Simulations and Simulation-Based Courseware

Logicdriven, LLC

Contents

Architectural History

The architecture for simulation-based courseware described in this document is the result of over fifteen years of refinement, starting with Toolbook-authored courseware developed for the US Government in the mid-1990's. Logicdriven personnel first adapted this design to simulation-based courseware in 2001 when developing the initial release of CAE's Simifinity software. Over the years, this architecture has been used successfully on over a dozen projects with hundreds of hours of courseware with simulations and players built in a variety of tools, including C++, Visual Basic, C#, Adobe Flash, and Unity 3D.

Simulation Training in Context

Simulation-based courseware content is only one part of a training solution. Training individuals to operate or maintain complex technical systems involves teaching theoretical information, conceptual imagery, knowledge-based checks on learning and formal assessments, doctrine and situational awareness training, crew resource management training, understanding the support environment, and more. Such material can be presented as standalone units of instruction, or can be blended into simulation-based content.

Logicdriven's Impression toolchain includes a complete authoring and playback framework for both simulation and non-simulation based content, allowing seamless blending of all types of learning material into either a single piece of instruction or an entire course.

The authoring environment supports common instructional production requirements, including associating content with learning objectives, referencing source documentation, and relating supporting media items to source materials, including original engineering drawings.

Impression's open data architecture and runtime framework ensure that the software can be used and extended in a nearly unlimited number of ways, from coupling production management systems (including discrepancy reporting, asset request and management, and customer review systems) to creating support utilities and training artifact generation tools.

Impression's simulation-agnostic support framework is unique within the industry; no other product that we are aware of provides the ability for content creators to author simulation-based content as part of the toolchain in the same fashion as they would author non-simulation-based content. We believe that using Impression to create simulation-based courseware will provide new opportunities to delight your customers.

Simulation-Based Courseware

Overview

To understand how simulation-based courseware works, we need to consider the structure of a simulation and how the courseware interfaces with it.

A *simulation* (or *simulator*, if you're looking at it from a hardware point of view) consists of several areas of functionality. These can include a mechanism for managing state data, rules or executable code for changing state data based on the behavior to be simulated, a control mechanism for startup/shutdown/reset, and an interface—either software-based, hardware-based, or both—to display the simulation state and allow users to interact with the simulation. The specifics architecture of any given simulation will depend on the design goals and constraints of the simulation, but all will generally include the functionality described above.

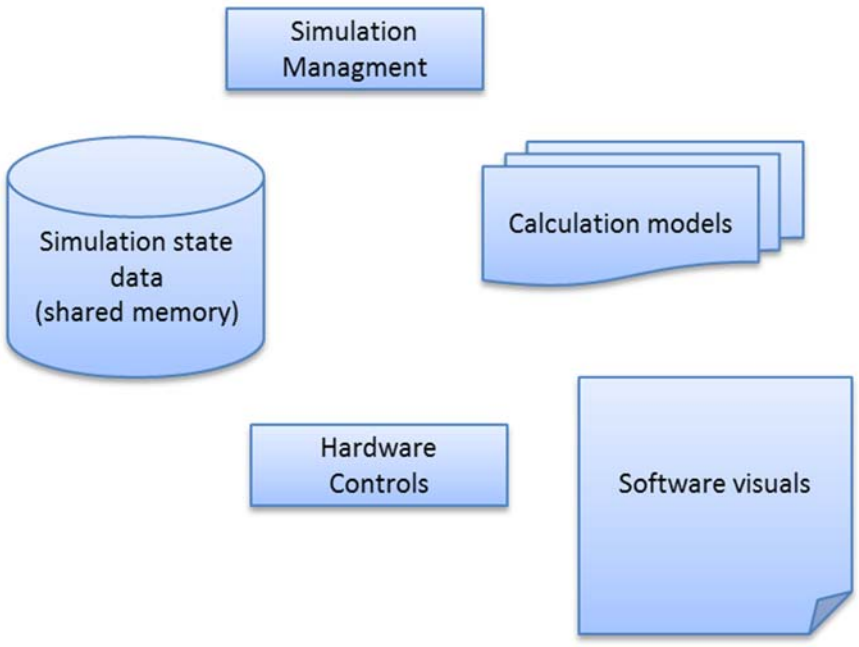


Figure 1: Simulation components

When considering the pedagogical value of a simulation, we can look at typical use cases for a learner (student) and an instructor with the simulation. The instructor may begin by showing the student how a specific task is performed using the simulation; in this case, the student is a passive observer while the instructor performs all interactions with the simulation. The instructor may have the student perform the task step by step with coaching; that is, the instructor will explain, step by step, what actions the student is to perform, how those actions relate to learning objectives, and, if the student performs an incorrect action, why the action was incorrect (in terms of the procedure, the learning objective, or both) and what the correct action should be. The instructor may simply evaluate the student as they perform the task, providing no guidance or feedback during the performance; instead, all feedback and scoring information is provided to the student when their performance is complete. And, of course, the instructor may simply allow the student unfettered access to the simulation without providing any guidance or feedback.

The use cases listed are not always provided, nor are they always as clearly defined as shown above. Instructors may not allow students to "just mess with" the simulation; or they may allow the student to practice a task with no guidance or feedback, stopping and remediating them only if the student performs an action incorrectly. The instructor may also need to reset the simulation or place the simulation into a specific state in order to teach a specific task.

In any event, to provide proper training, the instructor will prepare the simulation, monitor the student's interaction with the simulation, and make pedagogical decisions based on both the student's actions and the current state of the simulation as they relate to the task.

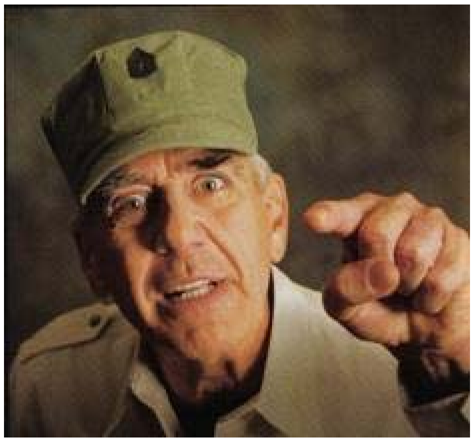


Figure 2: Pedagogical Remediation (instructor-led)

Courseware Control

In simulation-based courseware, we replace the living instructor with software. The software must perform the same actions as the instructor: prepare the simulation, show the student how to perform a task, monitor the student's interactions, evaluate those actions in the context of both the learning objective and the current state of the simulation, and provide appropriate feedback or remediation to the student based on the learning mode.

To meet these goals, the courseware must be able to:

- Set the simulation state
- Evaluate all interactions with the simulation's user interface
- Examine the simulation state

If we return to the simulation model described earlier, we can simplify that model by grouping all simulation functionality into one of two areas: the *simulation process* and the *simulation user interface*. The simulation process contains all functionality not directly related to display and interaction, including simulation state data, rules for changing the state data, and the control mechanism. The simulation user interface consists of the hardware or software interface used to both display and change state.

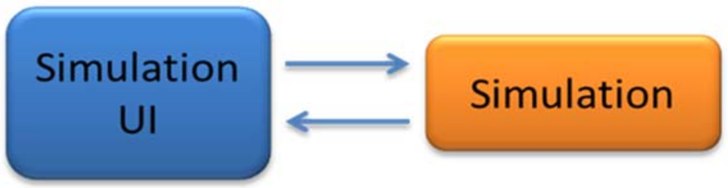


Figure 3: Simulation and Simulation UI

With this simplified model, we can see that there must be a communication mechanism to allow the user interface to both display simulation state and pass user-driven changes back to the simulation process. There must also be a way for the simulation process to pass relevant state changes (either user-driven or rule-based) back to the user interface; whether this is done by polling or by using a notification mechanism is irrelevant. The simulation process must also support non-interface-related changes to the simulation, including simulation startup and shutdown, freeze and reset, and fault injection.

It is into the boundary between the simulation process and the simulation user interface that the courseware controller is inserted.

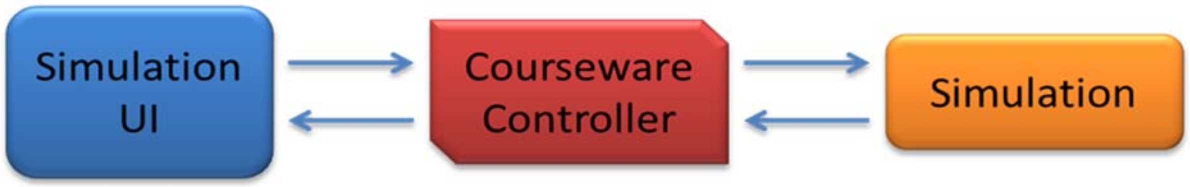


Figure 4 Courseware Controller inserted into simulation

From this location, the controller can intercept the user-driven interface changes and evaluate them. Although not strictly required, if the controller acts as an intermediary between the user interface and the simulation process, user-driven interface changes can be blocked and prevented from affecting the simulation (if, for example, the student performed an incorrect action). Since changes may be blocked by the controller, we'll call these *change requests*.

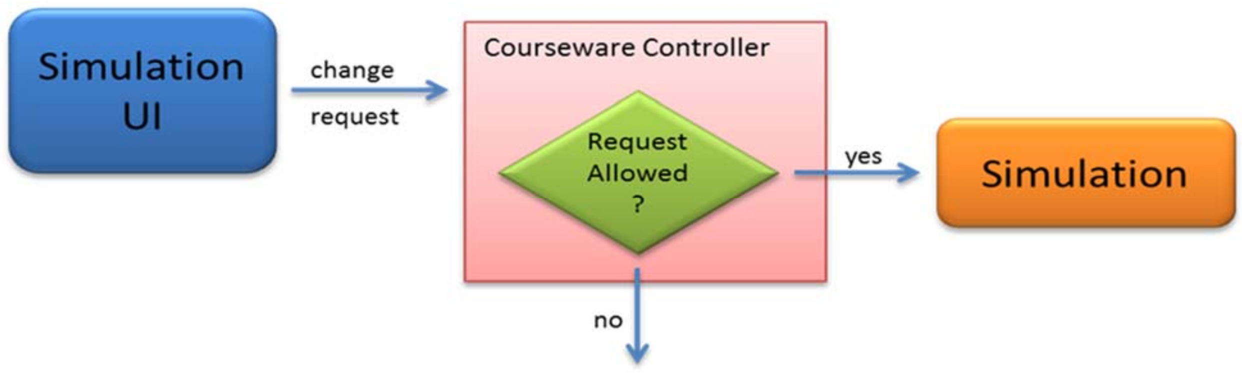


Figure 5: Evaluating change requests

The controller can also set user-driven interface changes programmatically. In the case of hardware user interface elements (switches, *etc*.), the design of the hardware may limit this capability.

