Remote Raiders

Software Design Document

Group Members	Jeremy Bennett, Vincent Engard, Zach Howell, Michael Irizarry, Tomas Rodriguez, Matt Walker
Faculty Advisor	Dr. Filippos Vokolos, Ph. D.
Project Stakeholder	Dr. Michael Wagner, Ph. D.

Revision History

Name	Date	Reason for Change	Revision
Jeremy Bennett, Vincent Engard, Zach Howell, Michael Irizarry, Tomas Rodriguez, Matt Walker	1/19/2016	First Draft - Sections outlined	0.9
Jeremy Bennett, Vincent Engard, Zach Howell, Michael Irizarry, Tomas Rodriguez, Matt Walker	1/30/2016	Initial Version - More UMLs and Class Descriptions added	1.0

Table of Contents

1. Introduction

1.1 Purpose of Document

This document is to describe the implementation of Remote Raiders software as described in the Remote Raiders Requirements document. Remote Raiders is designed to be a party game played in a room with a group of friends.

1.2 Scope of Document

This document describes the implementation details of the Remote Raiders Software. The software will consist of several systems that are mainly split between different namespaces. The two major namespaces are client and server. Code in the client namespace is intended to be run on client machines (smartphones). Code in the server namespace is intended to be run on the server machine (computer). This document will not specify the testing of the software.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Game State

Main Game - Refers to part of game where players can fully move around the map, collect items, and try to complete their objects.

Round - This refers to the amount of time players are in a single instance of the main game. At the end of a round the winner(s) are determined and the players may play a new main game instance.

Scenes - A scene is a screen displayed on either the client interface or server interface.

1.3.2 Player

Avatar - Virtual representation of the player as displayed on the server.

Inventory - This is where players can view and use the items they have collected. The player interacts with the inventory on the client.

Objectives - Goals for players to accomplish within a round. Depending on the objective, it may be hidden (viewable only by the player on their controller) or global (all players see the objective on their controllers).

Controller - Physical object used by player to control the avatar. The controller will act as the client and display a GUI that features the inventory, objectives, and information about the player avatar.

1.3.3 Game Environment

Map - Arena that is displayed on the server. Maps are composed of multiple rooms and hallways that avatars can move freely through. Certain rooms have dedicated functions like spawning items or spawning avatars.

Items - Objects that can be collected by an avatar and used by the player. Items initially are spawned on the server as item boxes so the item identity is hidden to all players. Upon contact with an item box the avatar will receive an item.

Switches - Object that avatars can can be activated or deactivated through avatar contact. Switches have two states: active and inactive. Upon contact a switch will change states.

Traps - Environmental hazards that avatars can trigger by activating switches.

2. System Overview

2.1 Description of Software

Remote Raiders is designed to be a party game with up to 10 players. Each player will have a set of objectives to complete within a round. Players will be able to control their avatars in the game by using their phone as a controller. Players can interact with other players through items, traps, and their avatars.

2.2 Technologies Used

Remote Raiders will use smartphones as input devices. Smartphones with Android will be supported. Smartphones will communicate with Remote Raiders over a local network (phones and server are both on the same network). Multiple smartphones and a computer are required to run Remote Raiders.

Target platform for the computer is Microsoft Windows, for smartphones the target platform is Android. The development environment is Unity 3D game Engine 5.0+ and Microsoft Visual Studio 2015. Version Control is handled through Perforce.

3. System Architecture

3.1 Architectural Design Components

Networking - This system allows for the client and server machines to communicate with one another. The networking system is composed of two main components Network Manager and Network Messenger.

Network Messenger - Network Messenger is responsible for making server command and Client RPC calls specific to our software. Server command calls can only be sent from the client to the server. Client RPC calls can only be sent from server to client.

Network Manager - Network Manager handles connection and disconnection of clients to server.

Player State - This system keeps track of the various “states” a player’s avatar can be in.

Items & Traps - Items on the server can interact with and affect avatars. Items on the client can be used by the player to use an item on the server. Traps exist only on the server and can only be interacted with through avatar contact.

Inventory - This system has two components that exists on both server and client. For each player, there is a corresponding server inventory and client inventory. The client inventory can be interacted with directly by the player via the client interface. The server inventory allows for avatars to use and collect items on server. The server and client inventory for any given player are always in sync.

Objectives - The objectives system is responsible for assigning the player initial objectives, keeping track of objectives completed by the player, and assigning the player a new objective upon completion of an objective.

Client - The client system is largely composed of the client interface that the player interacts with to use items in their inventory, view objectives, move their avatar, attack with their avatar, and monitor their avatar’s health. While the client is tied to one player, the thin client merely sends and receives messages rather than managing state.

Server - The server displays the map, avatars, items, traps, and menus used to navigate through the different scenes of the software. Items and traps only affect objects on the server. Objectives are completed by having avatars perform specific actions on the server.

