Joel Cright

Quest Creator and Editor

Contents

[Overview – Setup 2](#_Toc507344396)

[Overview – Data 3](#_Toc507344397)

[Overview – Logic 4](#_Toc507344398)

[Main Form 5](#_Toc507344399)

[Quest Form 6](#_Toc507344400)

[Step Form 7](#_Toc507344401)

[Detail Form 8](#_Toc507344402)

# Overview – Setup

Setup is done in three parts:

1. Ensure you have the latest version of the editor
2. If SQL Server is not installed, install it and create a database
3. Change the app.config file’s connection string to match your database’s connection string

# Overview – Data

The Quest System is composed of three parts:

1. Quests
2. Steps
3. Details

A **Quest** is a high level overview of these parts. It contains just a few pieces of data, and is used as a tool to bind the rest of the system together.

A **Step** is exactly what it sounds like, a step towards completing a goal. It is a more focused view, and are low level parts, and where the player will get most of their information. They are low level parts in this system.

A **Detail** is comprised of the requirements for a **Step**. They are, indivudally, very simple things, but you can create a very complex **Step** by creating many **Details**. The separation of known data (a **Step**) and unknown data (**a Detail**) should allow for interesting **Quests** by hiding various twists.

When a Quest file is built, it is outputted to C:\ParallelZodiac\Quests\

# Overview – Logic

When a **Quest** is built, a script is created. This script is named after the **Quest** ID. Internally, the class name is the name of the **Quest**, and it inherits from a class called Quest, while creating objects of the classes Step and Detail. The class contains functions to create both **Steps** and **Details**, and the Quest System writes one line of code for each of those. The generated code is very simple. Everything else is written to always function within these confines.

Quest.cs contains functions to complete/fail quests, and move forward a step. The next step function tells the step object to complete, then iterates the CurrentStep (how it keeps track of where the player is), and finally calls a startup function on the NEW current step.

Step.cs contains most of the code. The startup function brings up dialogue, updates the journal, etc. There is a completion function, which gives the player their rewards for completing the **Step**. It hasfunctions to complete and fail Details, which in turn can complete or fail the Step. It also has a function to reset the step (on a failure. It also contains the function to create Details (but again, I want to move this), which looks like: "Details.Add(CreateDetail(12, 1, Player, 5, -1, -1, Doug));". That goes DetailID, StepID, Active, Action, Timer, DateLimit, Passive. The Action number is the index from the Action list.

Detail.cs is a small class. It has functions to create a timer, and a callback function which fails the Detail (and notifies the Step) when the timer is completed. It also creates an observer for the passive entity, as well as another callback function. That observer callback ticks up the amount of times the action has been completed, checks if is equal to the amount desired, and if they are it notifies the Step that the Detail is complete (as well as stopping the timer and changing its state).

The Quest Creator is basically a GUI that lets you make, make **Steps** within a **Quest**, and make **Details** within **Steps**, and pushes this data to a DB. It then has a button that will build each quest using that DB’s data, which generates a single script to be placed somewhere in Unity.

The system uses Unity’s PlayerPrefs for persistence.

In the TODO section, we have:

* Branching/failing Steps. This will require a simple UI change, and a medium effort back end change, but will result in much better gameplay.
* Step startup function. We have no journal, no dialogue system, nothing for it to update currently.
* Step reset. Do I keep track of everything’s original position? Does everything keep track of its own original position? This also only matters for quests you can attempt more than once. Maybe those quests will be of the type where resets don’t need to happen (ie, related NPCs/object don’t move, or are abundant enough it doesn’t matter if they move).
* Have the Quest (and possibly Step and Detail) classes inherit from monobehaviour)
* Host the database online. This removes two steps in the install, and allows everyone to share their quests without hassle.

