Talin: A Framework for Dynamic Tutorials Based on the Skill Atoms Theory

Batu Aytemiz, Isaac Karth, Jesse Harder, Adam M. Smith, and Jim Whitehead
Design Reasoning Lab
University of California, Santa Cruz
{baytemiz,ikarth,jhharder,amsmith}@ucsc.edu, ejw@soe.ucsc.edu

Abstract

Most tutorials in video games do not consider the skill level of the player when deciding what information to present. This makes many tutorials either tedious for experienced players or not informative enough for players who are new to the given genre. With Talin, implemented as an asset in the Unity game engine, we make it possible to create a mastery model of an individual player's skill levels by operationalizing Dan Cook's skill atom theory. We propose that using this mastery model opens up a new design space when it comes to designing tutorials. We show an example tutorial implementation with Talin assembled using only graphical components provided by our framework, without the need of writing any code. The dynamic tutorial implementation results in the player receiving information only when they need it, whenever they need it. While the novice player is given all the information they need to learn the system, the expert player is not bogged down by tooltip pop-ups regarding mechanics they have already mastered.

Introduction

The opening minutes of any video game, which often take the form of a tutorial level, are extremely crucial. It is critical that the tutorial captures the attention of the player and teaches them how read the game's visual interface and usefully act in the game world. If the pace of this initial level is either too slow or too fast the player loses interest: tutorials have a major impact on player progress for games with unfamiliar mechanics, particularly when the tutorials are context-sensitive. At the same time, tutorials can be detrimental in games where players are already familiar with the genre (Andersen et al. 2012). Therefore, it is in the designers' interest to have tutorials that can provide context-sensitive help when the player is unfamiliar with the game but also completely disappear when the player is comfortable.

In this paper we introduce the Talin framework,[1] in the form of an asset for the Unity game engine,[2] to help designers implement adaptive tutorials without the need of any programming skills. Tutorials built with our framework continuously monitor evidence of player skills and act in the game world to give targeted feedback just where needed. In effect, they can bring some of the designer's intelligence out of the studio and apply it during play-time.

Not everyone requires an explanation of equivalent depth when it comes to learning new game concepts – not every player has identical gaming literacy. An experienced player will complain if the game locks away the interesting parts behind a series of simple tasks, such as camera control lessons, whereas a person new to video games will feel lost if not enough time is spent teaching the very basics. Rather than binning players into coarse classes (e.g. newbie versus expert) or projecting them onto a single axis of skill, Talin-supported tutorials can distinguish players along many dimensions, supporting players with distinct constellations of previous experience.

Meanwhile, in many video games, the tutorial is not integrated into the whole experience but rather exists only in the first few levels, or pops into view whenever a new mechanic is introduced. This front-loaded structure makes it difficult for players to remember the components of the game if they ever take a break or if a friend who was just watching the earlier interaction grabs the controls later on. When a player returns to any game after a hiatus, the natural instinct is to pick the game up from where they left off. In most cases this means they will spend some time trying to remember how to interact with the system, making very little progress. In severe cases, they will have to go back and replay the tutorial levels they already completed to bring themselves up to speed. Neither of these circumstances are desirable. Talin-supported tutorials can allow tutorial content to be sprinkled throughout a game's level progression (or even baked into the behavior of common level design elements) with the knowledge that it will only be seen by players who need to see it.

Even if the player does not take a break, it is difficult to gauge the player's understanding of the more complicated concepts. The game designers have to expertly craft their levels to ensure the desired progression is accurately reflected in the player's experience. In most cases, the system has no explicit feedback acting as a model of a player's mastery over the skills, which makes it impossible to give personalized help. The designers have to work with a one-

[1] The public download URL for our framework is withheld for blind review.

[2] https://unity.com/

design-fits-all structure which inevitably loses some players through the cracks.

The Talin system provides a solution to these problems by offering a new way of designing tutorials and making it seamless for any designer to implement. The system achieves this by operationalizing Dan Cook's skill atoms theory to build a mastery model representing the player's current understanding of the game. The mastery model can be queried to decide which information the player is lacking at any given moment.

