Stealth Assessments' Technical Architecture

Seyedahmad Rahimi[1], Russell G. Almond[2], Valerie J. Shute[2]

[1]University of Florida, [2] Florida State University

Abstract

With advances in technology and the learning and assessment sciences, educators can develop learning environments that can accurately and engagingly assess and improve learners' knowledge, skills, and other attributes via stealth assessment. Such learning environments use real-time estimates of learners' competency levels to adapt activities to a learner's ability level or provide personalized learning supports. To make stealth assessment possible, various technical components need to work together. The purpose of this chapter is to describe an example architecture that supports stealth assessment. Toward that end, we describe the requirements for both the game engine/server and the assessment engine/server, how these two systems should communicate with each other, and conclude with a discussion on the technical lessons learned from about a decade of work developing and testing a stealth-assessment game called *Physics Playground*.

Key words: Stealth assessment, software architecture, Physics Playground, ECD

Introduction

Learning and engagement theories—such as the zone of proximal development (Vygotsky, 1978) and flow (Csikszentmihalyi, 1990)—suggest that challenges in a learning environment should match learners' ability. With advances in technology, as well as in the learning and assessment sciences, educators can develop learning environments that can accurately assess and support learners' knowledge, skills, and other attributes (Shute et al., 2016; Shute & Rahimi, 2017). Such learning environments rely on real-time competency estimates to adapt challenges to learners' ability levels or to provide appropriate supports to maximize learning.

Stealth assessment (Shute, 2011) uses games or other technology-rich environments as a vehicle to assess learners' emerging competencies (e.g., creativity, problem-solving, physics understanding). Stealth assessment is based on an assessment design framework called evidence-

centered design (ECD; Almond et al., 2002). ECD allows stealth assessment designers to define the competency (unobservable) they are interested to assess (i.e., the competency model), identify good in-game indicators (observables), which can be statistically linked to the competency model (i.e., the evidence model), and define and create tasks that can elicit the evidence needed for the evidence model (i.e., the task model). When these three core models are established and implemented in a system, observations made in the context of stealth assessment tasks provide evidence of competency levels, allowing the system to update competency estimates in real-time.

The ongoing performance data are collected in log files as a learner interacts with the game. The stealth assessment then automatically scores and accumulates the collected data using statistical methods (e.g., Bayes nets), and makes real-time inferences about the learner's current level of targeted competencies, adapting the game difficulty accordingly (see Rahimi et al., in press; Smith et al., in press for more details). To make stealth assessment possible, various technical components need to seamlessly work together. The purpose of this chapter is to describe the various components (i.e., the architecture) of an adaptive environment in the context of a game called *Physics Playground* (*PP*; Shute et al., 2019). *PP*'s architecture can be used to create other educational games equipped with stealth assessment.

PP is a 2D game with simple game mechanics (e.g., drawing lines and creating physics simple machines; see Figure 11.1). The goal in this game is to direct a green ball to a red balloon. There are two level types in PP: *sketching* and *manipulation*. To solve sketching levels, learners draw simple machines (i.e., ramps, levers, pendulums, and springboards) to guide the ball to the balloon (Figure 11.1a). To solve manipulation levels, learners interact with various sliders to

change physics parameters (i.e., gravity, air resistance, mass, and bounciness of the ball), and also manipulate external forces exerted from puffers or blowers (Figure 11.1b).

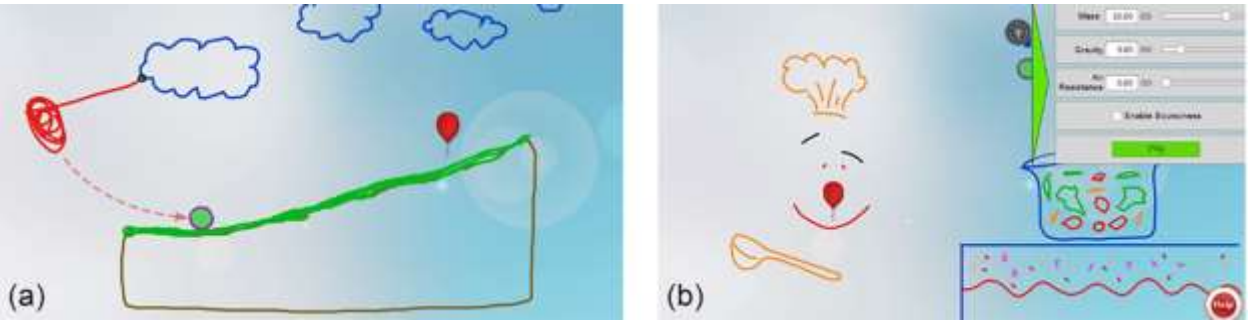

Figure 11.1: Sketching level (a) and Manipulation level (b)

Through an iterative process, we designed hundreds of game levels (explained later) and developed numerous learning supports in *PP* to assess and enhance learners' physics understanding related to specific concepts (Figure 11.2). Learners earn a gold coin for an elegant solution using a small number of objects or attempts, a silver coin for a regular solution with more than a predefined number of objects or attempts, or nothing for a failed attempt.