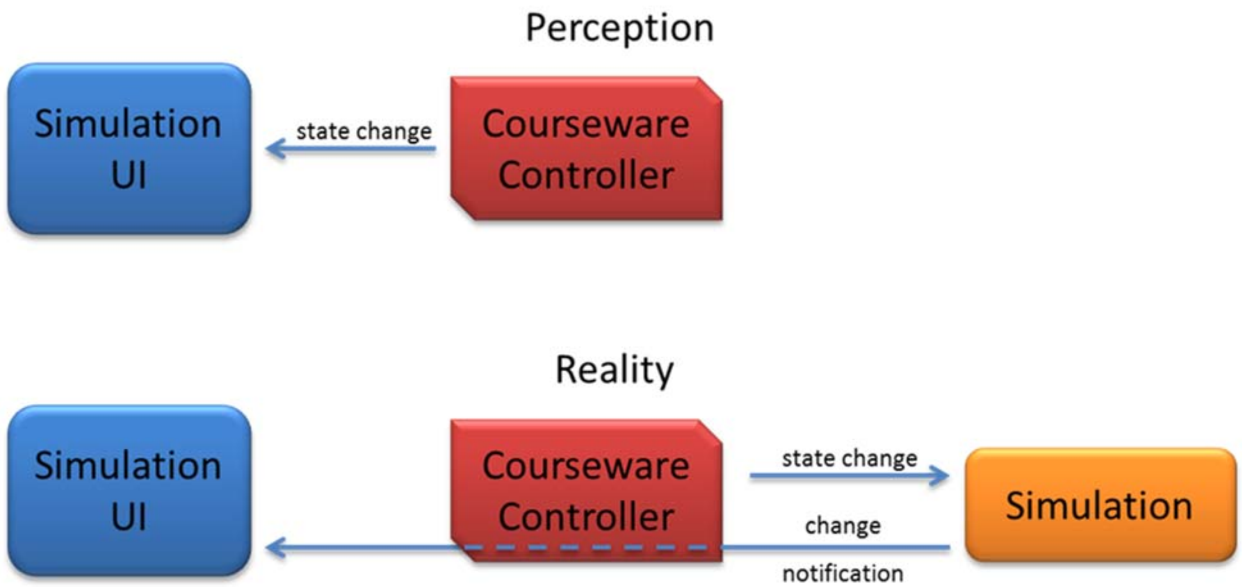


Figure 6: Changing user interface state (how it looks vs. how it usually works)

The controller can also receive simulation state change notifications by using the same communication mechanism used to pass changes back to the user interface. Finally, the controller can pass non-interface changes to the simulation. As we can see, by hooking into the communication mechanism at this point, the courseware controller is able to meet its design goals.

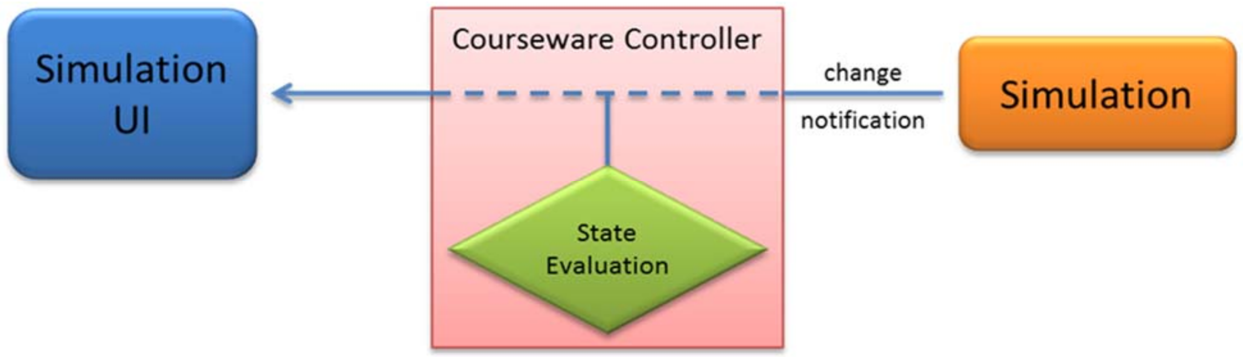


Figure 7: Simulation state notification processing

Courseware Data

Just as a living instructor uses course content (*lesson data*) to teach tasks to a student using the simulation, a courseware controller must also use course content to assist the learner. In the case of a physical instructor, lesson data may include simulation prep instructions, practical as well as theoretical or system supporting materials, task materials (including technical publications and job sheets), instructor guidance for providing the student with feedback, and lesson completion or success criteria. For a courseware controller, this information is provided to the controller in a structured format to allow the controller software to control the simulation experience for the student.

Note that although it is possible to design a series of "hard-coded" courseware controllers, one for each task to be performed on the simulation, efficiency and cost concerns usually drive the design of the courseware controller to use a data-driven approach. In this model, the controller reads the courseware data passed to it and adjusts its behavior based on the data.

Courseware Examples

Let's look at a simple example of using lesson data and a courseware controller to teach a task. Our simulation will be of a simple light and switch, and we will teach the student how to use the switch to turn on the light.

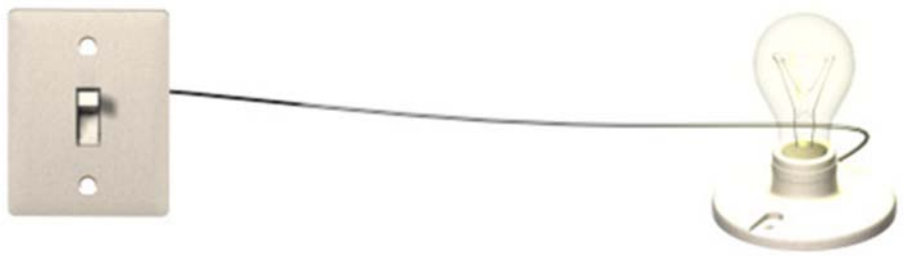


Figure 8: Simple simulation

We'll begin by defining the state data for a simulation model:

Data Model Element	Possible values
LIGHT_SWITCH	On, off
LIGHT_BULB	Lit, unlit

The code for the model will switch the state of the bulb based on the value of the switch.

Next, we'll define the simulation user interface:

Interface Element	Data Model Inputs	Data Model Outputs
Light switch	LIGHT_SWITCH	LIGHT_SWITCH
Light bulb	LIGHT_BULB	*n/a*

The courseware data will contain the following:

1. **Initial simulation state**: `LIGHT_SWITCH = off`. Since the simulation will change the state of the bulb based on the switch value, we do not need to specify the value of the LIGHT_BULB data element.
2. **Allowed student actions**: `LIGHT_SWITCH = on`.
3. **Required (ending) simulation state**: `LIGHT_BULB = lit`. Note that we specify the state of the bulb rather than simply checking or including the state of the switch. This may be done because the content author is aware of the behavior of the simulation and does not need to include the switch state; or it may have been omitted in more complex examples because there are multiple ways to place the simulation into the desired ending state and the author does not wish to prescribe the specific method the student must use to achieve the desired state.

The courseware controller will begin the lesson by setting the simulation to its initial state. It will then monitor both the simulation process and the simulation user interface, looking for changes.

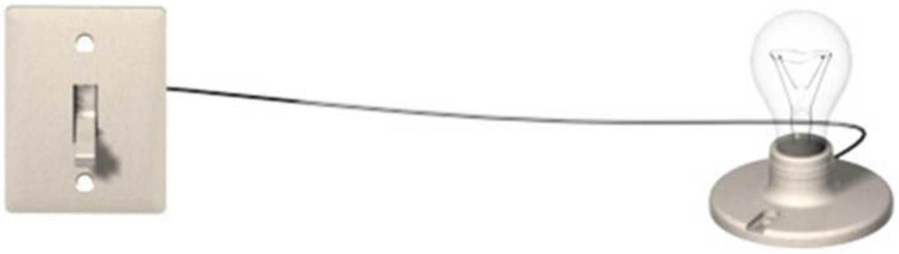


Figure 9: Initial state (power off)

As some point, the student will attempt to change the value of the light switch from `off` to `on`. When the student flips the switch, the user interface will broadcast this change request. The courseware controller will intercept the request before it is passed to the simulation process and evaluate it. In this case, changing the switch value is an allowed student action, and so the controller will let the change request pass from the user interface to the simulation process.

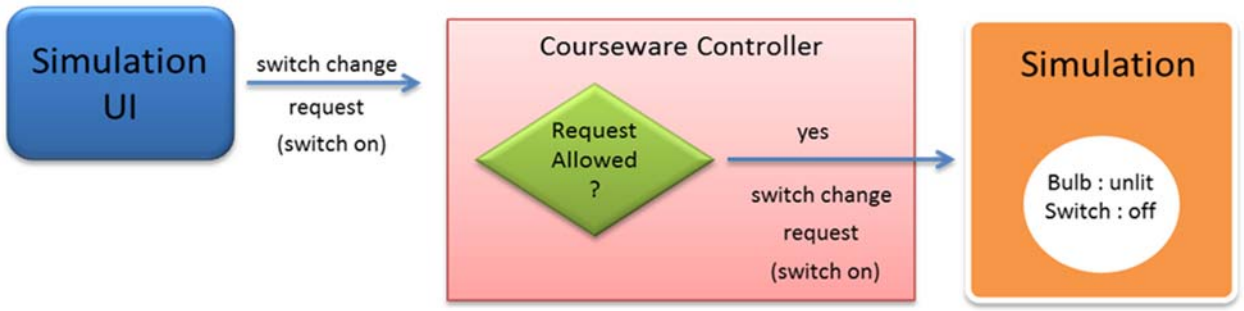


Figure 10: Approving the change request

Upon receiving the change request from the user interface or controller, the simulation process will save the switch position as part of its state data. On the next calculation cycle, the simulation process will change the value of the light bulb from `unlit` to `lit`. The simulation process will then broadcast the state change. The user interface will receive the change notification and light the bulb. The courseware controller will also receive the change notification, comparing the new value to the required simulation state. Since the new simulation state now matches the required state, the controller will recognize that the task has been completed.