On the high level, the way the system tracks the user's skill mastery is as follows: The designer defines what *skill atoms* needs to be tracked, for example attacking breakable doors. The skill *mastery* for each skill is represented as a scalar.[3] Then the designer adds a *detector* to the breakable door object. The detector activates whenever the player is detected by the detector, in this case, when the player is near a breakable door. The detector will then decide how to *adjust* the skill mastery value, either increasing it or decreasing it. If the player is near a breakable door but they are not breaking the door, the skill mastery value will decay. If the player attacks the breakable door, the skill will be considered exercised and the value will increase. The designer can decide to react to the player based on the value of the skill mastery value by activating predetermined *hints*. For example, if the skill mastery value is below a threshold, the breakable door might start shimmering as a form of a subtle hint. If the value decays even further, a textual pop-up might appear reminding the button press for attacking. Through the combination of skill atoms (keeping track of individual proficiencies), detectors (understanding when a skill is relevant) and hints (the way the game responds) designers track and respond to individual players.

We built the Talin framework as a Unity asset and focused extensively on usability and expandability. Skill atoms, detectors, and hints can be configured using the engine's graphical editor and placed into useful locations in the 2D/3D scenes of a game. We understand that, in most cases, video game tutorials are designed and implemented by game designers who are not proficient programmers. In order to ensure the tool is accessible and easily introduced into the workflow, we structured Talin to require no programming expertise. We also ensured that the framework is easily extensible if a developer with expertise wants to use the system for more specialized purposes (such as game-specific detector logic). Later in this paper, we walk through a no-code implementation of dynamic tutorialization for the Unity-provided *2D Game Kit*[4] example game.

This paper operationalizes Daniel Cook's skill atom theory into a designer-friendly tool. With the Talin framework we hope to streamline the process of making personalized tutorials for game designers and developers alike. The addition of this type of adaptive tutorials should increase the player retention rate as every player could get a tutorial tailored to fit their knowledge level. Adaptive tutorials will also help designers ensure that the player is effectively utilizing all the tools they have created. Overall, we believe this framework can greatly improve the process of authoring personalized tutorials and help create better game experiences.

[3]The designer may interpret a value between 0 and 1 as a probability that the player has mastered a given skill, but the framework does not enforce this interpretation.

[4]`https://unity3d.com/learn/tutorials/s/2d-game-kit`

Background

Our approach to a tutorial system is based on the knowledge model of the player. The initial inspiration for the project was the skill atoms player model developed by Daniel Cook (2007). The player's model of the game system is disassembled into atomic components, individual skills that the player learns and combines to form more complicated skills. The player interacts with the game in a feedback loop, as the actions the player takes update the state of the simulation, which gives feedback to the player. The player uses the feedback to update their internal model of the game and uses that to inform their next decision. Each individual unit of action-simulation-feedback is a skill atom. In Cook's model, skill atoms can be chained together to describe more complex skills that are assembled out of multiple atoms: for example, learning how to stack blocks efficiently in Tetris depends on first learning the skills to move and rotate the blocks.

The skill atom model was originally developed as part of an effort to create a game grammar, and it has been adopted and adapted several times. For example, in the HCI field the model has been amended by Sebastian Deterding to consist of "goals, actions and objects, rules, feedback, emergent challenge, and motivation" and combined with "design lenses" and "intrinsic integration" to arrive at a theory of intrinsic skill atoms for "gameful design" (Deterding 2015).

Importantly for our approach, the model has also been inverted by Isaac Karth (2014), replacing the emphasis the original model puts on goals and with the idea of the aporia: the gaps in the player's understanding. Rather than viewing the player as an informed agent seeking a known goal, we model the player exploring an unknown system via the process of play. As the inexperienced player naturally has little knowledge of the possible future goals, this agency-and-play-centric model is a more natural fit for our system, foregrounding that the player's primary interaction is playing with the unknown.

Naturally, skill atoms are hardly the first attempt to model user knowledge. For example, a formal knowledge model of a user's understanding can be represented as a graph of the knowledge space (Doignon and Falmagne 2012). The introduction of Intelligent Tutoring Systems (ITS) was predicated on personal computers becoming fast enough to generate intelligent computer-assisted instruction based on cognitive science models of how a student acquires new cognitive skills (Anderson, Boyle, and Reiser 1985). In contrast, rather than being concerned with the formal accuracy of our player-knowledge model, our approach is oriented toward the process of designer-accessible model specification and the output of the dynamic tutorials in a widely adopted game making environment. In this sense, our work is related to the Vixen project (Drenikow and Mirza-Babaei 2017) which

aims to support visual analytics directly from within the Unity editor UI.

