CHAPTER 9

IDENTIFYING OBSERVABLE OUTCOMES IN GAME-BASED ASSESSMENTS

Russell Almond
Florida State University

Valerie J. Shute
Florida State University

Seyfullah Tingir
Florida State University

Seyedahmad Rahimi
Florida State University

Simulation and game-based assessments generate lots of data; however, those data only become useful if they can be turned into evidence by connecting them to aspects of proficiency that are of interest to an educator. The fundamental principle is simple: A behavior which is likely to occur at high proficiency states and unlikely to occur at low proficiency states is

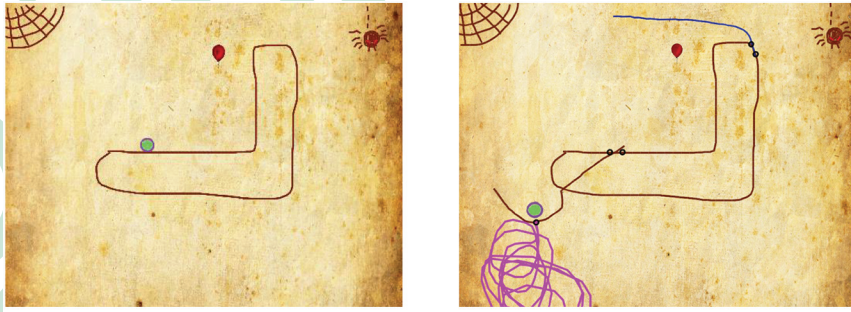
Innovative Psychometric Modeling and Methods, pages 163–192

good *evidence* for that proficiency state. Simulation and game-based assessment cause students to exhibit a large stream of behaviors, most of which have low evidentiary value. The challenge is one of *evidence identification*: isolating and recording the behaviors which have high evidentiary value.

As a running example for this paper, we will use the game *Physics Playground* (Shute & Ventura, 2013). As the name implies, the goal of the game is to teach physics. In the game the players interact with a 2-dimensional world which follows the laws of physics. Their goal in each case is to get the ball to a target, represented by a balloon. Figure 9.1a shows a typical level from Version 1 of the game. Figure 9.1b shows a possible solution. Here the player has created a springboard with a weight attached. Detaching the weight will cause the ball to fly up in the air and strike the balloon.

Although the goal of the game-based assessment is to infer knowledge about qualitative physics (Ploetzner & VanLehn, 1997), what the game provides is a stream of events: mouse clicks and button presses. The game engine converts these low-level events to higher level events: for example, shapes drawn on the screen, movements of the ball and the ball striking the balloon. A recent field trial involving around 300 players playing for approximately 5 hours each generated over 4.5 million logged events. From this stream of high-level events it is difficult to determine which sequences of events correspond to an understanding of physics. However, if inferences can be drawn about the players' ability relative to the dimensions of interest, those inferences could be used to guide the selection of levels in the game or learning supports to the players.

There are a number of approaches to processing these game logs. One approach is to ignore all but the highest-level events. In *Physics Playground*, the game awards a gold or silver coin for a successful solution of a level (gold for a solution that is both successful and efficient). Using only the



(a) Initial position for level Spider Web. (b) Solution for Spider Web.

Figure 9.1 Example level and solution from *Physics Playground* Version 1. *Player Goal: Get the ball to the balloon by drawing ramps, levers, springboards, and pendulums.*

coin events to make inferences about proficiency is easy to implement, but potentially leaves a lot of valuable evidence unevaluated.

A second class of approaches involves using machine learning techniques to extract relevant evidence from the game logs. Machine learning techniques come in supervised and unsupervised flavors. Unsupervised techniques merely require large data sets; however, like exploratory factor analysis, they produce unlabeled patterns of events that are seen in many logs. Sometimes these may make sense from a cognitive perspective and other times they may be artifacts of the particular sample used for training. The construct validity argument for inferences from these unsupervised models is weak: It relies on correlations between the presence of a particular pattern and the construct being assessed. Supervised machine learning techniques require a human to label patterns of interest, and the algorithm attempts to find ways of recognizing those patterns. The labeling process is almost always expensive.

This chapter takes a third approach: the use of rules of evidence to parse the event stream and set the values of *observable* variables. The terms *rules of evidence* and *observables* come from evidence-centered assessment design (ECD; Mislevy, Steinberg, & Almond, 2003). HyDRIVE was a simulation-based assessment that was one of the exemplars used in the formulation of ECD (Mislevy & Gitomer, 1996). The rules of evidence for HyDRIVE were written in the logic-based programming language, Prolog, so they were literally if–then style rules. These rules were used to set the value of observable variables in a Bayesian network which was then used to draw inferences about the student competencies. This is the same architecture we are implementing for Version 2 of *Physics Playground*. These if–then style rules can be written by the physics experts and students in instructional design who are working on the game levels and learning supports; team members with minimal exposure to formal computer programming.

This chapter looks at the construction of a scoring model for an updated version (Version 2) of *Physics Playground*. Section 2 presents a simplified version of ECD, focusing on the relationship between the elements of the design and the requirements for the scoring software. Section 3 describes the process used to establish the design elements for Version 2 of *Physics Playground*. Section 4 introduces a language for describing rules of evidence. Section 5 summarizes observations on how these algorithms performed in a recent moderately sized field test.

FOUR ELEMENTS OF ASSESSMENT DESIGN

The ECD process (Mislevy et al., 2003) centers assessment design around a construct validity argument for the assessment by forcing the designers to

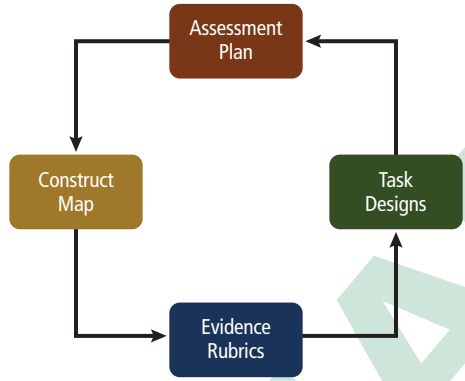


Figure 9.2 Four elements of the conceptual assessment framework.

explicitly state how observable outcomes from each task (or item) produces evidence for the construct of interest. This argument is built first at a qualitative level in the domain model, and then in a more quantitative way in the *conceptual assessment framework* (CAF; for details see Chapters 2 and 12 of Almond, Mislevy, Steinberg, Yan, & Williamson, 2015).

The CAF contains four elements as shown in Figure 9.2. The arrows are drawn in a counterclockwise direction, as the reasoning for design often travels in this direction. It is shown as a circle because this is an iterative process.

