

SMARTFOXSERVER 2X FPS TUTORIAL

Thomas Hentschel Lund

© 2010-2012 gotoAndPlay() www.smartfoxserver.com and Full
Control www.fullcontrol.dk

Users Guide

Table of Contents

| | |
|---|----|
| Unity 3D Client Installation | 1 |
| Authoritative Server | 5 |
| UDP and TCP Usage | 6 |
| Dynamic Room Creation | 7 |
| Extension Communication | 7 |
| Shooting | 8 |
| World Simulation Model..... | 11 |
| Interpolation and Extrapolation of Remote Players | 12 |
| Prediction..... | 15 |
| Server Time Loop | 15 |
| Server Cheat Prevention | 15 |
| Extrapolation of Rotation | 16 |
| Server Side World Collision Model | 16 |

Tutorial Installation and Setup

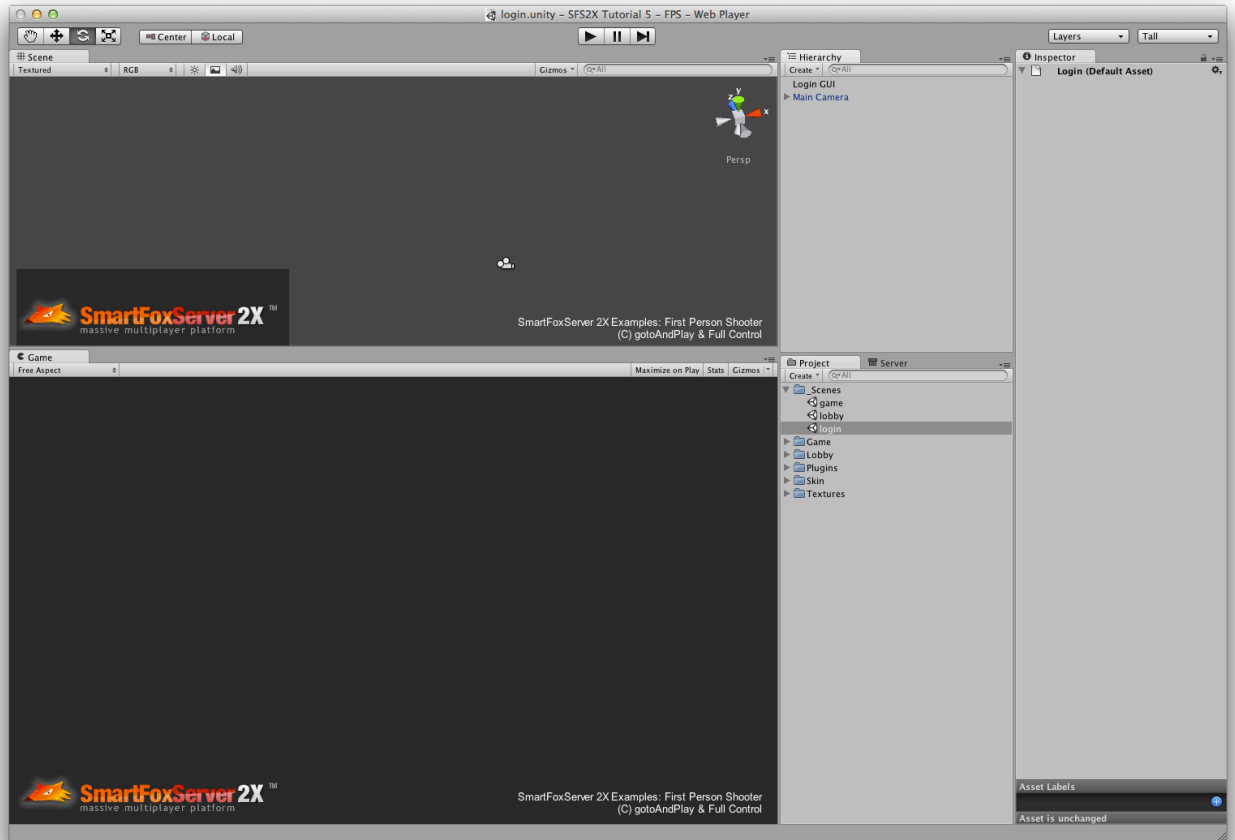
This tutorial showcases an first person shooter (FPS) implementation on SmartFoxServer 2X (SFS2X) with Unity 3D. The tutorial contains all the source code, and contains a SFS2X extension and a Unity project.

