Computers & the Humanities 281

Quizzes in Revolution

Any type of instruction would not be complete unless it was accompanied by some type of assessment. Computers provide several means by which this may be accomplished. We will discuss several types of assessment methods as created and implemented in Revolution. As before, we will concentrate on the methods and not deal with issues of content. The methods described below are by no means the only effective way to address the issue of knowledge assessment.

True/False

The basic premise behind true/false questions is that the user is given a statement and must decide whether it is true or false under the given circumstances. The first step, then, would be to create some fields containing statements and name them "Question1", "Question2", etc. The most obvious tool that we would use next would be radio buttons, as they can be set to be mutually exclusive (as most true/false questions you encounter are either one or the other). We could create a pair of radio buttons (one named "True", the other "False"), set the properties such that their borders were not showing (important later), then group each set and make sure the **Hilite one radio button at a time** property for the grouped pair is checked. Once that was set, we could duplicate the group for each statement and name them accordingly: "TFGrp1", "TFGrp2", etc.

Now we need to design a method by which the answers input by the users can be evaluated. One approach would be to create a custom property for each statement field, naming it "corrAnswer." We would then set the value of corrAnswer to true or false, depending upon the correct answer for that particular question. Then we would create a check box button which would give the users the option of showing or hiding the correct answers to each question. This is where the **showBorder** property becomes important.

When scripted properly, the checkbox (when selected) would draw a box around the correct answer for each question, thus visually indicating to the user the correct choice. When the checkbox is deselected, the boxes would disappear. While this is not the most secure method of assessing a student's knowledge (i.e., students can cheat when we allow them to check their own answers), it certainly is one approach, especially in a low-stakes situation. In this type of set-up, it would not take a lot of extra scripting to enable the computer to do the assessment and compile a score.

Even though we allow the users to check their own answers, we need a way to erase the answers they input in order to avoid influencing the next user that may come along. When the preOpenCard message was introduced, it was intimated that preOpenCard handlers are often used to initialize, set up, clean up, or reset the card when the users first access it. It is often preferable to write cleanup or reset handlers as separate handlers and place them in the stack script so that we can invoke them when and where we would like, particularly if we have several quiz/exercise cards within a stack. Cleanup is typically initiated when the user finishes the exercise/activity, and setup would typically be invoked from the handler that takes the user into the activity (as was done in the Golden Age stack). We would also want setUp to be available to allow the user to restart the activity if so desired.

For our true/false quiz as presently constituted, what does a cleanUp handler need to do on this card?

- hide the answers
- unhilight the True and False buttons
- also unhilight the "Show Answers" button--can do it with the same loop

And what does a setup handler need to do?

- for this exercise type, same as cleanup
- consequently, just invoke setup

Even though the cleanup would logically be invoked each time a user finishes, we would still want to execute the same statements in a setup just to make sure it was done properly (remember to account for all contingencies). In the development stage, it would be best to make two buttons to invoke cleanUp and setUp respectively. This would allow us to verify that they do indeed operate as we would wish. In a real application we would eventually delete these and invoke them as explained above. It would be useful to remember that grouped radio buttons have a **hilitedButtonName** property which returns the name of the hilited button within that group. If we `set the hilitedButtonName` of group *groupName* to "none" then we can effectively erase the answers input by the previous user.

With appropriately named handlers on the stack script, we can invoke the handlers by sending the appropriate message, which enables us to execute the statements within these handlers from any object on the card/stack. The renders unnecessary the need for distinct handlers in each of the objects.

Multiple Choice: First Iteration

The principles operating behind multiple choice questions are similar to true/false principles. Consequently we could use radio buttons and just add more to the group to accommodate the number of possible responses. The type of feedback would be similar: display visually the proper answer in some manner. This would be an effective approach.