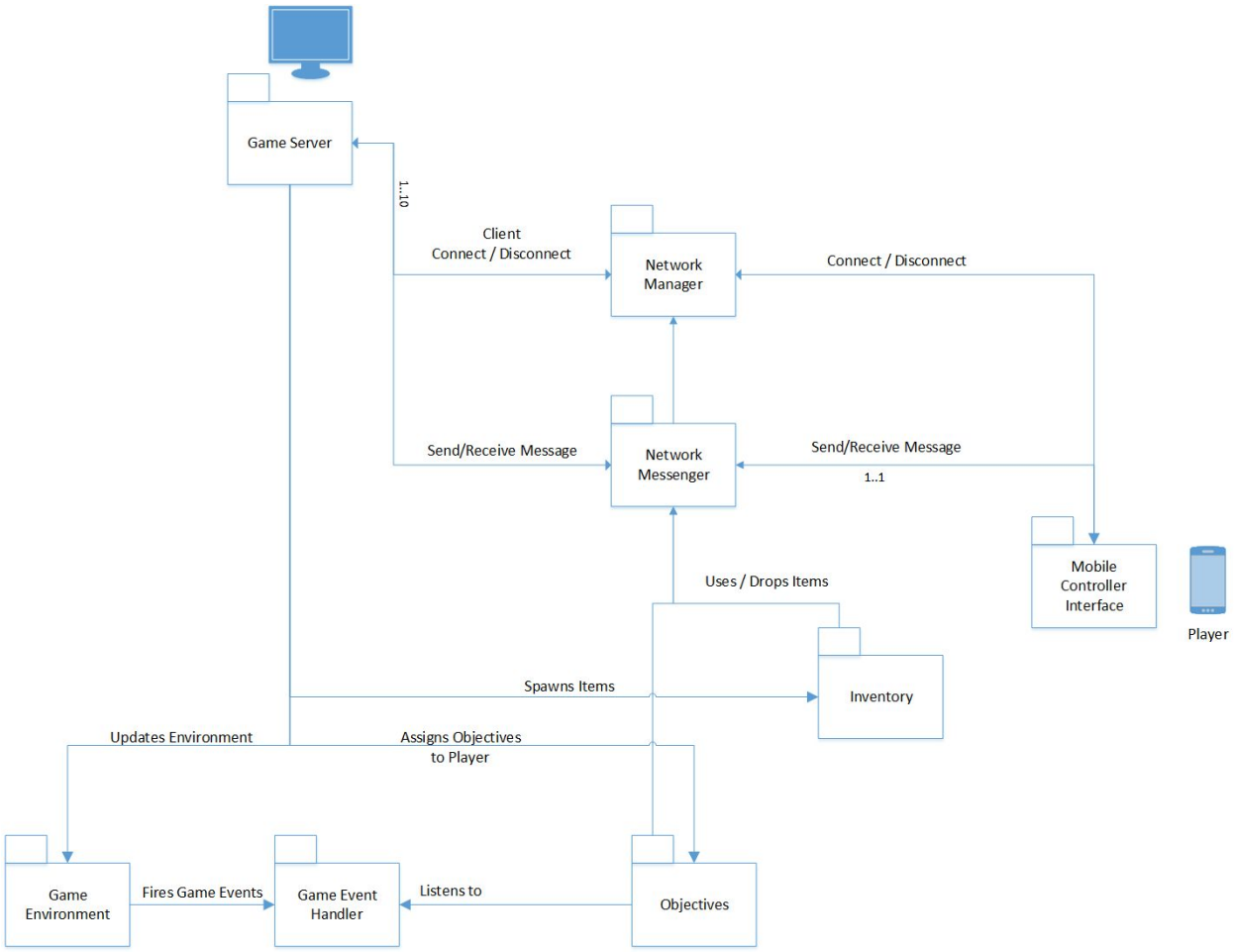

Figure 3.1

3.2 Design Rationale

Why Wifi only?

The customer made it clear that the main use-case for the game will be with people in the same room. As the computer and phones would at minimum need to be able to connect to the internet to communicate, having computers and phones on the same Wifi should be an easy assumption. Allowing the host computer to bypass firewalls and broadcast on the internet creates several additional networking and securities problems as well.

Why download a client vs Internet Browser?

Users have to download a client app to their phone. An interesting alternative is for users to merely access a webpage and play the game in their web browser. Preliminary research (making a demo game using HappyFunTimes) showed that, while such an approach was possible, the support was significantly worse than networking within Unity.

Using an app also gives access to native functionality and allows the player to access the game while offline. Storing local content over multiple sessions could allow users to view their previous records, and phone peripherals like vibration add polish.

Why Game Event handlers?

Game Event handlers allow relevant components to subscribe to specific changes in the global game state. This allows us to more efficiently implement global objectives and other requirements that rely on global state. This paradigm is also very easily extensible in terms of allowing different kinds of items and objectives without combinatorial component growth.

4. Component Design

4.1 Overview

In this section, more details on each component are given. For each component, UML and a brief description are given.

4.2 Network Messenger

The Network Messenger is responsible for sending information specific to our game over the network. Most of the methods found in Network Messenger are either “Command” or “ClientRpc” methods. A “Command” indicates a server command where the code contained in the method is run on the server. A “ClientRpc” indicates a client call where the code contained in the method is run on the client. All methods starting with “Cmd” are commands and any method starting with “Rpc” is a ClientRpc.

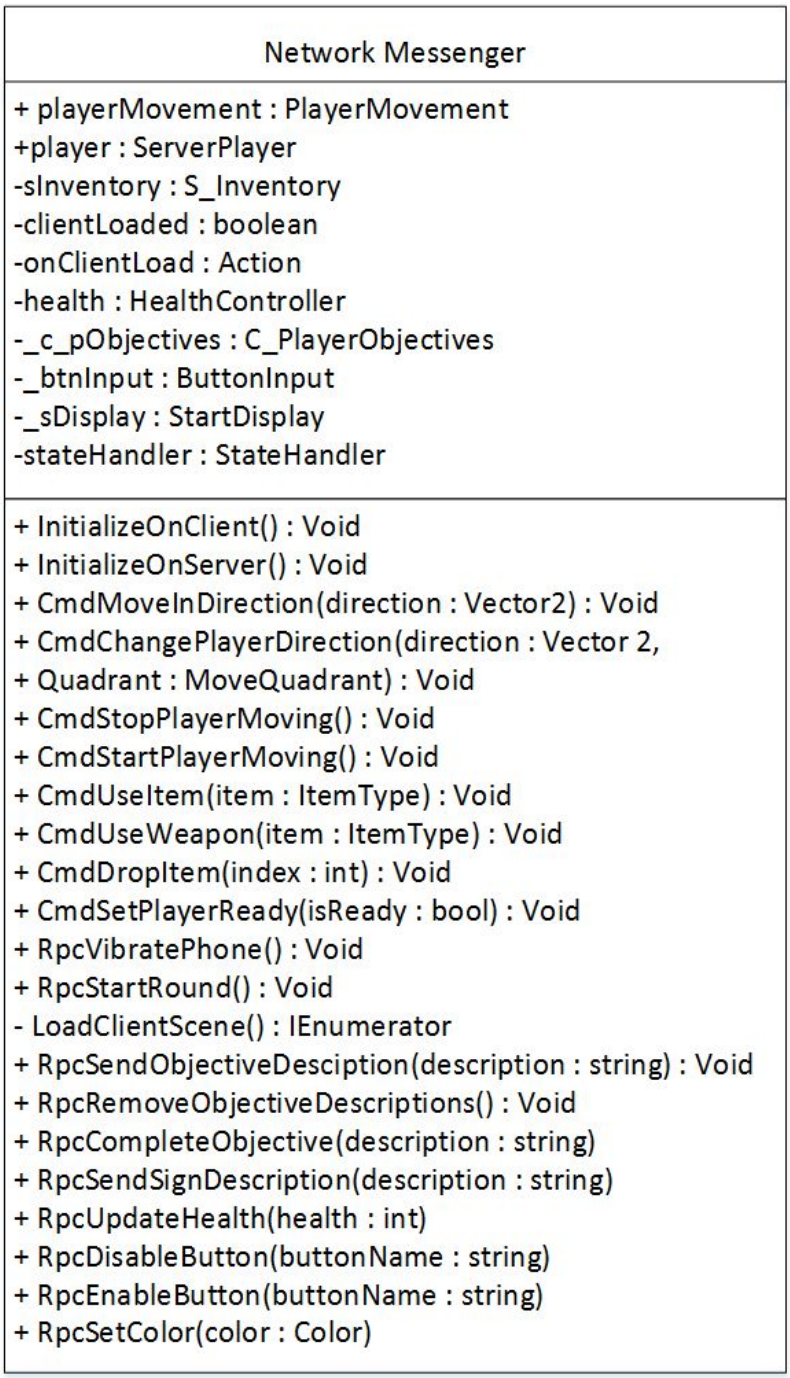
Network Messenger
+ playerMovement : PlayerMovement
+player : ServerPlayer
-sInventory : S_Inventory
-clientLoaded : boolean
-onClientLoad : Action
-health : HealthController
-_c_pObjectives : C_PlayerObjectives
-_btnInput : ButtonInput
-_sDisplay : StartDisplay
-stateHandler : StateHandler
+ InitializeOnClient() : Void
+ InitializeOnServer() : Void
+ CmdMoveInDirection(direction : Vector2) : Void
+ CmdChangePlayerDirection(direction : Vector 2,
+ Quadrant : MoveQuadrant) : Void
+ CmdStopPlayerMoving() : Void
+ CmdStartPlayerMoving() : Void
+ CmdUseItem(item : ItemType) : Void
+ CmdUseWeapon(item : ItemType) : Void
+ CmdDropItem(index : int) : Void
+ CmdSetPlayerReady(isReady : bool) : Void
+ RpcVibratePhone() : Void
+ RpcStartRound() : Void
- LoadClientScene() : IEnumerator
+ RpcSendObjectiveDesciption(description : string) : Void
+ RpcRemoveObjectiveDescriptions() : Void
+ RpcCompleteObjective(description : string)
+ RpcSendSignDescription(description : string)
+ RpcUpdateHealth(health : int)
+ RpcDisableButton(buttonName : string)
+ RpcEnableButton(buttonName : string)
+ RpcSetColor(color : Color)