Running the Unity 3D Client

Once that the SFS2X server is up and running with the proper extension installed, we can use the Unity 3D client applications to join the FPS world.

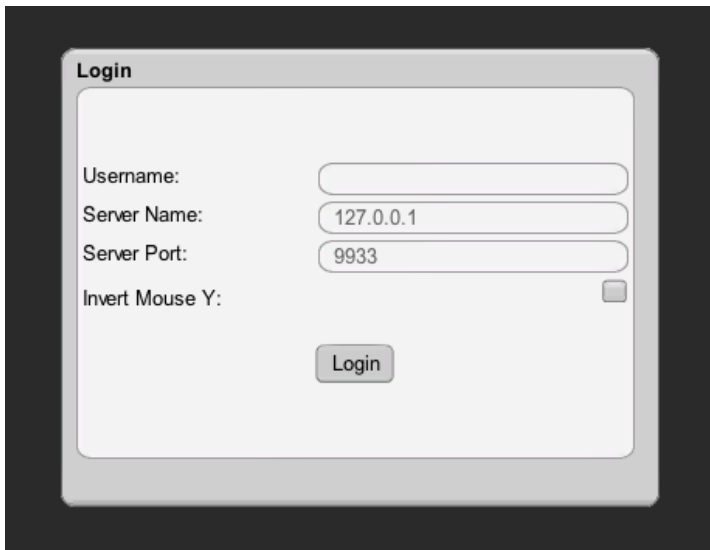
1. Open the folder in Unity 3D, as it is the root folder of the Unity client application
2. Once inside of Unity, open the 'login' Scene and start the Scene in Unity.

SMARTFOXSERVER 2X FPS TUTORIAL - USERS GUIDE



At this point, you should see the login screen (as seen below). Enter the username of your choosing, the IP address and port of your SFS2X server, and press the Login button

*** Select to 'Invert Mouse Y' will switch the mouse to 'Airplane Controls, (i.e. up is down, down is up).



A login window titled "Login" with a light gray background. It contains four input fields: "Username:" (empty), "Server Name:" (containing "127.0.0.1"), "Server Port:" (containing "9933"), and "Invert Mouse Y:" (with an unchecked checkbox). A "Login" button is positioned below the input fields.

3. We are now at the Lobby Screen where we can ...

- View Logged in Users
- View/Join Created Games
- Chat with Logged in users
- Create New Games
- Logout



The lobby screen features a large "Chat" window on the left with a message from "Thomas" saying "Lets play some shooter!!!". To the right are two smaller panels: "Game List" showing "No games available to join" and a "New game" button; and "Users" showing "Thomas" and a "Logout" button. At the bottom left is the "SmartFoxServer 2X" logo with the tagline "massive multiplayer platform". At the bottom right, text reads "SmartFoxServer 2X Examples: First Person Shooter (C) gotoAndPlay & Full Control".

4. To create a game, click the 'New game' button.

SMARTFOXSERVER 2X FPS TUTORIAL - USERS GUIDE

5. At this point, you will be automatically entered into the new game you have just created. Other players can join by running an instance of the unity client, logging in, going to the lobby and then picking your game out of the 'Game List'.



