Project Zomboid

API for Inventory Items

Document version 1.0

Contents

Introduction

This document is for the modders for Project Zomboid and it has the new system of Inventory Items manipulation described, which was developed to prevent cheating.

All inventory Items processing in multiplayer are transferred to the server side. That means, that while it’s still possible for the modification on the client side to create an Inventory Item and put it to the inventory, such an item won’t be available for interaction and will be deleted from the inventory after relogin.

All items should be created on the server side, and then transferred to the client. Thus an Inventory Item will be in the player’s inventory on both client and server sides.

1. General description

The main way of an Inventory Item creation, deletion and manipulation is to create it in a Timed Action. The Inventory Item creation, deletion and modification should be done on the server side, modifying the player’s inventory on the server side. The player’s inventory on the client side is modified subsequently to be identical to the one on the server side.

An alternative way of an Inventory Item creation, deletion and manipulation is to send a command using “`sendCommand`“ or “`sendClientCommand`“ functions. To do that you have to implement your command processor, that will receive the necessary data, perform cheating checks, create/delete/modify the Inventory Item on the server side, and send the corresponding packets to clients for synchronisation. This way could come in handy while dealing with the Inventory Item manipulation that doesn’t come from the user's actions, e.g. admin powers.

2. Timed Action Architecture

The new implementation of Timed Action allows to execute Timed Action on both the server and the client.

To adapt the existing Timed Action to the new architecture the following should be done:

1) Move the Timed Action file from `media/lua/client` folder to `media/lua/shared` one. This will allow the script to be loaded by the client as well as the server.
2) Make sure there is a variable of the same name with the same value for each `new` function argument.
3) Create the `getDuration` function, which returns the execution time of the Timed Action.
4) Move some code from the ‘perform’ function to the new ‘complete’ function.
    a) ‘perform’ function:
        i) performs the actions that are only needed on the client, e.g. animation and sound management;
        ii) doesn’t contain any manipulations with any items or objects;
        iii) is performed on the client as well as in singleplayer mode;
    b) ‘complete’ function:
        i) contains manipulations with items and objects only;
        ii) is performed on the server as well as in single-player mode;
        iii) is executed after ‘perform’ in single-player mode;
5) Add the call of the function sending changes to clients to the ‘complete’ function (see section 4).

Let’s take a closer look at the required changes.

2.1 Modification of the 'new' function

When running a Timed Action, the game sends it to the server side and restores the Timed Action Lua object. This is necessary for execution of 'getDuration' and 'complete' functions on the server side. The server receives the list of the 'new' function arguments and searches for their values in the variables (fields) of the object.

Let's assume there is a following code:

```
function ISPlaceTrap:new(playerObj, trap, damage, maxTime)
      local o = ISBaseTimedAction.new(self, playerObj);
      o.square = character:getCurrentSquare();
      o.weapon = trap;
      o.damage = damage / 20;
      o.maxTime = maxTime;
      return o;
end
```

Code piece 1 - Example of the incorrect 'new' function

The game won't be able to restore such an object on the server side, because the argument names aren't the same as the names of the variables in the object.

The following changes should be made in order for the server to be able to send a Timed Action to the server:

1) Rename `playerObj` to `character`. There is a realisation of the `ISBaseTimedAction.new` function In *Code piece 2,* and we could see that this function saves the argument into the variable named 'character'.
2) Rename the "`trap`" argument to "`weapon`", because the value of the "`trap`" argument is saved into the "`weapon`" variable.
3) It is necessary to get rid of saving the changed "`damage`" value into a variable with the same name. Otherwise, if we run such a timed action with "`damage`" equal to 1000, then an object will be created with the "`damage`" variable equal to 50. When this object is transferred to the server side, this constructor is called again, and it'll get the value of 50 as an argument of damage. As a result, the server will get an object with

damage value equal to 2. To prevent this from happening, the incoming values should be kept unchanged.

4) The execution time of the Timed Action also shouldn't be transferred as an argument, as it created a vulnerability where a cheater can perform Timed Actions instantly changing the time argument. To avoid this, the game uses the "`getDuration`" function to calculate the Timed Action execution time on the server side.

```
function ISBaseTimedAction:new (character)
      local o = {}
      setmetatable(o, self)
      self.__index = self
      o.character = character;
      o.stopOnWalk = true;
      o.stopOnRun = true;
      o.stopOnAim = true;
      o.caloriesModifier = 1;
      o.maxTime = -1;
      return o
end
```

Code piece 2 - realisation of the ISBaseTimedAction:new function

After following all the suggestions you'll get the code given in *Code piece 3.*

```
function ISPlaceTrap:new(character, weapon)
      local o = ISBaseTimedAction.new(self, character);
      o.square = character:getCurrentSquare();
      o.weapon = weapon;
      o.maxTime = o:getDuration();
      return o;
end
```