However, let's try something different. Let's give feedback in the form of a popup field. First we'll use fields for the possible answers, which will obviate the need for radio buttons. Then we'll create a sentence or two commenting on the users' choices, noting the level of their appropriateness. We could put a mouseUp handler in each field to show/hide the specific feedback, but this would create an inordinate amount of work for us, especially if we would like to have several questions.

Instead, let's create a more general handler and place it further down the hierarchy where it can serve us from several locations (which would also eliminate the need to script/change several handlers). As before, we could write two custom handlers, one `on wrongAns` and the other `on rightAns`. For wrong answers we would first put feedback into a field, show the field that indicates their answer is wrong, then hide the field after they've had time to read it. We would change the cursor to watch to indicate to the users that the computer is working and that they need not get too anxious (the cursor will change back on its own at the conclusion of the handler, indicating that the computer has finished its work). The method of handling a right answer is similar, except that the feedback would change appropriately.

With these two handlers in the card or stack script, we could then invoke the appropriate handler from each field as needed, obviating the need to do extensive scripting for each field. This simplifies immensely the scripting that we have to do for each field. Anytime we would like to change the function of each answer we would be able to do so by changing one general handler only instead of several.

Multiple Choice: Second Iteration

Despite the fact that the previous design works and works well, it is definitely not the most elegant design. It works wonderfully for canned questions and is and could be duplicated and adapted to situations with more questions. However, there is definitely another approach to presenting multiple choice questions. One would be to employ the services of a question bank, where all the possible questions and answers are stored in a hidden field or external file, retrieved at appropriate times, and displayed for the users (much like what happened in the vocabulary drill).

To this end we would first create a question-item bank field. As before, fields of this type can be scrolling style (especially if you have several questions and answers) and have a small font to hold lots of text (remember that the users will not see this particular field). Convention dictates the organization of the data within the field:

- put the question first
- follow it with the correct answer
- follow that with three "distractor" choices
- separate all by commas or distinctive delimiter

The last convention is essential, as using delimiters makes each into an item, which allows us to parse the question bank by referencing item units (e.g., the first item of each line is the question, the second item is the correct answer, etc.). Such a design ensures that the data for each question is organized in a consistent manner.

Now let's create a handler to access a line in the question bank and fill all the appropriate fields with the correct data. At times it helps in scripting to first get one aspect of the handler to work before working on another portion. This is an effective way to avoid trouble with isolating bugs within the entire handler. Initially, then, for this script, concentrate on just getting it working for one line of the question bank.

One approach would be to start with a variable into which we place 1. Put that line of the question bank into another variable (again, we'll have to account for the other lines in the question bank later). Put item 1 of that variable (the question) into the question field. The remaining items would go into corresponding answer fields. This would be facilitated by naming the answer fields appropriately (e.g., "Answer1", "Answer2", etc.). This could all be contained within a custom handler (`on pickQuestion` or something similar) in the card script that could be invoked from a control object on the card.

Multiple Choice: Third Iteration

If you haven't picked up on the most blatant limitation of this design, the users quickly will: The correct answer is always the first choice. To solve this we could randomize the order in which the answers are loaded into the answer fields. This gets a little dicey: We not only have to keep track of where the correct answer is, but we must also (following the design of our activity) script accordingly the fields containing the answers.

To this end we would need to modify our handler which picks the questions. We could create a variable containing the names of the available answer fields (remember: "Answer1", "Answer2", etc.). Inside the repeat loop within our handler (I hope you empolyed one), we can generate a random number based upon the number of fields we have named (similar technique as what we encountered in the vocabulary drill activity). Once we obtain a field name from the list, we need to delete it from the list so we don't encounter it again. We don't have to worry about checking if the list is empty because we're working in a fixed environment: There will always be four fields and always four possible choices from the question bank.