ITSs have been the basis for several tutorial generators. One recent example is the Thought Process Language (TPL). While TPL can "generate explanations for a given problem" it requires that an algorithm for solving the problem first be encoded in TPL (O'Rourke et al. 2015). Further, it assumes that the goal of the tutorial system should be to give the user complete understanding of a specific problem solving process. In contrast, our system is designed to be added to an existing game without reimplementing the gameplay elements, focusing on the elements of tutorial generation that are more important for games. Importantly, Talin is designed to suggest ways for the player to interact playfully with the game, not to instruct the player on the optimum solution.

For a game-specific approach, Michael Green et al. suggested ways in which AI techniques could be applied to the tutorial generation problem in the General Video Game AI (GVG-AI) framework (Green et al. 2018). Our goal differs from this project: rather than inferring the game rules that the tutorial explains, we assume that the developer is best equipped to manually specify the elements using our tutorial development toolkit. Our focus is on the dynamic presentation of the tutorial to the player, using the skill atom graph to dynamically focus the tutorial on only the elements that the player needs to have explained.

Another related research area is dynamic difficulty adjustment (DDA) (Hunicke 2005). While dynamic difficulty adjustment systems implicitly have a player model, they are more concerned with keeping the player in the flow channel rather than tracking which skills the player knows about or strategizing how to introduce an unknown mechanic to the player. Dynamic difficulty adjustment can be complementary to a dynamic tutorial system- perhaps the system displays hints before later escalating to adjusting the difficulty, while using the knowledge model to better track if the player is finding particular skills to be too frustrating or boring.

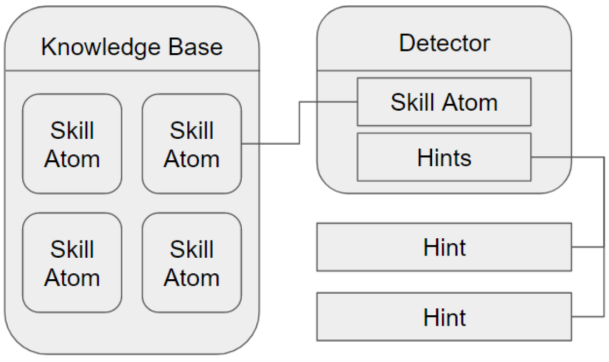

Figure 1: Overview of how skill atoms, detectors, and hints are structured.

Talin Technical Design

The Talin framework consists of three modular building blocks that the game designer manipulates to create the dynamic tutorials (Fig. 1). Skill Atoms capture a player's current mastery of a skill as a scalar value, Detectors decide when a skill mastery update is relevant, and Hints activate whenever the skill mastery value crosses a designated threshold.

Consider the mechanic of attacking a breakable wall in order to remove it. This is a game mechanic that can benefit from being taught in a dynamic fashion for several reasons: First, it is a common game mechanic that seasoned players will have internalized. Second, the mechanic affords a sense of discovery that could be undermined if a tooltip pops up and reveals the salient information too soon. Third, it is a context specific skill that is only applicable when a breakable wall is present. Due to the skill not being in the core gameplay loop, it can be easily forgotten when the player takes an extended break from the game.

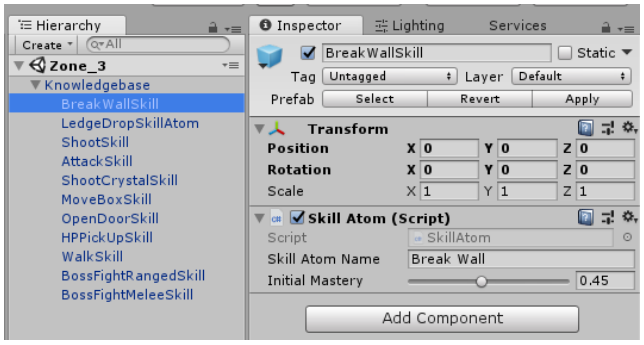

Figure 2: In-editor Talin designer interface showing how the Skill Atoms are initialized.

The implementation process starts with the designer creating the skill atom representing the BreakWall skill (Fig. 2). The designer then initializes the skill atom with a starting value that represents the initial skill mastery of the player. The designer is free in deciding how to initialize the skill mastery level. If they wish, they can assume prior knowledge of a common game mechanic, such as moving around and initialize the skill at a high value. Otherwise, if the skill in question is more domain specific, they can initialize the skill at a much lower initial value. The skill mastery value range is set to be between 0 and 1, where 1 represents total mastery of the skill and 0 represents lack of any understanding regarding the skill. Regardless of the initialization, as the player progresses through the game this skill mastery value increases or decreases to reflect the understanding of the player for the wall-breaking mechanic.