The four elements, going counterclockwise from the left, are:

1. The *Construct map* defines the space of possible patterns of skills that a student (in the target audience) might possess. (This was called the proficiency or competency model in older ECD papers.) The word "skill" is used for one dimension within the construct map with the understanding that the construct to be measured might not normally be described as a skill (e.g., engagement might be a target "skill").
2. The *evidence* describes *observable outcome* variables that could provide evidence for the skills of interest. (This was called the evidence model in older ECD papers.) The evidence is linked with a statistical model to the skill variables in the construct map, and with *rules of evidence*, or a scoring rubric, to the outcome space of a particular task.
3. *Task Designs* provide the description of a class of environment in which the evidence can be observed. (These were referred to as task models in older ECD papers.) Note that each design corresponds to a collection of different tasks which can be manipulated by changing features of the task (Kim, Almond, & Shute, 2016).

4. The *assessment plan* is a set of rules for constructing a complete form of the assessment. (This was called the assembly model in older ECD papers.) Note that in an adaptive assessment, this is described by a series of rules: Rules describing how the collection of tasks must span the construct and rules for determining when sufficient evidence has been gathered for the purpose of the assessment.

Two of the original six ECD models were left out of Figure 9.2, the *presentation model* and the *delivery system model*, but the design considerations they represent remain important. The presentation model describes the capabilities of the simulator or game engine. In *Physics Playground 1*, the ball was always the same mass. That made it difficult to assess the students understanding of the relationship between force, mass, and acceleration (Newton's Second Law). So for Version 2, the capability of changing the mass of the ball was added. The delivery system model describes among other things, the population targeted by the assessment. In our case as the target population is usually middle and high school students, so they had not been exposed to vectors in their mathematics classes. This was an important design consideration for the assessment.

Four Processes of Assessment Delivery

A completed conceptual assessment framework should provide complete specifications for building the assessment. The four process architecture for assessment delivery (Proc4; Almond, Steinberg, & Mislevy, 2002) provides an idealized model of the processes (human or computer) necessary to implement the assessment. The four processes can be placed in between the four elements as shown in Figure 9.3.

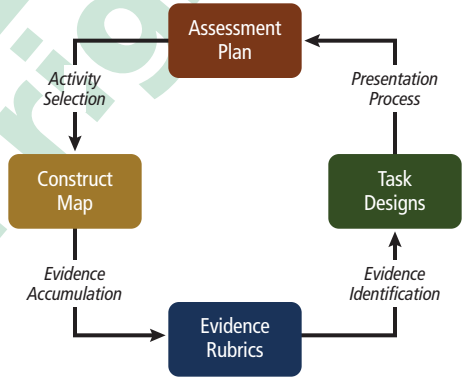


Figure 9.3 Four processes for assessment delivery.

Starting from the upper left, the *activity selection* process (AS) consults the assessment plan to determine the first task to present to the student. Then the *presentation process* deliveres that task and logs the student interactions. (Note that this could either be an assessment task, designed to gain knowledge about student skills as well as allowing the student to exercise the target skills, or a learning support task designed to more passively convey knowledge. In either case, the system logs the interaction, even if that merely consists of pressing "OK.") Next, the *evidence identification* process (EI) applies the *rules of evidence* appropriate to the task context to the stream of events. The EI it sets the values of key observable variables and passes them along. The *evidence accumulation* process (EA) takes the stream of observables from the EI process and attempts to locate the student on the construct map. Control then returns back to the AS process, which chooses the next level. In an adaptive assessment, it chooses the level based on the current estimate of the students skills; in a linear assessment, the tasks are presented in a predetermined order.

There are any number of ways that these processes can be implemented. For *Physics Playground 2* the following implementation strategies were used:

- *Presentation Process*—The game engine was written in Unity 2D® (Unity Technologies, 2019) and implemented as a client–server system where the game levels were displayed in the player's web browser. The game would send logging information back to the Unity server, where the events would be logged in a Learning Locker® ("Learning Locker Documentation," 2018) record store. In addition to presenting the game level, the presentation process presented learning supports often by streaming videos from YouTube® .
- *Evidence Identification*—The EIEvent package (Almond, 2019a) ran on a separate scoring server. It periodically downloaded events from the Learning Locker record score from the Unity server, filtered those events, and then ran the *rules of evidence* (described in Section 4 to determine the value of the *observables*. The observables were logged in a database collection which served as the input queue to the EA process.
- *Evidence Accumulation*—We employed a Bayes net based system that followed the algorithm described in Chapter 13 of Almond et al. (2015). It fetches the observables from the queue in the database, and uses this information to update the *student model*—the player specific copy of the construct map which locates that particular player in the skill space. When the updating was complete, the EA process would save statistics summarizing the player's location in the skill space to the database. This could be queried by both the game

AU: Where does the closed parentheses go?

engine (for displaying student progress on the scoreboard) and for the activity selection process. The code ran on the same server as the evidence identification code, and used the Peanut (Almond, 2019b) package and the Netica® Bayes net engine (Norsys, 2012).

- *Activity Selection*—The original plan was to use the expected weight of evidence algorithm used in ACED (Shute, Hansen, & Almond, 2008) and described in Chapter 7 of (Almond et al., 2015). Because of time constraints, a simpler algorithm was used and implemented inside the game engine. The levels were assigned to topics corresponding to skills in the construct map. If the corresponding statistic was high then the system would deliver difficult levels within the topic, and if it was low, then easier levels were delivered. If the corresponding variable became sufficiently high then the player would graduate from that topic and move to the next. When a player graduated from all topics, they would go into an endgame where the AS process would select unplayed levels in an arbitrary fashion.

The four processes are not meant to be a rigid partitioning of the work necessary for the assessment, but rather a guideline for organizing the work. *Physics Playground* required code for identifying whether an object a player had drawn on the screen was a ramp, springboard, lever, pendulum, or something else. Although this could be regarded as an EI task, it was easier to accomplish within the game engine as it required detailed state information about the position and motion of the object in the two-dimensional world. Similarly, it proved more efficient to put the entire activity selection process inside the game (the presentation process) than create a separate process.

Proc4 Messages

The four process (Proc4) architecture assumes that the processes communicate with each other through a series of messages. In the implementations we have worked with, these messages are divided into a header and a data portion. The header has a series of standard fields which are used to route and prioritize the message. The data portion can be anything, and in general its value will be determined by the message. Listing 9.1 shows a prototypical message in JSON (java script object notation; Bassett, 2015) format.

LISTING 9.1: A typical Proc4 message in JSON format

```
{
app: "ecd://epls.coe.fsu.edu/PP",
uid: "Student 1",
context: "SpiderWeb",
sender: "Evidence Identification",
mess: "Task Observables",
timestamp: "2018-10-22 18:30:43 EDT",
processed: false,
data:{
  trophy: "gold",
  solved: true,
  objects: 10,
  agents: ["ramp","ramp","springboard"],
  solutionTime: {time:62.25, units:"secs"}
  }
}
```

The `data` field is the body of the message. The other fields are headers. They have the following meaning:

app	A globally unique identifier (guid) for the assessment application. The URL-like structure is intended to allow each organization to issue its own app IDs.
uid	An identifier for the student or player.
context	An identifier for the context in which the message was generated. In *Physics Playground* this corresponds to game levels, but it might have other meanings in other applications.
sender	Which process generated the message.
mess	A subject line for the message indicating its content.
timestamp	A timestamp for the message. Generally, messages for the same uid need to be processed in chronological order.
processed	A flag that can be set after the message has been processed.