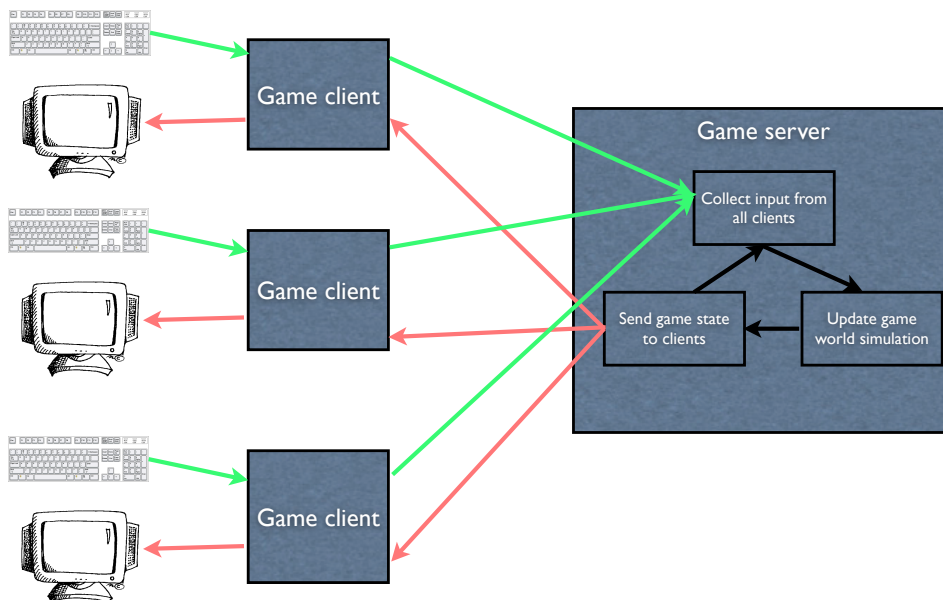
Tutorial Features and How it Works

Now that the game is installed and running, its time to take a look at how things work from a code perspective. This chapter will explain how the client and server communicate and where things are configured, so it is easier for you to modify the example into your own game.

Authoritative Server

The extension acts as authoritative server for the game clients. This means that the simulation runs on the server side, and all clients only have a representative simulation running on their clients. The server always is right. This is one of the best ways to prevent e.g. cheating.

The way this works is that the clients do not run the game code. They take input from the player (mouse, keyboard etc) and check these for validity. If valid, they are applied to the world simulation on the server. State changes are then sent to the clients on a need-to-know basis. The client purely acts as an input/output device.



In a real world scenario, some things can be put onto the client like basic collision checks and similar – as long as they are verified on the server side.

In the FPS tutorial we have cheated somewhat to reduce complexity. The clients run geometry collision checks and only the shooting is determined on the server. There is no verification of any input data either, although there is the following hook to add your own code for this in World.java:

```
private boolean isValidNewTransform(CombatPlayer player, Transform newTransform) {  
    // Check if the given transform is valid in terms of collisions, speed hacks  
    etc  
    // In this example, the server will always accept a new transform from the  
    client  
    return true;  
}
```

Optimally the client should not send a transform, but a request to walk forward. In the above code hook we could check for speed hacks and similar. But since the client sends a complete transform, we would not be able to directly check for aim bots or similar cheats.

UDP and TCP Usage

SFS2X supports extension communication using both UDP and TCP. For a shooter it is imperative to have the fastest possible transformation synchronization possible, while at the same time not missing displaying animations or having an up to date health status. UDP while fast and small is also unreliable, and thus one has to use it selectively.

For the tutorial it was decided to use UDP for sending transforms, and TCP for sending everything else (animation synchronizations, health updates, shot messages, chat etc). Missing a few transforms will not be vital for the game, as they are being sent multiple times a second and the game clients will interpolate/extrapolate movement anyways to smooth out anything that is missed.

The sending of UDP messages is very simple. When sending the ExtensionRequest, you as a developer simply add a “true” boolean as parameter, and the given request is sent via UDP.

