリプトを変更する。基本的に、Player オブジェクトは各クライアントで生成されるが、中身が同じであるため、ローカルプレイヤーである場合のみキー入力処理を行うようにスクリプトを変更する。

まず PlayerController スクリプトへ、名前空間 UnityEngine.Networking を追加し、基底クラスを MonoBehaviour から NetworkBehaviour に変更させる。

その後、Update 関数内で、ローカル Player オブジェクトを判別するために、NetworkBehavior にて用意されている "isLocalPlayer" を用いて、ローカルプレイヤー以外はキー入力を受け付けない処理を付加する。

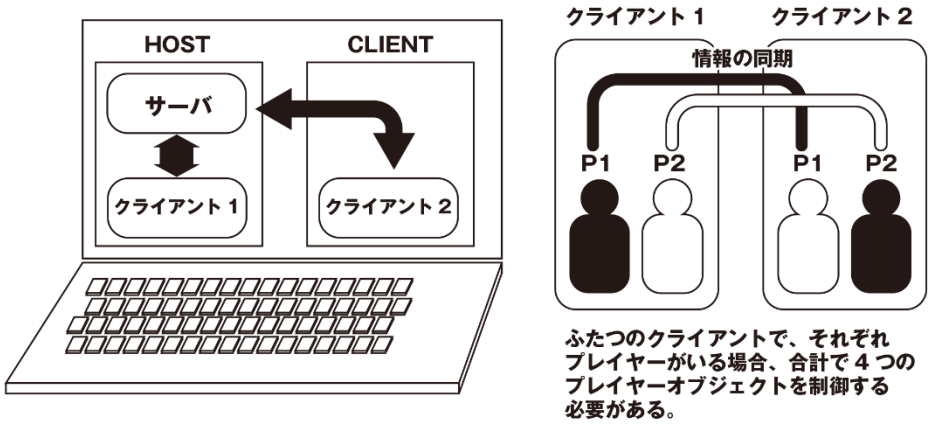
PlayerController.cs　Network 対応

```
1   using UnityEngine;
2   using UnityEngine.Networking;          // Network 用  namespace 追加
3
4   public class PlayerController : NetworkBehaviour    // 基底クラスの変更
5   {
6       public float SpeedRate      = 150.0f;
7       public float RotateRate     = 3.0f;
8
9       void Update()
10      {
11          if( !isLocalPlayer ) return;       // ローカルプレイヤーでなければ以降処理しない
12
13          var x = Input.GetAxis("Horizontal") * Time.deltaTime * SpeedRate;
14          var z = Input.GetAxis("Vertical") * Time.deltaTime * RotateRate;
15
16          transform.Rotate(0, x, 0);
17          transform.Translate(0, 0, z);
18      }
19  }
```



4.2.5. 現在の状況

現時点では、クライアント兼サーバ 1 台、クライアント 1 台の構成になっている。



この場合だと、合計 4 つのプレイヤーの位置を管理しなくてはならない。この同期をとるコンポーネントが“NetworkTransport” コンポーネントである。

PlayerPrefab を Hierarchy ビューに置き、下記の手順でアタッチ後、適用し Hierarchy ビューから削除しておく。

Compornent ＞ Network ＞ NetworkTransform

この状態で動作させることで、片方のクライアントの動きが、別クライアントの画面に反映される。

参考 NetworkTransform

4.2.6. ローカルプレイヤーの判別

これで、基本的なネットワークゲームの基本的な動作が設定できた。しかし。画面上の Player はどれも同じで、捜査対象である自機の判別が難しい。そこで、自機のみ色を変更する。

そのためには、NetworkBehavior に用意されている、自機の精製時に呼び出される関数である“OnStartLocalPlayer” 関数をオーバーライドすることで、自機が生成された場合の Material 情報を書き換えることにする。

PlayerController スクリプトに下記の関数を追加する。

PlayerController.cs　OnStartLocalPlayer 関数の処理追加

```
1   using UnityEngine;
2   using UnityEngine.Networking;          // Network 用  namespace 追加
3
4   public class PlayerController : NetworkBehaviour    // 基底クラスの変更
5   {
6       public float SpeedRate      = 150.0f;
7       public float RotateRate     = 3.0f;
8
9       void Update()
10      {
11          if( !isLocalPlayer ) return;       // ローカルプレイヤーでなければ以降処理しない
12
13          var x = Input.GetAxis("Horizontal") * Time.deltaTime * SpeedRate;
14          var z = Input.GetAxis("Vertical") * Time.deltaTime * RotateRate;
15
16          transform.Rotate(0, x, 0);
17          transform.Translate(0, 0, z);
18      }
19
20      // LocalPlayer 色変更
21      public override void OnStartLocalPlayer()
22      {
23          Transform.find(“Body”).GetComponent().material.color = Color.blue;
24      }
25  }
```

これで、各クライアントで表示される自機の色が青色に変更される。実際に動作させ、確認しておこう。

参考資料　NetworkBehavior

4.2.7. 弾丸の制御

4.2.7.1. 弾丸"bullet"の準備

まず適当に弾丸の Prefab を準備する。

シーン上に “cube” を作成し。名前を、"Bullet" に変更する。適当な大きさに変更した後、Rigidbody コンポーネントを追加した後、Rigidbody コンポーネントの Inspector ビューで、"Use Gravity" のチェックボックスを外し false に設定する。その後、Prefab 化しておき、Hierarchy ビュー上から削除しておく。