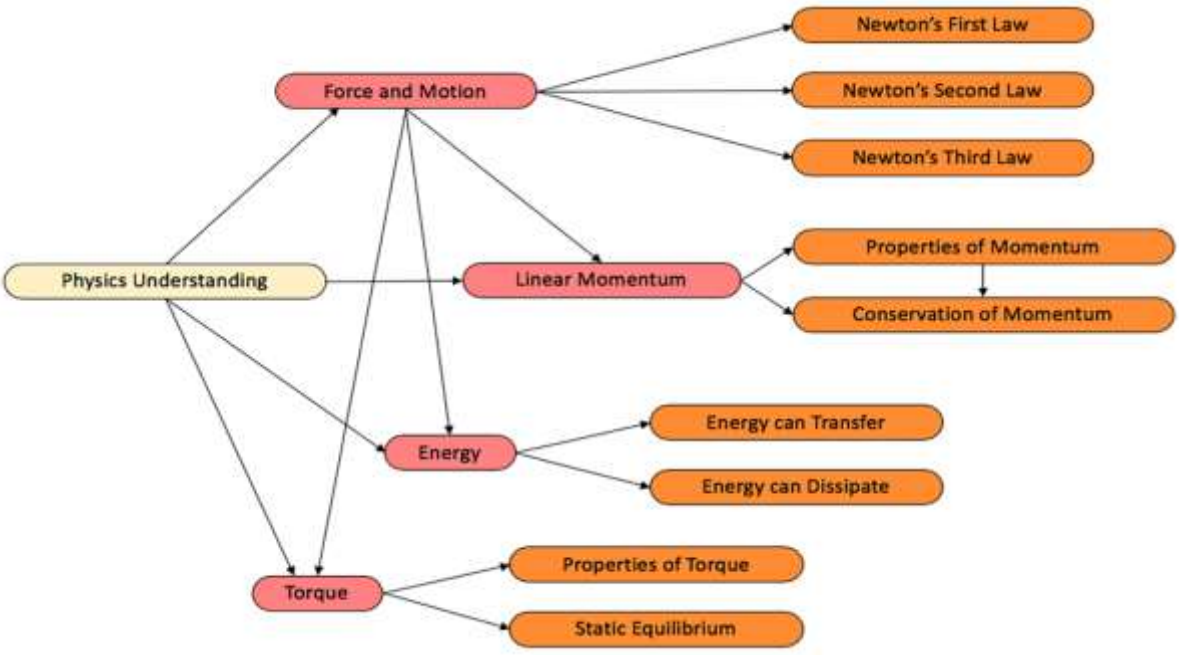

Figure 11.2: Physics understanding competency model

Currently, *PP* gathers learner gameplay data in log files, calculates each learner's overall and specific physics understanding per topic (i.e., the low-level, orange nodes in Figure 11.2), and finally, uses those estimates as the basis for adapting game difficulty to learners' current competency levels. Moreover, *PP* uses the stealth assessment estimates to provide timely learning support, and inform learners of their progress in physics understanding via an in-game dashboard. All of this is possible via a complex architecture— the focus of this chapter.

PP's Architecture

Almond, Steinberg and Mislevy (2002) defined a generic four-process architecture for assessments (Almond, 2020). The four mechanisms provided by the assessment system are (a) *(Activity) Presentation*—presenting the tasks (e.g., game levels) and capturing the responses, (b) *Evidence Identification*—extracting observed outcomes from the log file per task, (c) *Evidence Accumulation*—combining evidence across tasks, and (d) *Activity Selection*—deciding on the next activity (assessment or learning task) for the learner. Any assessment implementation like this must define the software architecture to carry out these tasks and assign them to an appropriate computer (which could be a client or a server in the network). The specific software architecture of stealth assessment systems may not include the same components as we employ. However, the general architecture components that we discuss in this chapter are intended to be generalizable to other stealth assessment systems.

As shown in Figure 11.3, *PP*'s architecture resides on two separate servers: (1) the *PP server*, which stores the game engine and the main databases; and (2) the *Assessment Server*, which stores data and output for the four processes in what is called the Proc4 database. The Assessment Server also houses the *Assessment Engine* (i.e., Evidence Identification and

Accumulation software) and a *Dongle,* which is a collection of scripts facilitating communication between the two servers. The components of each server are described below.

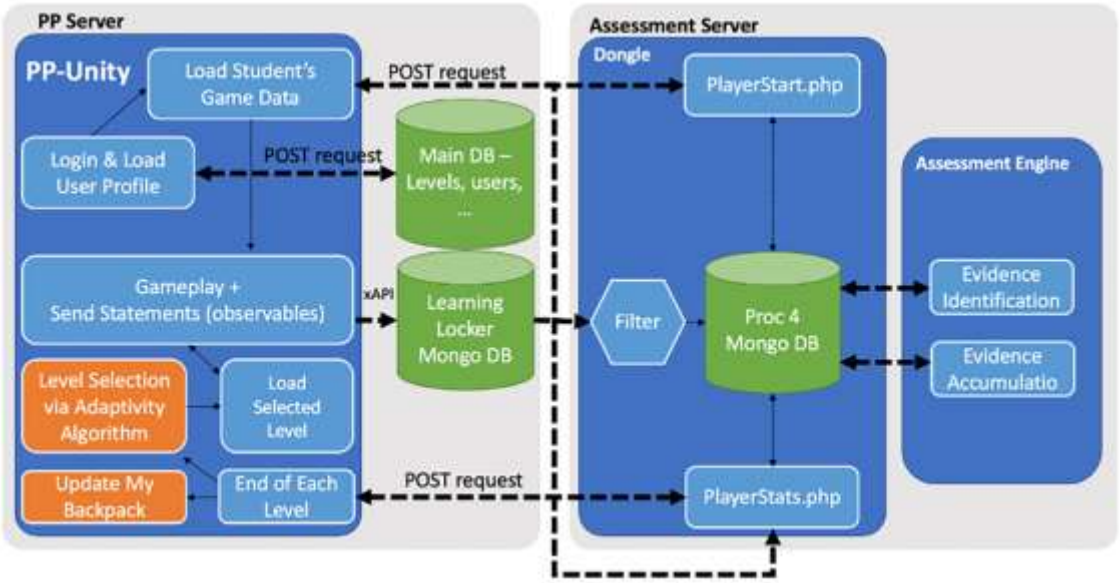

Figure 11.3: *Physics Playground*'s architecture

PP Server

Programming Languages and Platforms. The *PP* server's components are written in and managed by the following programming languages and frameworks. First, C# Unity (Unity Technologies, 2021) is used for developing *PP*'s *game engine*, *level editor*, and the communication processes among the game engine and other external components (e.g., the assessment server and learning supports per level). Second, PHP, HTML, and Yii, are used for writing the *Admin site*, which can be seen as a window to our main database, storing information about game levels, users, and user profiles (discussed later). Third, JavaScript is used for adding browser-based functionalities (e.g., showing a YouTube video on top of the game in the browser in a way that the learners feel the video is being shown to them inside the game environment).

Finally, the Experience Application Programming Interface (aka, xAPI; Betts & Smith, 2018) is employed to write statements in the game engine and send them into the log files residing on the Learning Locker (i.e., the log file platform).

The Production and Operation Process. *PP* is a dynamic game where most of its content can be modified from outside of its game engine without the need to write new code in Unity. That is, information about the levels (e.g., background image, positions of the ball and the balloon, and colors of the objects) dynamically get fetched from a database to the game environment. While this makes the programming harder, making changes to the game for non-technical researchers or future users (e.g., teachers) is easier. As shown in Figure 11.4, *PP*'s level production-operation process involves five components: (1) the level editor, (2) the admin site, (3) the main database, (4) the game engine, and (5) the log database—Learning Locker.