Not every skill that is required by the game is relevant throughout the experience. BreakWall skill is only of interest when the player is near breakable walls. This is where detectors come in. Detectors allow designers to specify when a skill atom is exercised (the atom's value is increased) or decayed (its value decreased). Designers can choose from several built in detectors covering the majority of simple cases.

Figure 3: Visual representation of the proximity detector, indicated by the green circle. The proximity detector is attached to the breakable wall, and linked to the BreakWall skill.

Those with programming experience can even extend an existing detector class to cover complex edge cases. The most commonly used detectors are proximity detectors and input detectors.

As the name suggests, the proximity detector allows the designer to check whether an object, often the player, is near the detector. For the BreakWall skill, a sensible option is to attach a proximity detector to the breakable wall object and decay the BreakWall skill whenever the player is within a set radius (Fig. 3). The proximity detector represents the idea that if a player is standing right next to breakable wall, and not breaking it, they might not realize that the wall is actually breakable. The longer the player actively lingers right next to the breakable wall without making any progress, the more likely it becomes that they need a hint to figure out that the wall is indeed interactable.

Yet, it is not enough to only decay the breakable wall skill value. The designer also needs a method to increase mastery level whenever the player shows that they know how to break a wall. To compliment the proximity detector, an input detector can be used. The input detector tracks whether a player successfully attacks and breaks a wall. Using the combination of the proximity detector and the input detector, the designer is be able to update the wall breaking mastery of the player simply by observing the play pattern. Due to the context specific implementation, the updates only happen when the skill itself is relevant. In order to allow designers to customize how the skill mastery values change we offer different ways of manipulating the value itself. The designer can pick between a linear change (increase/decrease by a constant amount), logistic change (reduce the distance to a target value by a constant fraction), or exponential change (scale the value by a constant fraction).

Hints are the structured way to create in-game events that trigger in relation to the given skill mastery level. A hint can be a simple tooltip pop up or even a sound cue. Or they can be much more intricate, resulting in substantial changes within the game systems. A hint activates when the mastery value of a skill crosses a designated threshold. It is up to the designer to define their own hints and at which mastery level the hints trigger. There is no limit as to how many hints can be attached to a particular skill atom. This allows the designer to combine different hints as they see fit. Using multiple hints with distinct thresholds, designers can start by introducing subtle hints and gradually become more explicit as the player shows signs of struggling.

Within the tool we offer several pre-built hints such as activating a particle system, playing a sound cue, or introducing a tool tip. For example, for the BreakWall skill, the designer can decide to activate a hint that creates a particle effect system emphasizing the cracks on the wall when the skill atom decays past a certain threshold. If the player stays within the radius of the proximity detector without breaking the wall even further, a second, more explicit hint can activate. Since the player missed both the cracks on the wall and the animated particles, the new hint has grounds to be very direct: a tooltip explicitly stating that the wall is breakable appears. For extreme scenarios, we can even imagine a hint implementation that assumes control of the player character to physically demonstrate the desired action (something the player might perceive as a cutscene).

Using the combination of skill atoms, detectors, and hints, a designer creates levels that adjust to the players. The skill atoms give the designers a method to keep track of what the player knows. The detectors allow the designer to manipulate the skill values. Finally, the hints introduce a convenient way to trigger user-defined in-game events.

Tutorial Implementation

Talin can be used without any programming knowledge. All of the skill atoms, detectors and hints can be implemented using the visual interface that Unity offers (Fig. 4). We believe it is crucial for the tool not to require programming to function. In most cases the person who is tasked with creating the tutorial is a game designer who might or might not be comfortable with programming. It is important that the tool offers enough flexibility out of the box to seamlessly adapt into the workflow of whoever is using it. If someone can create the rest of their tutorial scene with graphical scene editing tools, those tools should also serve them in making the tutorial dynamic.

In this section, we walk through a no-code implementation of dynamic tutorialization for the Unity-provided *2D Game Kit* example game. This example game includes an existing, static tutorial design that we adapt via integration with Talin-provided building blocks.