Figure 4.1

4.2.1 Attributes

Name	Type	Description
playerMovement	PlayerMovement	Used to access methods in PlayerMovement script
player	ServerPlayer	Used to access methods in ServerPlayer script
sInventory	S_Inventory	Used to access methods in S_Inventory script
clientLoaded	Boolean	Boolean indicating whether client has loaded or not
onClientLoad	Action	Set of delegates related to client
_health	HealthController	Allows modification of avatar health
_c_pObjectives	C_PlayerObjectives	Allows for modification of objective list
_btnInput	ButtonInput	Accesses buttons on UI
_sDisplay	StartDisplay	Allows modification of UI display

4.2.2 Methods

Void InitializeOnClient()	
Input:	Void
Output:	Void
Description:	Method sets clientLoaded to true and resets onClientLoad to an empty set

Void InitializeOnServer()	

Input:	Void
Output:	Void
Description:	Upon server startup initializes all non-client attributes and starts client UI if a main game instance is underway

Void CmdMoveInDirection(Vector2 direction, moveQuadrant quadrant)	
Input:	A two dimensional vector representing the direction an avatar will move, a moveQuadrant indicating which of the eight quadrants the avatar is facing
Output:	Void
Description:	Moves an avatar in a given direction and uses the quadrant information to change avatar animations

Void CmdChangePlayerDirection(Vector2 direction, moveQuadrant quadrant)	
Input:	A two dimensional vector representing a new direction an avatar will move, a moveQuadrant indicating which of the eight quadrants the avatar is now facing
Output:	Void
Description:	Player changes the the direction of their avatar's movement and the quadrant the avatar is facing

Void CmdStopPlayerMoving()	
Input:	Void
Output:	Void
Description:	Stops the avatar from moving

Void CmdStartPlayerMoving()	
Input:	Void
Output:	Void
Description:	Starts the avatar moving in the last direction they moved

Void CmdUseItem(ItemType item)	
Input:	The type of item the player is using
Output:	Void
Description:	Uses the item in the avatar's inventory

Void CmdUseWeapon(ItemType item)	
Input:	The type of weapon the player is using (specific ItemTypes are weapons)
Output:	Void
Description:	Uses the avatar's weapon

Void CmdDropItem(int index)	
Input:	The index of the item in the inventory to drop
Output:	Void
Description:	Removes the given item from an avatar's inventory and drops it on the map

Void CmdSetPlayerReady(bool isReady)	
Input:	Boolean determining status of player
Output:	Void
Description:	Changes the client's scene based on if they are ready

Void RpcVibratePhone()	
Input:	Void
Output:	Void
Description:	Vibrates a player's phone

Void RpcStartRound()	
Input:	Void
Output:	Void
Description:	Loads the client scene on client

IEnumerator LoadClientScene()	
Input:	Void
Output:	Method waits for client scene to load before resuming operation
Description:	Loads client scene on client

Void RpcSendObjectiveDescription(string description)	
Input:	String to display as objective description
Output:	void
Description:	Sends objective description to client

Void RpcRemoveObjectiveDescriptions()	
Input:	Void
Output:	Void
Description:	Removes objectives from client

Void RpcCompleteObjective(string description)	

Input:	String representing objective completed
Output:	Void
Description:	Removes completed objective from objective list

Void RpcSendSignDescription(string description)	
Input:	String representing text on sign
Output:	Void
Description:	Sends text from sign to the client

Void RpcUpdateHealth(int health)	
Input:	Value of avatar health
Output:	Void
Description:	Client UI is updated to reflect avatar health

Void RpcDisableButton(string buttonName)	
Input:	The name of the button to disable
Output:	Void
Description:	Disable the target button on the client UI

Void RpcEnableButton(string buttonName)	
Input:	The name of the button to enable
Output:	Void
Description:	Enables the target button on the client UI

Void RpcSetColor(Color color)	
Input:	The desired color to apply to the avatar

Output:	Void
Description:	Sets the avatar's color to the given color

4.3 Objective Architecture Design

The Objective component is responsible for creating and assigning objectives to all the players. S_PlayerObjectives manages all assigned objectives on the server. Each assigned Objective subscribes to GameEvents to see if it has been completed. S_ObjectiveList aggregates a list of all possible objectives.