In *Physics Playground* these messages were stored in a Mongo® database ("The MongoDB 4.0 Manual," 2018). This document-oriented database allows indexes to be built for the header fields, while allowing the data fields to be unrestricted (unlike a relational database, formal schemas are not required). This allowed the database to serve as queue for the other processes, as they could simply extract the oldest unprocessed message and process them.

Note that fields such as `context`, `mess`, and `sender` as well as the `data` field can take on arbitrary values. In practice, the three header fields will be limited to a certain *vocabulary*—set of legal values with specific meanings in the context of the operation. The set of legal data values will depend on the other fields. The `app` field controls the vocabulary, defining what values are legal for the other fields. This allows the same generic software to work with assessments with potentially quite different requirements.

PHYSICS PLAYGROUND CONCEPTUAL ASSESSMENT FRAMEWORK

Physics Playground is a moderately large game, with over 150 game levels, learning supports, and associated pretests and posttests. This effort has involved a large number of people (over 20), many of whom have joined the project in progress. Here, a formal assessment design is necessary to co-ordinate the efforts of all the people, as well as pass on the knowledge of key players who graduate from the project team. *Physics Playground* has a full conceptual assessment framework, but this chapter will concentrate on the pieces relevant to the development of the evidence rules.

The first step is to get clear definitions of the target constructs (Section 3.1), including defining evidence for them. When the first inventory was complete, it was clear that the existing sketching tasks were insufficient to span the construct, so new manipulation tasks were created (Section 3.2). Section 3.3 describes some other features of the game environment that need to interact with the EI process.

Contstruct Variable Definition Spreadsheet

A critical part of any assessment design is clearly defining the construct to be measured. After all, how can an assessment have construct validity if the construct is ill-defined? Table 9.1 is an excerpt from the spreadsheet we used in defining the competency variables for *Physics Playground 2*. At this stage the model is a three level hierarchical model, with and overall physics construct broken down into competencies and sub-competencies. The work was done primarily by the physics educators on the team, with other team members facilitating the progress, and asking clarifying questions.

AU: "a

Even though the focus was on defining the competencies, it was important to think about how the game could provide evidence for the competencies. The evidence column was in the spreadsheet from the beginning. This column would be later used to inspire task design and observable definitions.

TABLE 9.1 Construct Variable and Evidence Spreadsheet (excerpt)

Competency	Sub-Competency	Description	Evidence
Force and Motion	Newton's 1st Law	Static equilibrium (a = 0 and v = 0)	Player applies or adjusts a force (e.g., nudge, blow, gravity, air resistance) to keep an object stationary in at least one dimension.
Force and Motion	Newton's 2nd Law	Net force and acceleration are directly related	Player applies or adjusts a force acting on an object to cause it to accelerate at a desired rate.
Linear Momentum	Properties of momentum	Momentum is directly related to mass	Player adjusts the mass an object to affect the amount of momentum it transfers to a second object after the two collide.
Energy	Energy can transfer	Energy can transform from one type to another (e.g., GPE to KE)	Player changes parameters (e.g., mass, position, speed) to transform more or less energy of one type to another (e.g., KE, GPE, EPE) of the same object.
Torque	Properties of torque	Force and torque are directly related.	Player adjusts the magnitude of a force to cause a corresponding change in the magnitude of a torque.
Science and Engineering Practices	Use iterative design to solve a problem	Solve a problem by making variations on previous strategies	Player makes successive adjustments of the same parameter to solve a level.

The next task was to organize the proficiencies into a statistical model of the construct space. *Physics Playground* uses a Bayesian network (Almond et al., 2015) for this purpose. The Bayesian network expresses not only the possible skill profiles for the players, but the probablity that a player chosen from the target population will have a particular skill profile. Figure 9.4 shows the final competency network for *Physics Playground 2*. The graph is similar to a structural equation model, with nodes in the graph representing the competency variables, but has additional conditional independence restrictions. These restrictions allow efficient computations that make Bayesian network an ideal candidate for the EA process (Almond et al., 2015). Again, the student model, which tracks the skill profile of each player, is a player-specific copy of the competency network.

In additional to the graphical structure, conditional probability tables are requried for all nodes. This was initially done using input from the physics educators with the plan to estimate them from data after the field testing was complete.

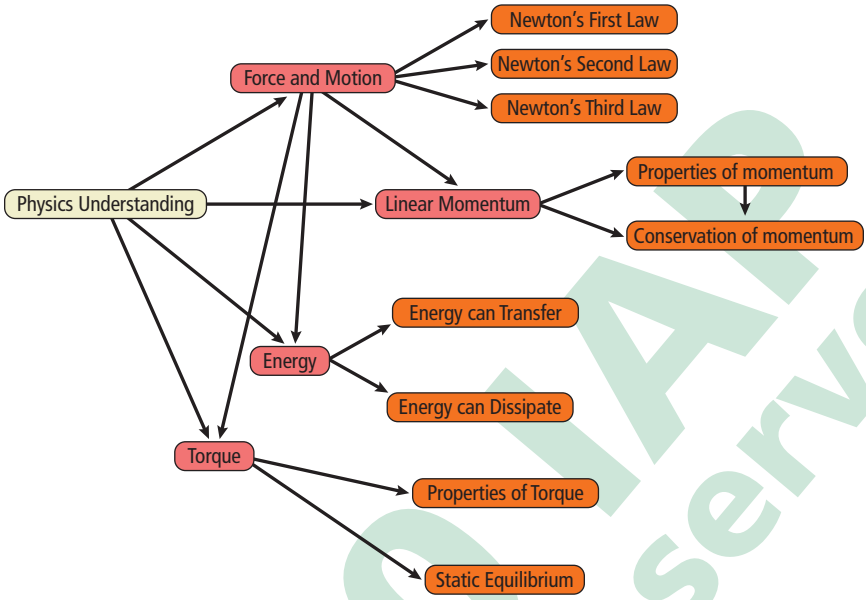


Figure 9.4 Final Competency Network.

In addition to the competency network, the EA process needs an evidence net for each game level. This is a Bayes net fragment which links the observable variables from the current level to the competencies measured in that level. Figure 9.5 shows the evidence net for the Spider Web level (Figure 9.1). Note that this is actually a fragment of the complete network as the orange nodes are references to the competency network.

The pale yellow nodes are *observables*, and rules of evidence are needed to define how their values are calculated. Section 4 addresses this task.

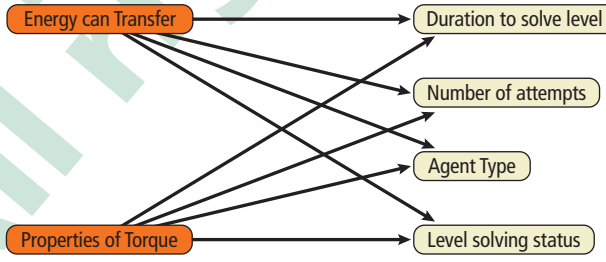


Figure 9.5 Evidence Net for Spider Web level.