In general, *PP*'s level production-operation cycle (Figure 11.4) is as follows. A level is first created using the level editor, then the level is uploaded to the admin site and simultaneously stored in the main database (where the admin site is a window to the main database). When a learner signs in to play the game, the game levels data assigned to the learner's user ID is requested from the main database. As the learner interacts with the game, the game records events describing all learner-game interactions into the Learning Locker, the log database.

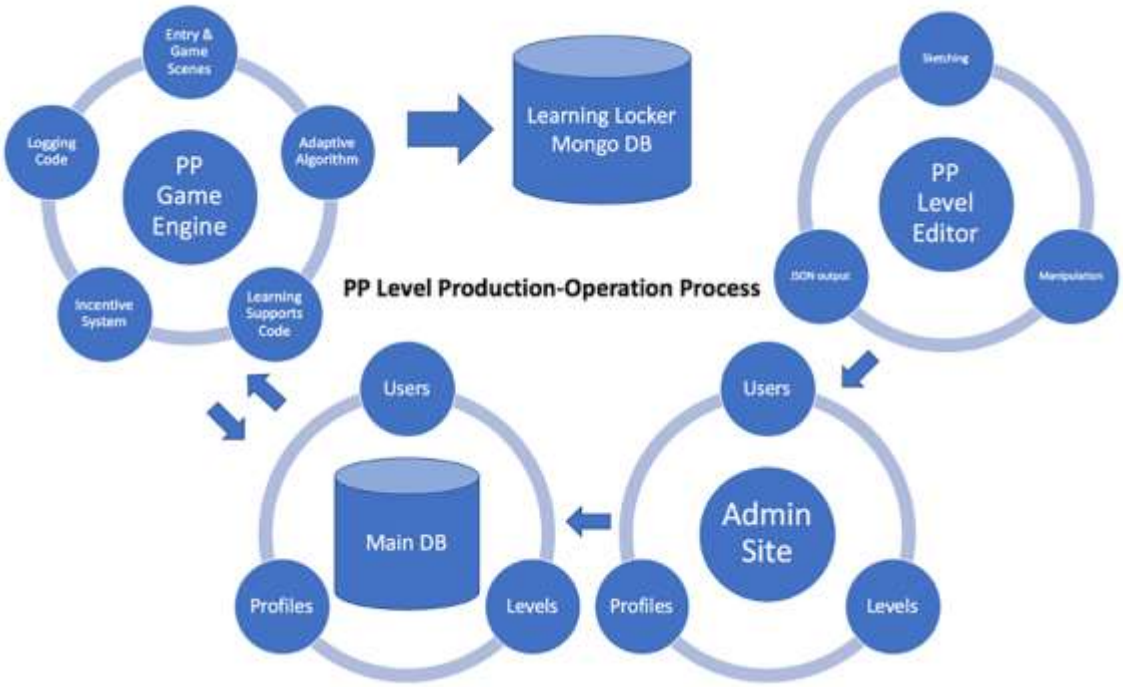

Figure 11.4: *Physics Playground*'s level production-operation process

The Level Editor. *PP* supports two different types of levels: *sketching*, where learners can draw objects (e.g., ramps and levers) to bring the ball to the balloon (see Figure 11.1a); and *manipulation*, where learners change three physics variables (i.e., mass, gravity, or air resistance) or manipulate a blower or a puffer to alter the path of the ball to the balloon (see Figure 11.1b). Using C# Unity, we developed a level editor (Figure 11.5) that allows non-technical users to design new levels. To create a level, the level designer selects the level type (i.e., sketching or manipulation), drags and drops the ball and the balloon to desired positions on the screen, uses the tools available (shown on the top left corner of the screen in Figure 11.5) to draw dynamic or static objects (with various colors) between the ball and the balloon. When designing manipulation levels, the level designer can add other objects (e.g., blowers and puffers) and set the initial values for the mass, gravity, and air resistance sliders. Once the level designer is satisfied with the game level, saving it produces a JSON file that contain the properties and

positions of the objects, and a thumbnail-size picture of the level created. To add the new level to the system, these files are uploaded to the admin site and stored in the database. Later, when a learner is assigned to play that particular level, the game engine requests that level's data from the database and sends it the learner's browser to set up the game.

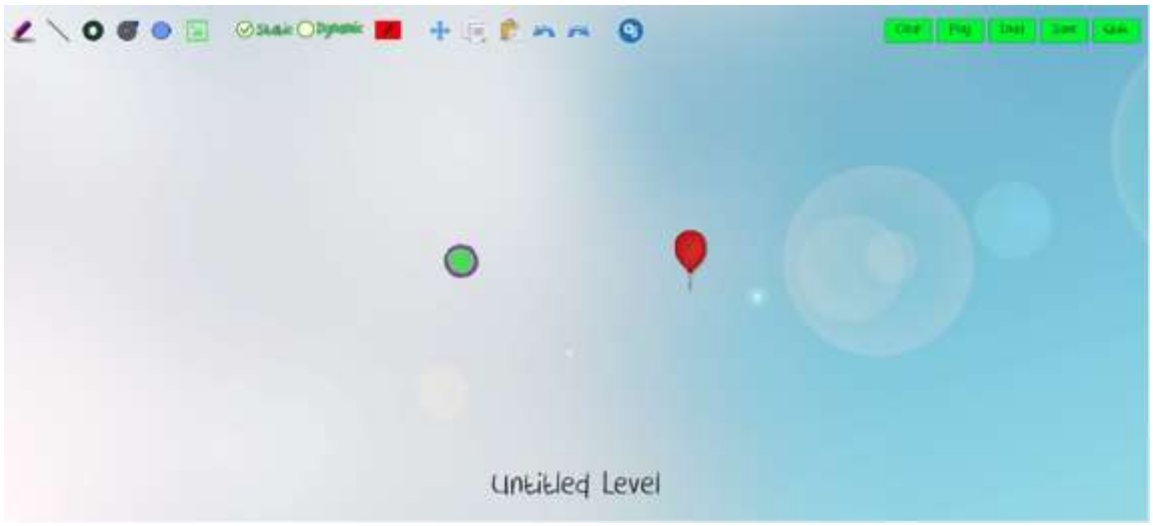

Figure 11.5: *Physics Playground*'s level editor

Admin Site. The admin site (Figure 11.6) is a tool used to manage *PP* level collections and users. This website is written using Yii (a PHP-based framework) and provides the following functions: (1) uploading levels—the levels created in the level editor can uploaded to the admin site along with learning support information (e.g., levels' learning supports), (2) creating profiles—collections of game levels (playgrounds) and rules for moving between them, and (3) registering users—creating user names and passwords for learners and assigning a profile to each user. Updating a profile in the admin site generates a new JSON file (stored in the main database) which contains all the needed information for a profile (i.e., list of levels and each level's data) so that it can be sent to the Unity engine as a package for each learner when they log in. That is, when a user signs in using the username and password created for them in the admin

site, the server sends the game profile assigned to that learner to the game engine; so, the learner can play the assigned game levels.