4.2.7.2. Player への弾丸発射実装

次に。PlayerController スクリプトを開き、弾丸発射の動作を追加する。

“bullet” と “バレットの spawn 位置” を設定するフィールドを設定した後、発射スクリプトを組み込む。

PlayerController.cs　fire 関数の処理追加

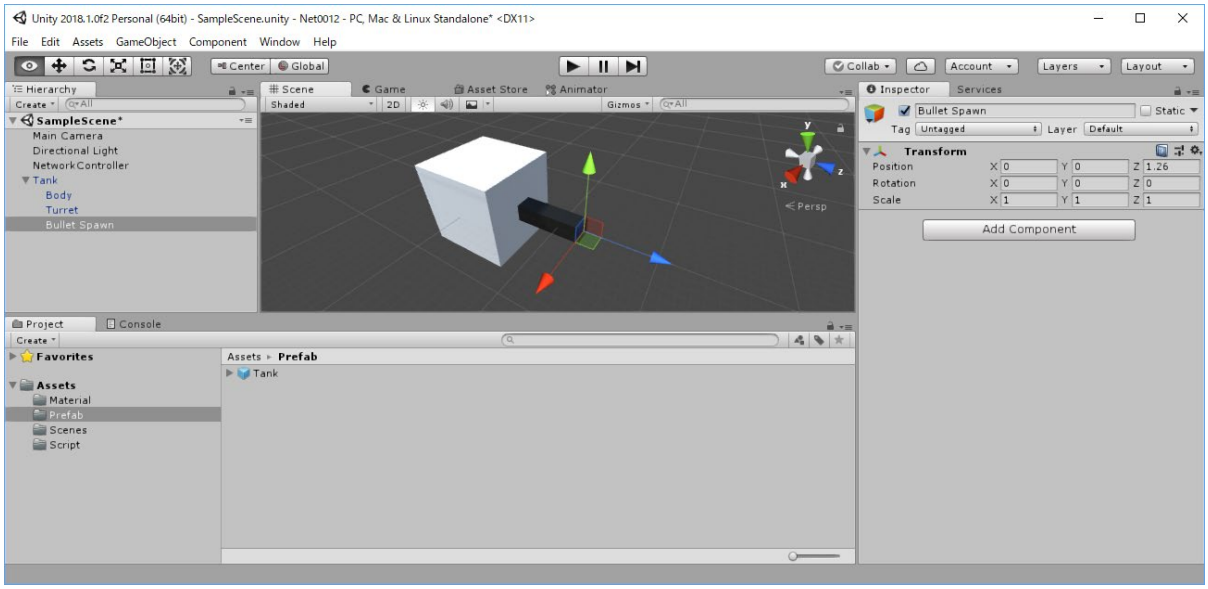
```
 1    using System.Collections;
 2    using System.Collections.Generic;
 3
 4    using UnityEngine;
 5    using UnityEngine.Networking;
 6
 7    public class PlayerController : NetworkBehaviour
 8    {
 9        public float SpeedRate = 150.0f;
10        public float RotateRate = 3.0f;
11
12        public GameObject bulletPrefab;              // bulletPrefab の設定
13        public Transform bulletSpawn;                // bullet 用 spawn ポイントの設定
14
15        void Update()
16        {
17            if (!isLocalPlayer) return;       // LocalPlayer 以外は以下処理しない
18
19            var x = Input.GetAxis("Horizontal") * Time.deltaTime * SpeedRate;
20            var z = Input.GetAxis("Vertical") * Time.deltaTime * RotateRate;
21
22            transform.Rotate(0, x, 0);
23            transform.Translate(0, 0, z);
24
25            // 弾の発射
26            if (Input.GetKeyDown(KeyCode.Space))
27            {
28                Fire();
29            }
30        }
31
32        // 弾発射関数
```

```
33          void Fire()
34          {
35              // Bullet プレハブから Bullet を生成する
36              var bullet = (GameObject)Instantiate(
37                  bulletPrefab,
38                  bulletSpawn.position,
39                  bulletSpawn.rotation);
40
41              // 弾の速度を増加させる
42              bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward * 6;
43
44              // 2 秒後に弾を破壊する
45              Destroy(bullet, 2.0f);
46          }
47
48          // Local Player 色変更
49          public override void OnStartLocalPlayer()
50          {
51              transform.Find("Body").GetComponent<MeshRenderer>().material.color =
52                  Color.blue;
53          }
    }
```