Manipulation Levels

Version 1 of *Physics Playground* had only one type of game mechanism: the sketching interface shown in Figure 9.1. In building the initial competency distributions, it became clear that it would be difficult to get evidence for certain competencies within this environment. For example, the "Linear Momentum" competency required players to adjust the mass of the ball. However, this was not possible using the Version 1 game mechanics. Consequently, Version 2 added a new type of level: manipulation levels. In manipulation levels, players can adjust the mass of the ball, the force of gravity, the air resistance, as well as the force exerted on the ball by blowers (steady force) and puffers (momentary force). Figure 9.6 shows a typical manipulation level. The key step in this level is changing the coefficient of restitution (i.e., bounciness) of the ball and adjusting the other parameters so that the ball bounces off the alligators and into the balloon.

These new types of tasks required new evidence nets, with new types of observables (e.g., counting slider manipulations rather than classifying types of objects drawn). Consequently, we classified the tasks (levels) into sets that would require similar evidence rules. Figure 9.7 shows the evidence net fragment for the Florida level.

Learning Supports and Other Game Features

Version 2 of the game included an incentive system, game store (Figure 9.8), and an extensive system of learning supports. Solving a level the

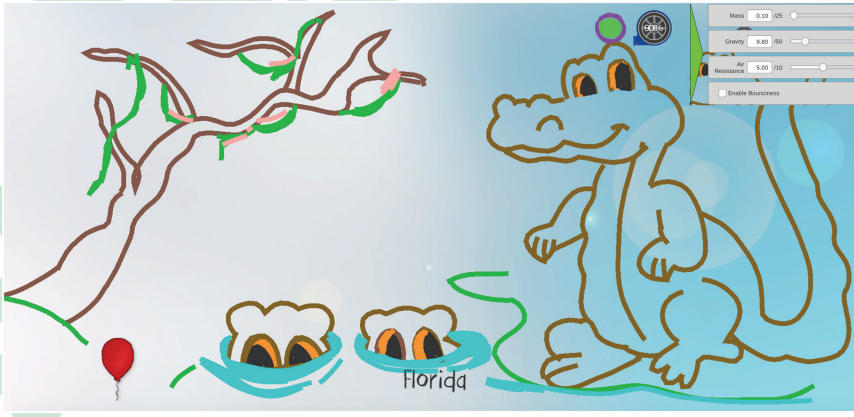


Figure 9.6 A sample manipulation level (Florida). Goal: Get the ball to the balloon by manipulating the ball's mass, gravity, air resistance, and/or bounciness.

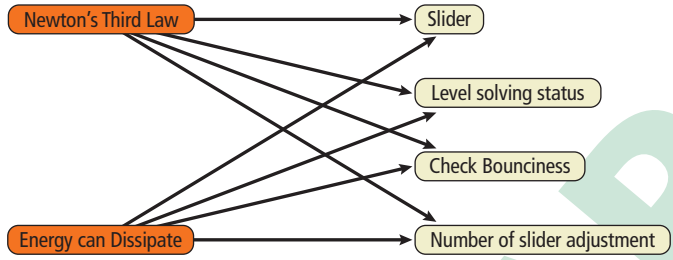


Figure 9.7 Evidence Net for Florida level.

Figure 9.8 My Backpack Physics understanding scores (top) and Store (bottom)

first time would earn a player $10 (silver coin) or $20 (gold coin). The player could spend the money in the game store to make cosmetic changes to the game (e.g., change the ball to a soccer ball, change the background image or background music). However, this added a new requirement that the game be able to keep track of which levels the player had solved and their bank balance across playing sessions.

The learnings supports included interactive definitions of physics terms, videos illustrating physics concepts using the game mechanics as well as hints and worked examples (i.e., short videos of expert solutions). Learning

support was tied into the reward structure. Viewing the physics videos or definitions would earn money (+$10 or +$5, respectively), but looking at worked examples cost money (–$60).

The reward system generated a challenge for the browser-based game engine. When the player logged off from the system and returned the next day, all locally cached information about the player's progress would be lost. The game engine needed to consult the server to restore the state of the system (i.e., landing the student on the last level they solved and restoring the player's bank balance).

Through the use of special trophy hall observables (a list of what levels the player had compeleted as well as the player's bank balance), the EI process could track the needed information. It then saved that information to a database on the scoring server which was consulted when the player logged in to restore the state of the game.

The end result of all of this modeling is a list of *observable* variables that form the output of the EI process. This included some human language description of the observables. The challenge was to transform these human descriptions into computer code—*rules of evidence*—that can set the value of the observables.

EVIDENCE RULES

The goal of EI is to filter the raw stream of events that comes from a game into a few key *observable* outcomes. These then become inputs to the statistical models in the EA process. Observables can be divided into *final observables* that are reported out by the EI process and *intermediate observables* (called *flags* and *timers*, see below), which are used to calculate the final observables.

In general, final observables are defined for one of three reasons.

- They are used as input to the EA process. (In *Physics Playground*, these are nodes in the Bayes Nets).
- They are used for context-level feedback. In *Physics Playground*, the player's bank balance was a feedback observable. It was not used in scores, but it was stored in the database, so the game engine could restore the bank balance when the player resumed play.
- They are logged for research purposes. In *Physics Playground* the complete sequence of simple machines constructed in a sketching level was logged for later inspection, but not scored. Note that these observables do not need to be calculated in real-time, but can be done by post-processing.

Defining Observables

The first step in defining the evidence rules is defining the observables. In *Physics Playground* these definitions were kept in a spreadsheet. The observable definitions must be sufficiently precise that they can be used as requirements for the evidence rules. This often requires the design team to work through some possibly tricky questions. Some examples from *Physics Playground* include the following:

- When is a drawn object considered a lever? This question involved identifying torques and rotation done by the object, so it proved easiest to include lever identification code in the game engine.
- What constitutes an "attempt" at a task? Is restarting the game level without leaving it an attempt, or does the attempt require navigating to a different level and then returning?
- Where should we setting cut scores for counts and continuous variables used as ordinal variables in Bayes nets? For example, what are long, medium, and short completion times for level? What are low, medium, and high numbers of manipulations for a slider? Note that these cut scores can be implemented in either the EI or EA process.

AU:

The following examples span the types of observables used in *Physics Playground*:

Obs 1	What is the maximum value coin (gold, silver, or none) that the player has earned for this level?
Obs 2	Did the player manipulate the gravity slider?
Obs 3	How many objects did the player draw?
Obs 4	How much time (excluding time spent on learning supports) did the player spend on the level?
Obs 5	Did the player attempt to draw a springboard?

Obs 1 and 2 can be read directly from the event stream. Obs 3 and 4 require keeping track of the state of the system (Section 4.4). Obs 5 requires detailed knowledge of the internal state of the game engine. To solve this problem, the EI code was put in the game engine and the output of the identification code was logged to the event stream.