# Main Form

**MENU ITEMS**

* Build All: Builds all Quests
* Build Selected: Build the Quest currently selected in the Quests data grid
* Build Out of Data: Builds all Quests whose versions don’t match the version in the database

**QUESTS BUTTONS**

* Add Quest: Opens a form to create a new Quest
* Edit Quest: Opens a form populated with the selected Quest’s info
* Delete Quest: Removes the selected Quest from the database

**QUESTS DATA ITEMS**

* ID: The Quest’s index in the database
* Name: The name of the Quest
* Description: The description of the Quest
* Current Step: The Step of the Quest the player is currently attempting

**STEPS BUTTONS**

* Add Step: Opens a form to create a new Step
* Edit Step: Opens a form populated with the selected Step’s info
* Delete Step: Removes the selected Step from the database

**STEPS DATA ITEMS**

* ID: The Step’s index in the database
* QuestID: The ID of the Quest the Step belongs to
* StepName: The name of the Step

**DETAILS BUTTONS**

* Add Detail: Opens a form to create a new Detail
* Edit Detail: Opens a form populated with the selected Detail’s info
* Delete Detail: Removes the selected Detail from the database

**DETAILS DATA ITEMS**

* ID: The Detail’s index in the database
* Active Entity: The name of the Entity performing the action
* Action: The ID of the action required
* Amount: The amount of times the action is required to be performed. -1 is the same as 1
* Timer: How long (in in-game minutes) the player has to complete the action before the Detail triggers a failure
* Passive Entity: The name of the Entity the action is happening to

# Quest Form

**WHAT IS A QUEST**

This is the high level view of a series of actions a player must complete.

**QUEST NAME**

The name of the Quest. Currently does not support special characters.

**QUEST DESCRIPTION**

The overview of the Quest. Supports special characters.

**CREATE/EDIT QUEST**

Saves the Quest data to the database. The text on the button is dependent on if you opened the form from “Add Quest” or “Edit Quest”

**CANCEL**

Closes the form, does not save anything.

# Step Form

**WHAT IS A STEP**

This is the action the player is told they must do.

**STEP NAME**

The name of the step. Does not support special characters.

**DIALOGUE**

This is the dialogue that will occur when the Step begins. Formatting is being worked on.

**EXCEPTION DIALOGUE**

This is custom dialogue that will occur when the Step fails or the player attempts to turn in an uncompleted Step. Formatting is being worked on.

**USE GENERIC EXCEPTION DIALOGUE**

If checked, the step will (in addition to custom and typed exception dialogue) use exception dialogue that is flagged as “Generic” (eg, should work anywhere). This can save time with repeatable quests, or more generic quests that get implemented often.

**USE TYPED EXCEPTION DIALOGUE**

If checked, the step will (in addition to custom and generic exception dialogue) use exception dialogue that is flagged as with the same action as the Details of the Step. This is more relevant dialogue than generic. This can save time with repeatable quests, or more generic quests that get implemented often.

**AMOUNT OF ATTEMPTS**

How many tries the player has at this step. If all attempts are used up, the Quest fails. Branching failures (eg, failing the Step merely opens a new Quest line instead of failing the Quest) will come in the next build.

**REWARD & AMOUNT**

The name of the reward and the amount given. You can add as many rewards and amounts as you would like. Use -1 to specify the item should be removed (quest related gear, or it gets stolen, etc).

**CREATE/EDIT STEP**

Saves the Step’s data to the database. The text on the button is dependent on if you opened the form from “Add Step” or “Edit Step”

**CANCEL**

Closes the form, does not save anything.

# Detail Form

**WHAT IS A DETAIL**

A Detail is the nuts and bolts of a Step. The player does not have access to this information, so you can use it to surprise them. You can add multiple details to a Step, to create more complex Steps. For example, to create an Escort Step, you would add a Detail for player to protect the NPC, and then a Detail for the NPC to travel somewhere.

**ACTIVE ENTITY**

The entity performing the action. This will usually be the player, but not always. It must have a GameObject representation.

**ACTION**

This is the action to be performed by the Active Entity. This is a list of the basic actions you can perform, and is predetermined. Actions CAN be added to the list without hassle.

**PASSIVE ENTITY**

This is the entity the action is being performed on. If the action is Travel, this is the location the Active Entity must travel to. It must have a GameObject representation.

**AMOUNT**

This is the amount of times the action must be performed. -1 is the same as 1. 0 gets changed to -1.

**TIMER**

How many in-game minutes the player has to complete the Detail. If they run out of time, the Detail triggers a Step failure, which checks how many attempts have been completed and then either resets or triggers a Quest failure.

**TIME DONE BY**

Not yet implemented. We need an in-game calendar first.

**CREATE/EDIT DETAIL**

Saves the Detail’s data to the database. The text on the button is dependent on if you opened the form from “Add Detail” or “Edit Detail”

**CANCEL**

Closes the form, does not save anything.