4.2.7.3. bullet の Spawn ポイントの設定

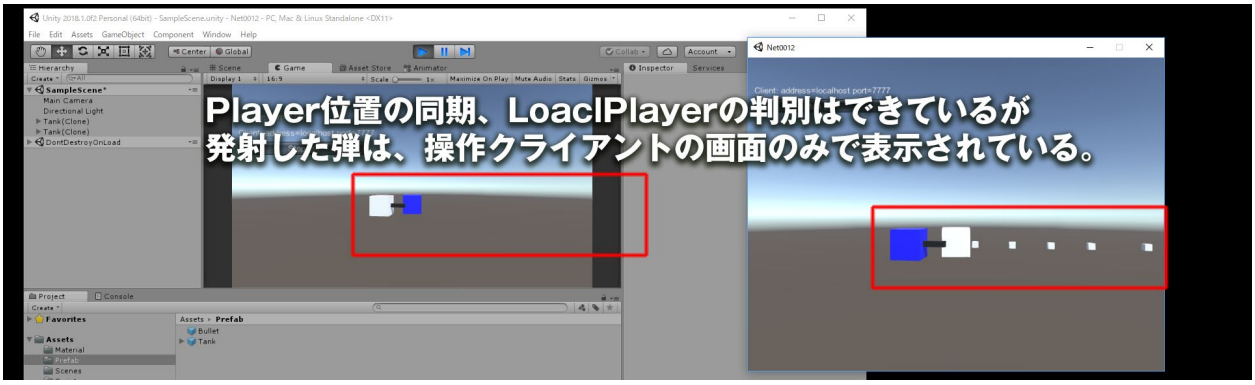
次に、bullet の Spawn ポイントとして、Player の子に空のゲームオブジェクトを作成した後、“BulletSpawn” に名前変更し、その位置を弾を出したい場所に調整したあと Prefab 化ししておく。

下図 BulletSpawn 設定した後の画面



Player Prefab を Hierarchy ビュー上に配置し選択した後。Inspector ビューで “PlayerController スクリプト” の、”bulletPrefab”、“bulletSpawn“ それぞれに設定する。この状態でで準備した、bullet と BulletSpawn に各 Prefab を設定して実行するとクライアントで弾丸が出ることを確認する。

その後、Network 上での動作を確認したものが下図のようになる。ここまででは、操作しているクライアントで弾が表示されるが、片方のクライアントには弾が表示されていないことを確認する。



次は、この弾丸の Network 対応を行う。

4.2.7.4. 弾丸の Network 対応

Bullet Prefab はネットワーク上で一意的に特定可能であるためには “NetworkIdentity コンポーネント” をひとつ持っている必要があり、また弾の位置と角度の同期のために “NetworkTransform コンポーネント” もひとつ必要となる。そしてプレハブ自体は、 生成可能プレハブとして NetworkManager に登録されている必要がある。

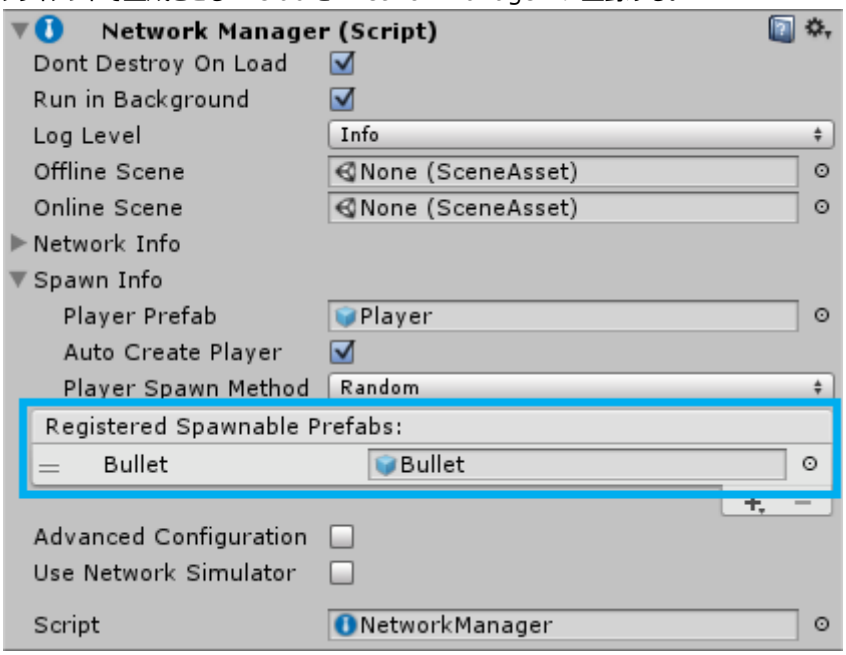
そのため、”Bullet Prefab” に NetworkIdentity コンポーネントと。NetworkTransform コンポーネントをアタッチしておく。

ただし、弾丸の動きに関して、発射された後は速度、方向は一定のため、弾の位置は正確に算出することが可能になる。実は逐一同期する必要はない。そのため、先ほど追加した”NetworkTransform” コンポーネントの Inspector ビューを開き、同期情報を送信するタイミング。“Network Send Rate” を ”0” に設定しておく。

Network Send Rate を 0 に設定することで位置がネットワークを介して同期されなくなり、各クライアントが弾の位置を計算できるようになる。

このように設定することで、ネットワーク全体の転送量が削減され、ゲームのパフォーマンス向上に繋がることになる。

クライアントで生成させる Prefab を NetworkManager に登録する。



4.2.7.5. スクリプトの Network 対応

現在の実行環境は、

「サーバー及び全てのクライアントが、同じゲームオブジェクト、同じスクリプト、同じコードを同時に実行している」

状態である。

その状態で、4.2.4 章で説明したとおり、各クライアントが、 ”isLocalPlayer” を用いて、操作する自機 Player オブジェクトを判別・選択的に制御している。