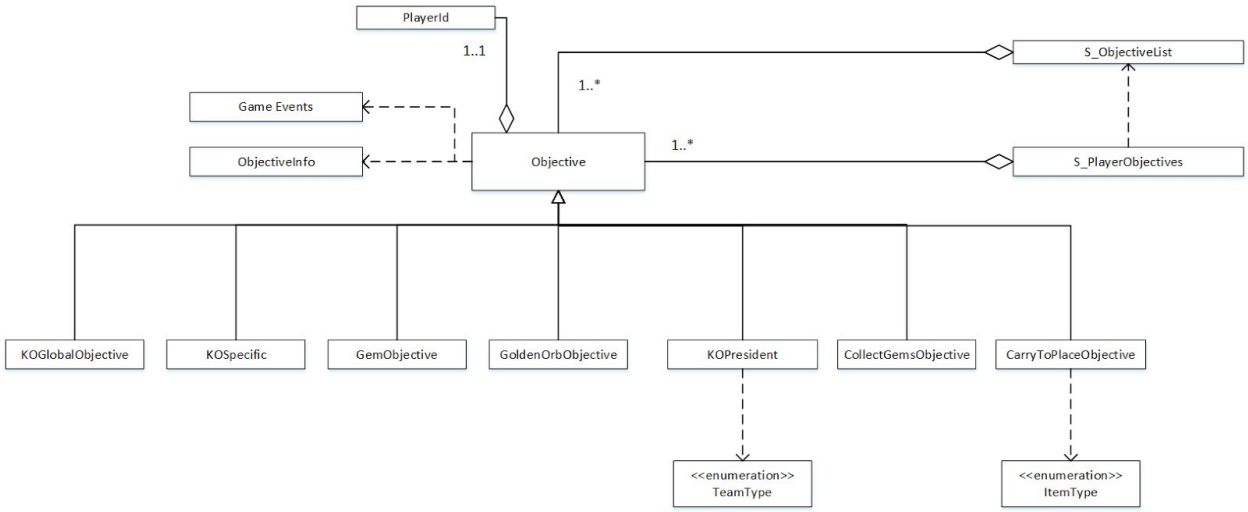

Figure 4.2

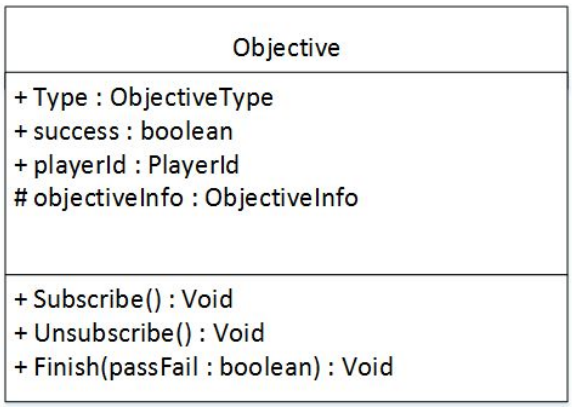

Figure 4.3

4.3.1 Objective Attributes

Name	Type	Description
Type	ObjectiveType	An enum of the current Objective.
success	boolean	Whether objective has been completed.
playerId	PlayerId	The Player who corresponds to this objective. Each objective object is for a single player.
objective	ObjectiveInfo	Description of the objective

4.3.2 Objective Methods

Void Subscribe()	
Input:	Void
Output:	Void
Description:	The Objective subscribes to a particular GameEvent and passes it a handler to see if the objective has been completed

Void Unsubscribe()	
Input:	Void
Output:	Void
Description:	Unsubscribes to the current GameEvent it's listening to

Void Finish(bool passFail)	
Input:	Boolean detailing pass or fail of objective
Output:	Void

Description	Sets success attribute to passFail value

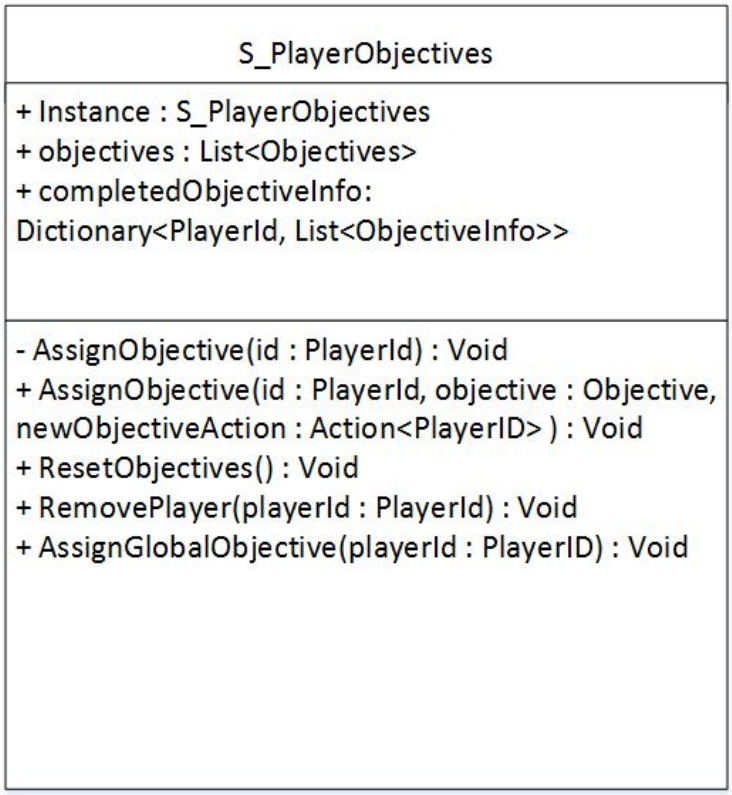

Figure 4.4

4.3.3 S_PlayerObjectives Attributes

Name	Type	Description
Instance	S_PlayerObjectives	Singleton to access class
Objectives	List	List of all active objectives assigned to players
completedObjectiveInfo	Dictionary	Map of all completed objectives to players

4.3.2 S_PlayerObjectives Methods

Void AssignObjective(PlayerId id)	

Input:	Id of the player
Output:	Returns the objective assigned to the player
Description:	Randomly assigns a non-global objective to the player.

Void ResetObjectives()	
Input:	Void
Output:	Void
Description:	Clears the objective and completed objective lists and sends messages to remove the objectives from the client.

Void RemovePlayer(PlayerId playerId)	
Input:	Id of the player.
Output:	Void
Description:	Clears the objective and completed objectives for a certain player.

Void AssignGlobalObjective(PlayerID playerID)	
Input:	Id of the player.
Output:	Void
Description:	Creates an objective (the global objective) for the player.

4.3.5 GameEvents

Figure 4.5

Name	Type	Description
Instance	GameEvents	The public singleton getter for GameEvents.

_instance	GameEvents	The private singleton instance of GameEvents.
pickupEvent	delegate	The function that is called when a pickupEvent is fired.
dropEvent	delegate	The function that is called when a dropEvent is fired.
playerKOEvent	delegate	The function that is called when a playerKOEvent is fired.
onPlayerDeathEvent	event	The event that is fired when a player dies.
onItemPickupEvent	event	The event that is fired when a player picks up an item.
onDropEvents	event	The event that is fired when a player drops an item.
onAltarDropEvent	event	The event that is fired when a player drops an item on the altar.
onRoundEndEvents	event	The event that is fired when the round ends.