Referring back to our example, if a designer wants to implement the BreakWall skill they start by instantiating a Knowledgebase prefab[5] in the editor and attaching a skill atom GameObject to it as a child. The skill atom game object has input fields for its name and the initial skill mastery value. The Knowledgebase is simply a container for all skill atoms. Then, they will attach a proximity detector game object to the breakable wall prefab and link it with the Break-

[5]In Unity, prefabs are the editor's abstraction of game object templates. In a lower-level game engine, our framework primitives might be realized as base classes to be instantiated or extended.

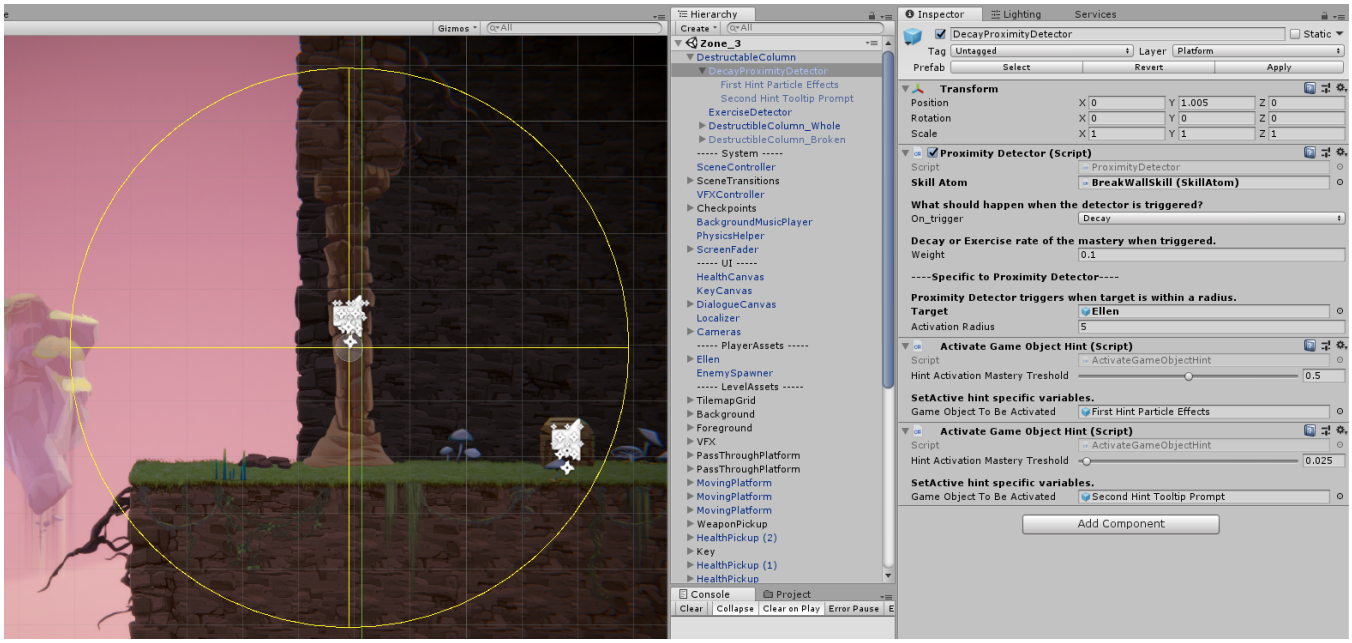

Figure 4: In-editor Talin designer interface. The proximity detector is displayed in the level editor view on the left as a yellow circle. In the center, the Detector is listed in the scene hierarchy, parented under the breakable object. On the right is the interface for the proximity detector properties, which allows the designer to completely specify the behavior without needing to use code.

Wall skill atom through the visual interface. The ability to add a detector as a child of another game object makes it so that only one change to a prefab propagates through all instances of said object throughout the game. Even if the tutorial is being implemented long after other levels are completed, implementing the tutorial at the prefab level makes it relatively easy to embed the dynamic tutorial all throughout the game with very little effort. After linking the skill atoms and the detector all that remains is to set up the desired hints. The hints are also added to the detector object as a script. This allows for the designer to have the ability to have different sets of hints at different detectors. In the case of the BreakWall skill, the designer can select the predefined hint that activates a game object, and through the visual interface set it up so that whenever the BreakWall skill decays under a certain threshold a tooltip gameobject activates.