In the tutorial we do this in the NetworkManager.cs here:

```
public void SendTransform(NetworkTransform ntransform) {  
    Room room = smartFox.LastJoinedRoom;  
    ISFSObject data = new SFSObject();  
    ntransform.ToSFSObject(data);  
    ExtensionRequest request = new ExtensionRequest("sendTransform", data, room,  
true); // True flag = UDP  
    smartFox.Send(request);  
}
```

Beware, that you should not overuse UDP. In the opinion of the author (me) UDP was invented in a different millennium to run games over 14.4k modems. Especially when running behinds certain firewalls, filters and

similar, UDP can actually turn out to be slower than TCP and more taxing on network equipment hardware. Additionally UDP packets will certainly get lost once in a while. Thus never use UDP for requests that **have** to get to the server and back.

Dynamic Room Creation

The FPS tutorial implements a generic lobby system and can host multiple FPS game instances. To be able to do this the lobby has to be able to dynamically create and destroy rooms and attach the FPS server code to each of these rooms. This way each room will host its own little world simulation that is independent from the other room instances.

In more advanced scenarios one could even run different server extensions in different rooms, be it different games (e.g. having some rooms be FPS, some rooms hosting racing games and others being chat rooms) or maybe different FPS games with different levels.

The dynamic creation and attachment of the rooms used in the FPS tutorial all happen client side inside the LobbyGUI.cs.

```
if (GUI.Button (new Rect (0, 70, 80, 20), "New game")) {
    showNewGameWindow = false;

    if (roomName.Equals("")) {
        roomName = smartFox.MySelf.Name + " game";
    }
    RoomSettings settings = new RoomSettings(roomName);
    settings.GroupId = "game";
    settings.IsGame = true;
    settings.MaxUsers = (short)numMaxUsers;
    settings.MaxSpectators = 0;
    settings.Extension = new RoomExtension(NetworkManager.ExtName,
    NetworkManager.ExtClass);
    smartFox.Send(new CreateRoomRequest(settings, true, smartFox.LastJoinedRoom));
}
```

The code waits for the user to press the “New game” button in the popup window where the user could type in a room name and maximum number of players for the room.

These values are put into the RoomSetting object together with the information that this is a game room (allowing such things as spectators), the maximum number of players – and most importantly it requests the server to add the FPS extension to this room that we copied into the servers extensions folder earlier.

Extension Communication

Clients communicate with the game code on the server side via extension requests and gets data back via extension responses. This model is pretty simple to use but ultra flexible. The data that is sent or received are SFObjects, which are highly optimized data objects that can be nested.

This is all best illustrated with an example.

Shooting

When the player in the game client wants to shoot, he presses the left mouse button as usual. In the ShotController.cs this is handled in the code shown below. The code does some initial checks of ammo existing (no need to send a shot request to the server if there is no ammo in the gun).

```
void Update () {
    if (loadedAmmo > 0 && Input.GetMouseButtonDown(0)) {
        DoShot();
    }
    ...
private void DoShot() {
    NetworkManager.Instance.SendShot();
}
```

The NetworkManager is the centralized code for all communication with the SFS2X server in this tutorial. Here we thus handle the actual shot request. The ExtensionRequest takes the parameters for “request command”, “data to be sent to server” and an optional room to send this request to.

In this case we send a “shot” command with empty data to the current room that we joined, which is our game instance room.

```
public void SendShot() {
    Room room = smartFox.LastJoinedRoom;
    ExtensionRequest request = new ExtensionRequest("shot", new SFSObject(), room);
    smartFox.Send(request);
}
```

On the server side we will receive this request in the FpsExtension class. This is the main entry point for all client communication on the server side. When this class was initialized, it registered to listen for “shot” commands and if any of those would arrive, the execution of what should happen would be handled in the ShotHandler class. The registration code is shown below

```
@Override
public void init() {
    world = new World(this); // Creating the world model

    // Subscribing the request handlers
    ...
    addRequestHandler("shot", ShotHandler.class);
}
```

The shot handler is very simple. It tells the World that a given user requested to shoot his gun

```
@Override
public void handleClientRequest(User u, ISFSObject data) {
    RoomHelper.getWorld(this).processShot(u);
}
```