そしてその他の Player オブジェクトは、サーバからの情報をもとに、各クライアントにある NetworkManager コンポーネントにより、生成され、NetworkTransform コンポーネントにより位置情報がサーバに集約した後、サーバから各クライアントに配布され表示されている事をまず頭に入れておこう。

では、弾丸も同様に処理すればいいのだろうか？弾丸のようなオブジェクトには、下記のような特徴がある。

〇Spawn ポイントは、Player の位置に依存
〇発射された後は、ユーザの操作に依存せず移動する。
〇生成するオブジェクト数が膨大な数になる。

実は、Player 同様に生成した後の位置を逐一同期させた場合、簡単に Network の帯域を使い果たしてしまう可能性があり、パフォーマンスへの甚大な影響が予想される。

そのためここでは、NetworkBehavior に用意された [Command] 属性を利用する。

4.2.7.6. [Command] 属性

[Command] 属性は、それに続く関数がサーバ側で動作するよう設定される。

利用の際には、クライアントから、関数呼び出しのメッセージをサーバに送信するだけで実行されることとなる。

また、サーバに設置する関数名は “Cmd” から始まる必要があるが、引数の引数も自動的に送信される。

そのため “PlayerController スクリプト” の下記の 2 カ所を修正する。

元の表記

```
// 弾発射関数
 void Fire()
 {
     // Bullet プレハブから Bullet を生成する
             :
```

変更後

```
// 弾発射関数
 [Command]
 void CmdFire()
 {
     // Bullet プレハブから Bullet を生成する
```

元の表記

// 弾の発射
if (Input.GetKeyDown(KeyCode.Space))
{
Fire();
}
:

変更後

// 弾の発射
if (Input.GetKeyDown(KeyCode.Space))
{
CmdFire();
}
:

また、NetworkServer.Spawn 関数を追加することで、サーバに接続された、全てのクライアントで弾丸を生成し、サーバがで、オブジェクトを削除すると、自動的に全てのクライアントから当該オブジェクトを削除することが可能になる。

最終的なスクリプトは以下の通りとなる。

PlayerController.cs　fire 関数の処理追加

```
 1  using System.Collections;
 2  using System.Collections.Generic;
 3
 4  using UnityEngine;
 5  using UnityEngine.Networking;
 6
 7  public class PlayerController : NetworkBehaviour
 8  {
 9      public float SpeedRate = 150.0f;
10      public float RotateRate = 3.0f;
11
12      public GameObject bulletPrefab;              // bulletPrefab の設定
13      public Transform bulletSpawn;                // bullet 用 spawn ポイントの設定
14
15      void Update()
16      {
17          if (!isLocalPlayer) return;       // LocalPlayer 以外は以下処理しない
18
19          var x = Input.GetAxis("Horizontal") * Time.deltaTime * SpeedRate;
20          var z = Input.GetAxis("Vertical") * Time.deltaTime * RotateRate;
21
22          transform.Rotate(0, x, 0);
23          transform.Translate(0, 0, z);
```

```
24
25              // 弾の発射
26              if (Input.GetKeyDown(KeyCode.Space))
27              {
28                  CmdFire();
29              }
30          }
31
32          // 弾発射関数
33          [Command]
34          void CmdFire()
35          {
36              // Bullet プレハブから Bullet を生成する
37              var bullet = (GameObject)Instantiate(
38                  bulletPrefab,
39                  bulletSpawn.position,
40                  bulletSpawn.rotation);
41
42              // 弾の速度を増加させる
43              bullet.GetComponent<Rigidbody>().velocity = bullet.transform.forward * 6;
44
45              // Client 上に弾を生成する
46              NetworkServer.Spawn(bullet);
47
48              // 2 秒後に弾を破壊する
49              Destroy(bullet, 2.0f);
50          }
51
52          // Local Player 色変更
53          public override void OnStartLocalPlayer()
54          {
55              transform.Find("Body").GetComponent<MeshRenderer>().material.color =
56                      Color.blue;
57          }
58      }
```

4.2.8. ダメージ処理

Network シューティングゲームで、各プレイヤーのステータス管理も重要な要素となる。弾丸を出すことはできたが、その弾丸が Player に命中したならば、ダメージを受ける。その受けたダメージは、ダメージを受けた以外のクライアントにも同期させる必要がある。そこでまず、シングルプレイヤーで弾丸の処理を実装した後に Network 上でのダメージによるステータス管理の実装を行う。

4.2.8.1. Player の体力

まずは、Player Prefab に、Compornent > Physics > BoxCollider を追加した後下記の通り、体力管理のスクリプトを追加し、Apply しておく。

Health.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3
4   using UnityEngine;
5
6   public class Health : MonoBehaviour
7   {
8       // 体力最大値
9       public const int maxHealth = 100;
10      // 現在の体力
11      public int currentHealth = maxHealth;
12
13      public void TakeDamage(int amount)
14      {
15          currentHealth -= amount;
16
17          // 死亡処理
18          if (currentHealth <= 0)
19          {
20              currentHealth = 0;
21          }
22      }
23  }
24
```

4.2.8.2. 弾丸の衝突判定

Bullet Prefab に対して、敵 Player に当たった際の動作を設定する。まず単一クライアントでの衝突処理を実装してみよう。そのために前章で作成した Bullet Prefab に、下記の通り弾丸が、何らかのコライダーに衝突したときの処理を追加する。

BulletControl.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3
4   using UnityEngine;
5   using System.Collections;
6
7   public class Bullet : MonoBehaviour {
8
9       void OnCollisionEnter(Collision collision)
10      {
11          // 衝突先のコライダー取得
12          var hit = collision.gameObject;
13          // 衝突先の Helth クラス取得
14          var health = hit.GetComponent<Health>();
15
16          if (health  != null)
17          {
18              health.TakeDamage(10);
19          }
20          Destroy(gameObject);
21      }
22  }
```