Learning Record Store

While observables are the output of the EI process, the events in the learning record store are the input. This is an extension of the simple Proc4

message (Listing 9.1) adding two new fields `verb` and `object` (described below). A *learning record store* is simply a database which accumulates such records. Learning Locker is a learning record store which provides additional services (such as buffering logging requests, and some reporting functionality). Part of the specifications of the presentation process (game engine) is that it should log to the learning record store when certain events occur.

While most of the fields are inherited from the general Proc4 message, the `verb` and `object` are borrowed from the xAPI format (Betts & Smith, 2018). The idea is that the verb and object should give a good idea of the content of the message. Some example verb–object pairs from *Physics Playground* include "initialized" "game level" (new level started), "identified" "game object" (game engine identified a ramp, lever, springboard or pendulum), and "manipulated" "control" (player adjusted one of the sliders). The verb–object pairs are useful because events can be filtered on these fields, reducing the number of events that need to be more carefully processed.

The Proc4 event format (Listing 9.2) is a simplification of the xAPI format in several ways. First, xAPI makes the `verb` and `object` fields more complex objects, including long url-like globally unique identifiers (guids), instead of simple strings. In the Proc4 version, the app field is a guid, but the other fields can be simple strings. Thus, the database can build indexes on those fields. Furthermore, while xAPI has several places where extensions can be added, in the Proc4 events, all extensions are added to the free form `data` field.

In the *Physics Playground 2* implementation, a daemon process periodically (several times per minute) grabbed new events from the Learning Locker store on the game server, translated them to the Proc4 event format, and did some filtering to remove unused verb–object pairs. The results were put into

LISTING 9.2: A typical event record

```
{
app:"ecd://epls.coe.fsu.edu/AssessmentName/StudyCondition",
uid:"Student/User ID",
timestamp:"Time of event",
verb:"Action Keyword",
context: "Context ID",
object:"Object Keyword",
data:{
 field1:"Value",
 field2:["list", "of", "values"],
 field3:{part1:"complex", part2:"object"}
}
}
```

a Mongo database collection. The EIP used that collection as an input queue, processing messages starting with the oldest unprocessed message.

Determining Tasks From Event Traces

One other key difference between xAPI and Proc4 event format is the context field (which is optional in xAPI and required in Proc4 although null values are allowed). While the meaning of context is vague in the xAPI specification ("An optional property that provides a place to add contextual information to a Statement [event]," xAPI Specification, Section 2.4.6, Betts & Smith, 2018), in the Proc4 context it is designed as an ECD task identifier.

The term *task* was introduced in ECD (Mislevy et al., 2003) to convey the idea that some assessment tasks, particularly in simulations and game, are more complex than simple items. Such tasks might have complex work products (e.g., an event stream) and multiple observed outcome variables. Each assessment would need to define what a *task* was for that application, but the basic rule was that a task was a unit of activity at which messages would flow around the four process architecture (Almond et al., 2002).

The term *context* is used instead of task, because in game and simulation based assessments, the task might emerge from the state of the game or simulator (Mislevy, 2013; Mislevy et al., 2015). In particular, actions of the player or changes in the system state might change the assessment context (the appropriate evidence processing instructions), even though the player is still following one task, such as the set of instructions and ultimate goal of the game.

Consider a flight simulator. The player (or pilot trainee) might work on a single *task* : fly the plane from one airport to another. However, within this single task, there are a number of different *contexts* in which the pilot needs to focus on different kinds of actions and instruments. A partial list might include:

1. Pre-flight Checks
2. Take-off
3. Cruising
4. Thunderstorm
5. Cruising (return to this context)
6. Approach
7. Landing

The same event might have different evidentiary value in different contexts. For example, lowering the landing gear is necessary during landing, but probably a bad idea during a thunderstorm.

This puts another requirement on the EI process. If the context is not (or cannot be) supplied in the logged event, then the EI process must infer it from the stream of events resporting the state of the game or simulator. In *Physics Playground* the context is the game level, but the game level was not present in all of the messages logged to Learning Locker®. In particular, the EI process used the "initialized" "game level" message to identify that the context had changed.

It also was convenient to define *context sets*—sets of contexts for which similar evidence identification logic could be used. For example, sketching levels (where the player draws objects) and manipulation levels (where the player manipulates simulation parameters to solve the level) are two natural context sets. Similarly, "bounciness levels," where the player must make the ball bouncy to solve the level, are the only ones in which events involving the bounciness control are relevant.

Tracking System State

Consider Obs 3 (number of objects drawn) and Obs 4 (duration of time on task) defined above; both require summarizing over multiple events. Obs 3 requires the EI process to maintain an object counter within a context (game level). Obs 4 requires the EI process to maintain a timer which is paused when the player enters a learning support and resumed when they return to the game context. In both cases, separate counters/timers are needed for each player and the counters and timers must be reset when the player enters a new level.

To store this person–context specific status information, the EI process needs to maintain a *state* object for each person. This is not a complete reproduction of the state of the player inside the game, but rather just contains the intermediate observables need to calculate the final observables. Listing 9.3 gives the JSON schema for a state.

LISTING 9.3: A high-level schema for a state object

```
{
  app:"Global identifier for the application"
  uid: "Identifier for player",
  context: "Task or context identifier",
  oldContext: "Task or context identifier.",
  timestamp: "Timestamp of last event processed."
  timers:{...},
  flags:{...},
  observables:{...}
}
```

The state object contains three collections of special variables: *flags*, *observables*, and *timers*. Flags and observables are similar in many ways, the only real difference being that observables are usually reported out from the EI process, and flags are usually internal to it. They could be numeric or character values, or lists of such values or even more complex object structures. The idea is that the various evidence rules will update their values. Timers are special variables meant to calculate the time spent in various kinds of activities. Rules can stop and start the timers as well as reset their values. All flags, observables, and timers have a name, so their values can be referenced by name in rules (e.g., `state.flags.gravityChanged`, `state.observables.objectsDrawn`, `state.timers.levelTime`).

Note that the state has both a *context* and *oldContext*. This allows the rules to determine when the state has changed from one context to another (e.g., "initialized" "game object" message). Often this is a trigger that the EI process needs post the observables for another process.

AU: EI process needs to post ?.

Rules of Evidence

A *rule of evidence* is a rule that defines how the player's state should change in response to a given event. It takes a state and an event as input and outputs an updated state. These are if–then type rules, which contain a set of *conditions* that must be met for the rule to fire. If the conditions are met, then the rule's *predicate* is executed to update the player's state.