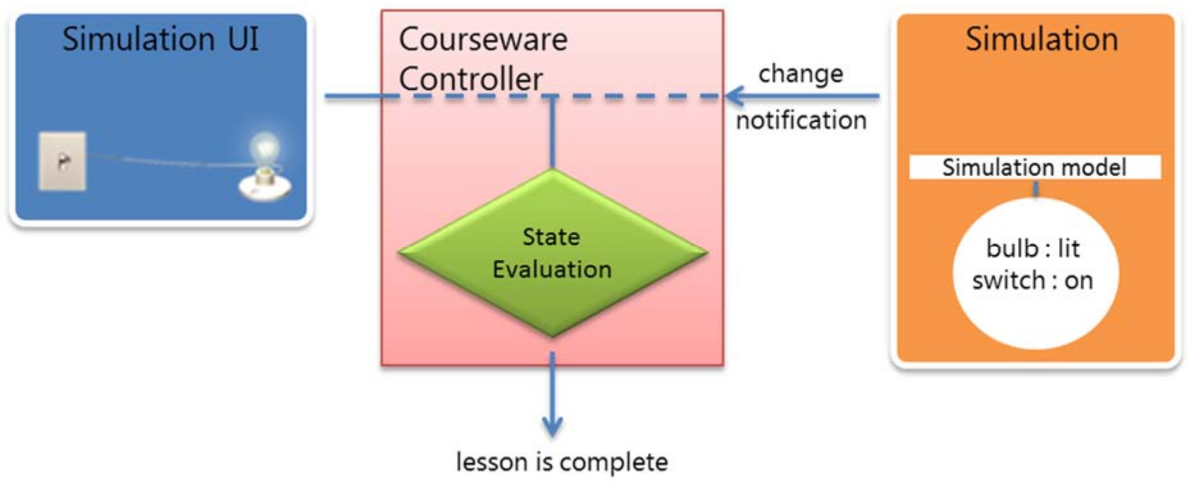


Figure 11: Lesson complete

The example above is a trivial one; there is very little decision-making required by the student, as our simulation has only one interactive element. Let's extend the example above to illustrate how a multi-step procedure can be managed by a courseware controller. This example will also include remediation information.

We will extend the simulation model to include a circuit breaker that controls whether or not power is flowing to the switch. The new model state data is:

Data Model Element	**Possible values**
MAIN_POWER	On, off
LIGHT_SWITCH	On, off
LIGHT_BULB	Lit, unlit

The code for the model will not light the bulb unless both main power and the switch are on.

The extended user interface is:

Interface Element	**Data Model Inputs**	**Data Model Outputs**
Circuit breaker	MAIN_POWER	MAIN_POWER
Light switch	LIGHT_SWITCH	LIGHT_SWITCH
Light bulb	LIGHT_BULB	*n/a*

The procedure will be to first turn on the circuit breaker, and then turn on the switch. The courseware data will contain the following:

1. **Initial simulation state**: `MAIN_POWER = off`, `LIGHT_SWITCH = off`.
2. **Step 1 allowed actions**: `MAIN_POWER = on.`

3. **Step 1 remediation**: Display "Turn on the circuit breaker before changing the switch." if the student attempts to change the light switch.
4. **Step 1 required (ending) simulation state**: `MAIN_POWER = on`.
5. **Step 2 allowed actions**: `LIGHT_SWITCH = on`.
6. **Step 2 remediation**: Display "Do not turn off main power." if the student attempts to change the circuit breaker.
7. **Required (ending) simulation state**: `LIGHT_BULB = lit`.

As with our first example, the courseware will begin by setting the initial simulation state, and then begin monitoring both the user interface and the simulation process for state changes and change requests. Because the courseware data specifies multiple steps for the task, the controller will (having just started) use the data for Step 1.

Let's assume that the student begins by attempting to change the value of the light switch. Since this is not an allowed action, the courseware controller will prevent the change request from reaching the simulation process and display the remediation message to the student.

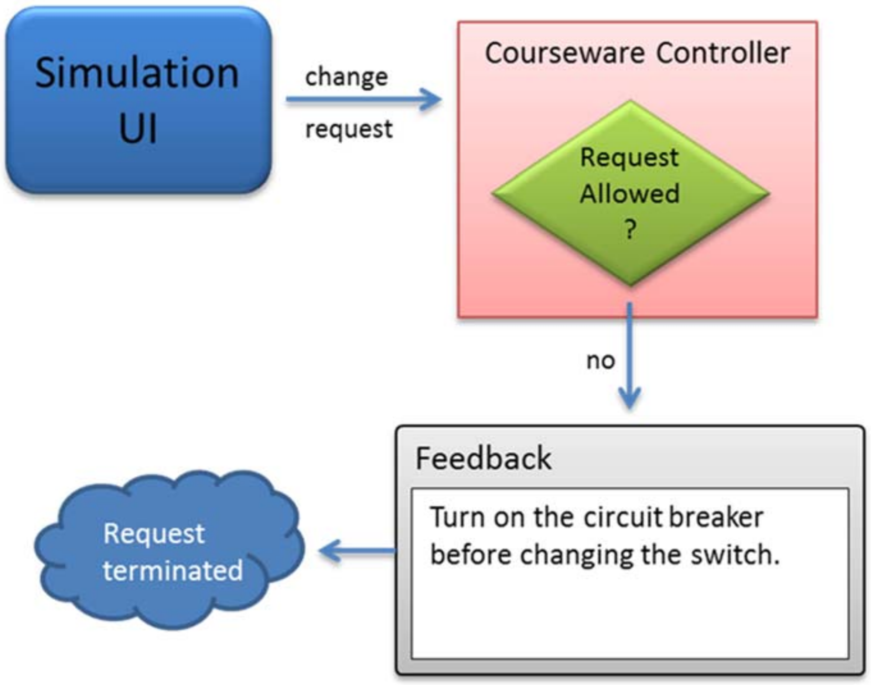


Figure 12: Unallowed change request processing

The student, reacting to the remediation message, then attempts to flip the circuit breaker. This is an allowed action, so the courseware controller will permit the request to reach the simulation process. The simulation process will store the change, the controller will note that the Step 1 required simulation state has been reached, and will then monitor change request and simulation state changes for Step 2.

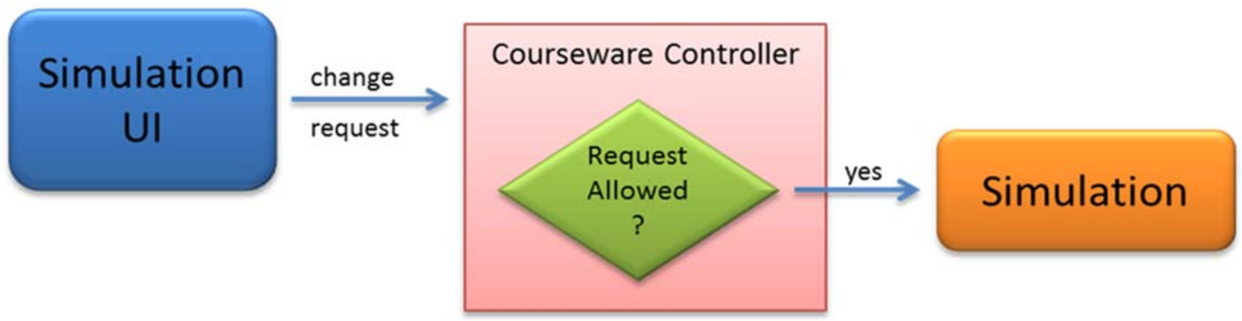


Figure 13: Allowed change request processing

Now that main power is active, the student must now flip the light switch. They may attempt to turn the circuit breaker back off, which will result in cancellation of the action and display of the remediation message. Eventually, though, they will turn the light switch on, which is an allowed action for Step 2. The courseware controller will allow the change, receive the bulb state change notification from the simulation process, and determine that Step 2 is complete. Since this is the last step, the controller will note that the task is complete.

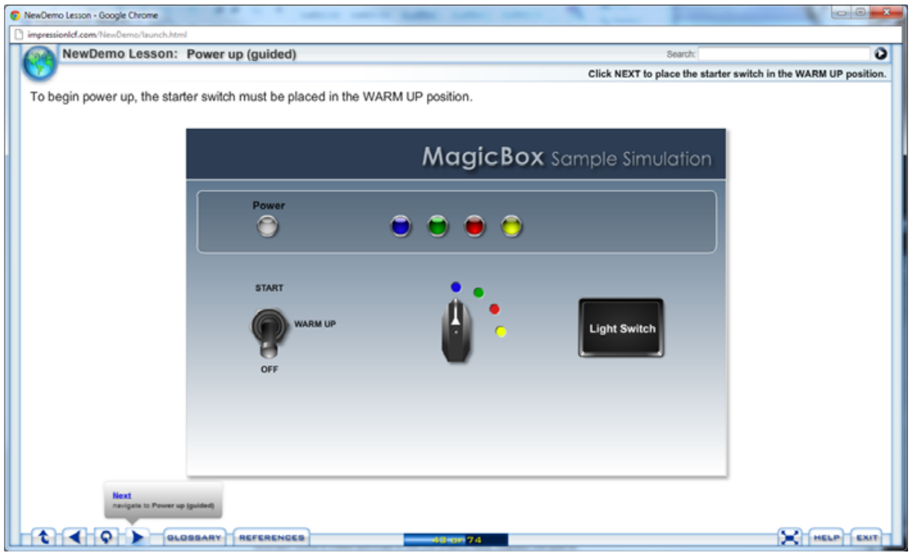
Procedural Variations

Both examples above imply that the student is engaging with the simulation in "practice" mode—that is, the student is expected to know the steps to perform to complete the task. The courseware controller will only provide instructional guidance if they perform an incorrect action. By adding additional courseware data, we can show how this task can be presented to the student in a variety of learning modes.

Show Me mode

In the show me mode, the instructor (or, in our case, the courseware controller) is responsible for performing the task; the student is a passive observer. For this mode, the student needs to have some mechanism for controlling the flow of a lesson; typically, a Next or Continue button is used.

In this mode, rather than monitoring the student's interactions with the simulation user interface, the courseware controller will programmatically make the appropriate changes to the user interface.


NewDemo Lesson: Power up (guided)

Click NEXT to place the starter switch in the WARM UP position.

To begin power up, the starter switch must be placed in the WARM UP position.