Each of the above events have a corresponding Subscribe, Unsubscribe, and OnEvent methods. Subscribe adds an anonymous function listener to the event (often called by Objectives), while Unsubscribe removes a subscribed listener. OnEvent is called by a variety of scripts, in the location where that event happens. For example, an KOSpecificObjective calls SubscribeToPlayerDeath event on creation, waits for any PlayerHealth script to call OnPlayerDeath upon a avatar's health reaching 0, and then calls UnsubscriceToPlayerDeath when it's complete.

4.4 Items & Inventory Architecture Design

The Items & Inventory section manages items on the server and in the client's inventory. The ItemType enum matches to separate server items and client item lists (ItemSpawner and ClientItemList, respectively). ItemSpawner contains all of the SpawnableItems usable or droppable on the server. Many of these SpawnableItems have an Effect on avatars who collide with the item. These Effects are shared with the Trap component. One Inventory and one

S_Inventory exist for each player, with Inventory managing Inventory across server and client using the aforementioned ItemType enum, while S_Inventory pulls from ItemSpawner to create SpawnableItems.

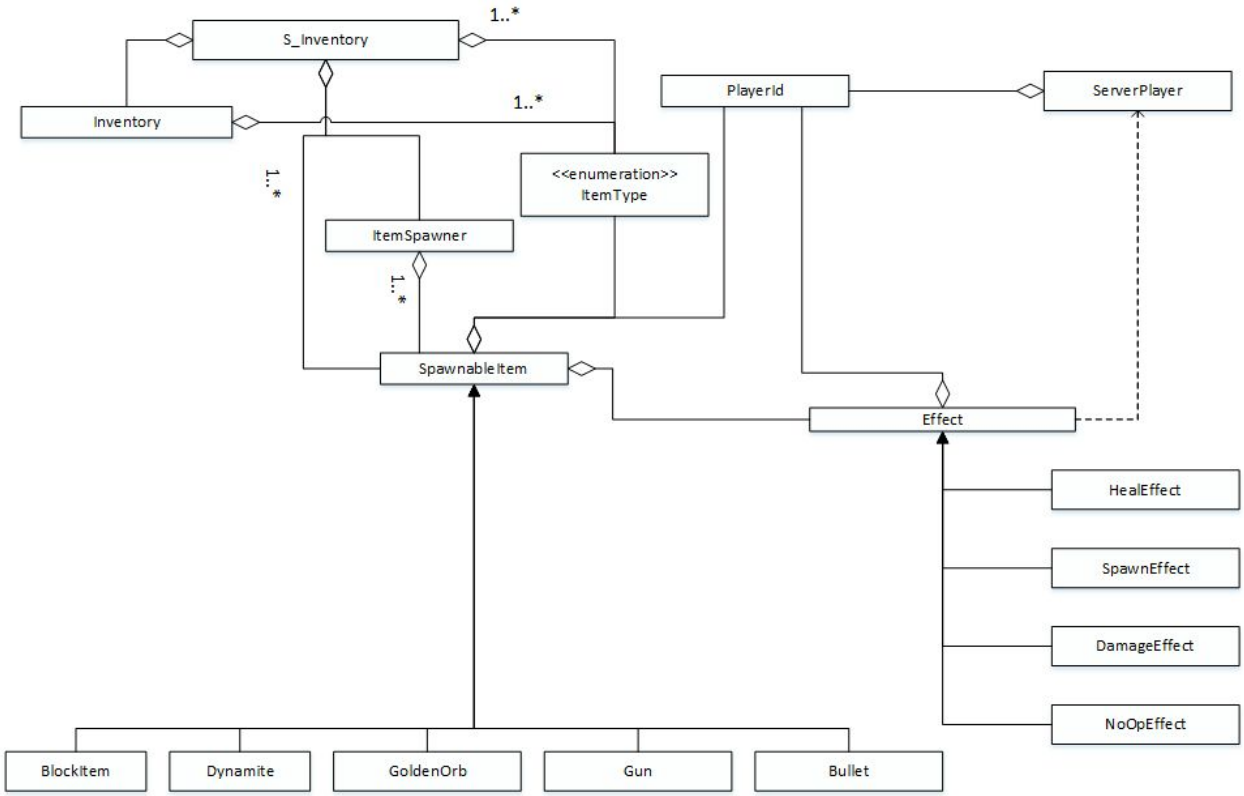

Figure 4.6

**Unless otherwise noted, the multiplicity for an aggregation is one-to-one.*

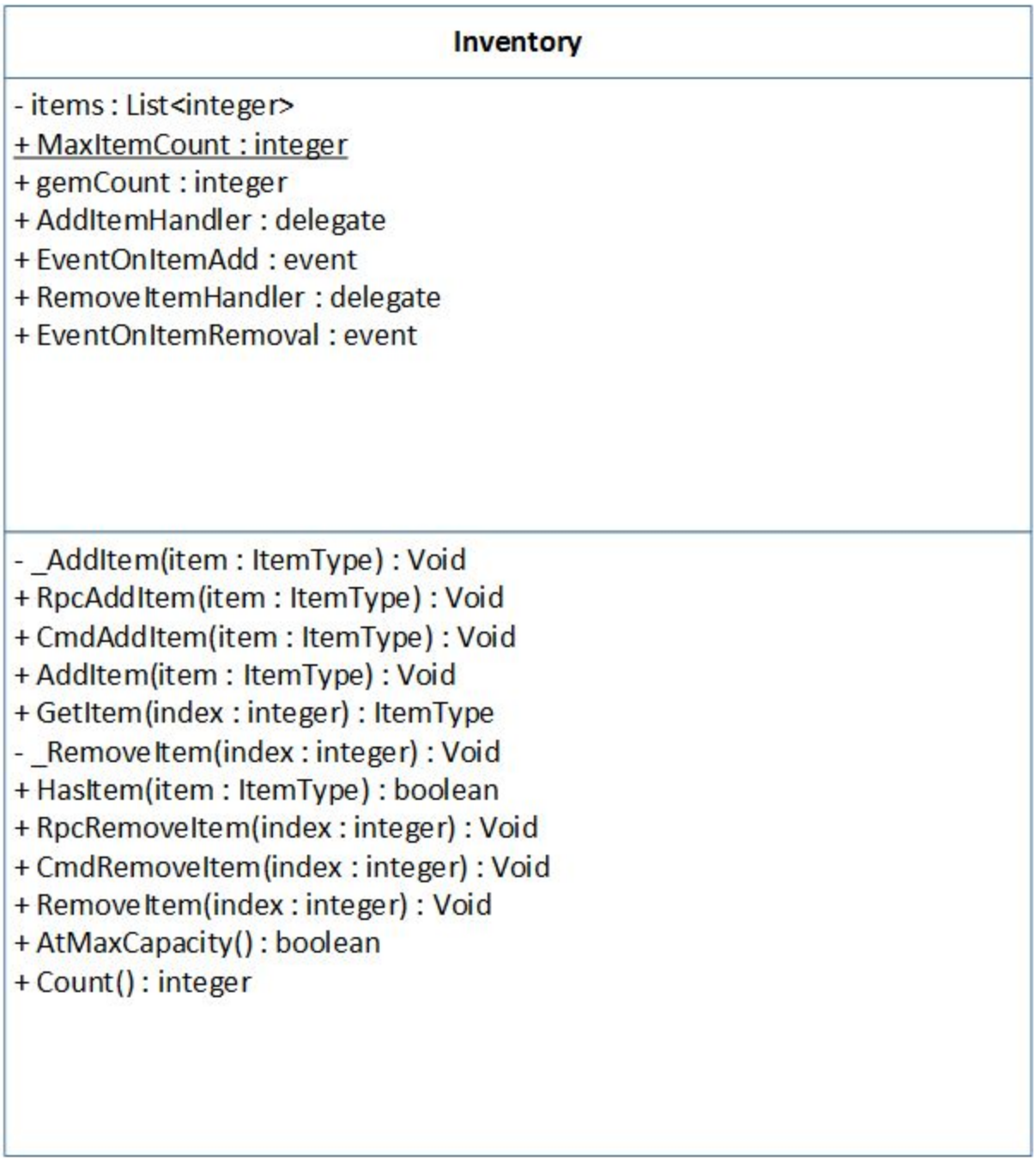

Figure 4.7

4.4.1 Inventory Attributes

Name	Type	Description
items	List<integer>	Internal representation of the inventory.

MaxItemCount	integer	The maximum number of items a player can have in their inventory
gemCount	integer	The amount of gems the player has in their inventory
AddItemHandler	delegate	A function that is called when an item is added to the inventory.
EventOnItemAdd	event	The event that is fired when an item is added to the inventory
RemoveItemHandler	delegate	A function that is called when an item is removed from the inventory