Listing 9.4 describes the basic structure of a rule (with Listing 9.5 and Listing 9.6 detailing the condition and predicate parts of the rule). The `app` field is used to associate a rule with a particular assessment, and the `name`

LISTING 9.4: The basic Rule Schema (condition and predicate are detailed below)

```
1   {
2   app:"Global identifier for the application",
3   ruleName:"Human readable identifier",
4   doc:"Human language description",
5   context:"Context or Group Keyword",
6   verb:"Action Keyword or ALL",
7   object:"Object Keyword or ALL",
8   ruleType:"Type Keyword",
9   priority:"Numeric Value",
10  condition:{...},
11  predicate:{...}
12  }
```

and `doc` fields are there to aid in maintaining the rules. The `ruleType` and `priority` fields are used to control the sequencing of the rules (see below). The `verb`, `object`, and `context` are technically part of the condition, however, they are separated so they can be used for filtering.

A rule is applicable to a given event if the `verb` and `object` fields of the rule and event match (or have the special value "ANY"). It is also only applicable when the current `context` of the state matches the context of the rule. Note that the rule's context could be a context set, so any context within that set matches, including the set of all contexts, "ALL." The `condition` and `predicate` fields of the rule are effectively an interpreted programming language that is described below.

Rule Types and Priority

Although the rules are similar to rule-based programming languages like Prolog, the sequencing of the rules can actually be important to their correct functioning. In order to make the sequence predictable, the rule execution is divided into five phases, with a different type of rule executed (if the conditions are met) in each phase. The *priority* field is used to establish the sequence within each phase: with rules being run in the order of the numeric priority variable.

The five types of rules are in sequence:

1. *State Rules*—Update flags and timers.
2. *Observable Rules*—Update observables.
3. *Context Rules*—Check for changes in context.
4. *Trigger Rules*—Send messages to other processes.
5. *Reset Rules*—Clean up state when context changes.

State and observable rules are actually identical (except for the order in which they are run). Although the intention is that state rules update flags and timers, and observable rules update observables, both types of rules can update all three kinds of values. The context rules are specifically for updating the `context` field. Note that the `oldContext` field is not changed until all rules are run, so that the later rules can check whether or not the context was updated. The reset rules, in particular, are designed to be run only after the context changes (so that context-specific flags and timers can be reset).

The trigger rules play a special role in the EI process. These indicate that the process should send an outgoing message. In particular, EI needs to send a "New Observables" message to the EA process, to indicate that new observables are ready to be processed, but it may send other messages to other processes (e.g., learning support systems). By default, the data field for that message will be the observables of the current state for the player; however, a subset of observables can be selected in the predicate of the trigger rule.

Conditions

The condition is the "if" part of the if–then rule. It lists a number of fields in either the event (referenced as `event.data.<name>`) or in the state object (referenced as `state.flags.<name>`; `state.observables.<name>`; or `state.timers.<name>`). Instead of an exact value, the rule can specify a comparison operator `?op`, inspired by the query documents used by the Mongo database ("The MongoDB 4.0 Manual," 2018). Listing 9.5 shows a few possible query syntaxes.

There are a number of possible query operators including: `?eq`, `?ne`, `?gt`, `?gte`, `?lt`, and `?lte`, `?in`, `?nin`, `?exists`, `?isnull`, `?isna`, `?any`, `?all`, `?not`, `?and`, `?or`, and `?regex` (which matches regular expressions). If the operator is omitted, `?eq` (equals) or `?in` (is in) are used as the operator depending on whether the value is a single value or a list. The `?where` operator is an extension operator: it runs a function in the host language (in the current implementation R (R Core Team, 2018). This can be used for more complex queries that are not easily expressed in the simple query language.

If multiple conditions are present in the condition field of the rule, then all of them must be satisfied for the rule to be applied. This includes the implicit conditions established by the verb, object, and context fields of the rules. This is effectively a logical conjunction. A logical disjunction can be created with multiple rules.

Predicates

The predicate is a set of instructions that is run only if the condition (including the implicit conditions of the `verb`, `object`, and `context` fields) is satisfied. The instructions usually involve setting one of the flags, timers, or observables of the state object. In each case, the operator is followed by a list of field value pairs that indicate which fields are to be updated and how. Listing 9.6 shows the general syntax along with a few common examples.

The most basic operators are `!set` and `!unset`, which set the value of a particular field. The field to be set must be in the state, but the value can reference a data field from the event. The operators `!incr`, `!decr`, `!mult`, `!div`, `!min`, and `!max` manipulate numeric fields, and the operators

LISTING 9.5: Several example condition clauses from a rule

```
1  condition:{
2    <field 1>:<value 1>,
3    <field 2>:[<value-list>],
4    <field 3>:{"?op":<value3>},
5    "?where":<function>,
6    ...
7  g
```

LISTING 9.6: Several examples of the predicate part of a rule

```
predicate :f
  <update operator>:{ <field1>: <value1>, ... },
  "!set":{ state.flags.<logical>: true},
  "!set":{ state.timers.<name>.running: true },
  "!incr":{ state.flags.<count>: 1},
  "!setCall":{ state.flags.<name>: <function>},
  ...
}
```

`!addToSet`, `!pullFromSet`, `!push`, and `!pop` operate on list-valued fields. For timers, the subfields `state.timers.`*`name`*`.running` and `state.timers.`*`name`*`.time`, or the special operators `!start` and `!reset` can be used. The extension operator `!setCall` runs a function in the host langauge (R); allowing the set of operators to be expanded as needed.

Because the current EIEvent implementation is written in R (R Core Team, 2018), the set of query and update operators is straightforward to extend. R is a reflexive language, that allows functions to call other functions by name (Chambers, 2004). In EIEvent, there are R functions bound to the query operator (e.g., `?eq`) and update operator (e.g., `!set`) names. The condition checking and predicate execution code calls these functions by name (using the R `do.call` function). Thus, the set of query and predicate operators can be easily extended by simply writing functions for the new operators.

Extended Example: Counting Air Slider Manipulations

Consider the problem of counting how many times the player has manipulated the air resistance slider within a given level. The goal is to set an observable variable—`state.observables.airManip`—to the number of manipulations. We will assume that at the beginning of the level, a reset rule sets this observable to zero. The rule shown in Listing 9.7 increments the counter when appropriate.

The `name` and `doc` fields describe the rule, and the `app` field shows which application uses this particular rule instance. The `verb`, `object`, and `context` field allow the EI process to filter out this rule when it is applicable. The condition will only be checked if the verb of the event is "Manipulate" and the object is "Slider." Also, the context of the current state must be a game level that is in the "Manipulation Levels" set.

Assuming that the event is applicable to this event and context, the conditions will be checked. Here there are two requirements. First, the slider that was moved (the `gameObjectType`) must be the "AirResistanceValueManipulator."

LISTING 9.7: Count Air Resistance Manipulations Rule

```
{
app: "ecd://epls.coe.fsu.edu/PPTest"
name: "Count Air Resistance Manipulations",
doc: "Increment counter if slider changed.",
verb: "Manipulate",
object: "Slider",
context: "Manipulation Levels",
ruleType: "Observable",
priority: 5,
conditions: {
 event.data.gameObjectType:"AirResistanceValueManipulator",
 event.data.oldValue:{"?ne":event.data.newValue}
},
predicate: {
 "!incr":{state.observables.airManip:1}
}
}
```

Second, the oldValue cannot be equal to the `newValue`; that is the player must have actually moved the slider and not just kept it in the same place.