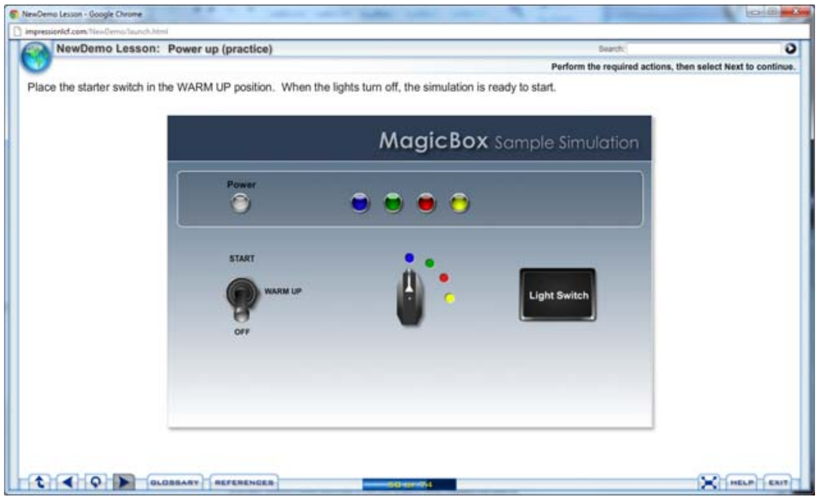
NewDemo Lesson - Google Chrome
impressionlcf.com/NewDemo/launch.html
Search:
MagicBox Sample Simulation
Power
START
WARM UP
OFF
Light Switch
Next
navigate to Power up (guided)
GLOSSARY
REFERENCES
HELP
EXIT

Figure 14: Show me

As with a physical instructor, the courseware controller will walk through the procedure one step at a time, stopping at appropriate intervals to explain the procedure or underlying theory. Since the courseware is self-paced, the student has plenty of opportunity to review the instructional material before proceeding to the next step.

Guided Practice mode

In the guided practice mode, the courseware will provide additional information to the student before each step of the task. The behavior of the courseware controller is identical to its behavior in the standard "practice" mode examples covered in the previous section; the primary difference is that, once the courseware controller begins monitoring the simulation process and user interface for changes on a particular step, guidance information (the initial directions for each step) are shown to the user.


NewDemo Lesson: Power up (practice)

Perform the required actions, then select Next to continue.

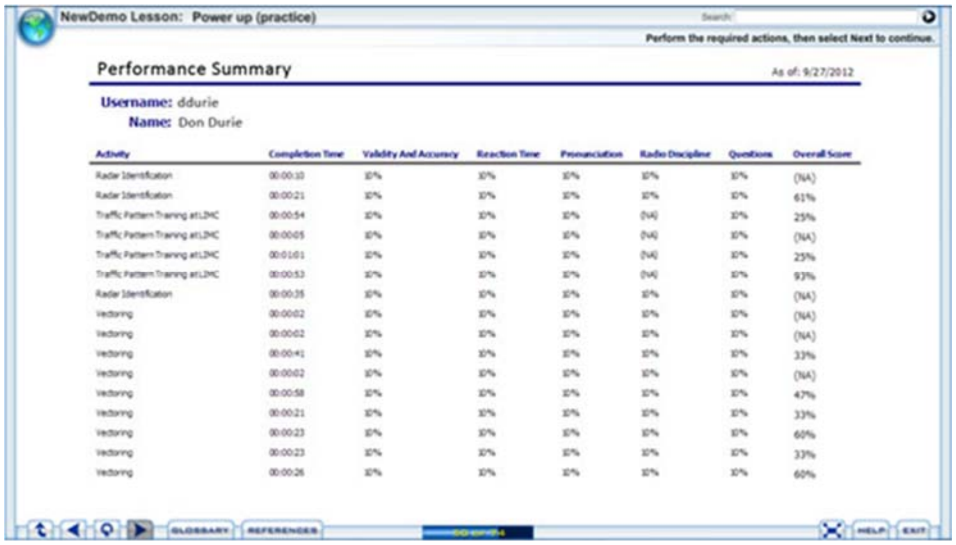
Place the starter switch in the WARM UP position. When the lights turn off, the simulation is ready to start.

NewDemo Lesson - Google Chrome
impressionlcf.com/NewDemo/launch.html
Search:
MagicBox Sample Simulation
Power
START
WARM UP
OFF
Light Switch
GLOSSARY
REFERENCES
HELP
EXIT

Figure 15: Guided practice mode

Evaluation mode

In evaluation mode, the courseware will not provide remediation messages to the student. Instead, incorrect actions will be tracked and used to create a score or pass/fail grade once the task is complete. Scoring information may be presented to the student, sent to a Learning Management System for archiving, or passed to an instructor for evaluation. Note that the courseware controller may use correct/incorrect action specifics to provide a detailed analysis of the student's performance.


NewDemo Lesson: Power up (practice)
Search:
Perform the required actions, then select Next to continue.
GLOSSARY
REFERENCES
HELP
EXIT

Performance Summary

As of: 9/27/2012

Username: ddurie
Name: Don Durie

Activity	Completion Time	Validity And Accuracy	Reaction Time	Pronunciation	Radio Discipline	Questions	Overall Score
Radar Identification	00:00:10	10%	10%	10%	10%	10%	(NA)
Radar Identification	00:00:21	10%	10%	10%	10%	10%	61%
Traffic Pattern Training at L2HC	00:00:54	10%	10%	10%	(NA)	10%	25%
Traffic Pattern Training at L2HC	00:00:05	10%	10%	10%	(NA)	10%	(NA)
Traffic Pattern Training at L2HC	00:01:01	10%	10%	10%	(NA)	10%	25%
Traffic Pattern Training at L2HC	00:00:53	10%	10%	10%	(NA)	10%	93%
Radar Identification	00:00:35	10%	10%	10%	10%	10%	(NA)
Vectoring	00:00:02	10%	10%	10%	10%	10%	(NA)
Vectoring	00:00:02	10%	10%	10%	10%	10%	(NA)
Vectoring	00:00:41	10%	10%	10%	10%	10%	33%
Vectoring	00:00:02	10%	10%	10%	10%	10%	(NA)
Vectoring	00:00:58	10%	10%	10%	10%	10%	47%
Vectoring	00:00:21	10%	10%	10%	10%	10%	33%
Vectoring	00:00:23	10%	10%	10%	10%	10%	60%
Vectoring	00:00:23	10%	10%	10%	10%	10%	33%
Vectoring	00:00:26	10%	10%	10%	10%	10%	60%

Figure 16: Scoresheet

Mixed Modes and Alternate Views

The modes discussed above do not need to be "locked" for a specific lesson; the courseware can switch between modes based on student performance or a student-directed control mechanism.

For example, a student could begin a lesson in standard practice mode. After two incorrect actions, the courseware could decide to switch to guided practice mode, believing that the student could benefit from the additional instruction. Alternatively, a student could manually choose to switch to guided practice after recognizing the need for more detailed instruction.

With a software user interface, simulation-based courseware need not limit itself to display of the virtual hardware. Such alternate views could be used to provide theoretical knowledge. One such example would be display of a "live schematic", showing the changes that each step makes in the systems affected by the task. In a student-directed mode, a live schematic view could include alternate mechanisms for performing actions. Such actions could be in the form of free-floating virtual hardware elements or an abstract selection/activation mechanism.

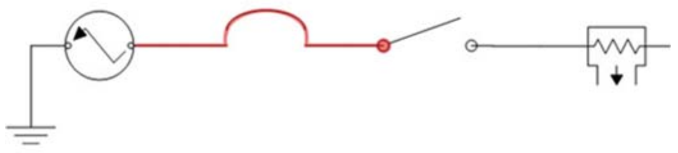


Figure 17: Live schematic

Non-Procedural Training

The earlier examples address procedural training requirements; that is, the need to teach the student a specific sequence of actions to achieve the desired result. Simulation-based courseware can also be used to teach non-procedural behaviors such as troubleshooting. An example of this follows.

We'll extend our simulation model to include a fault for a bad bulb. The new model state data is:

Data Model Element	Possible values
MAIN_POWER	On, off
LIGHT_SWITCH	On, off
LIGHT_BULB	Lit, unlit
LIGHT_BULB_BAD	True, false

The code for the model will not light the bulb unless both main power and the switch are on, and the bulb is not bad.

The extended user interface is:

Interface Element	Data Model Inputs	Data Model Outputs
Circuit breaker	MAIN_POWER	MAIN_POWER
Light switch	LIGHT_SWITCH	LIGHT_SWITCH
Light bulb	LIGHT_BULB	*n/a*
Remove and Replace Bulb button	n/a	LIGHT_BULB_BAD

The remove and replace bulb button will clear the fault.

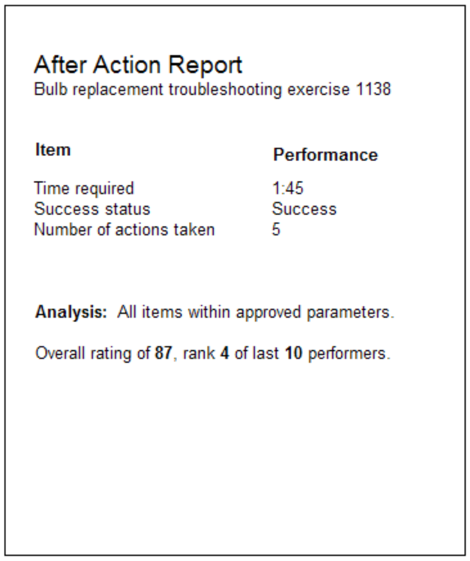
Our courseware data will also be different for this exercise. The courseware data will contain the following:

1. **Initial simulation state**: `MAIN_POWER = off`, `LIGHT_SWITCH = off`, `LIGHT_BULB_BAD = true`.
2. **Allowed actions**: *(all actions are allowed)*
3. **Initial directions**: Display "There is a problem with the light. Fix the problem, and turn on the light."
4. **Required (ending) simulation state**: `LIGHT_BULB = lit`.
5. **Time Limit**: 15 minutes.

The courseware controller will begin the lesson by setting the simulation to its initial state. It will then start its internal timer and monitor both the simulation looking for changes. Since all actions are allowed, the controller may simply ignore user interface change requests, or it may log them and use the data for post-training analysis.

Eventually, one of two outcomes will be reached. One possibility is that the time limit will be reached. When reached, the courseware may fail the student for the task, provide additional time at a penalty, or offer specific information to help the student solve the problem (note that data related to this outcome is not included in the example courseware data).

The other possibility is that, through a series of actions in no determinate order, the student will get the bulb lit and the task will complete. The courseware may take the logged student actions, including completion time data and present an after action report, comparing the student's performance to the optimal solution (again, such data is not shown in this example).



After Action Report

Bulb replacement troubleshooting exercise 1138

Item	Performance
Time required	1:45
Success status	Success
Number of actions taken	5

Analysis: All items within approved parameters.