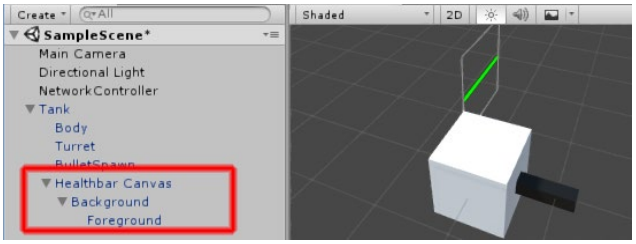
4.2.8.3. Player の体力ゲージ

これで、ダメージを受けるようになったが、ダメージを視覚化する必要がある。そこで、Player に、下記の手順で体力表示用のゲージを追加する。

- ○ PlayerPrefab、今回 TankPrefab を Hierarchy ビューに配置する。
- ○ TankPrefab を選択後その子に、GameObject > UI > Canvas にて Canvas を追加
- ○ Canvas 後、名前を “Healthbar Canvas” に変更する。
- ○ “Healthbar Canvas” を選択し、Inspector ビュー “Canvas” の RenderMode を ”WorldSpace”に変更する。
- ○ Canvas の子に GameObject > UI > Image を追加し、名前を “Background” に変更する。
- ○ Background が選択された状態で、
  RectTransform コンポーネント Width 100 、 Height 10
  Image コンポーネント Source Image をビルトインの InputFieldBackground に設定。
  Image コンポーネント Color を Red にする。

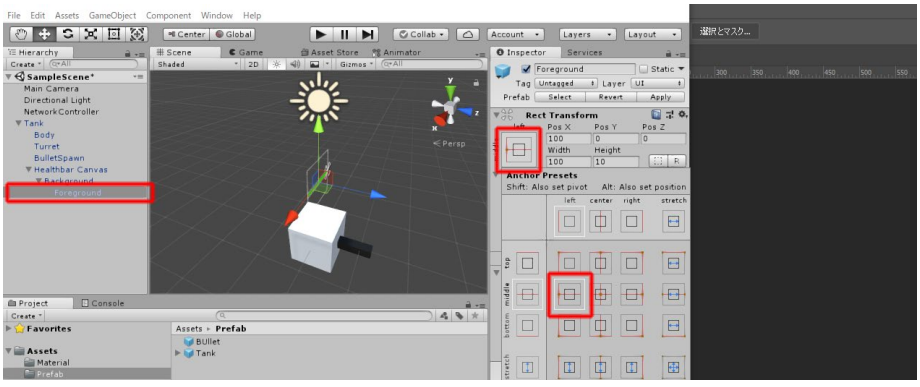
この際、Anchor Presets ウィンドウで Anchor や Pivot は変更しないように注意し、Background の複製を作成しておく。作成した Background の複製の名前を “Foreground” に変更し、Foreground を Background トの子にする。

この状態で体力ゲージの構造はこのようになっているはず。



Healthbar Canvas が選択された状態で、RectTransform コンポーネントをギアメニューとともにリセットし Scale を (0.01, 0.01, 0.01) に Position を (0.0, 1.5, 0.0) に設定する。

- ○ “Foreground” を選択し、Image コンポーネントのカラーを “Green” に設定しておく。
- ○ Anchor Presets ウィンドウを開き、下記の通り Position を Middle・Left に設定する。



- ○ その後 Pivot を (0, 0.5) から (1 .0.5) に変更する。

これで、Player の頭上に Helthbar ができ、ダメージ表示のための UI が準備ができた。

次に、Health スクリプトへ先ほど追加した UI コントロール処理を、下記の通り追加する。

Health.cs

```
1   using UnityEngine;
2   using UnityEngine.UI;                          // UGUI の利用宣言
3   using System.Collections;
4
5   public class Health : MonoBehaviour {
6
7       public const int maxHealth = 100;
8       public int currentHealth = maxHealth;
9       public RectTransform healthBar;            // 制御する Healthbar の取得
10
11      public void TakeDamage(int amount)
12      {
13          currentHealth -= amount;
14          if (currentHealth <= 0)
15          {
16              currentHealth = 0;
17          }
18          // Healthbar サイズのコントロール
19          healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);
20      }
21  }
```

スクリプト変更後、追加した Health コンポーネントの Healthbar フィールドに、先ほど作成した、UI の Forground を指定しておく。

これで、弾丸が Player に当たった場合、ダメージを受け表示がグリーンからレッドに変化するようになる。しかしこのままでは Healthbar の向きが、Player と同じ向きに向いてしまうため、その向きを調整することとする。

UI の向きを、常に画面表示を行っているカメラの方向に向ければ良い。これを Billbord といい、位置や回転などを管理している transform に実装されている LookAt() 関数を使って、メインカメラの向きを設定すればいい。