Code piece 3 - An example of the correct 'new' function implementation

Currently the game supports the following data types for the arguments of the 'new' function:

1. BaseVehicle
2. BloodBodyPartType
3. BodyPart
4. Boolean
5. CraftRecipe
6. Double
7. EvolvedRecipe

8. FluidContainer
9. Integer
10. InventoryItem
11. IsoAnimal
12. IsoDeadBody
13. IsoGridSquare
14. IsoHutch.NestBox
15. IsoObject
16. IsoPlayer
17. ItemContainer
18. KahluaTableImpl
19. MultiStageBuilding.Stage
20. PZNetKahluaTableImpl
21. Recipe
22. Resource
23. SpriteConfigManager.ObjectInfo
24. String
25. VehiclePart
26. VehicleWindow
27. null

Serialisation realisation is in the `PZNetKahluaTableImpl` class.

It should also be noted that while passing an object, the client sends information to the server to search for the corresponding object on the server side. After that the server searches for the corresponding object and uses it as an argument. For this reason, the game cannot transfer as an argument an object created on the client side.

2.2 Realisation of the “getDuration” function

The “`getDuration`” function is used by the server to fetch the time needed to execute the Timed Action. However, the client can also use this function, using the following code in the ‘new’ function.

```
o.maxTime = o:getDuration();
```

Code piece 4 - the usage of the getDuration function in the ‘new’ function to receive the execution time of a Timed Action

The “`getDuration`” function returns the execution time of the Timed Action in cycles. To convert the time in seconds to the value that the “`getDuration`” function should return, divide the time by 0.02. That is, for a Timed Action that should last 1 second, the “`getDuration`” function should return a value of 50. For an action that is executed instantaneously, it is recommended to return the value ‘1’.
For infinite Timed Actions the “`getDuration`” function should return the value ‘-1’.

Also this function should return the value 1 if the TimedActionInstant cheat is enabled.

Here’s an example usage of this function for Timed Action that lasts 1 second.

```
function ISPlaceTrap:getDuration()
	if self.character:isTimedActionInstant() then
	return 1;
	end
	return 50
end
```

Code piece 5 - Typical realisation of the getDuration function, returning the 1 second time

2.3 Split of the “perform” function into “perform” and “complete”

It is necessary to split the “`perform`” function into 2 functions: “`perform`” and “`complete`”. Both of these functions are executed at the end of Timed Action execution. The client executes only the “`perform`” function, and the server executes only the “`complete`” one. Singleplayer first executes “`perform`” and then “`complete`”.

The “`perform`” function must contain code that is not related to modifying items and objects. This can be code to control sounds, animation, interaction with UI, etc.

The “`complete`” function must contain code for items and objects modification. This function cannot contain code that cannot be called on the server side.

Let’s take a look at such a split, using ISAddFuelAction as an example.

```
function ISAddFuelAction:perform()
	self.character:stopOrTriggerSound(self.sound)

	self.item:setJobDelta(0.0);

	if self.item:IsDrainable() then
	self.item:Use()
	else
	self.character:removeFromHands(self.item)
	self.character:getInventory():Remove(self.item)
	end

	local cf = self.campfire
	local args = { x = cf.x, y = cf.y, z = cf.z, fuelAmt = self.fuelAmt }
	CCampfireSystem.instance:sendCommand(self.character, 'addFuel', args)

	-- needed to remove from queue / start next.
	ISBaseTimedAction.perform(self);
end
```

Code piece 6 - The obsolete realisation of the ‘perform’ function for the ISAddFuelAction timed action.

This function contains the code to control the interface and sound, manipulate objects, and the synchronisation code using the command.

Let’s keep in this function only the code for interface and sound control.

```
function ISAddFuelAction:perform()
      self.character:stopOrTriggerSound(self.sound)

      self.item:setJobDelta(0.0);

      -- needed to remove from queue / start next.
      ISBaseTimedAction.perform(self);
end
```

Code piece 7 - The new realisation of the “perform” function of the ISAddFuelAction timed action

We’ll create a new function called “`complete`” and put the code there to manipulate the objects.
We also have to perform an action that was previously executed by sending the “`addFuel`” command. If we look at the “`addFuel`” command realisation, we’ll find the following code:

```
function SCampfireSystemCommand(command, player, args)
      if command == 'addFuel' then
      local campfire = campfireAt(args.x, args.y, args.z)
      if campfire then
            campfire:addFuel(args.fuelAmt)
      end
…
```

Code piece 8 - Implementation of the “addFuel” command handler

Apparently, the coordinates of the campfire were sent to the server. Then the server was finding the campfire on the map and performed the “`addFuel`” function. As the “`complete`” function is going to be performed at the server side, we don’t have to send the command anymore. We can directly get an object and perform the required function.