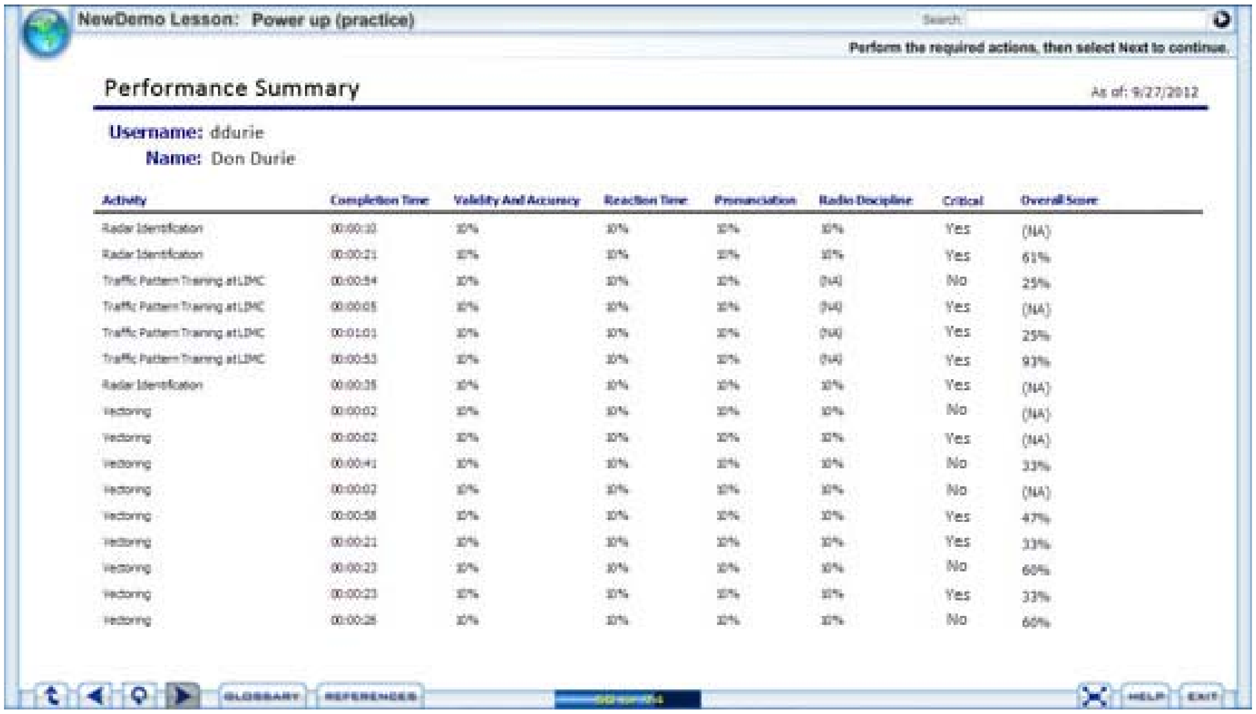
Overall rating of **87**, rank **4** of last **10** performers.

Figure 18: Troubleshooting scoresheet

Assessment and Evaluation

As part of its core design, the courseware controller must evaluate and approve or reject every action the student performs on the simulation user interface. Further, these actions must be evaluated in context; that is, to be approved, the action must be the correct action to perform at the correct moment in time.

All student actions (and the controller's evaluations of those actions) can be stored during performance of the procedure for later review. Scoring and weighting information can be associated with specific correct or incorrect actions to produce a passing or failing grade based on predetermined criteria—a safety issue (*e.g.* forgetting to power off high-voltage equipment before opening) can be weighted so that the student fails the lesson even if all other actions were performed correctly, while several non-critical actions could be omitted while still giving the student a passing grade. Also, as noted in the troubleshooting example above, additional criteria like "time to perform" can also be associated with courseware content.



Perform the required actions, then select Next to continue.

Performance Summary

As of: 9/27/2012

Username: ddurie
Name: Don Durie

Activity	Completion Time	Validity And Accuracy	Reaction Time	Pronunciation	Radio Discipline	Critical	Overall Score
Radar Identification	00:00:10	10%	10%	10%	10%	Yes	(NA)
Radar Identification	00:00:21	10%	10%	10%	10%	Yes	61%
Traffic Pattern Training at LJMC	00:00:54	10%	10%	10%	(NA)	No	25%
Traffic Pattern Training at LJMC	00:00:05	10%	10%	10%	(NA)	Yes	(NA)
Traffic Pattern Training at LJMC	00:01:01	10%	10%	10%	(NA)	Yes	25%
Traffic Pattern Training at LJMC	00:00:53	10%	10%	10%	(NA)	Yes	93%
Radar Identification	00:00:35	10%	10%	10%	10%	Yes	(NA)
Vectoring	00:00:02	10%	10%	10%	10%	No	(NA)
Vectoring	00:00:02	10%	10%	10%	10%	Yes	(NA)
Vectoring	00:00:41	10%	10%	10%	10%	No	33%
Vectoring	00:00:02	10%	10%	10%	10%	No	(NA)
Vectoring	00:00:58	10%	10%	10%	10%	Yes	47%
Vectoring	00:00:21	10%	10%	10%	10%	Yes	33%
Vectoring	00:00:23	10%	10%	10%	10%	No	60%
Vectoring	00:00:23	10%	10%	10%	10%	Yes	33%
Vectoring	00:00:28	10%	10%	10%	10%	No	60%

Figure 19: Assessment results

Simulation-Based Courseware Support in Impression

Now that we've discussed how simulation-based courseware works, we can discuss how to build a virtual (software) simulation environment capable of supporting simulation-based courseware using Impression.

Impression Architecture

Logicdriven's Impression Learning Content Framework is an open-ended system designed to support the development of advanced, highly interactive courseware, including simulation-based courseware. The Impression system's architecture is centered around the concept of separation of learning content from its presentation, which allows courseware materials to be developed for any end product, from self-paced and instructor-led content deployed from the web, delivered on standalone CD- or DVD-ROM, installed on workstations in a schoolhouse, and as printed materials.

The Impression product consists of two main components; the Content Creation Tool (CCT) and the Runtime Engine Developer's Kit (RTE DevKit). The CCT is a rich-client application used by instructional designers and subject matter experts to create learning material. The RTE DevKit is a collection of source code, utilities, and documentation designed to be used by software developers to create the courseware presentation system.

RTE DevKits are available for Adobe Flash Actionscript 3, HTML5/Javascript, and C# Mono-compatible environments (including Unity 3D).

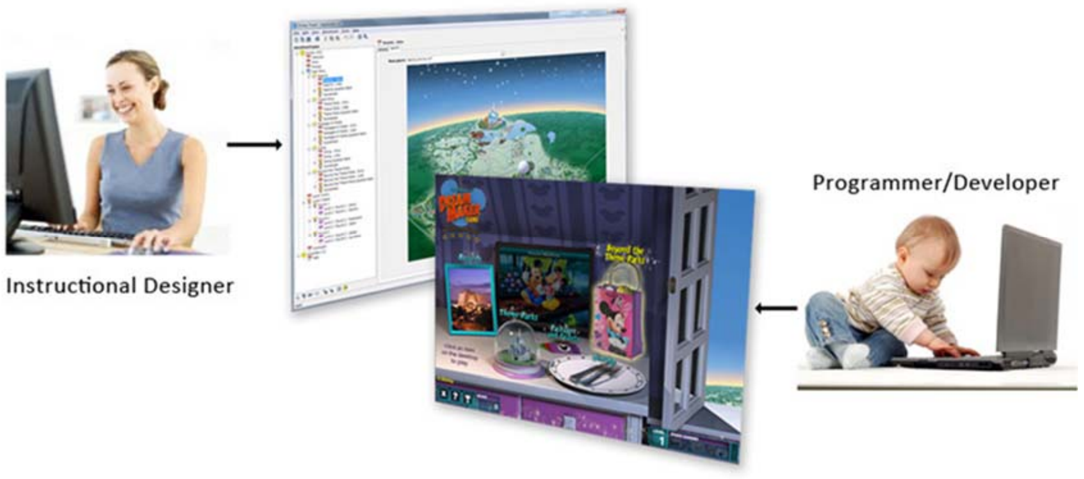


Figure 20: Impression Learning Content Framework

These presentation systems (called *Runtime Engines*, or *RTEs*) are typically designed as a single application that can read and parse courseware data developed with the CCT, present the data and supporting multimedia files, react to learner interactions, and report student progress and achievements to a Learning Management System. Since the CCT-developed content exists independently of the RTE, the single RTE can display any content created for a given project.

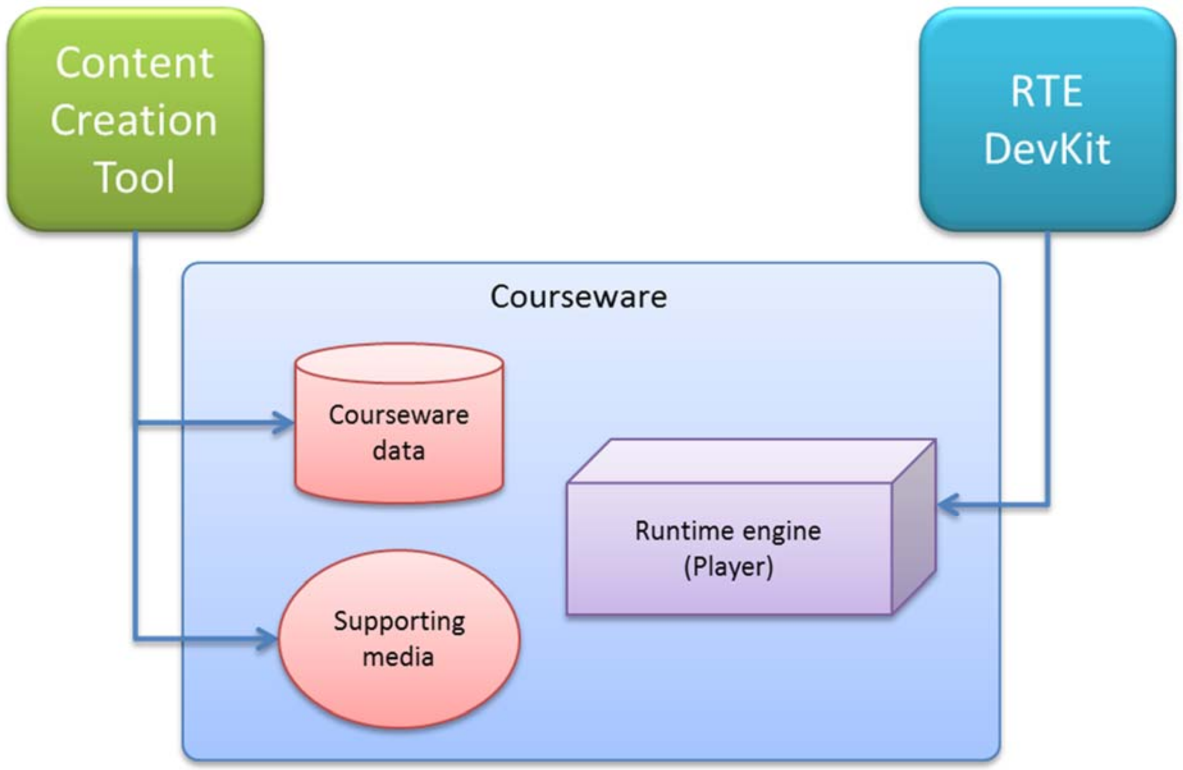


Figure 21: Using Impression to create courseware

Impression includes explicit support for simulation-based courseware. Simulation support includes a collection of programmatic interfaces for listening and controlling simulation processes and simulation user interfaces, managing simulation state snapshots and processing CCT simulation content. The SimulationPlayer class is responsible for directing and controlling explicit instances of these interfaces to provide a robust simulation-based learning environment. Logicdriven and their customers have successfully deployed simulation-based courseware on a number of projects using both Flash and Unity 3D toolchains, and have developed proof-of-concept demos using simulations and simulation user interfaces authored with Javascript and DiSTi's GL Studio.

