A Mixed-Initiative Tool for Designing Level Progressions in Games

Eric Butler, Adam M. Smith, Yun-En Liu, and Zoran Popović
Center for Game Science
Department of Computer Science & Engineering, University of Washington
{edbutler,amsmith,yunliu,zoran}@cs.washington.edu

ABSTRACT
Creating game content requires balancing design considerations at multiple scales: each level requires effort and iteration to produce, and broad-scale constraints such as the order in which game concepts are introduced must be respected. Game designers currently create informal plans for how the game's levels will fit together, but they rarely keep these plans up-to-date when levels change during iteration and testing. This leads to violations of constraints and makes changing the high-level plans expensive. To address these problems, we explore the creation of mixed-initiative game progression authoring tools which explicitly model broad-scale design considerations. These tools let the designer specify constraints on progressions, and keep the plan synchronized when levels are edited. This enables the designer to move between broad and narrow-scale editing and allows for automatic detection of problems caused by edits to levels. We further leverage advances in procedural content generation to help the designer rapidly explore and test game progressions. We present a prototype implementation of such a tool for our actively-developed educational game, *Refraction*. We also describe how this system could be extended for use in other games and domains, specifically for the domains of math problem sets and interactive programming tutorials.

Author Keywords
Game design; design tools; AI-assisted design; educational games.

ACM Classification Keywords
H.5.0 Information Interfaces and Presentation: General

INTRODUCTION
The design of interactive experiences that consist of a sequence of episodes building in complexity is an intricate, multi-dimensional design problem that often takes experts a long time to complete. This is particularly apparent in game design. Designing just a single gameplay element (e.g., puzzles, challenges, encounters, levels) requires many iterations and interactive playtests to uncover a satisfying result. Crafting a coherent and effective sequence of these elements into an entire game, called a *progression*, is even more involved because the experience of playing the game is highly dependent on the way individual components are connected. As game designers adjust their game at these different scales—altering details of a single gameplay element and scheduling appropriate introduction of those details across the game's entire progression—they wrestle with a mixture of formal and informal requirements. Keeping all of these concerns in mind while designing each level is difficult.

Game designers have adopted semi-structured practices for addressing this complexity. They often create informal notes and plans that capture a sketch of the progression's overall properties and then consult this document while designing individual levels. Although developers often create complex in-house tools for authoring game content (e.g., the Dragon Age Toolset[1]), these tools focus on individual levels and rarely explicitly model progressions. As designers improve their level designs in response to testing, the progression plan is rarely updated and the high-level structure of the final level progression is not articulated in general terms again.[2] As a result, the coherence and intent of the original plan can be lost.

We propose the creation of progression design tools that aid game designers through all stages of design: sketching, rapid exploration, iteration, and final authoring of a complete game progression while keeping the progression plan in-context and up-to-date. Such tools should allow editing of both individual elements and progressions over those elements. Building on the ideas of mixed-initiative planning and constraint-based game content generation and verification, we demonstrate a prototype tool using the educational game *Refraction* that combines existing single-level editors into a mixed-initiative progression design tool.

Our focus on tools for designers contrasts with current trends in game design automation research. Many projects aiming to reduce designer burden offer fully automatic generators for individual levels, often focusing on optimizing a fixed metric for the quality of the level [24]. These systems are not designed to optimize the relative placement of levels within a

[1] `http://social.bioware.com/page/da-toolset`

[2] In a very interesting exception to this trend, fans sometimes recreate visual progression plans for popular games, such as Piotr Bugno's detailed outline of the story and level progressions for *Portal 2*: `http://www.piotrbugno.com/2012/06/portal-2-timelines/`

progression. Other fully automated generators explicitly take user-configurable constraints as input [17, 20]. These generators are directable down to a fine scale but still require an external progression plan. Zook et al. [25] describes a system for fitting a progression of generated challenges in a training game to an ideal player performance curve; however, the system requires all design input to take the form of formally modeled properties such as evaluation functions and causal coherence constraints. We are interested in allowing the designer input at all scales, so fully automated methods will not suffice. Mixed-initiative design tools, such as Tanagra [21] and SketchaWorld [18], pair generative techniques with an interactive editing interface that allows the human and machine to take turns editing a shared level design. Such tools help the designer prototype new ideas and check constraints for quality assurance while still allowing designers to craft levels with subtle properties that are difficult to formalize. These tools focus on creating single levels, whereas we want to create a tool that allows designers to work at multiple scales. However, we follow similar design process pattern for the problem of designing coherent progressions of detailed levels.

