Copyright © Pixel Crushers

MULTIPLAYER GUIDE

Table of Contents

Multiplayer Guide 1
- Overview 1
- Instanced, Limited-Player Multiplayer Games 2
- Persistent Data Limited-Player Games 3
- Massively-Multiplayer Online Games (MMOs) 3

Overview

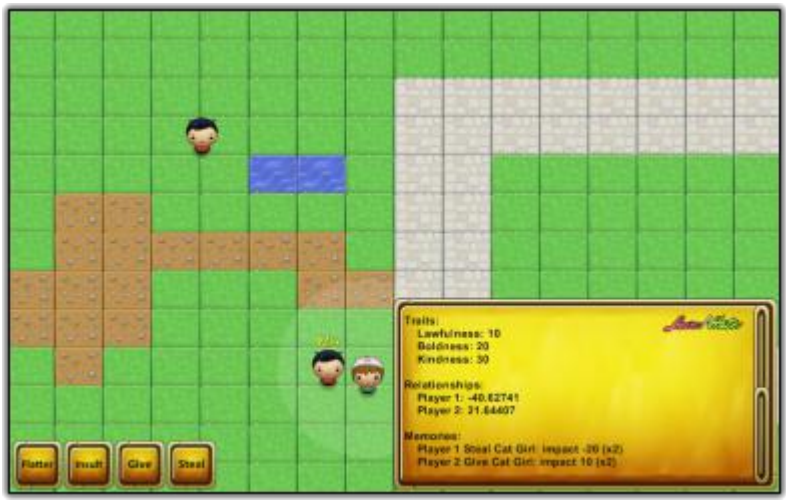
Multiplayer game development is primarily data management. Multiplayer games must do two things:

1. Identify what data need to be synchronized between players, and
2. Synchronize it in a reliable, secure, and timely manner.

How this is implemented depends on your multiplayer architecture: Is it a massively-multiplayer game like *World of Warcraft* or a limited-player game like *Left 4 Dead*? Is the data centralized on an authoritative or non-authoritative server, or mirrored across clients using peer-to-peer connections?

The remainder of this document describes implementation strategies for various architectures.

Instanced, Limited-Player Multiplayer Games



In an instanced, limited-player multiplayer game, like Valve's *Left 4 Dead* or BioWare's *Neverwinter Nights*, you may want all clients to have complete relationship information about all other PCs and NPCs.

To do this, create factions in your faction database for all possible players (e.g., Player 1, Player 2, etc., or “Bill” and “Zoey” as in *Left 4 Dead*). These will act as player slots in the faction database. Create additional factions for your NPCs. As each player joins the game, assign one of the player slots.

To keep the faction database and faction members in sync across the clients, add scripts that implement Love/Hate's event handlers such as IRememberDeedEventHandler and IModifyPadEventHandler. See the *Event Handlers* section for details. If a deed happens within view of two clients, the witness on both clients will raise an OnRememberRumor event and send a message to the remote clients.

If you change faction parents, for example using FactionDatabase.AddFactionParent(), you'll have to synchronize this information to the other clients yourself.

To save a game, refer to the Love/Hate manual's *Saving and Loading* section. Use FactionManager.SerializeToString() and FactionMember.SerializeToString() to get serialized data strings. To load, send this data back to the components using DeserializeFromString(). If you're also using the Dialogue System for Unity, you can rely on the Dialogue System's save subsystem instead.

The included example demonstrates how to keep data synchronized in a simple, instanced, limited-player scene.

Persistent Data, Limited-Player Games

Some games, like Blizzard's *Diablo*, use instanced worlds but also retain persistent character data. You may want to include Love/Hate data in your persistent character data. To do this, handle faction members as described in the *MMOs* section below. However, you may choose to record data only at checkpoints, when the host player saves, or when the player logs out, rather than sending and validating every change with the server as it occurs.

Massively-Multiplayer Online Games (MMOs)

In an MMO with potentially millions of players, it's neither practical nor even useful to create faction slots for all players. Player 1 cares about NPCs' relationships to Player 1, not to Player 999999 whom Player 1 has probably never met and never will.

Instead, each client should maintain Love/Hate data for the client's local player. Your faction database should have one Player faction, for the local player.

When the client logs in, the server should send faction manager and faction member data to the client as serialized strings. See the Love/Hate manual's *Saving and Loading* section for the format of the serialized strings. The client should then apply this using FactionMember.DeserializeFromString() and FactionMember.DeserializeFromString(). The server doesn't have to run Love/Hate; it only needs to send and receive data in the serialized string format.

Add scripts that implement Love/Hate's event handlers such as IRememberDeedEventHandler and IModifyPadEventHandler. See the Love/Hate manual's *Event Handlers* section for details. When a faction member on a client modifies its PAD, for example, your script should send the new PAD values to the server. The server can then validate the values and record them. If the values are invalid (an indication of cheating), the server can reject the values and tell the client to use different values instead.

When a client logs out, the data is already on the server, so you don't need to do anything else.

If your MMO has a huge number of factions that don't often need to know about each other, you might choose to send individual faction or faction member data as needed, rather than sending it all upfront to the client. In this case, you can add a small script to the faction member and/or faction manager that keeps track of whether the data is current with the values on the server. If not, it can pull the current data from the server.