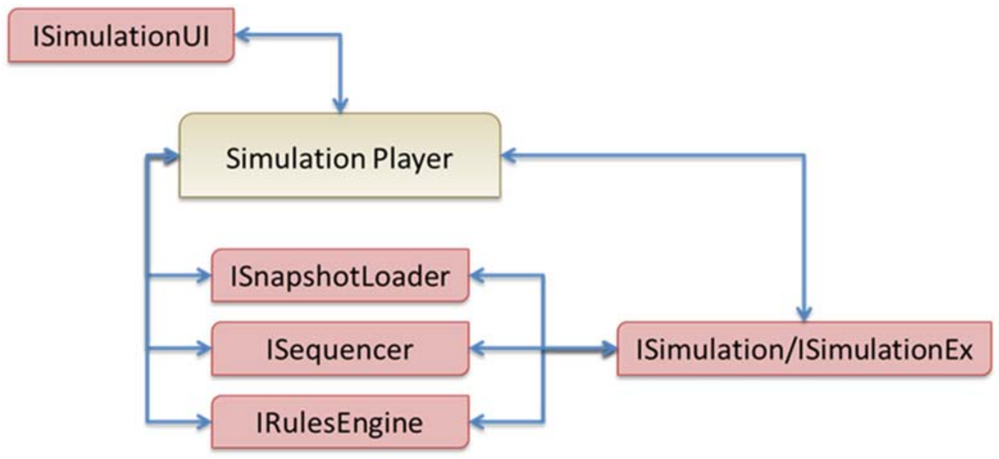


Figure 22: Impression runtime simulation classes

Simulation-Based Courseware Runtime Flow

Let's look at the generalized process that this runtime engine will use to initialize the simulation and courseware, present the instructional material to the student, and advance through the learning material. For this example, we will assume that the courseware material will consist of some introductory material, a multi-step simulation exercise, and a "results" screen that will be shown when the exercise is complete.

Courseware flow begins when the student launches the runtime engine, triggering the initialization process. On initialization, the runtime engine will launch the simulation process, which will begin standard initialization. The runtime will also load and initialize the simulation user interface. Once the process and interface are initialized, the runtime will feed the initial state data (from a snapshot file, a series of explicit data model element assignments, or both) to the user interface subsystem. The courseware data file will also be loaded and parsed. Once initialization is complete, the student will begin the exercise.

> *The actual exercise flow and control will depend on the courseware material. Refer to the previous section to review how the runtime may advance the student through the lesson.*

After completion of the exercise, the runtime hides the simulation interface and presents the results screen, using the data gathered during performance of the exercise. After review, the student closes the runtime engine, which shuts down the simulation process as part of its termination code.

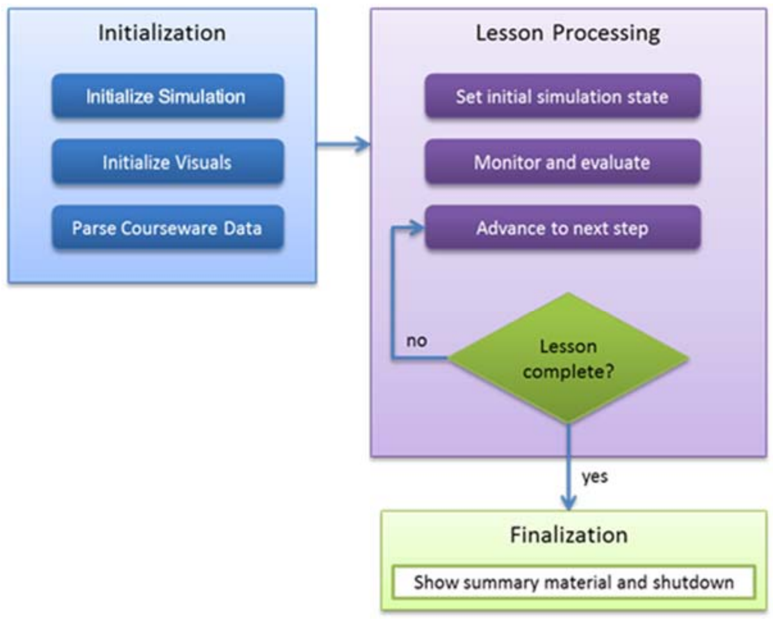


Figure 23: Courseware flow

Authoring Simulation-Based Courseware

So far, we have discussed the data used to drive simulation-based courseware in somewhat general terms; this section will cover the authoring process in more detail.

Creating Courseware Data

When a content creator (Instructional Designer or Subject Matter Expert) begins to author simulation-based courseware, they must consider several items, specifically:

- *What is the end goal?* That is, how will the simulation be used to teach the learning objectives? What learning mode (show me, guided practice, *etc.*) will be used? What non-simulation content should be added, and where?
- *How should the procedure be broken down into discrete learning steps?* Simulation-based courseware offers a lot of flexibility to the content creator; as a result, there are often several ways to guide a student's behavior through a procedure.
- *What data should be provided for each step?* As explained below, robust simulation-based courseware content is rarely as simple as "flip the switch, and then turn the dial."

It's worth noting that all of these decisions are affected by the programmed capabilities of the courseware runtime player and the fidelity of the simulation.

Simulation Step Data

Each step of a simulation-based procedure can consist of many different kinds of data, including:

Initial simulation state directives

Setting the appropriate initial state may involve loading a complete or partial snapshot of the simulation state from an external file; setting one or more simulation state values explicitly, or using a "targeted reset" command (essentially, a hardcoded snapshot embedded within the simulation). Subsequent steps of a multi-step procedure may not include any initial simulation state directives at all; in such cases, the simulation is assumed to have been placed in the proper state via a combination of state directives and user actions from previous steps.

Final simulation state values

The content creator needs to specify the simulation state to be achieved for successful completion of the step. This may include student-controlled user interface values (*e.g.* a switch must be in the correct position), non-interactive user interface values (the light must be on), or non-visual simulation state values (main power must be on).

Allowed student actions

The content creator may need to define which visual components the student may interact with, and what values the student may set. In some cases, this may not be necessary—for example, if the final simulation state calls for a toggle switch to be placed in the `on` position, the runtime may recognize that the student is moving the switch to the `on` position, and will automatically approve the request. More complex controls may require that the content creator explicitly allow intermediate states; moving a car's ignition switch from `off` to `start` requires first moving the switch to the `on` position. Moving the switch from `off` to `on` is a necessary, but not sufficient, action for a "start the car" step. In a less-restricted procedure, such as a troubleshooting lesson, all actions may be allowed.

Alternate student actions

The goal of simulation-based courseware is to teach the student how to interact with the hardware being simulated, not to teach the student how to master the virtual interface. As a result, the content developer may choose to override the default user interface behavior to reduce the level of effort required for the student to master the task.

Consider the example of a twin engine plane's throttle quadrant. In the throttle quadrant, there are two throttles, one for each engine. Suppose that the visuals were designed so that the student may click on the left or right throttle and drag the individual throttle up and down; if they click and drag in the space between the throttles, both throttles move together. In a procedure that requires both throttles to be moved (and is not teaching the behavior of the throttle quadrant), a content creator may choose to specify that both throttles be moved if only one is moved.

This provides the student with a larger "hit area". Students can click and drag on the left throttle, right throttle, or the space between the two; allowing the student to maintain focus on learning the procedure rather than concerning themselves with their mastery of the input device.

> *Another way to do this is to allow the content creator to specify the behavior of the user interface elements—in this example, to specify that the throttles should always be ganged together for this step. Whether this is accomplished by instructing the simulation process to treat individual throttle changes as ganged throttle changes or by specifying the behavior of the visual component is not important; in either case (as in the case of the runtime directive above), the content creator must define the control behavior for the step.*

Remediation and Feedback

The content creator should provide feedback to—at a minimum—let the student know they have performed an incorrect action. Without such feedback, the student will not know why their actions were not reflected by the simulation state; they may believe that a problem exists with the computer or with the simulation software.

Remediation and feedback may be generalized; for example, "That action is incorrect". Generalized remediation may include hints as to the correct action ("Incorrect. Click on the Master Caution switch to silence the alarm", or refer back to the procedure ("Incorrect. Wait for the glow plugs to fire before starting the engine "). Remediation may also be associated with a specific action ("Incorrect. That is the left throttle, advance the right throttle instead").

Remediation and feedback may include text and graphics; or the context in which remediation is displayed may change based on the specific action attempted. For example, an incorrect action that violates a safety procedure may be displayed in a popup window with a striped red border and the word "DANGER" shown in large bold print before the message.

The content creator will add as much or as little feedback as the procedure and project design guidelines call for. Adding effective feedback requires a feel for the material as well as an understanding of the common mistakes a student may make; it is impractical to define specific remediation for every possible action in a non-trivial simulation.

Scoring and Weighting

For certain types of assessment content, the content creator will need to specify scoring values for the step. This can include identifying a base value for completing the step without errors; time bonuses, and penalties for general or specific incorrect actions.

Non-Simulation Actions

In some cases, a procedure may require actions not modeled by the simulation. For example, the starting procedure for a single-engine propeller-driven airplane may require the pilot to stick his head out of the cockpit and yell "Prop Clear!" Content creators may need to define these non-simulation actions.