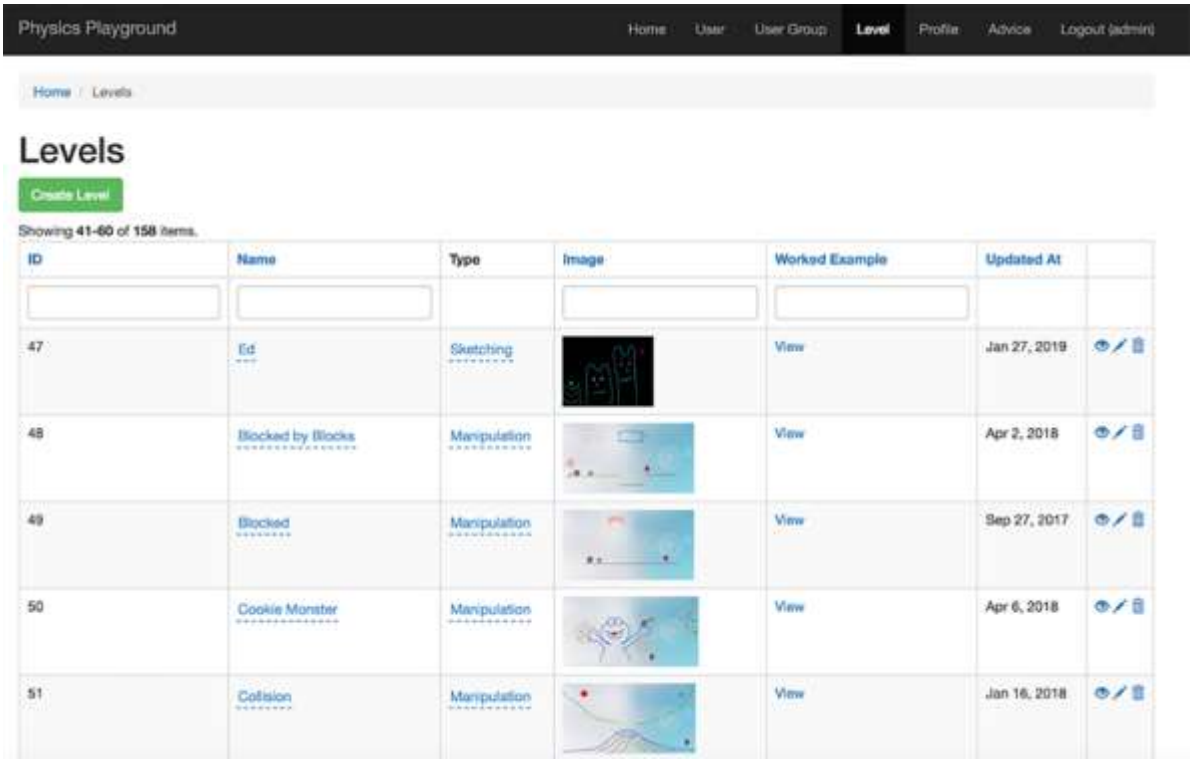

Figure 11.6: Physics Playground's admin site

The modularity and dynamic nature of *PP* serves two main purposes. First, a single game URL points to unlimited versions (profiles) of *PP*. That is, learners using the same URL to log in, might see different set of game levels as they might have different profiles assigned to them. This functionality allows researchers to easily create different versions of the game, and randomly assign learners to different conditions (via learner profiles). If teachers use *PP* in their class, they can assign their students to different collections of levels that focus on particular content, without the need to create multiple versions of the game. Moreover, the admin site ensures the integrity of the game configuration data in the main database. The stealth assessment designer will be prompted to include necessary supporting information and will not be allowed to create profiles that reference incomplete levels.

All the admin site data is stored in a SQL database (see Geschwinde & Schoenig, 2016 for an introduction to SQL). *PP*'s SQL database includes various tables related to *levels*, *users*, and *profiles*. The designer inserts data to tables through the admin site (e.g., when creating a new user ID) and the *PP* game engine queries the SQL database for username authentication, and for profiles and level data. The table for levels, for example, includes level information (e.g., level name, type, and difficulty) all of which gets stored using the admin site. Also, the JSON file exported from the level editor (i.e., including the position of the ball, balloon, and other objects on the screen) gets uploaded to the SQL database for each level as one variable. Similar to the levels table, the users table stores information related to learners (e.g., user ID, profile number).

PP Game Engine. Using the C# Unity programming language, we developed a game engine consisting of three main scenes (i.e., the environment where the game developers add game content and write code; a game can have multiple scenes): The first is the *Entry scene*, which is responsible for communicating with the main database and authenticating users' information. What the learner sees in the *Entry scene* are two text-entry fields for username and password. The *Entry scene* is linked to the code responsible for communicating to the SQL database to load all data for all levels, and communicating with the assessment engine for the learner's previous data (if applicable). If the learner has data from their previous gameplay sessions, the latest estimates of their physics understanding gets loaded to the game (discussed in more detail when we describe the components of the assessment server). The second is the *Menu scene* which is responsible for arraying the level's thumbnail pictures which allow learners to navigate through the playgrounds and choose a level to play. The third is the *Game scene* which includes multiple code files responsible for loading levels on the screen, applying physics engine laws (e.g., when an object is drawn, it will fall due to gravity), providing learning supports, and

providing a place where learners can check their progress and customize game options (i.e., via options available in *MyBackpack*). Also, the *Game scene* is linked to global code (accessible in all the scenes) which takes care of gathering the log data (i.e., important learner interactions with the game) that are sent back to the server to be cached in the Learning Locker.