If the conditions are met, then the predicate is executed. In this case the `airManip` observable is incremented by one. As this rule sets an observable, its type is "Observable," and as it is independent of other rules, it is given a moderate priority of 5.

To see this rule in action requires an event (Listing 9.8) and an initial state (Listing 9.9).

LISTING 9.8: Event for testing Air Manipulation Rule

```
{
app:"ecd://epls.coe.fsu.edu/PPTest",
uid: "Test0",
verb: "Manipulate",
object: "Slider",
context: "Air Level 1",
timestamp:"2018-09-25 12:12:28 EDT",
data: {
 gameObjectType: "AirResistanceValueManipulator",
 oldValue: 0,
 newValue: 5,
 method: "input"
}
}
```

LISTING 9.9: Initial State for testing the air manipulation rule

```
{
app:"ecd://epls.coe.fsu.edu/PPTest",
uid: "Test0",
context: "Air Level 1",
timers:{},
flags:{
  airUsed: true ,
  airVal: 5,
},
observables:{
  airManip:1
}
}
```

Note that the verb is "Manipulate" and the object is "Slider"; thus, the rule is applicable to this event.

Presuming that "Air Level 1" is in the set of manipulation levels, so this rule is applicable to this event and state. Next the EI process checks the conditions. Here `gameObjectType` is indeed "AirResistanceManipulation-Slider" and `oldValue` is indeed different from `newValue` so the condition is satisfied.

As the rule is applicable and the condition is satisfied, the EI process runs the predicate. In this case, the value of the `airManip` observable is incremented. Other flags and observables remain the same. Listing 9.10 shows the result.

LISTING 9.10: Final state after running Air Manipulation rule

```
{
app:"ecd://epls.coe.fsu.edu/PPTest",
uid: "Test0",
context: "Air Level 1",
timers:{},
flags:{
  airUsed: true,
  airVal: 5
},
observables:{
  airManip:1
}
}
```

This example is more than a simple illustration: It is part of a proof of correctness of the rule (Dijkstra, 1976). The rule is the program, and the input state and event are the preconditions. The output state is the post-condition. Thus, this set of JSON objects can be used to test that the rule is functioning properly. Note that while this shows a positive instance, a more complete test would include both positive and negative instances (places where the conditions are not satisfied). Producing an exhaustive set of tests requires a disciplined programmer.

However, even an incomplete set of such tests provides the advantages of unit testing (Runeson, 2006) to the design of the EI process. In particular, running this test suite can help discover regression problems if a rule was changed, or possible negative interactions among rules. Also, as play testing uncovers problems in the rules, new tests can be added to the suite to prevent the problems from reoccurring.

FIELD TESTING

In May of 2019, we conducted a large-scale field trial using 27[illegible]udents. The students each played the game for approximately 5 hours, generating over 4.5 million statements in the learning record store. The students were assigned to four study conditions, one of which was a no-treatment control group, with approximately an equal number (70) of students per condition. The three treatment conditions differed in how the game levels were sequenced: (a) a user controlled sequence (user control), (b) a computer-controlled sequence arrayed from easy-to-difficult (linear), and (c) a computer-controlled adaptive sequence. The adaptive sequence controlled which level was delivered next based on the estimated ability of the player, so it required the output of the EA process. The scoring was done on a dedicated scoring server (different from the Unity server) running Red Hat Enterprise Linux 7.5 with 16 cores available. The three conditions were assigned different application IDs, so that the EI and EA processes for those groups could be run on different CPUs on the server; so six of cores were used for scoring (while others handled communications and database tasks).

The adaptive sequencing algorithm required nearly real-time processing from both EI and EA processes, as the decision about which level to present next should be determined on the basis of the student's estimated ability on the skill which was the current focus topic. If the EI and EA processes have not completed by the time the game engine needed to determine the next level, the server would return a cached version of statistics. However, if the EI and EA processes fell badly behind, then the adaptive system would not be responsive to differences in student abilities. The other two branches of the study required the trophy hall observables (e.g., bank balance) from the

EI process to restore the game state at the start of a new playing session; but this only happened once a day, so there was more time to complete the task.

The initially deployed system had 15 observable outcome variables used by the EA process, and 23 rules of evidence to calculate those variables. In addition, six more rules of evidence were used to track the players' progress through the game (trophy hall observables), to aid in restarting on the next day.

Unfortunately, due to set backs in the development processes, the rules of evidence were not thoroughly tested. This resulted in numerous errors the first half day of testing. (Note that the initial day of testing included a pretest, so the subjects had only 1/2 class period of game play.) Some of the problems were inevitable, and could have been easily fixed with more testing. (For example, the name of a level was called "Rabbits' hole" in one data file and "Rabbit's Hole" in another. Others were bugs in the EIEvent code.

Another problem, however, had to do with the incomplete nature of the test sets. In particular, the people writing rules were not programmers, so they had little experience in writing test cases. Also, while the rules were tested in isolation (similar to the test in the previous level), there were few tests of the entire rule set to look at interactions. One efficient way to generate test cases is to have level designers play through the level, and then to capture the events generated by that user in the context of that level (the events can be captured with a simple database query). Making a tool that allowed a play tester to play a game level and then compare the observables immediately would help spot problems earlier. In addition to the bugs in the rules and the EI software uncovered by the serious testing from the first day, another problem arose related to timing. The first half day of testing produced several hundred thousand events for the EI process to work through, and it was moving much too slowly. One problem was the amount of logging turned on (it was still at the debug level for testing), which caused lots of time consuming disk writes. But by far the biggest problem was the number of events that were processed for which no rules were applicable. We set the debugged EI to reprocess the data from Day 1's testing on the evening of that date and it was unclear that process would be finished in time for Day 2 testing.

To expedite the process, we made some hasty changes to the scoring model. First the number of observables was reduced from 14 to four: the trophy received for the level, the time spent on the level, the last agent used (for sketching levels), and the number of attempts per level. This in turn dropped the number of rules of evidence required from 23 to 9. More importantly, it enabled the events to be filtered before coming into the EI process. The number of events dropped from hundreds of thousands to several thousand: a 500/1 reduction.

Although there are many ways the EIEvent engine can be rewritten to improve its efficiency, because of the real-time requirements, it is important to consider multiprocessing. Our initial deployment used only 3 of the 16 available processors for EI (and another 3 for EA), with one dedicated process for each of the three branches of the study. Unfortunately, R (R Core Team, 2018) provides only limited tools for multiprocessing. Although multiple threads could be assigned to each branch, it is important to make sure that the events for a single player are processed in sequence: Thus, more sophisticated resource queuing is needed.

As EI does not use any of the data analysis capabilities of R, a different programming language which offers more multiprocessing tools should be considered for the next version of EIEvent.