EventOnItemRemove	event	The event that is fired when an item is removed from the inventory

4.4.2 Inventory Methods

Void AddItem(ItemType type)	
Input:	The item you are trying to add
Output:	Void
Description:	Adds the item to the server and client inventory. The RpcAddItem and CmdAddItem are used in order to sync the addition of the item across the server and the client.

Void RemoveItem(int Index)	
Input:	The index of the item you are trying to remove.
Output:	Void
Description:	Remove the item to the server and client inventory. The RpcRemoveItem and

	CmdRemoveItem are used in order to sync the removal of the item across the server and the client.

Boolean AtMaxCapacity()	
Input:	Void
Output:	A boolean representing whether the inventory is at the max capacity or not.
Description:	Returns whether or not the inventory is at maximum capacity.

Boolean HasItem(ItemType item)	
Input:	The item that you are checking whether it exists in the inventory or not.
Output:	A boolean representing whether the item exists in the inventory or not.
Description:	Returns whether or not a specific item exists in the Inventory.

integer Count()	
Input:	Void
Output:	The count of items in the inventory.
Description:	Returns the count of the items in the inventory.

4.5 Trap Architecture Design

The trap component of the game is responsible for managing traps, switches and effects caused by traps. TrapComponents are triggered by a switch that has been paired with the TrapComponent. Upon avatar collision with a triggered TrapComponent, an Effect is applied to the avatar.

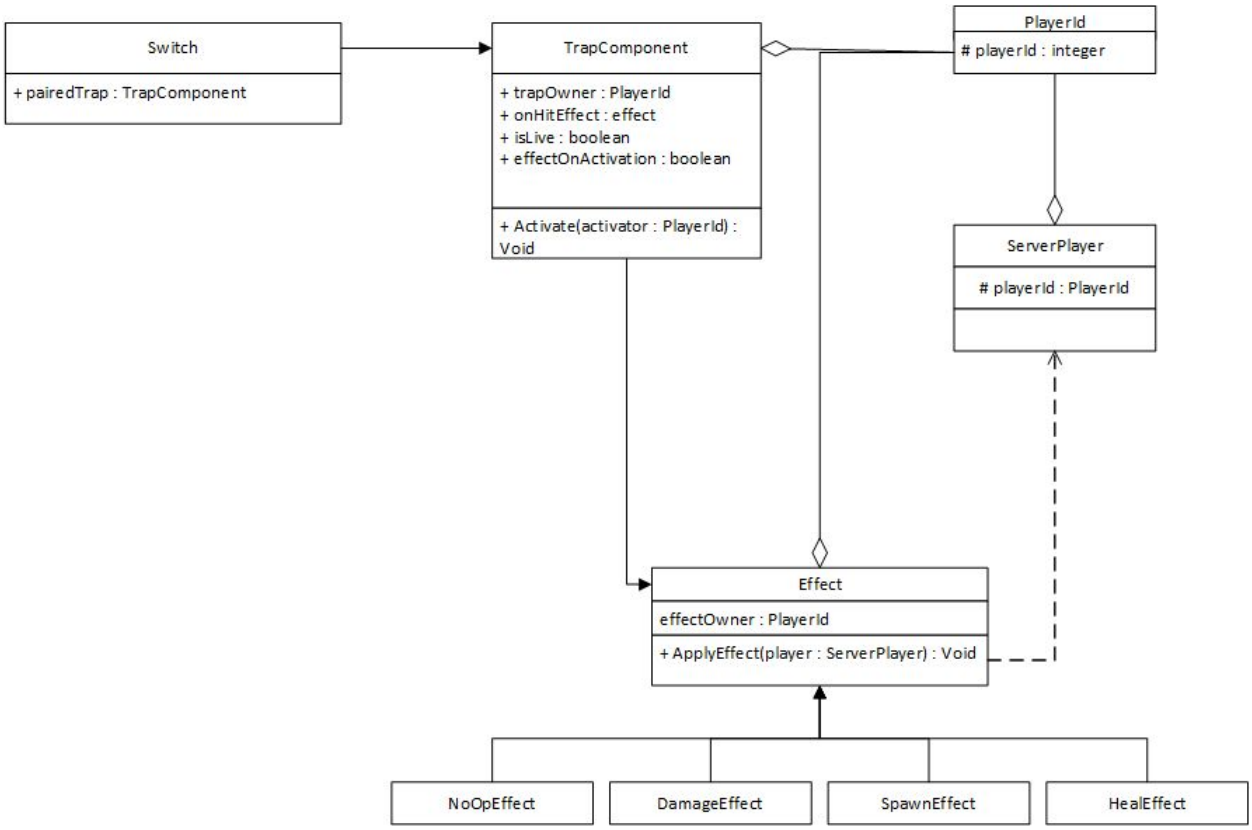

Figure 4.8

4.5.1 TrapComponent Attributes

Name	Type	Description
trapOwner	PlayerId	The player that triggered the trap.
onHitEffect	effect	The effect that is triggered when a player comes in contact with the trap.
isLive	boolean	Whether the trap is currently triggered or not.
effectOnActivation	boolean	Whether or not the effect is used immediately when the trap is triggered.