The Log Files Structure. *PP* logs its gameplay data using Learning Locker which is a Learning Record Store (LRS). Learning Locker maintains collections of *statements* (event records) in the xAPI format. An xAPI statement consists of: actor (i.e., user), verb (i.e., event), object (i.e., an object that the event is linked to), and extensions (which is a place for inserting extra data related to the event at hand—e.g., the coordinates of the object just drawn). Learning Locker uses MongoDB, which is a document database storing data in a JSON format. The vocabularies for possible "verbs", "objects" and the associated extension data are not specified by xAPI, but rather by each project that uses the xAPI format.

A key challenge is deciding which specific activities in the game to log. Too little detail means that key features that provide evidence of targeted competencies will not be captured. But too much detail will negatively affect the performance of both the game and the scoring engines. The key to getting the logging system right is to look forward to the next scoring step—*Evidence Identification.* The output of evidence identification is the assignment of values to a collection of variables (associated with each game level) called *observables.* An observable plays one of four roles in the stealth assessment. That is, an observable is used: (1) as input to the *Evidence Accumulation* process (used in scoring), (2) to trigger feedback (e.g., show a video related to a lever if the learner did not draw a lever), (3) for research purposes (e.g., what percentage of time did learners spend viewing learning supports), or (4) to calculate other variables (e.g., if an observable is whether the player spent more time using learning supports than playing the game,

the system must first calculate the amount of time spent with learning supports and then game play).

In principle, the game engine focuses on logging data and leaves the interpretation of the data to the assessment engine (i.e., through the evidence identification and accumulation processes). In practice, though, sometimes interpretation is best done in the context of the game engine. In *PP*, the system which identifies whether what the learner drew was a simple machine (i.e., ramp, level, pendulum, or springboard) was written as part of the game, not the assessment engine. The output of the identification was logged as a message ("identified machine" with data indicating which machine). Other critical messages include the start and stopping point of each level (and which coin was won) and the starting and stopping point of learning supports (important for many learning details). After identifying what needs to be logged and where the evidence identifications need to be done, various xAPI-compliant functions are written in the game engine inside the *Player.cs* code that are called when those events occur in the game (e.g., when a level was solved and a coin was achieved). These events are sent in the form of xAPI statements to the Learning Locker. Although putting evidence identification code into the game engine can be more efficient, allowing separate evidence processing on the event logs allows new observables to be added even after data collection.

Learning Locker. Learning Locker is a Learning Record Store (LRS) which stores statements generated by the xAPI-based learning activities (e.g., gaming interactions). Figure 11.7 illustrates a graph in LL from the number of logged statements from October 21st to October 28th in 2019.

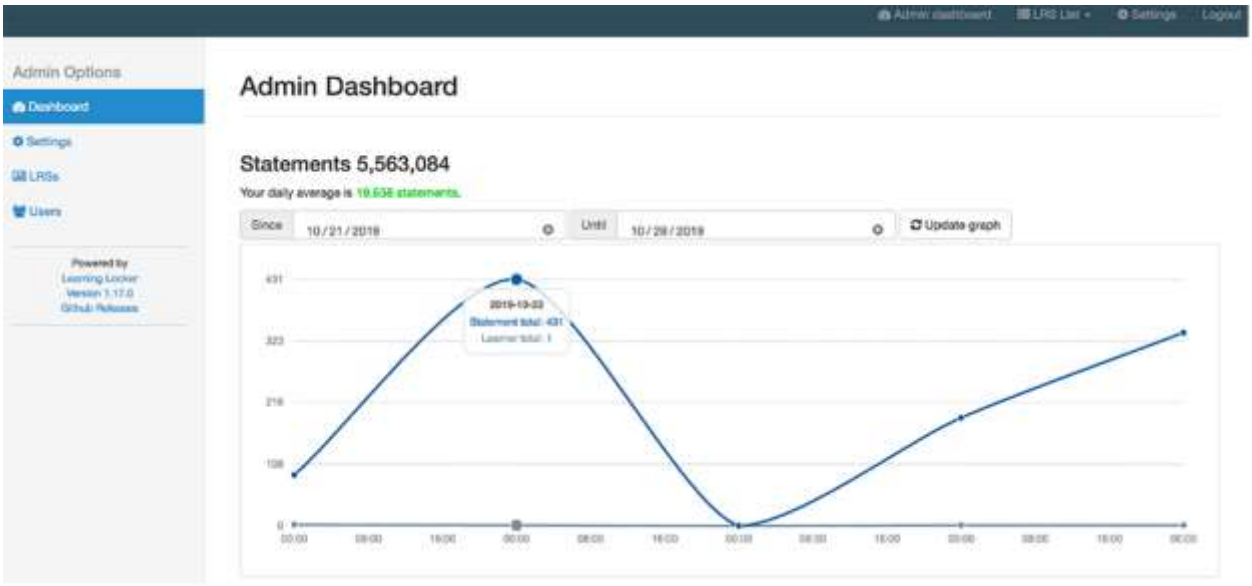

Figure 11.7: Learning Locker interface

The main advantages of using Learning Locker are: (a) buffering between the web server and the database so gameplay is not slowed by logging operations; (b) the live streaming and monitoring features which allows researchers to identify which learners are playing and who may need help logging in; and (c) data extraction tools which quickly return log subsets based on date ranges. For more complex queries, the underlying database supports a sophisticated query language and JSON export facility. And although it takes some time to figure out how to write queries against data in the xAPI format, once mastered it allows for more complex queries. A viable alternative to using Learning Locker would be to directly have the game server log event messages (in any desired format) to a database (possibly a document database like Mongo DB), but xAPI was at least a useful starting point for thinking about what to log.

Another challenge when using xAPI has to do with a lot of redundant meta-data which is included in every statement (event record; see Almond et al., 2020). In particular, actors, verbs and objects are all specified using URI-like globally unique identifiers (guids) to allow the same name to be used to mean different things in different projects. Almond et al. (2020) have

proposed a simpler language for events called *Proc4* which only uses long guids for the application ID, the other fields are assumed to be drawn from a vocabulary defined by the application. The last step of the logging process involves a small script (i.e., the *filter* in Figure 11.3) that runs periodically and (1) grabs all Learning Locker statements posted since the script last ran, (2) translates them into Proc4 event records, and (3) adds them to the input queue in the Evidence Identification database.