We now need to deal with the issue of scripting the fields to give appropriate feedback. There are a number of ways to deal with this, but one effective way is to set the script of objects with which we're working Rather than have fixed mouseUp handlers in each choice field, we'll have the handler build an appropriate mouseUp handler in a variable and then set the script of each choice field dynamically to contain that handler:

- First we'll put `on mouseUp` in the first line.
- Then we'll put `end mouseUp` in the third line.
- Between those two (in the second line) we'll put `RightAns` for the field that contains the correct answer and `WrongAns` for the others.

We know (because of the way we designed our question bank) that the first answer we take is the correct answer, so we just have to assure that the rightAns handler is assigned to the field that was randomly picked to hold this answer. All others, then, will indicate that the answer was incorrect. This handler is compiled and placed within a variable. Finally the script of the current field is set to this handler. In this manner all the fields dynamically receive the handler appropriate for their contents.

Multiple Choice: Fourth Iteration

Now that we have it working with one question, we can now alter the handler to move through the questions in the question bank. If we wanted the questions to be presented randomly, we could use the same random technique we used previously in the Vocabulary Drill. In this situation, however, the questions are supposed to be presented in order. In this case we need to increment the question number variable by one. This won't do any good, however, if we immediately initialize it to 1 at the beginning, so let's remove that line. We still need to initialize it somewhere, so let's do it in the setup handler. We notice immediately that our brilliant idea doesn't solve the problem. As you may recall, any variables created in a handler only exist while the handler is active. When the handler ends, its variables end their useful existence. In order to prevent this, we have to tell Revolution we want to keep the variable intact. We do this by declaring the variable as a global variable.

Once we have a global variable defined, any handler can use it. We need only to add a global declaration in that particular handler with the same variable name. Even though this is a global variable, we can still modify it. The fact that it's a global variable does not place any limitations on what can be done with it. In fact, just the opposite happens, as it is now available to any given handler which can then access its contents and modify them if so desired.

Multiple Choice: Fifth Iteration

Our multiple choice questions now work in cycling through the possible questions, but we haven't accounted for the time when we run out of questions. To solve this we can check to see if our question number is greater than the number of questions. We could encode in the handler the exact number of questions, but if we were ever to add or subtract questions, we would have to remember to change this number as well. It would be easier (and smarter) for us to do it dynamically, to have the application determine just how many questions are in the question bank. Due to the way we designed our bank initially, the number of questions corresponds exactly with the number of lines in the question bank field. Once we determine that the end has been reached, we could display an answer dialog informing the users that the end has been reached. We would then exit the handler, using the `exit to top` statement.

We also need to account for cleanup and setup of the quiz. This is as simple as emptying the answer fields to prepare for next questions. We also need to enable the a button to allow the user go to the next question. By dynamically changing the label of our button, it could alternately serve as the "Start Quiz" button and the "Next Question" button. By being aware of such properties and putting them to good use, we can to create a more useful and inviting interface.

Short Answer: First Iteration

We will now move on into the realm of short answer quizzes. These are designed to prompt users for a response, check the input, and then give appropriate feedback. For a short answer question, then, let's start out with:

- A Question field for the question
- An Answer field for the student to type their answer
- A Check Answer button for the student to click to check the answer
- A pop up feedback field where feedback on the answer is displayed

We would want to group all these objects so that we can create more questions. We need to be sure to set the **backgroundBehavior** property of the group to true. That will cause it to be automatically copied onto newly created cards.

The other two fields that we will utilize will normally be hidden. The anticipated field lists answers the author expects a student might enter. FeedbackList has a corresponding feedback line for each answer. While "right" and "wrong" are often appropriate, meaningful feedback that explains why the answer is right or wrong is sometimes desired, as it provides a learning opportunity for the users.

For our present design, a Check button (whose label is "Check Answer") would do most of the work:

- Takes what the student typed in the answer field
- Looks for it in the "Anticipated" field
- Determines which line matches the user input
- Puts the corresponding line of "FeedbackList" into the "Feedback" field
- Sets the cursor to watch, then back to hand.

What to do if the answer isn't there? Remember that the result will not be empty (it says "not found"). If we use the convention that the first anticipated answer is the best one, we can build a meaningful feedback line. This could all be contained in a custom handler in the card script.