The code in the World class now converts the SFS User to the CombatPlayer object associated with it (see world simulation model later in this document). It does various checks to see if the given CombatPlayer is even able to shoot his weapon. Then it tells the weapon to shoot. The second part of the method now handles the result of the shooting. As a minimum we need to send some information back to the user who fired his gun that the ammo count changed. Additionally we want to notify everyone in the simulation that the given player fired his gun (so they can run animations and play gun shot sfx). Last but not least we want to see if the shot actually hit someone.

```
// Process the shot from client
public void processShot(User fromUser) {
    CombatPlayer player = getPlayer(fromUser);
    if (player.isDead()) {
        return;
    }
    if (player.getWeapon().getAmmoCount() <= 0) {
        return;
    }
    if (!player.getWeapon().isReadyToFire()) {
        return;
    }

    player.getWeapon().shoot();

    extension.clientUpdateAmmo(player);
    extension.clientEnemyShotFired(player);

    // Determine the intersection of the shot line with any of the other layers to
    check if we hit or missed
    for (CombatPlayer pl : players) {
        if (pl != player) {
            boolean res = checkHit(player, pl);
            if (res) {
                playerHit(player, pl);
                return;
            }
        }
    }

    // if we are here - we missed
}
```

We will not go deeper into the actual handling of the shot – you can trace through the code from here yourself if interested. What we will do now is trace back one of the multiple responses that are sent – namely the update of the ammo.

The method that sends back the response is situated in the FpsExtension as one can see above and is called clientUpdateAmmo(). The code looks very similar to the client code sending the ExtensionRequest in the first place. This time we want to send back data from the world simulation model.

As we are running an authoritative server setup we send the absolute values to the client. In case anyone cheated and tried to hack e.g. maximum ammo values or similar, then those values would get overwritten on the client side.

The `send()` method on the extension class sends back a “response command”, “data to be send to client(s)” and “one or more SFS Users”. This case we only send the ammo count back to the client that is connected to the given `CombatPlayer` that shot the gun. This is by purpose in a need to know basis. Sending too much info to players that do not need to know will simply open up for cheating.

```
// Send message to clients when the ammo value of a player is updated

public void clientUpdateAmmo(CombatPlayer player) {
    ISFSObject data = new SFSObject();
    data.putInt("id", player.getSfsUser().getId());
    data.putInt("ammo", player.getWeapon().getAmmoCount());
    data.putInt("maxAmmo", Weapon.maxAmmo);
    data.putInt("unloadedAmmo", player.getAmmoReserve());

    this.send("ammo", data, player.getSfsUser());
}
```

SFS2X now takes care of the magic of transporting the data back to the client. Next place we see the response is again in the `NetworkManager` in the `OnExtensionResponse` method. That is the method that we registered for handling `EXTENSION_RESPONSE` events from the server.

```
private void SubscribeDelegates() {
    smartFox.AddEventListener(SFSEvent.EXTENSION_RESPONSE,
    OnExtensionResponse);
}
```

The parameters we send – the command and the data object – are easily retrieved from the event object, and the “ammo” command is then handled by reading out the data we put in earlier for current ammo count etc. and updated for our GUI to display.

```
private void OnExtensionResponse(BaseEvent evt) {
    try {
        string cmd = (string)evt.Params["cmd"];
        ISFSObject dt = (SFSObject)evt.Params["params"];

...
        else if (cmd == "ammo") {
            HandleAmmoCountChange(dt);
        }
...
    }

...
// Ammo count changed message from server
private void HandleAmmoCountChange(ISFSObject dt) {
    int userId = dt.GetInt("id");
}
```

```
if (userId != smartFox.MySelf.Id) return;

int loadedAmmo = dt.GetInt("ammo");
int maxAmmo = dt.GetInt("maxAmmo");
int ammo = dt.GetInt("unloadedAmmo");

ShotController.Instance.UpdateAmmoCount(loadedAmmo, maxAmmo, ammo);
}
```

That's it! All communication with the extension uses this model. Nice, simple and easy to work with.