Assessment Server

Almond (2020) describes a streaming process of scoring. First, the evidence capture (EC process or game engine) collects a series of events (xAPI statements); these are sent to the input queue of the evidence identification (EI) process. Next, the EI event process events for each learner until it reaches the end of a task (for *PP*, this was defined when the learner started a new level or exited the system); it then creates a new record of the observed outcomes for that level and posts it in the input queue of the evidence accumulation (EA) process. The EA process (in *PP*, the Bayes net engine) updates the learner model for the learner and posts new scores (i.e., statistics of the learner model) to the output queue. In the case of an adaptive assessment, the output queue of the EA process would be collected to the activity selection (AS) process.

For *PP* all queues were implemented as a collection of records; one for each supported process. Each record has a time stamp and a *processed* flag; note the next message in the queue is always the oldest unprocessed record. This allows the system to both keep track of scoring progress, but allows for easy rescoring (simply mark all records as unprocessed). In one of our big field trials, we assigned a unique application ID to each of the three study conditions. The EI

and EA process for each “application” was run on a different processor, allowing the scoring engine to take advantage of multiprocessing to improve throughput.

The Dongle. To make the game real-time, it is essential that when the game engine queries the scoring server to find out information about the learner it returns quickly. Internet connections always involve some waiting time, but it is important that the game does not freeze while waiting for the scoring server to finish scoring. To this end, we provide a lightweight *dongle*, an adapter between the database and the game which would execute simple web queries.

The key to the dongle design is that the main scoring database always contains the most up-to-date information about the learner available. If the scoring has not finished for the last level when the query comes from the game, the database will still have the results from the previously scored levels (note that when a learner first registers with the scoring database, initial records with default statistic values are saved in the database). Then a lightweight common gateway interface (CGI) script can be added to the browser to query the database for the latest record. The CGI scripts will be blocked by database writes and excess server traffic, but not by waiting for scoring processes to finish.

The *PP* implementation used two such scripts. *PlayerStart.php* runs when the learner (player) logged into the game. If the learner had previously played the game, then the recorded information from the previous sessions would be returned (i.e., levels played, coins collected, and money balance for the learner). If the learner was new, then new default records would be created for the learner. *PlayerStats.php* provides the current estimates of the learner’s competencies from the learner model statistics stored in the database; these were used to populate the scores for the nine physics concepts and overall physics understanding, and were displayed in the *My Backpack* dashboard.

***Evidence Identification*.** The EI process is responsible for task level scoring. The EIEvent system (Almond, 2021) uses a combination of rules and a finite state machine to process the events (Almond et al., 2020). When the EI process receives a level start event, it initializes the record for that learner. As the EI processes each event, it updates the appropriate fields in the learner record. For example, moving the gravity slider adds to a count of slider movements, earning a coin sets the value of the coin earned field, starting a learning support starts a learning support timer, and exiting the learning support stops the timer. When the EI process receives an event indicating that the learner has started a new level, it posts a message containing the observable for that level to the EA process input queue.

The logic for processing an event is given as a series of "if-then" rules expressed as JSON objects (Almond et al., 2020). These rules are stored in the database, with a separate collection of rules kept for each application. This allowed the rules to be updated later when new observables are added for research purposes or when the definitions of the observables need to be revised. The rule sets are maintained in a source code repository (github) and then transferred to the server for scoring runs.

In the scoring implementation, it turned out that there were a lot of events which would not trigger scoring rules (e.g., movements of the ball and other objects on the screen). As filling the database with these unused events slowed down queries, the filter part of the dongle was expanded to drop the unused events before they were added to the EI process queue. This increase scoring speed considerably. The complete implementation of the *PP* scoring engine is available (Almond, 2021), but the current implementation is slow and not designed to take advantage of parallel processing to speed up scoring.

Evidence Accumulation. The EA process was based on Bayesian networks (Rahimi et al., in press, chapter 12 in this book, describes the construction of the model). The EABN package (Almond, 2021) is a scoring engine in R (R Core Team, 2021) which uses Netica (Norsys, 2021), to perform Bayesian network calculations. The basic algorithm used for EA is laid out in Chapter 14 of Almond et al. (2015). Note that the learner model in this approach is a Bayesian network. After each game level, the EA process provides an estimation for each competency variable based on the evidence accumulated so far by learner's gameplay. When the learner starts to play, the learner model (aka, the student model) is initialized with a model based on some predefined specifications by the subject matter experts. As learner-specific evidence is added to the model, it tracks what is known about that specific learner. Statistics of this learner model—specifically, the learner-specific probability distributions over the possible states for the competency variables—are posted to the database after the observables from each game level are absorbed into the learner model. Thus, when the game engine queries the scoring server (often for the purpose of displaying learner progress; e.g., through *My Backpack* in *PP*) it will return the statistics from the learner model after the last scored level.

My Backpack

We designed a multipurpose dashboard in the game called *My Backpack* where learners can (1) see their current estimates of mastery related to the targeted physics concepts (Figure 11.8), (2) buy new background images, background music, and different ball faces using the coins earned in the game (Figure 11.9), and (3) see their progress in the game, including levels solved and coins earned. Each gold coin (given for an elegant solution which included a certain number of objects in their solution) earns the learner $20 in the game, and each silver coin (given

for a solution which did not meet the criteria needed for a gold coin) earns $10 in the game. Learners can use their game money to purchase items and game customizations in *PP*'s store.