Short Answer: Second Iteration

We now need to add a few finishing touches. We need to clean up the question area for the next user. We accomplish this by emptying the student's answer. For setting up, we need only to clean up and then insert the cursor in the appropriate field. One more thing we can do to make it more convenient for the users is to let them just press return, tab or enter after they type in their answer. These handlers, of course, would be in the script of the answer field.

Matching: Map Drag and Drop

For matching questions on paper-and-pencil tests, the classic form is usually a two-column presentation. The user is expected to match an item in the first column with an item in the second column, often by drawing a line from one to the other. However, Revolution allows us to modify this design in a number of ways, one of which is a labeling type of exercise.

First we start with a simple picture with a gaggle of labels nearby, for example, a map of Central America. The object is to drag the names of these countries to the appropriate location within the picture. First let's get it working for just one label. We'll create a field, give it the name of "Mexico", and type "Mexico" in the field (just like our labeling exercise). We now need a way to determine if the label is within the correct country. To make it easy on ourselves, we'll place buttons within the countries themselves, creating a type of target area indicating to the users where they need to place the labels. Each of these buttons will then be given a name that corresponds to their state.

The stage is now set. We just need to enable the users to drag the labels to the picture and determine if they have made a correct choice. This first part, allowing users to drag the labels around is slightly tricky, but simply think it through in English: We want the field to move with the cursor while the mouse is down. This is accomplished with the `grab` command within a `mouseDown` handler. The location of the field is dynamically changed through this command until the mouse is released.

We then need to determine if the user drops the field in the correct location. Again in English, when the mouse is up we need to check the location of the field and determine if it is similar to the location of the button to which it corresponds. With a `mouseUp` handler we can check the location of the field against the `rectangle`, or `rect` property of the button with the same name ("Mexico"). If the location is within the rectangular area of the button, then we have a match, and the field is snapped into its proper location. Otherwise, it's not a match and the field is returned to its original location.

Now that we have it working for one field, let's get it operational for all. Since we're performing essentially the same function for each label, this would indicate strongly to us that we need to place it in a script elsewhere. This would be an opportune time to utilize the card script. Make sure you put appropriate do-nothing mouse event handler in all other fields that you do not wish to be part of the activity. We also want to give the users some visual feedback as to whether they made a correct match or not, perhaps a button with a changing icon.

As mentioned earlier, we also need to return the field to its original location upon a wrong choice. If we have a field that contains all the original locations of the fields, then all we need to to is access the location that corresponds to the field and set it. To get that particular field set up, we could create a temporary button that generates the list of locations for us, organized by line (e.g., line 5 holds the location of field 5). We could also use this field of locations to reset the activity, a handler that would move through the list of locations and restore all fields to their original place in order to prepare the activity for the next user.

Concluding Matter

As you have probably surmised, there are numerous ways in which to create some type of assessment activity. Once more, these examples are not necessarily the most elegant or effective, nor was this designed to be comprehensive or exclusive. They were designed mainly to expose you to some of the multifarious methods of assessment. Hopefully you will find these general principles useful.

Your assignment, then, is to create some sort of assessment of a topic of your choice:

1. Create a stack and entitle it ***YourName*--Assessment**. Give the stack an appropriate label (based on the subject with which you are working).
2. Create some sort of assessment activity that includes at least five questions or assessment items.
3. Use your creativity and experience with Revolution development to improve on the examples presented in class. Consider improvements to layout and presentation, options and flexibility, and visual elements as well as more efficient scripting.
4. Put a copy of your finished stack in the **Assignments** folder.

As you are aware, you need to include some type of assessment activity in your final project. You may use any of the methods described above (excluding the true/false method outlined above) or create one of your own. You need to demonstrate some creativity and sophistication in your individual assessments and go beyond the basics employed in the examples.

Course Schedule
Main Page