Instructional Guidance

Depending on the training mode, content creators may need to include instructional guidance or explanatory material to assist the student in following the procedure. When alternate displays are available, explanatory material may differ based on the active display (*e.g.*, theoretical systems instruction would be shown if the display is a "live schematic", while procedural instruction would be shown if the simulated equipment is onscreen).

Contextual/Ordered Data

Even within a step, any of the data elements described above may be valid only based on simulation state or previous actions. For example, one remediation message may be shown for a specific incorrect action when an engine is cold; as the engine is warming up, the same incorrect action may need a different remediation message. Changes to simulation state within a step may be ordered, with some actions allowed only after the simulation has reached a specific state.

Defining Steps

Most procedures will consist of a number of discrete steps; however, the steps defined in the controlling documentation may not be the most "natural" way to create a learning sequence.

Take the example of starting a car's engine. The technical documentation may say something like "turn the ignition key to the start position, hold it there until the engine starts, and then release the key". This single procedural step can be defined in extensive detail over several steps, as in:

- Step 1: Turn the ignition key to the `start` position.
- Step 2: Wait for the engine to start.
- Step 3: Release the key (to the `on` position).

It can also be defined in a single step, as in:

- Step 1: Wait for the engine to be started and the ignition key to be in the `on` position.

Either choice is valid—which one is used is up to the discretion of the content creator.

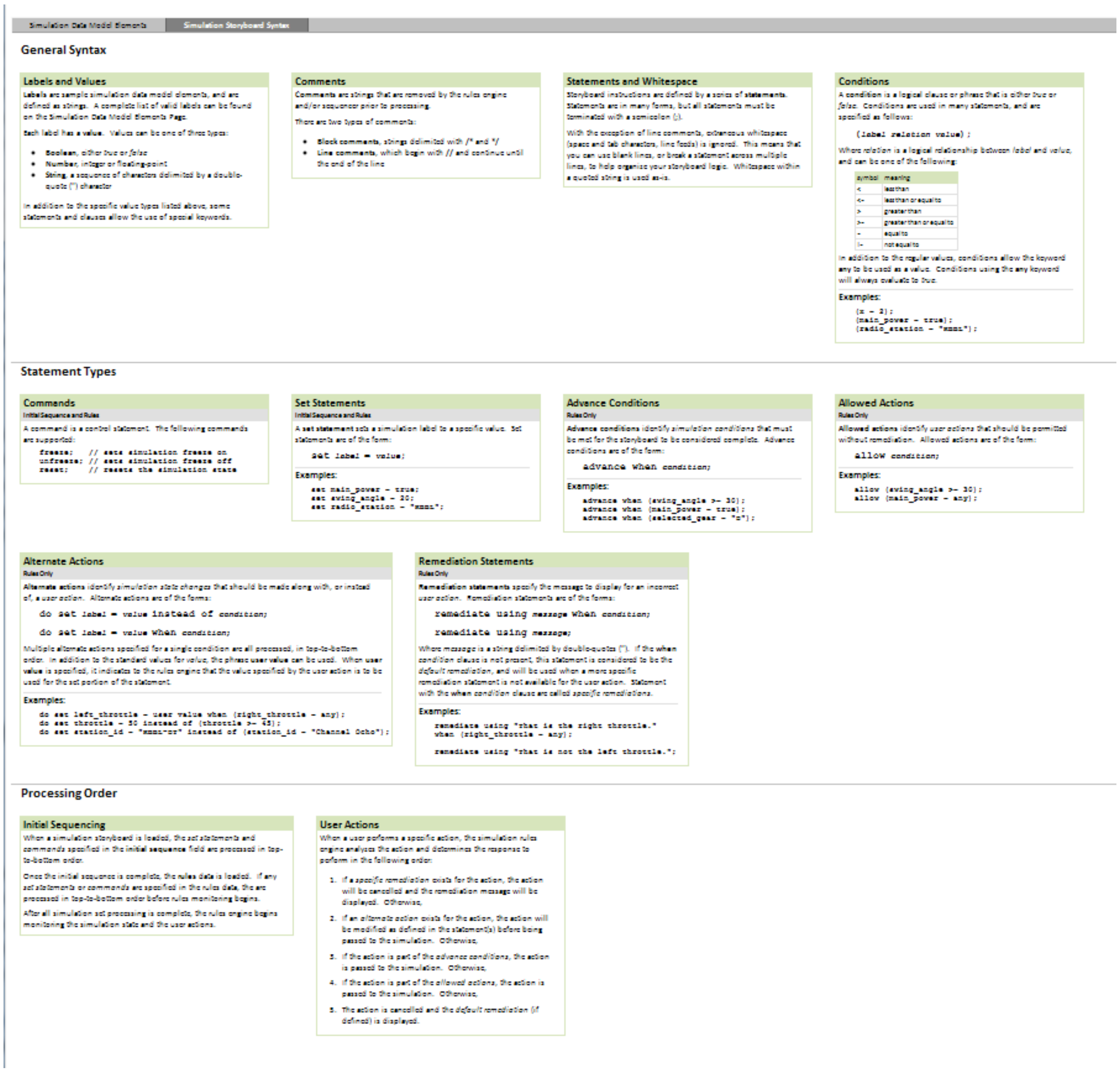
Determining How the Simulation Should be Used

As discussed earlier in this document, simulations can be used in courseware in one of several different modes: show me, practice/guided practice, and evaluation. Alternate displays of simulation state can be created and used to pass on theoretical knowledge in addition to procedural proficiency. The best use of the simulation toolset to meet the instructional objectives is a decision that ultimately rests in the hands of the content creator; but it should be noted that the more capabilities that are provided by the simulation and runtime environment, the more likely it is that an instructional designer or subject matter expert can develop an optimal solution.

Authoring Simulation-Based Courseware with Impression

Logicdriven's Impression Learning Content Framework includes support for creating simulation-based content using text statements. Each step (or *storyboard*, in Impression lingo) allows the content creator to define initial simulation state (including snapshot files), required or final simulation state, allowed actions, remediation and feedback, scoring, non-simulation actions, alternate displays, and instructional guidance. Storyboards are sequenced by the content creator, and the runtime environment can automatically advance to the next storyboard when the simulation is in the desired state, or it can require the student to perform a specific action (as when a "show me" mode is active and the student is a passive observer).

For those Runtime Engine samples that support the MagicBox sample simulation, we include a simple declarative language parser that is embedded within a component that implements the framework's IRulesEngine interface. This component is used by the runtime engine to examine and evaluate simulation state and user interface change requests, and to determine the appropriate response; whether to display additional feedback or remediation, switch learning modes, cancel the change request, or modify the simulation state.



General Syntax

Labels and Values

Labels are sample simulation data model elements, and are defined as strings. A complete list of valid labels can be found on the Simulation Data Model Elements Page.

Each label has a **value**. Values can be one of three types:

- **Boolean**, either *true* or *false*
- **Number**, integer or floating-point
- **String**, a sequence of characters delimited by a double-quote (") character

In addition to the specific value types listed above, some statements and clauses allow the use of special keywords.

Comments

Comments are strings that are removed by the rules engine and/or sequencer prior to processing.

There are two types of comments:

- **Block comments**, strings delimited with /* and */
- **Line comments**, which begin with // and continue until the end of the line

Statements and Whitespace

Storyboard instructions are defined by a series of **statements**. Statements are in many forms, but all statements must be terminated with a semicolon (;).

With the exception of line comments, extraneous whitespace (space and tab characters, line feeds) is ignored. This means that you can use blank lines, or break a statement across multiple lines, to help organise your storyboard logic. Whitespace within a quoted string is used as-is.

Conditions

A condition is a logical clause or phrase that is either *true* or *false*. Conditions are used in many statements, and are specified as follows:

```
(label relation value);
```

Where *relation* is a logical relationship between *label* and *value*, and can be one of the following:

symbol	meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
!=	not equal to

In addition to the regular values, conditions allow the keyword **any** to be used as a value. Conditions using the any keyword will always evaluate to *true*.

Examples:

```
(x = 2);
(main_power = true);
(radio_station = "xxxx");
```

Statement Types

Commands
Initial Sequence and Rules

A command is a control statement. The following commands are supported:

```
freeze;   // sets simulation freeze on
unfreeze; // sets simulation freeze off
reset;    // resets the simulation state
```

Set Statements
Initial Sequence and Rules

A **set statement** sets a simulation label to a specific value. Set statements are of the form:

```
set label = value;
```

Examples:

```
set main_power = true;
set swing_angle = 20;
set radio_station = "xxxx";
```

Advance Conditions
Rules Only

Advance conditions identify *simulation conditions* that must be met for the storyboard to be considered complete. Advance conditions are of the form:

```
advance when condition;
```

Examples:

```
advance when (swing_angle >= 30);
advance when (main_power = true);
advance when (selected_gear = "c");
```

Allowed Actions
Rules Only

Allowed actions identify *user actions* that should be permitted without remediation. Allowed actions are of the form:

```
allow condition;
```

Examples:

```
allow (swing_angle >= 30);
allow (main_power = any);
```

Alternate Actions
Rules Only

Alternate actions identify *simulation state changes* that should be made along with, or instead of, a user action. Alternate actions are of the forms:

```
do set label = value instead of condition;

do set label = value when condition;
```

Multiple alternate actions specified for a single condition are all processed, in top-to-bottom order. In addition to the standard values for *value*, the phrase **user value** can be used. When **user value** is specified, it indicates to the rules engine that the value specified by the user action is to be used for the set portion of the statement.

Examples:

```
do set left_throttle = user value when (right_throttle = any);
do set throttle = 50 instead of (throttle >= 45);
do set station_id = "xxxx-cr" instead of (station_id = "Channel Ocho");
```

Remediation Statements
Rules Only

Remediation statements specify the message to display for an incorrect *user action*. Remediation statements are of the forms:

```
remediate using message when condition;

remediate using message;
```

Where message is a string delimited by double-quotes ("). If the **when** *condition* clause is not present, this statement is considered to be the *default remediation*, and will be used when a more specific remediation statement is not available for the user action. Statement with the **when** *condition* clause are called *specific remediations*.

Examples:

```
remediate using "that is the right throttle."
when (right_throttle = any);

remediate using "that is not the left throttle.";
```

Processing Order

Initial Sequencing

When a simulation storyboard is loaded, the *set statements* and *commands* specified in the **initial sequence** field are processed in top-to-bottom order.

Once the initial sequence is complete, the **rules** data is loaded. If any *set statements* or *commands* are specified in the rules data, the are processed in top-to-bottom order before rules monitoring begins.