World Simulation Model

The primary part of the server extension is the world simulation. This is a data object representation of what is going on inside the game world.

The objects and their relationships can be seen in the following diagram. It also contains some of the primary attributes and methods on each class.

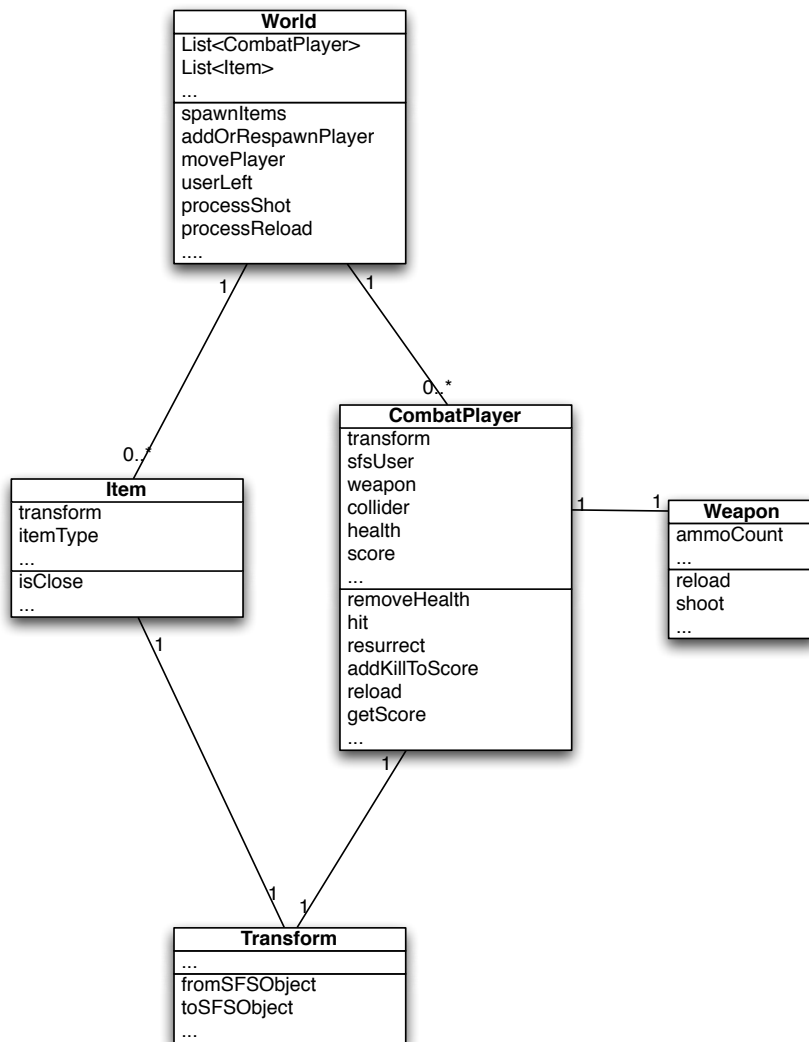
There is the World itself – which contains primarily a list of players and a list of the items that are spawned in the world.

Each Item has a type (HealthPack or Ammo) as well as a Transform for its position in the world

Each CombatPlayer is linked to a SFS2X User, so we can relate a given network command to the specific CombatPlayer in the simulation. Each player has a Weapon that he can fire, a Transform for where he is located in the simulation as well as attributes for score, health. Additionally there is a Collider object attached, which is used in the shooting collision detection.

Weapons are simply that – containing ammo counts and methods to reload, shoot etc.

The Transform class has 2 primary methods to transform itself to and from an SFSObject. These are the SFSObjects send back and forth between the clients and the server.



So basically any extension request (e.g. a player spawn) will change this world simulation model. It will be the definite and authoritative data representation of the game. Whatever the clients render and see is simply a copy of this model visualized with graphics and audio.

