

RPG Series: Multiplayer Framework

by Dave Young 2011 <http://www.daveyounggames.com>

Introduction

This document will serve as the basic documentation for the Multiplayer Framework I have been working on for the last couple of weeks. It should help you find specific things and give some small detail as to what the functions are and how to use them.

I'll only describe the areas and the more important functions/flow, and you have the code to examine for the rest. I would have liked to have given more detail, but so much has happened in the restructuring and refactoring that I didn't have time.

The new content is broken up into several areas:

Lobby: Host/Join

Game Server: Auto-Swapping Client Based Server

Platforms

Targetting

Net Objects

Game Lobby: List, Host, Join Games

We modified the connection process a bit. After connecting to the server in `Network_Connection.Connection_onLoop ()`, we make a call to `this.getSessionList ()`. This function fills a list HUD control with any game sessions which have more than 1 player in them.

From there, the user can either create a new game session or choose to join one of the existing ones. If they choose to join one of the existing ones, we change state to `SessionConnection`, which connects to the session. When the session is connected, a call is made to `this.ConnectedToSession ()`. `ConnectedToSession` check to see how many users are in the current session and tries to find one to ask about the server password. If it can find a user to ask, it sends out a call to that user, using `ServerAI.onVerifySessionPassword`. This user should be the server user, so they can confirm the password is correct or not. The result is passed back to the requesting user through `Network_Connection.onVerifySessionPassword`. If the password did match, the function `this.SessionVerified ()` is called, otherwise a `Reconnect` state is changed, which disconnects the user from the server and reconnects to generate another game list.

`SessionVerified` brings up the onscreen menu hud and calls `StartPlaying`, which is the main entry point to the actual scene loading and initial player creation. It calls `RPGMain.onStartGame`, which does quite a bit, but essentially fires off the rest of the game join process and creates the local player, initializes chat, etc.

At any time while playing, the player can reach the game lobby screen and choose to `Disconnect`, and then join or host another game.

When you decide to host a game, you are automatically made the server user, and can respond to join requests and check the join password yourself.

Game Server

One of the coolest things about this framework is that it allows any connected client to take responsibility to be the game server for the scene they're in. ServerAI is a new AI we added to the mix. It doesn't have very many functions just yet, but serves an extremely important function. Here's how it works.

When a new game is created, the user who created the game automatically becomes the Server user. This is always done with a call to `ServerAI.onSetAsServer`. This function is also used to take AWAY server status from a user, which can be useful if needing to force a new server to come in and take control. As soon as a user is designated as the server user, they make a call to `ServerAI.informServerObjects`. This tells any server-based objects in the scene that they are now being controlled by the local user, and to begin running their AI routines if they have them.

From then on, at an interval specified in code, the ServerAI calls `onProcessServerObjects`. This tells every server object to run a cycle of server logic, using a call to:

`object.sendEvent (hObject,sAIModel,"onUpdateAI")`

which can be anything from finding new targets to selecting new waypoints for navigation. It is recommended that you use this placeholder to perform as much as possible, and migrate things out of frame-dependent operations like `onEnterFrame` handlers. This will help your game run smoother in general. By choosing a good interval for processing server AI, you can balance the load quite well.

When any new user enters the scene, a handler in ServerAI sends out all the server objects under its control. This does two major things. One: it creates all the correct objects for joining players. Two: it tells the new player which specific objects are server objects. This is critical to maintain!

The ServerAI maintains a list of server objects, even if the current player is not the server user. Every time a new object is created, if it came from the server, the new player tells its own ServerAI to add the object to its list. This is done so that when a new server needs to be chosen, ALL the players have a current list of server objects and can therefore take full responsibility and begin processing those objects the very moment they are chosen as a new server user.

A new server user needs to be chosen when the old server user disconnects for any reason. This is detected in `Network_Connection.onUserLeaveScene`. If the user leaving was also the same user we knew to be the server user, we immediately call `Network_Connection.getNewServerUser`. This function simply goes through the connected user list for the current scene, and picks the user with the lowest userID. Every connected player will have the same list, and therefore will have picked the same userID to be the new server user. The result is sent to `Network_Connection.onSetServerUserID`. If the user being sent to that handler just happens to be the local player, we also send out an event to:

`user.sendEvent (application.getCurrentUser (),"ServerAI","onSetAsServer",true)`

The ServerAI knows it is now the server user, already has a full list of server objects to process, and takes over without missing a beat. That's it in a nutshell!

Platforms

Not necessarily a critical part of the multiplayer platform, networked platforms were added in as a way to test server-controlled objects. We added in a new entry to Entities.xml.