先ほど作成した“Healthbar Canvas”に“HealthBillboard” という名前のスクリプトを作成した後、追加、Apply しておけば良い。

HealthBillboard.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3
4   using UnityEngine;
5
6   public class HealthBillboard : MonoBehaviour {
7
8       void Update () {
9           transform.LookAt(Camera.main.transform);
10      }
11  }
12
```

現段階では、プレイヤーの「現在の体力」の変化は、それぞれのクライアントとサーバーに個別に適用される。Player が他の Player を撃った際、Bullet スクリプトと Health スクリプトは、それぞれのクライアントおよびサーバー上で独立して「ローカルに」機能する。

しかし、この状態では一切同期が行われていない状態であるが、弾丸は NetworkManager を介して、Spawn（生成）されたオブジェクトとして管理されており、衝突（コリジョン）を検知すると全てのクライアント上で破壊される。

弾丸は全てのクライアント上に存在しており、そのため弾丸とプレイヤーとの間にコリジョンが発生することが可能となり、プレイヤーがダメージを受ける。

ただし、別のクライアント上で衝突が検知される前に、特定のクライアント上で弾丸が破壊されてしまうことも起こり得る。弾丸は同期されている一方、プレイヤーの現在の体力（Current Health）は同期されていないので、各クライアントやサーバにおいてプレイヤーの Current Health が全く異なるという状況も起こり得る。

ここまでの作業が完了した時点で、2 体のプレイヤーの頻繁な撃ち合いを行うと、クライアントによってプレイヤーの「現在の体力」に相違が出るはずだ。

4.2.8.4. Player 体力の Network 化

各クライアントでの違いを無くすためには、Player の体力の変更はサーバーにおいてのみ適用されるべきもので、その後でクライアント上で同期するべきだ。これを “サーバー権限” での動作という。

Unet では、サーバ権限下でステート同期（State Synchronization）という仕組みがあり、Network 上で管理される GameObject に属する変数をサーバで管理し、ローカルに同期できる。これを用いることで現在の体力とダメージのシステムをネットワークに連携させ、サーバー権限の下で機能させる事ができる。

サーバで管理すべき変数をローカルで宣言するためには、[SyncVar]を指定すれば良い。

今回、体力をサーバ権限で管理する設定を行う。そのため、Health スクリプトを開き、スクリプトを Network に対応させ、体力を管理する変数を下記の通り設定する。

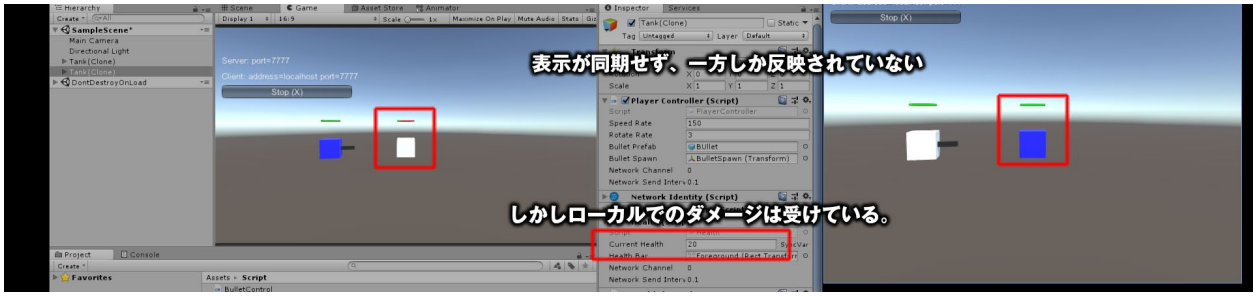
1.名前空間に UnityEngine.Networking を追加
2.基底クラスを NetworkBehaviour に変更する。
3.同期する変数に[SyncVar]属性を設定する。
4.TakeDamage 関数をサーバでのみ実行する。

最終的に、Health スクリプトは下記の通りに変更すれば良い。これで各プレイヤーの体力が全クライアントで同期する。

Health.cs

```
1   using UnityEngine;
2   using UnityEngine.UI;
3   using System.Collections;
4   using UnityEngine.Networking;      // 1.名前空間に  UnityEngine.Networking  を追加
5
6   // 2.基底クラスを  NetworkBehaviour  に変更する。
7   public class Health : NetworkBehaviour {
8
9       public const int maxHealth = 100;
10      [SyncVar]                          // 3.同期する変数に[SyncVar]属性を設定する。
11      public int currentHealth = maxHealth;
12      public RectTransform healthBar;
13
14      public void TakeDamage(int amount)
15      {
16          if (!isServer) return;         // 4.TakeDamage 関数をサーバでのみ実行する。
17
18          currentHealth -= amount;
19          if (currentHealth <= 0)
20          {
21              currentHealth = 0;
22          }
23          //  Healthbar サイズのコントロール
24          healthBar.sizeDelta = new Vector2(currentHealth, healthBar.sizeDelta.y);
25      }
26  }
```

ただしこの状態では、Healthbar はゲームの全てのインスタンスで機能していない事が確認できるが、各プレイヤーの体力管理変数 currentHealth はパブリックであり、エディター上で確認することができる。



これは、Foreground ゲームオブジェクトの RectTransform がネットワーク経由で同期されておらず、Healthbar の Size Delta を設定するコードがサーバーのみで実行されているためである。

Healthbar がホストクライアントで機能するためには、Healthbar の Foreground ゲームオブジェクトの RectTransform を同期する必要がある。

そこで、サーバにより各クライアントで同期された変数を、ローカルに反映させるために、Unet では SyncVar フックという機能が準備されている。これにより同期された変数と関数を関連付けるもので、ダメージを受けた際ゲージサイズを変更させる関数を独立させ [SyncVar(“関数名”)] によって指定すれば良い。

Health.cs