So we change the Code piece 9 to the code from the command handler you can see in Code piece 10:

```
CCampfireSystem.instance:sendCommand(self.character, 'addFuel', args)
```

Code piece 9 - Sending of the addFuel command from client to server

```
      local campfire = campfireAt(args.x, args.y, args.z)
```

```
	if campfire then
		campfire:addFuel(args.fuelAmt)
	end
```

Code piece 10 - Search and modifying of the object

We also change the call of the local "`campfireAt`" function to its realisation, which is given above.

```
local function campfireAt(x, y, z)
	return SCampfireSystem.instance:getLuaObjectAt(x, y, z)
end
```

Code piece 11 - Realisation of the campfireAt function

As a result we get the following function:

```
function ISAddFuelAction:complete()
	if self.item:IsDrainable() then
	self.item:UseAndSync()
	else
	self.character:removeFromHands(self.item)
	self.character:getInventory():Remove(self.item)
	sendRemoveItemFromContainer(self.character:getInventory(),self.item)
	end
	local                          campfire                          =
SCampfireSystem.instance:getLuaObjectAt(self.campfire.x,   self.campfire.y,
self.campfire.z)
	if campfire then
	campfire:addFuel(self.fuelAmt)
	end
	return true
end
```

Code piece 12 - The new realisation of the 'complete' function of the ISAddFuelAction timed action

3. Realisation features of the long Timed Actions

To implement Timed Actions that perform actions during execution, such as pouring liquids or reading books, it is necessary to use an additional Anim Event emulation API.

To implement such an action the implementation of the "`serverStart`" function is needed, which is executed on the server side when the Timed Action starts. In this function, call the "`emulateAnimEvent`" function to set up the AnimEvent emulator. After that, you need to write an implementation of the "animEvent" function, which will be called periodically to execute a part of the whole action. To stop such an action, the server-side function `self.netAction:forceComplete()` should be called. You could use the "`self.netAction:getProgress()`" function to get the progress of the action in the animEvent function on the server side.

Let's take a look at the examples. First here's the realisation of the serverStart function:

```
function ISChopTreeAction:serverStart()
      self.axe = self.character:getPrimaryHandItem()
      emulateAnimEvent(self.netAction, 1500, "ChopTree", nil)
end
```

Code piece 13 - Example of realisation of the serverStart function

The "`emulateAnimEvent`" function takes the following arguments:

1) "NetTimedAction" - always equals to `self.netAction`;
2) "duration" - a period in milliseconds;
3) "event" - a name, that will be sent to "`animEvent`" function;
4) "parameter" - a stock parameter for the "`animEvent`" function.

An example of implementation of the "`animEvent`" function is shown in the code piece 14.

```
function ISChopTreeAction:animEvent(event, parameter)
```

```
	if not isClient() then
		if event == 'ChopTree' then
			self.tree:WeaponHit(self.character, self.axe)
			self:useEndurance()
			if self.tree:getObjectIndex() == -1 then
			if isServer() then
					self.netAction:forceComplete()
			else
					self:forceComplete()
			end
			end
		end
	else
		if event == 'ChopTree' then
			self.tree:WeaponHitEffects(self.character, self.axe)
		end
	end
end
```

Code piece 14 - Example of realisation of the "animEvent" function

This function on the client only reproduces tree chopping effects using the "`WeaponHitEffects`" function. But in singleplayer, and on the server, this function calculates the damage the tree takes from axe hit using the `self.tree:WeaponHit(self.character, self.axe)` function.

Finally, when the tree is chopped down - it stops the infinite Timed Action:

```
			if isServer() then
					self.netAction:forceComplete()
			else
					self:forceComplete()
			end
```

Code piece 15 - Realisation of the end of the Timed Action

4. The use of the sendClientCommand

The “`sendClientCommand`” function is executed on the client and sends a command whose handler is on the server. If this function is called in a singleplayer game the handler of this command will also be called by the game.