Interpolation and Extrapolation of Remote Players

Every client in the game will always receive transforms of the remote players with a delay. When a given client sends its transform to the server at time T1, then the server receives this after some network delay and adds it to its model at time T2. The server then sends out all transforms to everyone else who then receive it after network delay at time T3. So at this time T3 the clients are capable of rendering the remote player where he was at time T1.

This delay is simply a matter of fact in any online game.

Additionally a modern computer can render a scene at 50-100 frames a second (or faster). Network messages are only send/received maybe 10-20 times a second. If a game client only renders the transforms as received by

the server, it would not only be drawing remote players at “old” positions, it would also render them 4-5-6 times at the same position before updating the position as received by the server. This is very noticeable. Sending transforms faster is **not** the solution.

There are basic 2 different techniques used to counter this in the example – interpolation and extrapolation.

Interpolation is used to move the position of the player between 2 received transforms in a smooth matter. So if the x coordinate of a transform is 5 and the next one received by the server is 10 – then the game client can calculate (with the current framerate) how much it should move a remote player from x=5 to x=10 to make movement smooth.

This counters the different in render speed vs. receiving the network transforms, but is adding an additional delay into the position of the client. Now we are even another time step behind the T1 position, as interpolation is done on 2 already received transforms.

This “lag” is not a problem in online worlds or games where 99% accurate positioning is important for the gameplay, but for first person shooters or racing games this becomes a problem.

To continue the example above, a given client would at T4 (in the interpolation case) see the player where he was at T1 (most likely at least 200-300 milliseconds). If the client aims at this remote player and shoots at time T4, then the shot request is sent to the server at T5, applied to the server game model at T6. The server would look at where the shot was aimed and compare to the position of the target player and declare it a miss.

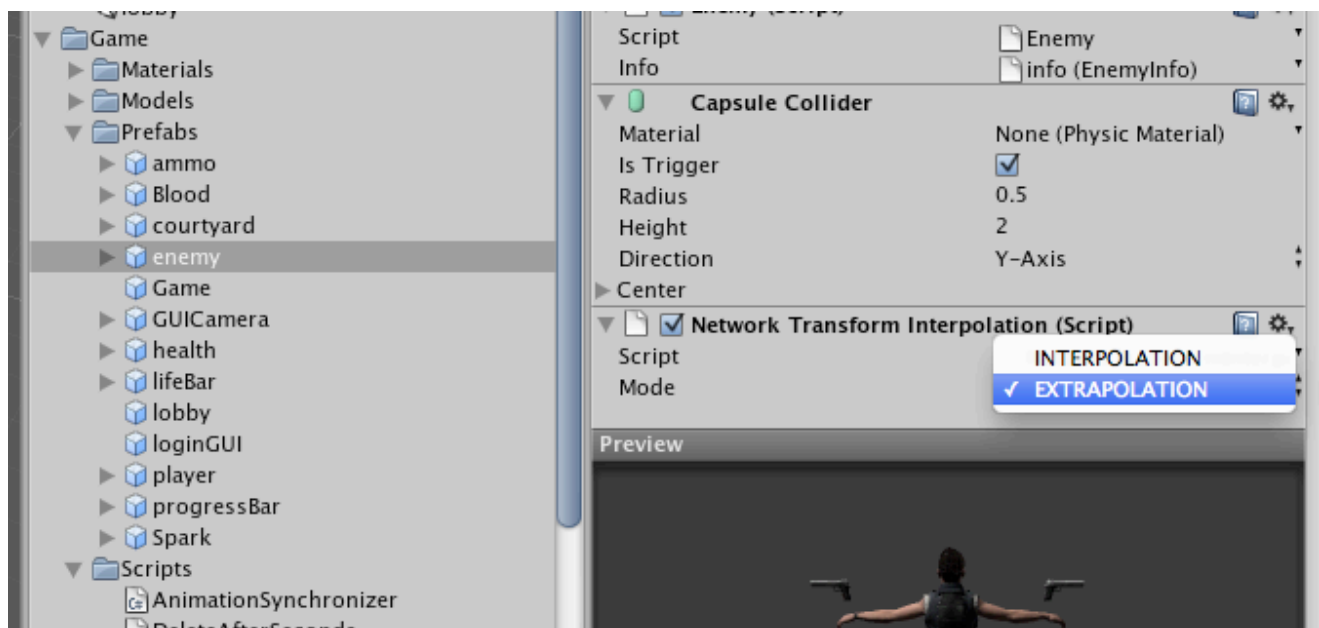
What the game client needs to render instead is a guess of where a player is **right now** – by extrapolating the remote players historic movement and take a wild guess at the current position. By drawing remote players at the position where it thinks that the player will be in 200-300 milliseconds, a shot aimed at a given point will on the server thus be compared to the real position – and be as precise as possible.