```
1   using UnityEngine;
2   using UnityEngine.UI;
3   using System.Collections;
4   using UnityEngine.Networking;
5
6   public class Health : NetworkBehaviour {
7
8       public const int maxHealth = 100;
9       [SyncVar(hook = "OnChangeHealth")]              // SyncVar フック指定
10      public int currentHealth = maxHealth;
11
12      public RectTransform healthBar;
13
14      public void TakeDamage(int amount)
15      {
16          if (!isServer) return;
17
18          currentHealth -= amount;
19          if (currentHealth <= 0)
20          {
21              currentHealth = 0;
22          }
23      }
24
```

```
25          //  Healthbar サイズのコントロール関数として独立させる
26          void OnChangeHealth (int health)
27          {
28              healthBar.sizeDelta = new Vector2(health, healthBar.sizeDelta.y);
29          }
30      }
```

これで体力ゲージが、各クライアントで同期し、表示ができるようになる。

4.2.8.5. Player の死亡と再配置

現時点では、プレイヤーの「現在の体力」がゼロになった際、何も起こらない。ここでは、「現在の体力」がゼロになった時点でプレイヤーが開始位置に戻されて体力が満タンに戻るように実装する。

[Command] 属性は、クライアントで呼び出され、サーバで実行するが、Player の死亡した場合呼び出すことができない。そのためサーバ側で死亡・回復処理を呼び出し、各クライアント上で実行する必要がある。

Unet ではその場合[ClientRpc] 属性を用いることで実現できる。

関数を ClientRpc コールにするには、[ClientRpc] 属性を使用し、 関数の名前にプレフィックス “Rpc” を追加する。こうすることで、この関数がサーバー上で呼び出された場合にクライアント上で実行されるようになり、全ての引数は自動的に ClientRpc コールの一部としてクライアントに渡される。

また、オブジェクトの再生成（respawn）を有効にするには、Health スクリプト内に Respawn 関数を作成し、プレイヤーの体力が 0 になった際の Respawn 関数への呼び出しを、サーバー上で動作する関数 TakeDamage 内に作成する必要がある。

Health.cs

```
1   using UnityEngine;
2   using UnityEngine.UI;
3   using System.Collections;
4   using UnityEngine.Networking;
5
6   public class Health : NetworkBehaviour {
7
8       public const int maxHealth = 100;
9       [SyncVar(hook = "OnChangeHealth")]
10      public int currentHealth = maxHealth;
11
12      public RectTransform healthBar;
13
14      public void TakeDamage(int amount)
15      {
16          if (!isServer) return;
17
18          currentHealth -= amount;
19          if (currentHealth <= 0)
20          {
21              currentHealth = maxHealth; // 体力を初期値に戻す
22              RpcRespawn();           // サーバーで呼び出され、クライアントで実行される
23          }
24      }
25
26      void OnChangeHealth (int health)
27      {
28          healthBar.sizeDelta = new Vector2(health, healthBar.sizeDelta.y);
29      }
```