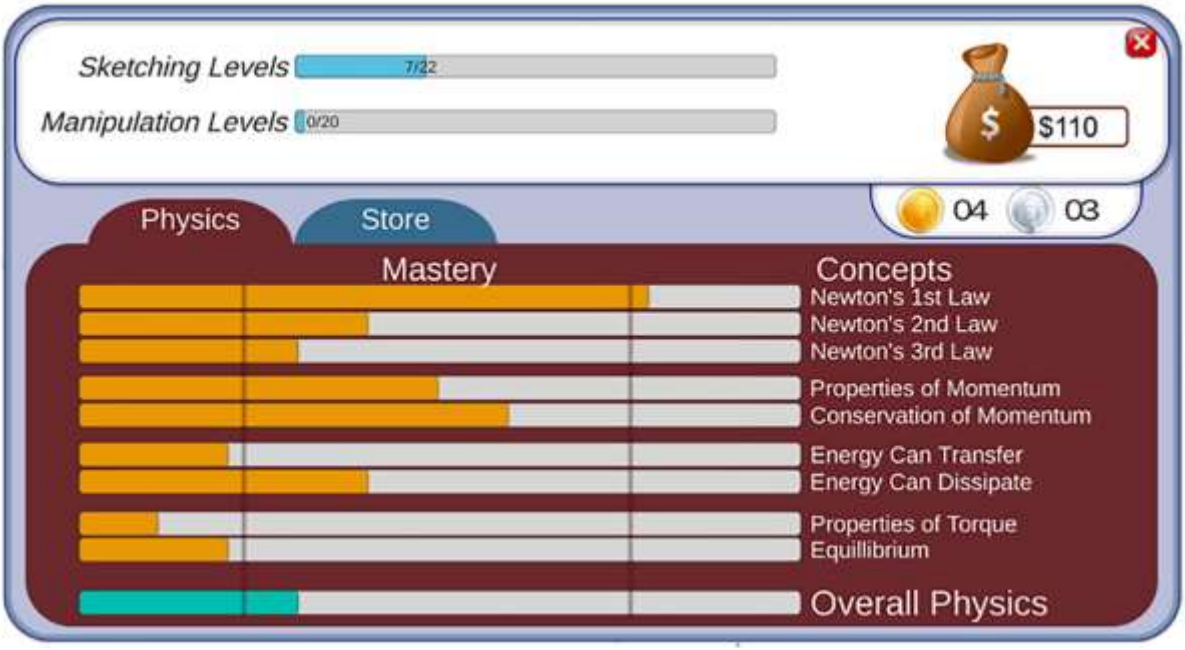

Figure 11.8: *My Backpack*'s physics tab with indicators of learner's level of competency

In *My Backpack*'s store (see Figure 11.9), learners can spend the in-game money that they earned through gameplay to customize their game aesthetics.

Figure 11.9: Game store in *My Backpack*

Lessons Learned

Test, Test, and Test Some More

One of the most important steps in designing a dependable stealth assessment system is to make sure all the needed information is being properly logged. In one study (Shute et al., 2020), we found a peculiar phenomenon that forced us to do a lot of post hoc work to clean the data. Specifically, we created three versions of the game: *adaptive* where the next level was shown to

the learners using an adaptive algorithm, *linear* where the learners would go to the next level based on a predefined order, and *free choice* where learners could choose any level to play next. There were no issues regarding the log flies for the logs coming from the free choice condition as the end-of-the-level events had more pauses between them when sent to the LL. However, the other two conditions generated an unexpected bug. At the end of each level, several functions would send various events to the LL (e.g., level solution status, coin earned, money earned). In the adaptive and linear conditions, these events seemed to reach the server later than initializing the next level. The main reason for this discrepancy was related to the way a learner would move from one level to another in these conditions. In the free choice condition, learners would come to the main menu and choose the next level, thus allowing more time for logging the events properly; whereas in the linear and adaptive conditions, the learners would jump to the next level as soon as they solved or quit the current level causing the bug discussed above. Therefore, we would see some event that belonged to the previous game level in the logs related to the next level. To fix this issue we had to write a code to manually adjust the events to the right order.

To avoid such situations, it is recommended to test the logging system multiple times and under all probable cases to make sure the system works properly. Moreover, if the games are web-based, it is possible that the browser or network issues can produce an error that blocks the log events from being sent to the server. For example, in our studies, we used a JavaScript code to show the learning support related to a game level on the screen. When we tested our game using an internet service without any restrictions, everything worked as planned. However, we identified an issue when we ran our study in a school which restricted learners from accessing YouTube (we hosted our learning supports on YouTube). To avoid this issue, you might want to host the videos you will use on a server that can be accessed without any restrictions in schools,

or coordinate with the schools or district Information Technology team to allow certain links from YouTube to pass through the school firewall of the school.

Backup and Update at the Same Time

As almost all professional coders use some sort of *git* (i.e., a remote repository to back up and keep track of changes made to their code; see https://git-scm.com/ for more information), they can use the same mechanism to update the game server. Git is also used for working on a programming project as a team. That is, a group of programmers can collaboratively develop a program during software development. We used git for keeping track of our source code and other files, backing up our projects, and updating our server. Figure 11.10 shows how we set up our local and cloud-based, git repositories to generate, backup, and update our source code as well as our final product—i.e., *PP*.