REFLECTIONS

The EIEvent package used in *Physics Playground 2* demonstrates the feasibility of using a rule-based approach to evidence identification in game-based assessments. The JSON-based rules described in Section 4.5 provide a solid, extensible starting point for writing custom code for this approach. Writing, testing, and debugging the rules for a particular observable takes on the order of one day (after the designer learns the system). Most importantly, the observables used in this model, unlike those used in machine learning approaches, can be justified from a construct validity standpoint.

The goal of designing the system so that level designers and not coders can write the rules was only partially successful. Although a number of members of the level design team contributed to the rule making, the test sets created with the rules were incomplete. Also, more co-ordination was needed between the game engine design and the rule design. Specifically, it was necessary to know what events were logged and when. Better documentation could have helped this process.

It is worth spending a fair amount of time on planning how the scoring process should be tested. A graphical tool that allows a tester to run simple test cases would enable nonprogrammers to do more of the testing. Also, a version of the game where the game engine displays the observables after a tester completes a level would help testers identify problems along with test cases for those problems.

The use of event filtering to speed up EI is important for systems which require real-time scoring. In particular, observables which are needed for the real-time adaptivity should be given precedence over observables for research purposes. The more events which can be pre-filtered before getting to the EI process, the faster the process will run.

Although these tools are still fairly early in their development, the R packages are available from the web sites listed below.

- Game demo and level editor: https://pluto.coe.fsu.edu/ppteam/pp-links
- Peanut and RNetica [Bayesian Network Tools]: https://pluto.coe.fsu.edu/RNetica
- Proc4 (General message and database functions), EIEvent (Rule-based Evidence Identification), and EABN (Bayes net Evidence Accumulation): https://pluto.coe.fsu.edu/Proc4

ACKNOWLEDGMENTS

Work on this paper was supported by the National Science Foundation grant DIP 037988, Val Shute, Principle Investigator. Version 1 of *Physics Playground* was supported by the Bill & Melinda Gates Foundation U.S. Programs Grant Number #0PP1035331, and concurent work on *Physics Playground* is supported by the Institute of Educational Science, Goal 1 039019, Val Shute, PI.

A large cast of people have worked on Physics Playground including: Fengfeng Ke (co-PI for the NSF grant), Adam LaMee, Don Franceshetti*,[1] and Mathew Martens (Physics Eduction Experts), Weinan Zhao*, Xinhao Xu*, Chen Sun, Xi Lu*, Ginny Smith, Curt Fulwider*, Zhichuan Lukas Liu, Megan Bayles*, Buckley Guo*, Jiawei Li, Jordan Charles*, Renata Kuba, Chih-po Dai.

We would also like to thank the teachers, adminstrators, and espcially the IT staff of the schools at which we did our playtesting. Their help has made this game a valuable educational experience for their students.

NOTE

1. People marked with an asterisk were formerly involved with the project.

REFERENCES

Almond, R. G. (2019a). *EIEvent: Evidence identifcation event processing engine* [Computer software manual]. Retrieved from https://pluto.coe.fsu.edu/Proc4

Almond, R. G. (2019b). *Peanut: Parameterized bayesian networks, abstract classes* [Computer software manual]. Retrieved from https://pluto.coe.fsu.edu/RNetica

Almond, R. G., Mislevy, R. J., Steinberg, L. S., Yan, D., & Williamson, D. M. (2015). *Bayesian networks in educational assessment.* Springer.

Almond, R. G., Steinberg, L. S., & Mislevy, R. J. (2002). Enhancing the design and delivery of assessment systems: A four-process architecture. *Journal of Technology, Learning, and Assessment, 1*(5). Retrieved from http://www.jtla.org/

Bassett, L. (2015). *Introduction to JavaScript object notation: A to-the-point guide to JSON.* Stebastopol, CA: O'Reilly.

Betts, B., & Smith, R. (2018). *The leraning technology manager's guide to xAPI* (2nd ed.) [Computer software manual]. Retrieved from https://www.ht2labs.com/resources/the-learning-technology-managers-guide-to-the-xapi/#gf26

Chambers, J. L. (2004). *Programming with data: A guide to the S language.* New York, NY: Springer.

Dijkstra, E. W. (1976). *A discipline of programming.* ???: Prentice-Hall.

Kim, Y. J., Almond, R. G., & Shute, V. J. (2016). Applying evidence-centered design for development of game-based assessments in Physics Playground. *International Journal of Testing, 16*(2), 142–163. Retrieved from http://www.tandfonline.com/doi/ref/10.1080/15305058.2015.1108322. (Special issue on cogntive diagnostic modeling)

Learning Locker Documentation (2nd ed.) [Computer software manual]. (2018). Retrieved from https://docs.learninglocker.net/welcome/

Mislevy, R. J. (2013). Evidence-centered design for simulation-based assessment. *Military Medicine, 178*, 107–114.

Mislevy, R. J., & Gitomer, D. H. (1996). The role of probability based inference in an intelligent tutoring system. *User-Modeling and User-Adapted Interaction, 5*, 253–282.

Mislevy, R. J., Oranje, A., Bauer, M. I., von Davier, A., Hao, J., Corrigan, S., . . . Joh, M. (2015). *Psychometric considerations in game-based assessment* (Tech. Rep.). GlassLab: Institute of Play. Retrieved from http://www.instituteofplay.org/work/projects/glasslab-research/

Mislevy, R. J., Steinberg, L. S., & Almond, R. G. (2003). On the structure of educational assessment (with discussion). *Measurement: Interdisciplinary Research and Perspective, 1*(1), 3–62.

Norsys, Inc. (2012). *Netica.* Retrieved from http://www.norsys.com

Ploetzner, R., & VanLehn, K. (1997). The acquisition of informal physics knowledge during formal physics trianing. *Cognition and Instruction, 15*, 169–206.

R Core Team. (2018). *R: A language and environment for statistical computing* [Computer software manual]. Retrieved from https://www.R-project.org/

Runeson, P. (2006, July). A survey of unit testing practices. *IEEE Software, 23*(4), 22–29. https://doi.org/10.1109/MS.2006.91

Shute, V. J., Hansen, E. G., & Almond, R. G. (2008). You can't fatten a hog by weighing it—or can you? Evaluating an assessment for learning system called ACED. *International Journal of Artificial Intelligence in Education, 18*(4), 289–316. Retrieved from http://www.ijaied.org/iaied/ijaied/abstract/Vol18/Shute08.html

Shute, V. J., & Ventura, M. (2013). *Stealth assessment in digital games.* Retrieved from http://library.oapen.org/handle/20.500.12657/26058

The MongoDB 4.0 Manual (4.0 ed.) [Computer software manual]. (2018). Retrieved from https://docs.mongodb.com/manual/

AU: Please update link.

AU: Please provide location.

AU: Please clarify publication location, format as City, State: Publisher if applicable.

AU: Please update link.

Unity Technologies. (2019). *Unity user manual* (2019.1-002V ed.) [Computer software manual]. Retrieved from https://docs.unity3d.com/Manual/index.html