Effective progression design tools could provide valuable assistance to the same expert designers who would normally have designed levels without such support. We expect that they would be almost required for very large, complex progressions where manual exploration becomes intractable. More importantly, however, they would open up possibilities for end-user progression design. Consider an experienced teacher who wants to directly manipulate which concepts will be used in a progression tailored to their class. They might prefer that the tool automatically generate the relevant challenges at the scale of individual puzzles everywhere except where one or two key levels should visually resemble examples previously shown in class. This property is unlikely to be supported by any fully automated system.

In this paper, we discuss the requirements of a progression design tool, and present our prototype implementation for our actively-developed educational game, *Refraction*, illustrating its utility through several case studies. We expect that these ideas might be explored in domains outside of games such as interactive programming tutorials and high-school algebra exercises. This paper makes the following contributions:

- We identify the need for progression design tools and describe their impact with respect to current practices.
- We sketch a general architecture for progression design tools, including the use of generative techniques on a per-level basis where available.
- We present a prototype implementation of our design tool, attached to our own active design project, *Refraction*.

REFRACTION

Refraction is a puzzle game, intended for use with elementary education of fractions, that involves splitting and combining laser beams to form beams of fractional power. The game consists of a sequence of puzzles in which the player must direct lasers into targets, a task that requires both spatial and mathematical problem-solving skills. A sample puzzle is shown in Figure 1.

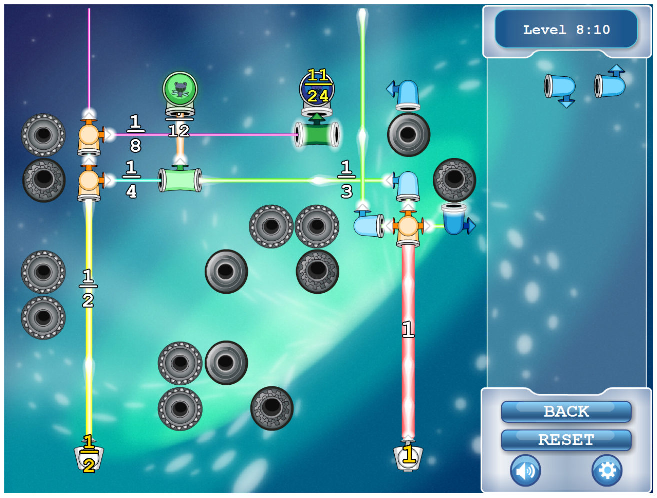

Figure 1. A level of *Refraction*, our actively developed game. We continue to create and release new level progressions for the game, driving our need for progression design tools.

Though *Refraction* has already been released to a wide audience, it is still under active development. We frequently need to create new level progressions for ongoing classroom studies and experiments, often involving adapting the game for different age groups or testing new mechanics. Thus, we personally have a strong need for tools that allow us to effectively and rapidly explore and iterate level progression designs.

CURRENT PRACTICES

Game developers have created a wealth of guidelines and best-practice suggestions for creating effective level progressions in games. In this section, we review current practices and discuss which aspects our system aims to improve. These practices are not well-documented, and there is not a clear accepted plan for tackling these challenges. Industry magazines such as Gamasutra[3] archive articles and discussions on game development written by industry members, from which we draw our information.