While the simple workflow of Talin requires no programming knowledge, the system allows for easy expandability. If a developer wants to have a certain detector trigger in more specific scenarios than simple radius or input situations, they can extend any of the base classes to include the desired functionality. Consider a case where the designer wants to introduce a PickUpHealthPack skill atom. In this context, a standard proximity detector will not be sufficient. If the player is lingering next to a health pick up while they are not missing any health points, the designer can not assume that the player doesn't know how the health pack functions, as they might be deciding not to pick it up so as to save it for later. Thus, the proximity detector can be extended with a few lines of code to decay only when the person is near a health pick up and they are missing a certain amount of health points. While this is a trivial example, the extendibility of the base classes allows the developers to create game specific structures to support the type of tutorials they want to achieve.

The tool also comes with simple debugging support, including visualizations of the values of each skill atom and how they change as the player progresses.

Example Personas

Compared to the existing static tutorial in *2D Game Kit*, the dynamic tutorial made by Talin offers different experiences to different players (Fig. 5). For example the expert player starts the game and immediately knows how to navigate the space. Because they move around with ease, the input detector recognizes the player exercising the movement skill and thus, no hint regarding movement appears.

When the expert player reaches the first breakable wall segment, it takes only a few seconds for them to realize there are cracks on the wall sprite. At this point, due to their previous experience the expert player discovers that they can attack the wall to destroy it. Similarly, when they reach the second breakable wall several levels down the line, they do not need any support.

Overall, the expert player, due to never needing any help, never sees any of the hints. This allows them to discover some of the game mechanics themselves hopefully increasing their enjoyment.

Whereas an expert player doesn't need any help, a novice player might need more pointers than we expect. When the game starts the novice player spends some time exploring the controls but doesn't figure them out. The initial detector

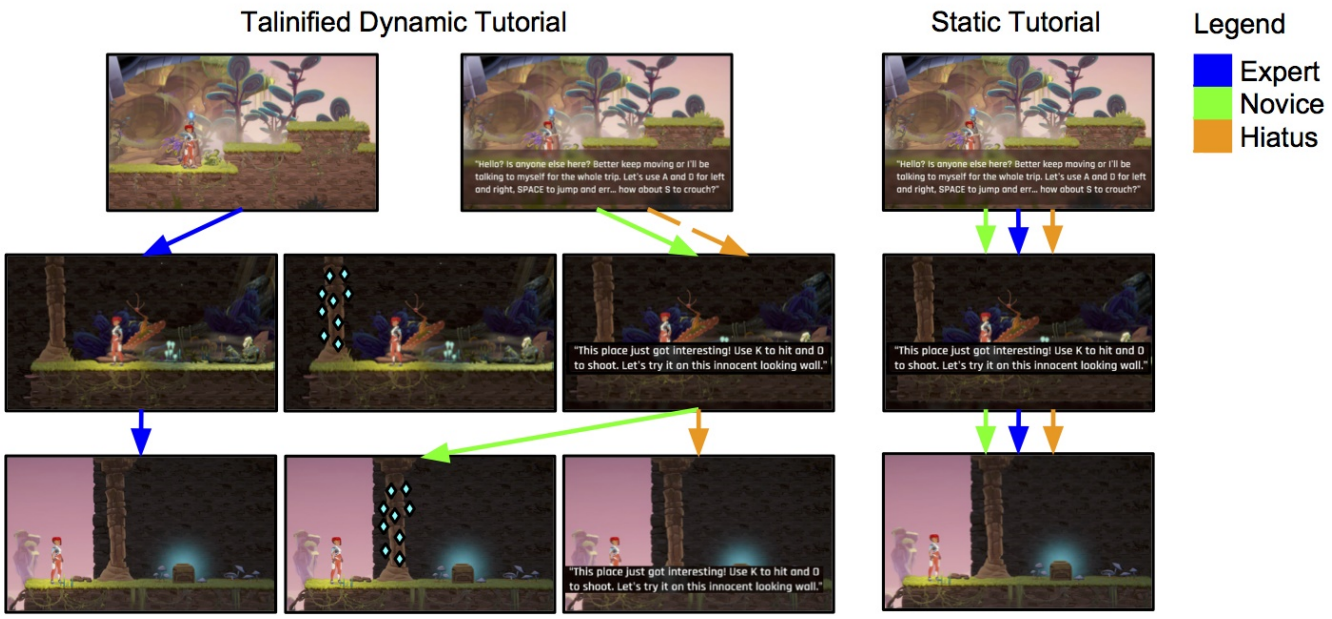

Figure 5: The experience of the different personas through different game moments, contrasting our dynamic tutorial experience with the traditional static tutorial.

tracks that the player is not making any progress and activates the first hint explaining how to navigate the game.