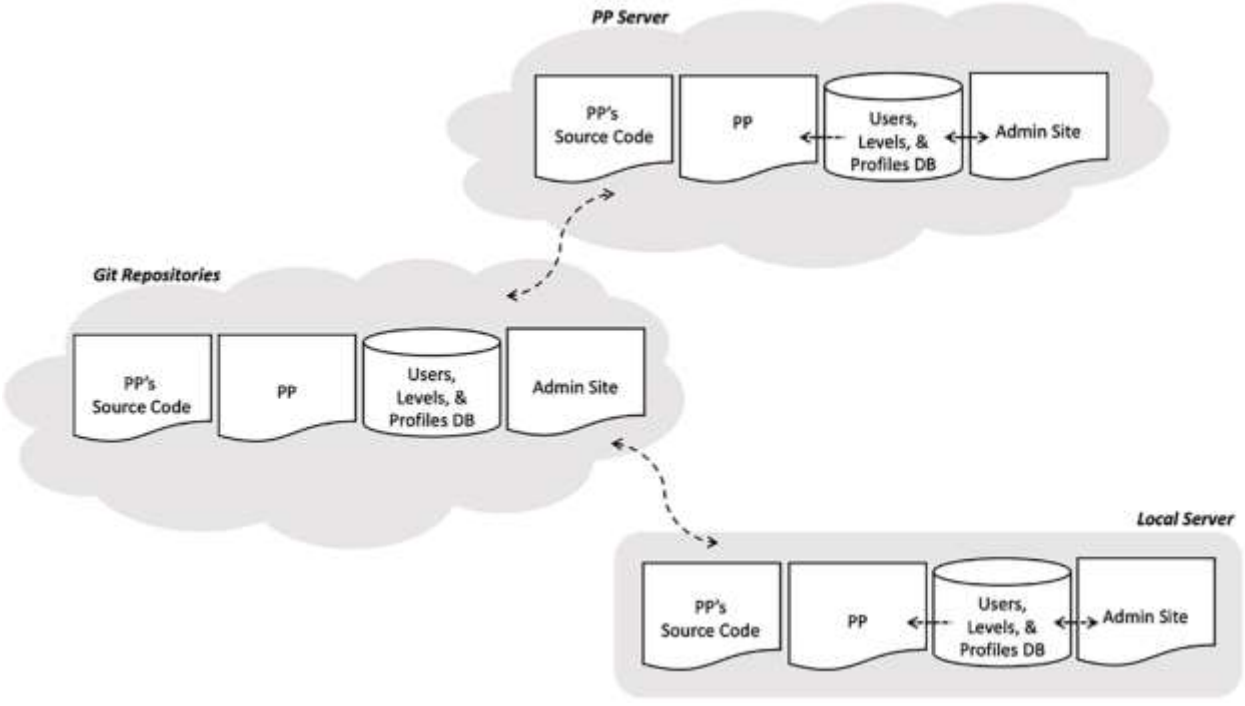

Figure 11.10: *PP*'s backup and updating mechanism

First, we created the remote repositories on a git system. Then, we connected our local server to those git repositories and then *pushed* our local server's content to the remote repository. Next, we connected the remote repository to our *PP* server and *pulled* the remote repository's content to our server. *Push* and *pull* are common git commands that can be used to send or receive content (e.g., any change done to a folder locally) to a remote repository. This allowed us to revert development changes if needed using a couple of git commands.

Note that the line between the local server and the git repository, and the line between the git repository and *PP* server in Figure 11.10 is bi-directional indicating that both the local and the *PP* servers can push to and pull from the git repository. However, most of the time we push content from the local server to the git repository; and pull content from the git repository to the *PP* server. Also note that there are directional lines (one and two-way) between the Admin Site, *PP*, and the Users, Levels, and Profiles database in the local and *PP* server but not in the git repository. The git repository just stores the components mentioned above—there is no internal communications in a git repository; hence the git repository is not active. However, on the local server, those components (i.e., the Admin Site, *PP*, and the Users, Levels, & Profiles database) become active. For example, learners' usernames, passwords, and their profiles are authenticated and sent to *PP* when *PP* sends a POST request to the Users, Levels, and Profiles SQL database—this request never happens on a git repository.

Conclusion

To be able to accurately measure learners' knowledge and skills in real-time, we designed and developed a complex architecture that can store, retrieve, and analyze learners' interaction data. This architecture works as a dynamic system with various interactive components to make such a

goal happen, adaptively assessing and enhancing learners' knowledge and skill acquisition. What we described in this chapter is not the only way to create a game with stealth assessment. For example, embedding stealth assessment into an existing game (e.g., Shute, Wang, et al., 2016; Smith et al., in press). The software architecture of those types of stealth assessment could be different from what described here. For instance, one could include the assessment engine in the game engine rather than including it as a separate system in a separate server. Each of these design decisions depend on the situation that assessment designers are dealing with.

Creation of a complex architecture such as what we described in this chapter needs multiple experts from multiple disciplines. However, we believe and hope to see some innovations in this area, which people can create games with stealth assessment in an automated manner which does not require a lot of programming. That is, reusable, programmed pieces that can go into the architecture of multiple games to make a stealth assessment of a particular competency possible.

References

Almond, R. (2021). *Proc4, EIEvent and EABN*. https://pluto.coe.fsu.edu/Proc4/

Almond, R. G. (2020). Scoring of Streaming Data in Game-Based Assessments. In *Handbook of Automated Scoring*. Chapman and Hall/CRC.

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*, (online).

Almond, R., Shute, V., Tingir, S., & Rahimi, S. (2020). Identifying observable outcomes in game-based assessments. In *Innovative Psychometric Modeling and Methods* (pp. 163–192). Information Age Publishing.

Betts, B., & Smith, R. (2018). *The leraning technology manager's guid to xAPI* (Computer software manual) [Computer software].

Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience*. Harper and Row.

Geschwinde, E., & Schoenig, H. (2016). *An introduction to SQL*. https://www.informit.com/articles/article.aspx?p=25959

Norsys, Inc. (2021). *Netica Application*. https://www.norsys.com/netica.html

Rahimi, S., Almond, R. G., & Shute, V. J. (in press). Getting the first and second decimals right: Psychometrics of stealth assessment. In M. P. McCreery & S. K. Krach (Eds.), *Games as Stealth Assessments*. DOI press.

Shute, V., Almond, R., & Rahimi, S. (2019). *Physics Playground* (1.3) [Computer software]. https://pluto.coe.fsu.edu/ppteam/pp-links/

Shute, V. J., Leighton, J. P., Jang, E. E., & Chu, M.-W. (2016). Advances in the Science of Assessment. *Educational Assessment*, *21*(1), 34–59. https://doi.org/10.1080/10627197.2015.1127752

Shute, V. J., & Rahimi, S. (2017). Review of computer-based assessment for learning in elementary and secondary education: Computer-based assessment for learning. *Journal of Computer Assisted Learning*, *33*(1), 1–19. https://doi.org/10.1111/jcal.12172

Shute, V. J., Wang, L., Greiff, S., Zhao, W., & Moore, G. (2016). Measuring problem solving skills via stealth assessment in an engaging video game. *Computers in Human Behavior*, *63*, 106–117. https://doi.org/10.1016/j.chb.2016.05.047

Shute, V., Rahimi, S., Smith, G., Ke, F., Almond, R., Dai, C.-P., Kuba, R., Liu, Z., Yang, X., & Sun, C. (2020). Maximizing learning without sacrificing the fun: Stealth assessment, adaptivity and learning supports in educational games. *Journal of Computer Assisted Learning*, *37*(1). https://doi.org/10.1111/jcal.12473

Smith, G., Shute, V. J., Rahimi, S., Kuba, R., & Dai, C.-P. (in press). Stealth assessment and digital learning game design. In M. P. McCreery & S. K. Krach (Eds.), *Games as Stealth Assessments*. DOI press.

Unity Technologies. (2021). *Unity - Manual: 3D.* https://docs.unity3d.com/Manual/Unity2D.html

Vygotsky, L. S. (1978). *Mind in society: The development of higher mental processes*. Harvard University Press.