Level Progressions

Many games consist of a sequence of distinct elements (e.g., levels, stages, challenges, scenarios or puzzles) that the player encounters. We call this sequence a *progression*. In *Refraction*, the elements are the individual puzzles (involving 2-5 minutes of play), and the progression is the entire sequence of puzzles (2-5 hours of play). There are many ways to discretize a game progression; puzzles in *Refraction* are very short so make a reasonable choice, but larger games might model progressions over smaller elements. In this paper, we will generally refer to the individual elements as *levels*. Levels cannot be designed entirely in isolation; how they relate to each other is critical for effective game design. Much of the effort spent during the iteration process focuses on altering individual levels to improve the overall progression.

As a result, game design often happens at multiple scales. At a broad scale, designers create a *progression plan*, which

[3] `http://gamasutra.com`

documents the features they want each of their levels to have. These features might include, for example, which mechanics appear in a level or which key dramatic events occur in the game's narrative for a story-oriented game. In *Refraction*, there are several different types of pieces the player may use, such as bending pieces, splitting pieces, and adding pieces. The pieces required to solve any particular level are a feature we consider during the design process. At a narrow scale, designers are editing the levels themselves, trying to craft levels that have the complex properties they desire.

There are many design considerations when creating progression plans. For example, many games introduce different mechanics slowly and deliberately to allow the player to learn each of them without becoming overwhelmed. Game designer Dan Cook associates these mechanics with *skill atoms* [6] and suggests diagramming the dependence between atoms to better understand how a game works. In many cases, the introduction of a new skill atom may depend on the player having previously mastered another. For example, in Nintendo's *Super Mario Bros.*, the player should learn to jump before they can learn to jump on an enemy. Similarly, overusing a certain atom can lead to burnout, and avoiding this requires additional constraints between levels to be checked. In *Refraction*, we wish to introduce pieces in a particular order and pace. Skill atoms may also have intra-level considerations: in *Refraction*, for example, some types of pieces cannot be used unless another type of piece is present in the level.

Another important design consideration of progressions is pacing. *Pacing* describes how elements such as complexity, difficulty, and intensity vary over the course of the game. Game designer Jenova Chen writes [5] about how Csikszentmihalyi's concept of *Flow* [7] applies to game progressions. If the difficulty increases too quickly, players can become frustrated; too slowly, they become bored and disengaged. Many articles have been written on Gamasutra analyzing and discussing techniques for effective game pacing [4].

Design Practices

Designers understand the value in explicitly planning their games' progressions. Progression design considerations and progression plans are often sketched out in a design document or whiteboard before production of levels. Designers have written articles endorsing planning game progressions before production, writing that teams that do not explicitly create progression plans often end up redoing work after user testing reveals problems with their progressions, resulting in a worse final product[5]. However, the progression may change greatly during production, user testing, and iteration of levels; levels and mechanics may be reordered, or mechanics may be dropped entirely or new ones added. So while planning progressions beforehand is very useful, if left unaltered, the progression plans become increasingly inaccurate. Although game developers frequently spend great effort creating in-house editors and other authoring tools for their games[6], to the best of our knowledge, they do not create editors for their progression plans that automatically sync with their levels. For designers to continue to work at this broad scale during development, they must first evaluate the current progression manually, a significant task[7]. After production, sometimes developers (or even players) recreate visualizations of the progression realized in the final game. While these may be used in post-mortem analysis, they are still created mainly by hand, and are of little utility for design of that game, since it has already been published and is unlikely to undergo large changes. Integrating an up-to-date progression plan as part of the standard editing environment could help the designer more effectively work at multiple scales.

Because the information about the progression plan is cumbersome to keep in sync with the actual level progression, ensuring that the current levels meet all progression design considerations is an expensive, error-prone task. Edits to levels frequently change their properties in ways which may, for example, disrupt desired pacing, or introduce game mechanics too quickly or in the incorrect order. If not noticed by the designer, these problems will eventually be discovered in user testing, a relatively expensive resource. Automated detection of these problems may save time and effort by finding issues quickly, without designer effort.

We expect having the ability to rapidly explore different progression plans would enhance the design process. However, if the designer wishes to make a minor adjustment to the progression, say moving the introduction of a mechanic to a later point in the game, this might require significant modifications to existing levels. While a large portion of game developers create level editors to help author levels, these adjustments are still expensive to make. Because of this, designers can be limited in how easily they can explore the design space.