After all simulation set processing is complete, the rules engine begins monitoring the simulation state and the user actions.

User Actions

When a user performs a specific action, the simulation rules engine analyses the action and determines the response to perform in the following order:

1. If a *specific remediation* exists for the action, the action will be cancelled and the remediation message will be displayed. Otherwise,
2. If an *alternate action* exists for the action, the action will be modified as defined in the statement(s) before being passed to the simulation. Otherwise,
3. If the action is part of the *advance conditions*, the action is passed to the simulation. Otherwise,
4. If the action is part of the *allowed actions*, the action is passed to the simulation. Otherwise,
5. The action is cancelled and the *default remediation* (if defined) is displayed.

Figure 24: Declarative Language Syntax

Although the declarative language is easy to learn and provides quite a bit of functionality, it may or may not meet the needs of any given project. The language can be extended or replaced within the runtime environment; this is as simple as writing a new component that implements the IRulesEngine interface. The Content Creation Tool is language-agnostic; it only provides a way for the content creator to enter the information; it does not attempt to determine if the information is syntactically valid.

Improving the Authoring Experience

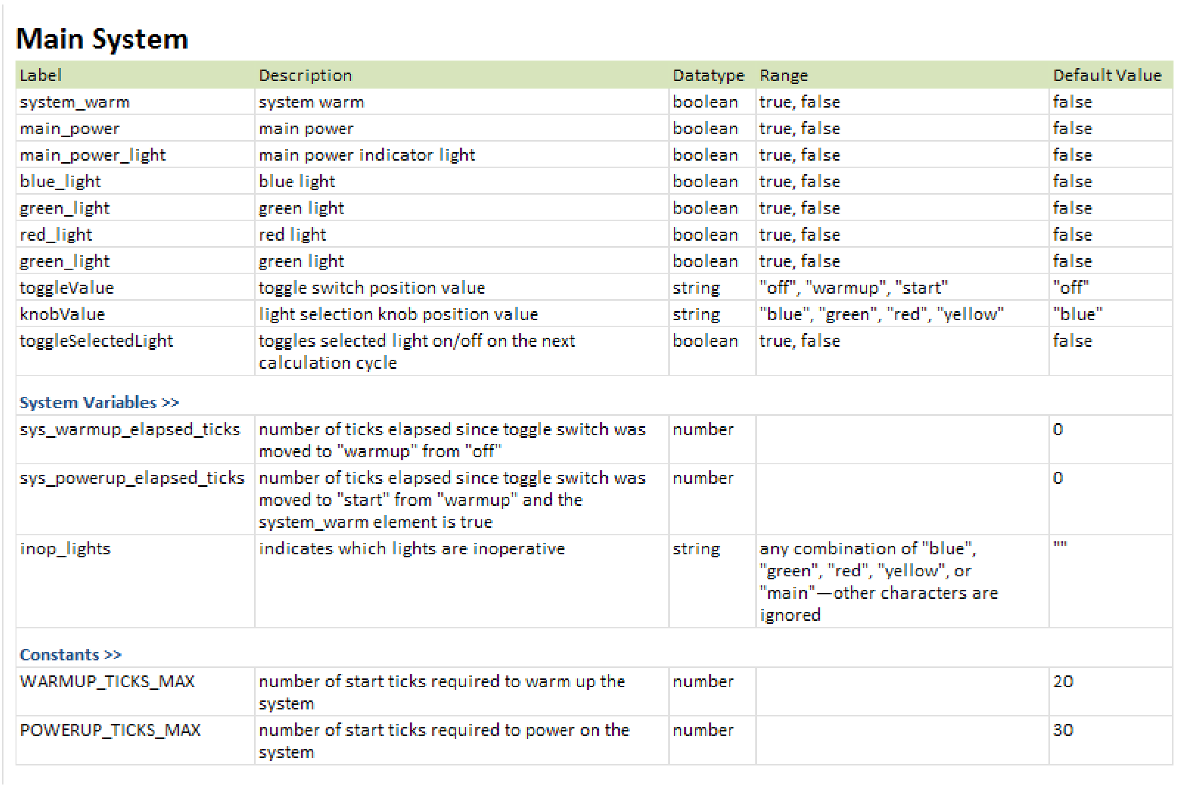
Even an easy-to-learn declarative language can impose a burden on content creators; they must not only understand the language, but be able to enter (and look up) specific simulation state data elements and their corresponding values. There are two major areas where authoring efforts can be reduced.

Simulation System Modeling

In many cases, simulation processes used with hardware trainers do not model the actual systems being simulated; rather, they use abstract logical models to accurately reproduce discrete output values given a specific set of inputs.

This approach can work well for hardware devices; in such cases, the actual simulation process itself is a black box and is not accessible to end users of the device. The only personnel concerned with the simulation process are the engineers and technicians responsible for building and testing the hardware. And their primary concern is the accuracy of inputs and outputs, not the realism of the underlying logic.

In contrast, developers of simulation-based courseware *are* concerned with the realism of the simulation process; having to translate a logical model to actual system state values imposes a significant burden on content creators, who themselves may or may not have a high degree of technical proficiency. Additionally, having a complete a model available provides content creators with more ways to "naturally" create simulation-based courseware, and improves the end result.



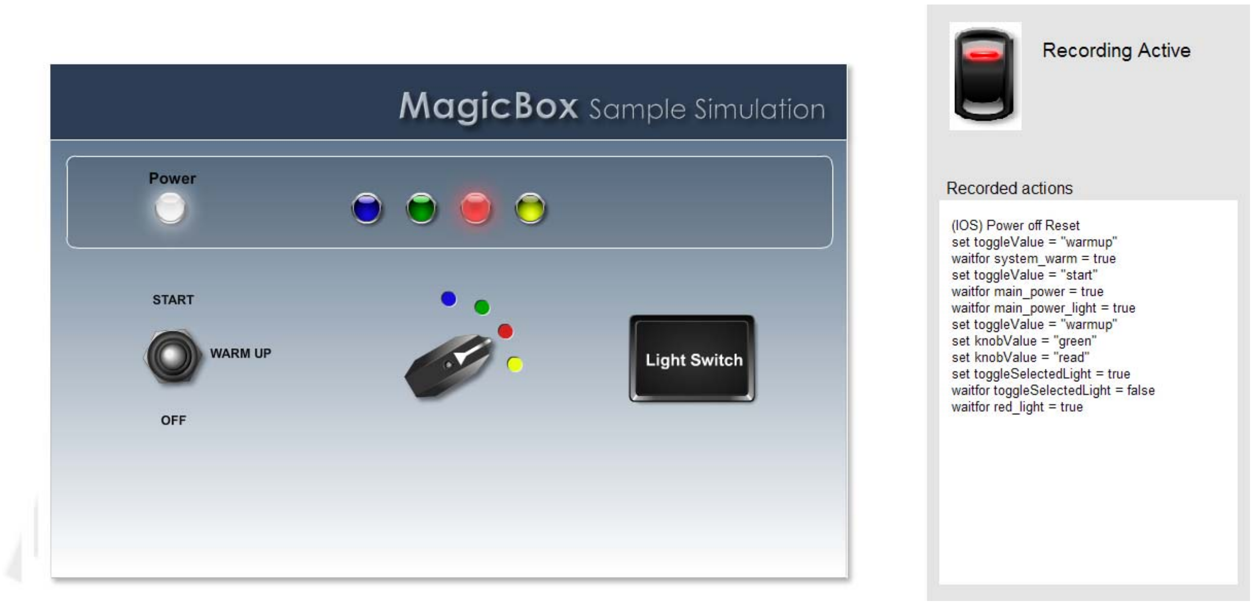
Main System

Label	Description	Datatype	Range	Default Value
system_warm	system warm	boolean	true, false	false
main_power	main power	boolean	true, false	false
main_power_light	main power indicator light	boolean	true, false	false
blue_light	blue light	boolean	true, false	false
green_light	green light	boolean	true, false	false
red_light	red light	boolean	true, false	false
green_light	green light	boolean	true, false	false
toggleValue	toggle switch position value	string	"off", "warmup", "start"	"off"
knobValue	light selection knob position value	string	"blue", "green", "red", "yellow"	"blue"
toggleSelectedLight	toggles selected light on/off on the next calculation cycle	boolean	true, false	false
System Variables >>				
sys_warmup_elapsed_ticks	number of ticks elapsed since toggle switch was moved to "warmup" from "off"	number		0
sys_powerup_elapsed_ticks	number of ticks elapsed since toggle switch was moved to "start" from "warmup" and the system_warm element is true	number		0
inop_lights	indicates which lights are inoperative	string	any combination of "blue", "green", "red", "yellow", or "main"—other characters are ignored	""
Constants >>				
WARMUP_TICKS_MAX	number of start ticks required to warm up the system	number		20
POWERUP_TICKS_MAX	number of start ticks required to power on the system	number		30

Figure 25: Trivial simulation data model elements

Authoring via Simulation Recording

One way to reduce authoring efforts is to provide a virtual authoring simulation environment that allows content creators to initialize the simulation and walk through a procedure while recording their actions. Once recorded, these actions can be saved to a lesson for further adjustment and form the basis of a simulation-based lesson. Using the Impression Connect extensibility architecture, such an environment can easily be created using the simulation process executable, the simulation visuals, and the Impression runtime interfaces for managing simulations.



```
(IOS) Power off Reset
set toggleValue = "warmup"
waitfor system_warm = true
set toggleValue = "start"
waitfor main_power = true
waitfor main_power_light = true
set toggleValue = "warmup"
set knobValue = "green"
set knobValue = "read"
set toggleSelectedLight = true
waitfor toggleSelectedLight = false
waitfor red_light = true
```

Figure 26: Example authoring recording interface

In many cases, additional information will need to be added by the content creator after the recording has been saved; items like remediation and feedback, scoring and response weighting, alternate and intermediate actions, and instructional guidance can be entered during recording, but such a "stop/start" approach to procedure recording can be non-optimal, depending on the system, the procedure, and the individual content creator. Additionally, content creators may wish to insert non-simulation content (for example, a multiple-choice check on learning) in the middle of a simulation sequence.

Logicdriven's experience with semi-automated simulation content development indicates lesson "tuning" is often required after recording. Because of this and the additional content requirements noted above, the results of recording should be stored in a human-editable form (for example, as a series of declarative language statements as described earlier).

Although the need for text-based editing remains, the more capabilities that can be added to the virtual environment, the more power and flexibility can be put into the hands of the content developers, allowing them to choose an authoring style most suited to their individual creation processes.