When the player arrives at the second moment, they spend some time near the breakable wall without making any progress. Due to the proximity detector this loitering incrementally decays their BreakWall skill. When the break wall skill crosses the first threshold a particle effect system activates drawing the attention of the player to the well. Yet, the player spends a little bit more time near the wall without any progress. Then the second hint activates, explicitly telling how to make progress at this step.

When it comes to the third moment, another breakable wall several levels into the game, the player still takes a few moments. This time however, they do not need the tooltip to appear as the particle effects are enough for them to remember the break wall mechanic.

Next we consider a player who has gone through the first two moments, then has decided to take a long break from playing the game before coming back. Their memory of the two moments and the tutorial prompts attached to them is blurry. When the player reaches the third moment, they do not remember the break wall mechanic. As they linger, the attached proximity detector decays their BreakWall skill. When the mastery level crosses the first threshold the particle effects appear, but in this case that is not enough. The player spends a few more seconds within the radius of the proximity detector which results in the explicit tooltip hint activating. The player remembers the controls and the mechanics. This allows them to keep on making progress, instead of getting frustrated and stopping playing the game altogether.

Overall every single player gets an experience that is specifically tailored to their skill level, which is made possibly by the constant tracking of the individual mastery levels.

Conclusion

In this paper we presented a new tool designed to help game designers create dynamic tutorials. We showed how Talin operationalizes Dan Cook's skill atom theory to create compartmentalized skill mastery tracking. We showed the steps outlining how to create a dynamic tutorial and discussed how different players engage with the dynamic tutorials.

While this paper mainly focused on contributing in the area of tutorials, we believe this granular level of skill mastery tracking can be an incredibly valuable dataset when it comes to understanding the holistic experience of the players going through games.

In future work we will be evaluating the dynamic tutorials to better quantify the benefits and will be exploring other uses of the tracked skill levels of the players to enhance both the development and playing experience of the games.

Talin currently requires the manual specification of skill chains. While this is consistent with our design goal of developer-control, a possible future direction is to build on the research by using Cognitive Task Analysis to identify skill chains (Horn, Cooper, and Deterding 2017).

We believe that it should be useful to model the player's mastery level for many more skills than are explicitly addressed with hints. Additional skills that are tracked behind-the-scenes should provide useful context for nuanced gameplay analytics. Beyond knowing where in a given level players are likely to abandon gameplay, we would like to know which concepts they were struggling with. Analytics based on skill level might suggest the need for additional dynamic hints far from the game's traditional tutorial level.

References

Andersen, E.; O'Rourke, E.; Liu, Y.-E.; Snider, R.; Lowdermilk, J.; Truong, D.; Cooper, S.; and Popović, Z. 2012. The impact of tutorials on games of varying complexity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM.

Anderson, J. R.; Boyle, C. F.; and Reiser, B. J. 1985. Intelligent tutoring systems. *Science* 228(4698):456–462.

Cook, D. 2007. The chemistry of game design.

Deterding, S. 2015. The lens of intrinsic skill atoms: A method for gameful design. *Hum.-Comput. Interact.* 30(3-4):294–335.

Doignon, J., and Falmagne, J. 2012. *Knowledge Spaces*. Springer Berlin Heidelberg.

Drenikow, B., and Mirza-Babaei, P. 2017. Vixen: interactive visualization of gameplay experiences. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 3. ACM.

Green, M. C.; Khalifa, A.; Barros, G. A.; and Togelius, J. 2018. "press space to fire": Automatic video game tutorial generation. *arXiv preprint arXiv:1805.11768*.

Horn, B.; Cooper, S.; and Deterding, S. 2017. Adapting cognitive task analysis to elicit the skill chain of a game. In *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, 277–289. ACM.

Hunicke, R. 2005. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 429–433. ACM.

Karth, I. 2014. Ergodic agency: How play manifests understanding. *Engaging with Videogames: Play, Theory and Practice* 205–216.

O'Rourke, E.; Andersen, E.; Gulwani, S.; and Popović, Z. 2015. A framework for automatically generating interactive instructional scaffolding. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 1545–1554. ACM.