Current design practices result in several problems we wish to overcome in the design of our system. Designers are not able to easily explore progressions because level authoring is time-consuming. Designs for progressions plans are often abandoned after level creation begins because they are too cumbersome to keep up-to-date. As a result, level edits often introduce problems that are not discovered without significant inspection and user testing.

PROGRESSION DESIGN TOOLS

In this section, we formalize our design problem and discuss our goals for an effective game progression design system.

Definitions

We will use *levels*, *progressions*, and *progression plans* as defined previously. Each level has several *properties*, features that the designer cares about. Examples include which game concepts are used in a particular level, or whether a particular narrative event occurs in a level. There are often a huge

[4]e.g., `http://www.gamasutra.com/view/feature/134815/` or `http://www.gamasutra.com/view/feature/132415/`

[5]e.g., `http://www.gamasutra.com/view/feature/3848/`

[6]For example, players can create complex mods and scenarios using *Starcraft 2*'s in-house level editor: `http://us.battle.net/sc2/en/game/maps-and-mods/`

[7]Recommendations to do just that come from `http://www.gamasutra.com/view/feature/132256/`

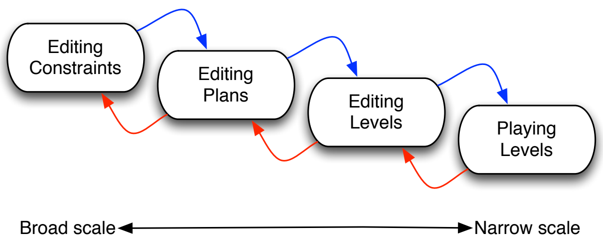

Figure 2. The ideal workflow for our system (a waterfall-like pattern). Designers must iterate at multiple scales, so our tool aims to allow the designer to move freely between broad and narrow-scale editing.

number of possible levels that share a particular set of properties. Likewise, there are many progressions that conform to a particular progression plan. The designer may have particular considerations for their game's progression, such as pacing or the ordering in which mechanics are introduced. We refer to all of these considerations definable at the level of the progression plan as *progression constraints* (or just *constraints*). We use the term (loosely) in the sense of optimization problems: the designer has a set of constraints in mind, and the design problem is to generate a progression best satisfying those constraints. Constraints may be hard (never to be violated), soft (flexible in exchange for satisfying other constraints at some cost), or unmodeled (not tracked by the tool). Again, there will generally be many progression plans that equally satisfy a particular set of constraints.

Goals of the Design Tool

The tool is intended to be a full-fidelity game editor and for authoring the final levels seen by players. The system should support rapid iteration and exploration at multiple scales of the design: progression constraints, progression plans, and individual level designs. Designers should be able to playtest the current progression. Figure 2 demonstrates the desired workflow for the system. Many constraints and properties can be formally modeled, so we would like to draw on advances in procedural content generation to automate generation and analysis of content where possible. Level generation is not an option for all types of games, so it is an optional component of our tool. It is important to note that many critical design considerations, such as the moment-to-moment difficulty or affective impact on players, are subjective or cannot be easily modeled. Thus human editing *must* be available at all scales, so the system should integrate existing level editors that game designers already create. As the designer changes from broad-scale progression planning to narrow-scale level editing and back, the system should automatically keep other parts updated: Edits to levels should be reflected in the progression plan, and edits to the plan should notify the designer if plan-level constraints are violated.