The “`sendClientCommand`” function has the following arguments:

1) IsoPlayer “player” - optional argument, player’s object;
2) String “module” - the name of the module for which the command is sent;
3) String “command” - the command name;
4) KahluaTable “args” - table with arguments.

```
sendClientCommand(self.player, "vehicle", "getKey", { vehicle =
self.vehicle:getId() })
```

Code piece 16 - Sending the getKey command for the ‘vehicle’ module with an argument vehicle = self.vehicle:getId()

Upon receiving the client command the server calls an OnClientCommand event with the following arguments: module, command, player, args.

```
local VehicleCommands = {}
local Commands = {}

function Commands.getKey(player, args)
	local vehicle = getVehicleById(args.vehicle)
	if vehicle and checkPermissions(player, Capability.UseMechanicsCheat) then
	local item = vehicle:createVehicleKey()
	if item then
	player:getInventory():AddItem(item);
	sendAddItemToContainer(player:getInventory(), item);
	end
	else
	noise('no such vehicle id='..tostring(args.vehicle))
	end
end

VehicleCommands.OnClientCommand = function(module, command, player, args)
	if module == 'vehicle' and Commands[command] then
```

```
        Commands[command](player, args)
        end
end

Events.OnClientCommand.Add(VehicleCommands.OnClientCommand)
```

Code piece 17 - Example of implementation of the getKey command handler on the server side

The `Events.OnClientCommand.Add()` function call sets the `OnClientCommand` event handler. After executing this function, receiving any command will result in calling the function indicated as an argument. In this case the `VehicleCommands.OnClientCommand` function is the handler for the `OnClientCommand` event.

The `VehicleCommands.OnClientCommand` function checks if the module is set equal to "vehicle" and if there is a function with the name corresponding to the command name in the Commands table. Then the function calls the corresponding function by sending it the "player" and "args" arguments.

The `Commands.getKey` function is a handler of the "`getKey`" command. This function works only if the player has the `UseMechanicsCheat` permission. In case if the player has the appropriate permission and the required vehicle is found on the server side then it creates the key InventoryItem on the server side using the "`createVehicleKey`" function, and after that adds it to the player's inventory and sends it back to the client side.

5. Synchronisation of the creation, deletion and modification of items and objects

All object changes made in the ‘complete’ function should be sent to clients. The following functions are needed for that:

1. **sendAddItemToContainer** - adds the InventoryItem to the container;
2. **sendRemoveItemFromContainer** - deletes the InventoryItem from the container;
3. **syncItemFields** - synchronises the following variables: condition, remoteControlID, uses, currentAmmoCount, haveBeenRepaired, taintedWater, wetness, dirtyness, bloodLevel, hungChange, weight, alreadyReadPages, customPages, customName, attachedSlot, attachedSlotType, attachedToModel, fluidContainer, moddata;
4. **syncItemModData** - synchronises the moddata;
5. **syncHandWeaponFields** - synchronises the following variables: currentAmmoCount, roundChambered, containsClip, spentRoundCount, spentRoundChambered, isJammed, maxRange, minRangeRanged, clipSize, reloadTime, recoilDelay, aimingTime, hitChance, minAngle, minDamage, maxDamage, attachments, moddata;
6. **sendItemStats** - synchronises the following variables: uses, usedDelta, isFood, frozen, heat, cookingTime, minutesToCook, minutesToBurn, hungChange, calories, carbohydrates, lipids, proteins, thirstChange, fluReduction, painReduction, endChange, reduceFoodSickness, stressChange, fatigueChange, unhappyChange, boredomChange, poisonPower, poisonDetectionLevel, extraItems, alcoholic, baseHunger, customName, tainted, fluidAmount, isFluidContainer, isCooked, isBurnt, freezingTime, name;
7. **transmitCompleteItemToClients** - add an object to the map;
8. **transmitRemoveItemFromSquare** - delete an object from the map;
9. **sync** - synchronise the object change on the map;
10. **transmitUpdatedSpriteToClients** - synchronise the changed sprite.

Here are some examples of the code for how to create, delete and manipulate items:

```
      local candle = instanceItem("Base.Candle")
      self.character:getInventory():AddItem(candle);
      sendAddItemToContainer(self.character:getInventory(), candle);
```

Code piece 18 - Adding InventoryItem to the inventory

```
      self.character:removeFromHands(self.weapon)
      self.character:getInventory():Remove(self.weapon);
      sendRemoveItemFromContainer(self.character:getInventory(),
self.weapon);
```

Code piece 19 - Deleting the InventoryItem from the inventory

Here are some examples of the code for how to create, delete and manipulate objects.

```
      local    trap    =    IsoTrap.new(self.weapon,    self.square:getCell(),
self.square);
      self.square:AddTileObject(trap);
      trap:transmitCompleteItemToClients();
```

Code piece 20 - Adding the IsoObject to the map

```
      self.trap:getSquare():transmitRemoveItemFromSquare(self.trap);
      self.trap:removeFromWorld();
      self.trap:removeFromSquare();
```

Code piece 21 - Deleting the IsoObject from the map

```
      self.generator:setActivated(self.activate)
      self.generator:sync()
```

Code piece 22 - IsoObject synchronisation on the map