On the other hand a remote player can suddenly change movement pattern, and the clients running the extrapolation will not know this until they get the transforms. Thus extrapolation becomes slightly jittery as the server tries to correct its extrapolation position to what really happened.

The tutorial code supports very basic extrapolation by looking at the last 2 transforms, their time difference and the current simulation time. It creates a basic movement vector and multiplies player speed to make a guess at where the remote player is.

This is most likely the simplest extrapolation possible and is not perfect. But it can serve as a basis for making a more advanced algorithm.

The code supports switching between interpolation and extrapolation using a dropdown on the NetworkTransformInterpolation component on a enemy prefab as shown below.



So if you need precise and fluid rendering of remote players at the cost of accuracy, then use interpolation. If you want to sacrifice precision to get accuracy then use extrapolation.

Improvement Suggestions

The example code is not a game. It is not meant to be a finished game. It contains shortcuts and deficiencies and is not a “press play button to make a FPS MMO”. Due to time constraints and keeping code simpler there are places that you need to write code if using the example as a skeleton. This chapter points at a few known places where you have to do “something”.

Prediction

Game clients in real games are never ever to be trusted. In the tutorial we trust the clients to send us their transforms, and this can lead to severe cheating attempts. There is nothing preventing a client from using speed hacks and aim bots (by sending us fake transforms).

Server Time Loop

The server simulation currently runs without a scheduled timer loop. Whenever anything is happening (requests from clients), responses are sent out to everyone connected to the same game.

This also has a scalability problem built-in, as every time a client sends its transform – then the server immediately applies it to the model and sends out the new transform to everyone. The more players connected, the more individual transforms are being sent around.

In a real life scenario, the server simulation would have a e.g. 30 millisecond scheduled update loop, where it takes all received requests and processes these, applies them to the game world and sends out status responses to clients. This also adds the possibility to batch the transforms into a single response containing all and is much more efficient.

Server Cheat Prevention

There is a dummy method on server to put in validation of received transforms from clients before sending them to the other clients. Here it is possible to add e.g. simple speed hack checks.

This is discussed briefly in another part of the document.

To take cheat prevention even further, one could also rework the sending of the transforms into sending the keypresses to the server for validation **before** they are applied to the transform. That would give the server a way to validate any kind of input but at the cost of responsiveness. To counter this responsiveness one would then use a technique called prediction to make the individual client avatar models move right away and then later correct their positions if e.g. the server decided to deny a movement command.

Extrapolation of Rotation

Only the position part of the transform is extrapolated. The rotation of players could thus be made smooth by adding interpolation or extrapolation to the rotation too. The code would be very similar to the existing position interpolation/extrapolation code.

On a side note, the separate head animation is not synchronized at all in the tutorial code.

Server Side World Collision Model

Optimally the server runs all collision checks instead of the players also for level geometry. This would prevent users from sending transforms that would make them walk through walls and similar.

To do so one could write editor scripts in Unity that export all collision meshes to a data file, which is then imported on the server and used for collision checks.