The interesting new functions we added to AIEntity were:

- getRotation
- setRotation
- getRotationSpeed
- setRotationSpeed
- getVelocity
- setVelocity
- onSetRotation
- onSetRotationSpeed
- onSetVelocity
- onSetIsAI
- setMyParent
- setupClass

For server objects, we really wanted to avoid using dynamics for movement, so are relying on linear and calculated positioning using tables for velocity, rotation, and rotation speed. Shiva doesn't have a proper vector variable type, or a 3D point type, which is why I decided to use tables. If any of these values are set, they are read and processed in AIEntity.updateAIMove. This function only occurs for AI, and it is a frame dependent function called in onEnterFrame. If there is a velocity, for example, a new position is calculated based on that velocity, and similar for rotation and rotation speed. Rotation is to set a target rotation, while rotation speed is a constant speed of rotation. Now, adding NetObjectAI to the object does the job of making it networked. More on that later.

To act as a platform which moves, we needed to simply tell the current player to set the object as its parent. This is done as a check upon landing from a fall or jump in the AIEntity.updateDynamicics function. If the object we landed on has an AI attached to it (and is therefore capable of self-directed movement), we set it as our parent. From then on we inherit its velocity. With some extra coding (also in updateDynamics in the if(this.hOnObject ()) block near the top), we determine the rotation to inherit also, while maintaining our own individual rotation and movement ability. We need to also track the velocity of the parent object, so that it can be subtracted from our own total velocity in order for the correct move animation to play.

When we jump, our parent is set to nil. That's it!

There is one problem with the parenting over the network, in that the remote player's position is out of synch. We should be using the velocity of the parent object to help predict where the player should be standing, but are not yet. The platform's view on remote clients is

already 'behind' due to internet travel time, so we look like we are standing in the past. It should be relatively simple to modify this, as we can easily send more information over the network about our parented object and we know its velocity.

The TargettingAI was added to provide a view in the main HUD to our current target, and some simple buttons were added to it. When we click on an object, we look at its class (defined in Entities.xml and maintained in AIEntity.sClass) and decide to show a special set of buttons if it's a 'Vehicle' type. If so, we show the test buttons which let us modify the velocity and rotation speed of the platform. Have fun!

Targetting

The Targetting AI has been added as a new User AI. In InputCore2, we added a new handler called onTouchedObject, which is called whenever the RPGJoypad.onTouchSequenceEnd determined that an object was touched. InputCore2 then tells the Targetting AI Targetting.onSetTarget, which stores the handle of the target and calls Targetting.changeTarget.

This is when we show some information about the target, and can check the class of the target to see if we need to show some special HUD. It is a placeholder example, feel free to modify. FYI, there is a commented-out line in AIEntity.updateAIMove which was sending velocity and rotation speed information back to the targetting AI. It is:

```
--user.sendEvent ( application.getCurrentUser ( ), "Targetting", "onSetVehicleInfo", vX, vY, vZ, rvY )
```

It's only appropriate if the current user is targetting the object, so is commented out. Feel free to experiment!

AI Targetting

AIEntity.makeTargetLocs (player objects do this when created)

AIEntity.updateTargetLocations (updated by player objects on a scheduled post interval)

AIEntity.getFreeTargetLocation (run by objects needing to target the player, recursive)

A nice little subsection to targetting is the new targetting functionality added to AI Objects which a server controls. Instead of running right at the target, Server based objects can find the nearest target with AIEntity.getNearestTarget. This gives them a target player to attack.

To add to the complexity, I created a target node system. When a player object is created, he makes a call to AIEntity.makeTargetLocs. This takes a radius and a desired number of target nodes, and creates an orbit of helper nodes around the player, aligned to the global rotation axis at all times.

When an object wants to target the player, he tries to find the closest target node which does not already contain another targetting object (ie another attacking monster) and is not obscured by collision. If no node is available because maybe the player is swarmed/surrounded, the monster will just use the nearest target node and wait for his turn to beat on the player. When the monster has a target node, he looks at it and changes his velocity to get to it.

On the player side, since he is being targetted by creatures, needs to update this set of nodes based on an interval, and checks to see if any are obscured by a collision, or are filled by a deleted object. A node could be filled with a deleted object if the player killed off a creature attacking him.

We are using TAGS instead of object handles for the targetting, as we can always check to

see if there is an object with a specific tag in the scene, but trying to use a handle for a deleted object will get you nothing but errors.

Net Objects

I took the time to separate the networking functions for AIEntity out into a new AI, called NetObjectAI. This makes the AIEntity class much simpler. Basically, anything that is created with NetObjectAI will also be networked, according to the parameters set in Entities.xml. Work still needs to be done to allow object-specific network commands to be sent from the AIEntity. I recommend using specific event types (numbers) for AIEntity-based networking events, and to send them to NetObjectAI to process. This will allow you the greatest flexibility and allow you to filter out class-specific networking from base object networking.

One of the nicer additions to the networking is the ability to set and modify the packet send and polling frequency for objects. This is handy if you want an object to have a very low update rate, versus one (like a player) which needs high fidelity.