4.5.2 TrapComponent Methods

Void Activate()	
Input:	Void
Output:	Void
Description:	Make the trap live, which will make the player able to interact with it. If effectOnActivation is true, the effect of the trap is automatically used.

4.6 Player State Component

The avatar's state is controlled by the StateHandler. States of the avatar dictate when certain actions can and cannot be performed, as well as triggering certain GameEvents. All of the avatar states are managed through a finite state machine. States are specified in more detail in the Requirements Specification [1].

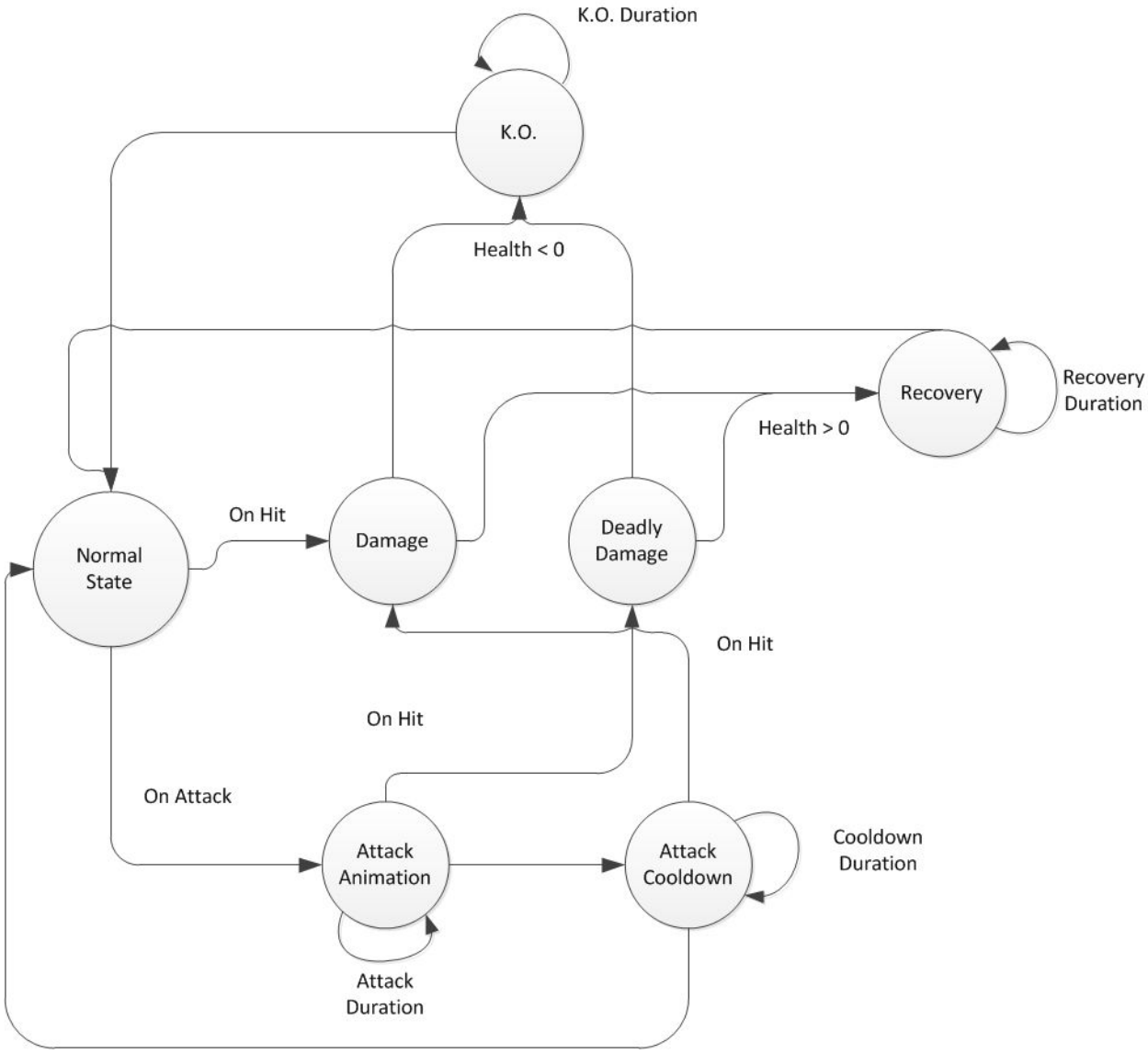

Figure 4.9

5. Human Interface Design

5.1 Overview of User Interface

The screenflow and screenshots for Remote Raiders are specified in more detail in the Requirements Specification [1]. For more implementation details, each screen will approximately exist within one Unity scene. The canvas UI system provided after Unity 4.6 will be used, especially to ensure a responsive design for mobile.

5.2 Screen Objects and Actions

Server
The intended input method for use on the server is a mouse. There are not very many buttons on the server - after the initial menu the whole game should be accessible via the client controllers.
Client
The intended input method for use on the client is a touch screen. Additional care should be made to ensure the menus are responsive for a variety of phone screen sizes.

5.3 Server Menu Flow

Below is a listing of the server's screen flow for convenience. Implementing these menus in Unity involves mostly simple scripts to hide and show objects. See the Requirements Specification for more details [1].