```
30          // サーバーで呼び出され、クライアントで実行される関数の定義
31          [ClientRpc]
32          void RpcRespawn()
33          {
34              if (isLocalPlayer)
35              {
36                  // ゼロ地点に戻る
37                  transform.position = Vector3.zero;
38              }
39          }
40      }
```

この状態では、ローカルの Player ゲームオブジェクトの位置をクライアントが制御している。これは、この Player ゲームオブジェクトがこのクライアントにおける ローカル権限 を持っているためである。

プレイヤーの currentHealth が 0 になった際、そのため単純にサーバーが直接その Player ゲームオブジェクトを元の位置に戻したとしたら、権限を持っているこのクライアントはサーバーをオーバーライドしてしまう。

そのようなことをを回避するため、ClientRpc コールという形で、サーバーから該当クライアントに対して、このプレイヤーのゲームオブジェクトを再スタート位置に戻すよう命令を出すことになる。ClientRpc コール設定後は、このプレイヤーゲームオブジェクトの NetworkTransform の働きにより、全てのクライアント上でこの位置に同期が行われることになる。

これで、「現在の体力」が 0 になるまでダメージを受けると、ダメージを受けた方のプレイヤーゲームオブジェクトが元々の位置に転送され、「現在の体力」は最高値に戻る。

4.2.8.6. プレイヤーの Spawn 位置を設定する

このままでは、全ての Player は、初期生成及び死亡した場合同じ場所に生成されることになる。そのため、ゲームの開始時には、プレイヤーのどれかを動かした後に別のプレイヤーが参加するということのない限り、全てのプレイヤーが同じ位置に存在する状態になっている。

しかし、プレイヤーはそれぞれ異なる位置に生成されるようにするのが理想的で、Unet では生成位置をハンドルする機能がビルトインされた NetworkStartPosition コンポーネントを使用して行うことができる。

ここでは、まず 2 つの異なる Spawn 位置を設定し、初期生成時、各プレイヤーを異なる場所に生成する機能を実装する。2 つの異なる生成位置を作成するには、それぞれに NetworkStartPosition コンポーネントの添付された新規ゲームオブジェクトを、下記の手順で 2 つ作成する必要がある。

- 空のゲームオブジェクトを新規作成し、ゲームオブジェクトの名前を “Spawn Position 1” に変更する。
- Spawn Position 1 ゲームオブジェクトが選択された状態で、
  ... Network > NetworkStartPosition コンポーネントを追加する。
  ... Transform Position を (3, 0, 0) に設定する。

- 上記処理が終わったら Spawn Position 1 ゲームオブジェクトの複製を作成し、名前を “Spawn Position 2” に変更する。
- Spawn Position 2 ゲームオブジェクトが選択された状態で、
  ... Transform Position を (-3, 0, 0) に設定する。

- Hierarchy ウィンドウで Network Controller ゲームオブジェクトを選択。
- Network Controller ゲームオブジェクトが選択された状態で、
  ... Spawn Info のドロップダウンメニューを開く。
  ... Player Spawn Method を Round Robin に変更する。

Spawn Position ゲームオブジェクトは NetworkStartPosition コンポーネントを持っているので、NetworkManager によって自動的に検知されている。そのため、NetworkManager は、クライアントがサーバーに接続されると、そのクライアントのゲームオブジェクトに添付された NetworkStartPosition の Transform Position によって設定される位置に、新しいプレイヤーゲームオブジェクトを生成する。

Player Spawn Methods（プレイヤーの生成方法）は、Random と Round Robin の二種類が設定できる。

Random は、使用可能な NetworkSpawnPosition の位置の中からランダムに選んだ位置を使用するのに対し、Round Robin は、（同じ位置をランダムに繰り返し使うことなく、）使用可能な全ての生成位置を順番に（周期的に）使用する。

Random の場合、複数のプレイヤーゲームオブジェクトに同じ生成位置が使用されることも十分にあり得る。これは、接続されたクライアントの数にもよるが。Round Robin の場合、生成位置の数よりクライアントの数が多い場合以外は、生成位置が重複して使用されることはない。

ゲームが開始されると、シーン内の異なる場所にプレイヤーが生成される。

4.2.8.7. Respawn 位置の設定

前項と同じように Spawn Position ゲームオブジェクトに NetworkStartPosition コンポーネントを使用した、ReSpawn 処理を実装する。

再生成位置の保存方法として、生成位置の配列を作成し、プレイヤーの再生成時の位置をその中から選ぶことができるようにする、そこで Start 関数内のローカルプレイヤーの生成位置を ReSpawn 位置として確保する方法を実装する。

この方法では、Network Manager で Round Robin Player Spawn Method により選択される生成位置は、該当プレイヤーによって固定されることになる。

再生成の仕組みを作成するには、配列を作成し、Start 関数で NetworkStartPosition がアタッチされたゲームオブジェクトを検索して配列に追加し、そのオブジェクトの Transform Position を再生成位置として使用する。

これは、先ほど実装した Network Manager によって自動的に行われる開始位置の処理と異なり、Round Robin ではなく Random を使用する。

これらを実装したものが下記のスクリプトになる。

Health.cs

```
1   using UnityEngine;
2   using UnityEngine.UI;
3   using UnityEngine.Networking;
4   using System.Collections;
5
6   public class Health : NetworkBehaviour {
7       public const int maxHealth = 100;
8       public bool destroyOnDeath;
9
10      [SyncVar(hook = "OnChangeHealth")]
11      public int currentHealth = maxHealth;
12
13      public RectTransform healthBar;
14
15      private NetworkStartPosition[] spawnPoints;
16
17      void Start ()
18      {
19          if (isLocalPlayer)
20          {
21              spawnPoints = FindObjectsOfType<NetworkStartPosition>();
22          }
23      }
24
25      public void TakeDamage(int amount)
26      {
27          if (!isServer) return;
28
29          currentHealth -= amount;
```

```
30              if (currentHealth <= 0)
31              {
32                  if (destroyOnDeath)
33                  {
34                      Destroy(gameObject);
35                  }
36                  else
37                  {
38                      currentHealth = maxHealth;
39
40                      // サーバーで呼び出され、クライアントで実行される
41                      RpcRespawn();
42                  }
43              }
44          }
45
46          void OnChangeHealth (int currentHealth )
47          {
48              healthBar.sizeDelta = new Vector2(currentHealth , healthBar.sizeDelta.y);
49          }
50
51          [ClientRpc]
52          void RpcRespawn()
53          {
54              if (isLocalPlayer)
55              {
56                  // 生成位置指定用変数
57                  Vector3 spawnPoint = Vector3.zero;
58
59                  // 生成位置の配列があって、それが空でない場合は、ランダムに 1 つ抽出する
60                  if (spawnPoints != null && spawnPoints.Length > 0)
61                  {
62                      spawnPoint =
                            spawnPoints[Random.Range(0, spawnPoints.Length)].
                            transform.position;
63                  }
64
65                  // プレイヤーの Position を、選択された生成位置に設定する
66                  transform.position = spawnPoint;
67              }
68          }
69      }
```