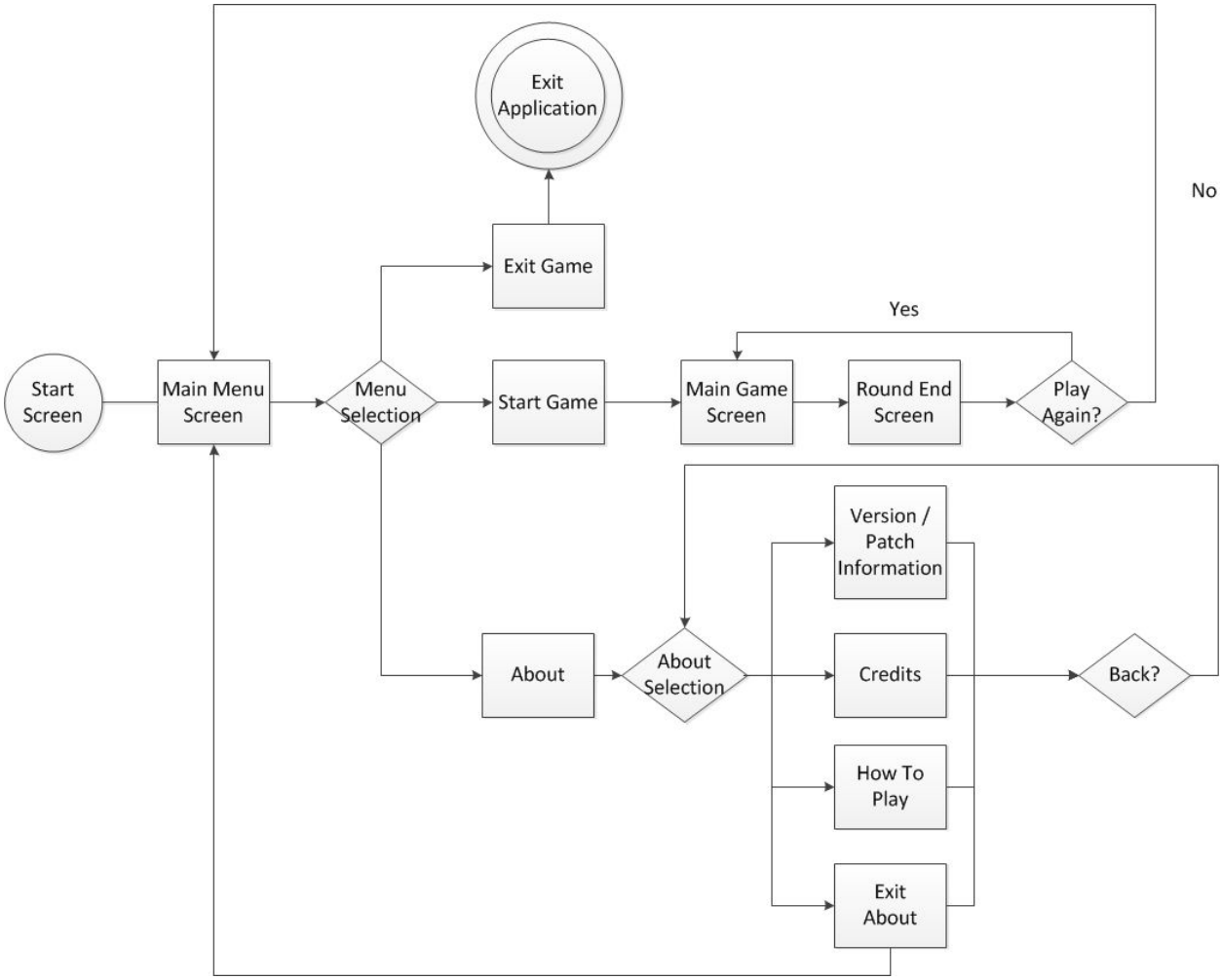

Figure 5.1

5.4 Client Menu Flow

Below is a listing of the client's screen flow for convenience. See the Requirements Specification for more details [1].

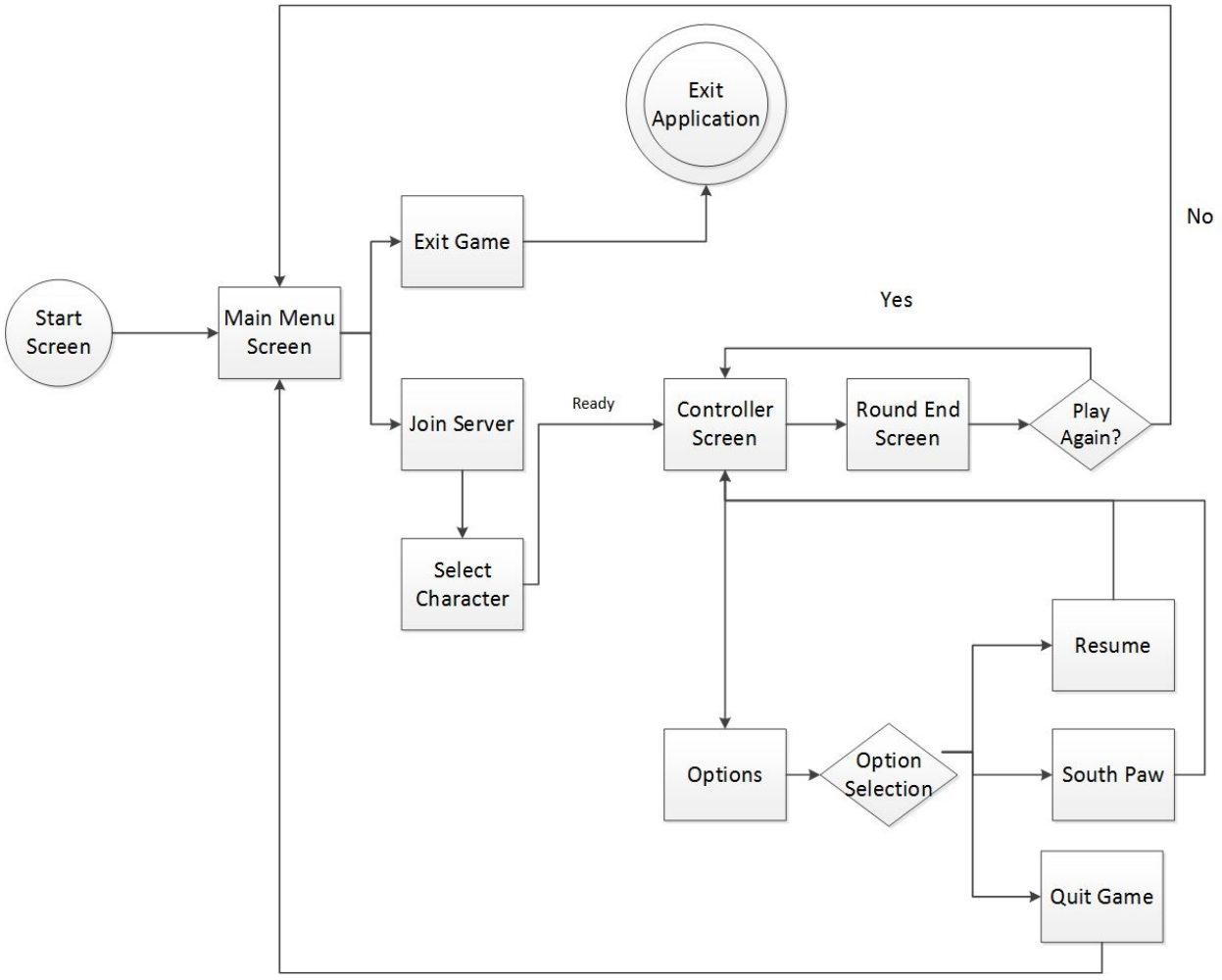

Figure 5.2

6. References

[1] Bennett, et. al. (Remote Raiders Requirements Specifications Document). Fancier Fish Productions. Philadelphia, PA, United States, 2015.

[2] http://docs.unity3d.com/Manual/index.html