APPLICATION TO REFRACTION

To ground this discussion about progressions, levels, and constraints, we discuss how these ideas apply in the design of our game, *Refraction*. In this section, we discuss progression constraints and level properties specific to *Refraction* that we currently use as well as which game-specific components we

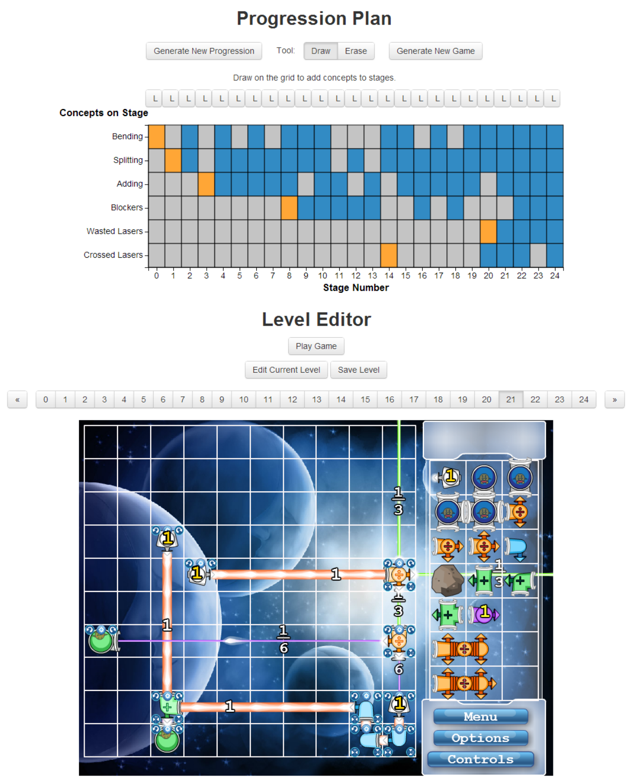

Figure 3. Screenshots of two of the three browser-based editing environments of our tool, each for editing the game at a different scale. On top is the *progression plan editor*, used for directly manipulating the plan. On the bottom is the *level editor*, which embeds *Refraction*'s custom editor. There are additional views for editing constraints and playtesting.

needed to provide for the implementation. Figure 3 shows screenshots from our prototype intended to demonstrate how a designer might control and view plans and levels. We delay discussing the system components until the next section.

Constraints and Properties

All level properties we model in this prototype describe whether a particular gameplay *concept* (or *skill atom*, to use Cook's [6] terminology) is required to solve a level. Table 1 lists a sample of these concepts. Because we currently only model concepts, all of our level properties are binary, describing whether a concept is required. Many additional properties have not yet been modeled, such as how closely the pieces must be placed together in all possible solutions. Some can-

Concept	Description
Bending	Must use bending pieces
Splitting	Must use splitting pieces
Adding	Must use adding pieces
Blockers	Must contain obstructive pieces
Wasted Laser	Leaves some laser beams unused
Crossed Laser	Laser beams are unavoidably crossed

Table 1. A sample of the properties we model for levels in *Refraction*. These properties are binary, describing whether or not a particular gameplay concept is required to solve a level.

not be directly modeled, such as whether a level is fun or aesthetically pleasing. Though not tracked by our tool, these can still be addressed through manual edits to level designs.

We have implemented four types of progression constraints.

Prerequisites
Prerequisites are *inter*-level constraints that define a partial ordering over the introduction points of the concepts. For concepts A and B, if A is a *prerequisite* of B, then the level of A's first appearance must precede the level in which B first appears. For example, we want to introduce pieces that bend the laser before introducing pieces that split the laser.

Corequisites
Corequisites are *intra*-level constraints on which concepts can appear together. A is a corequisite of B if A must show up in any level that has B. For example, one concept is to require that the laser be looped around to cross itself. This is only generally possible if bending pieces are available. Thus, "Bending" is a corequisite for "Crossed Laser."

Concept Count
For each level in the progression plan, we compute a illustrative proxy measure for "level intensity" (a term often occurring in other designers' progression plans) as a sum of the concepts that are in the level. Although this metric does not correspond directly with real-world difficulty, it is nevertheless useful for controlling pacing of the progression at a broad scale with a few quick adjustments. The constraint assigns target number of concepts for each level. Our interface allows control via a spline editor, as seen in Figure 5.

Concept Introduction Rate
This constraint controls the rate at which new concepts are introduced, describing, for each point in the progression, the number of concepts that should have been introduced by that point. This constraint, expressed via spline editor, can be used to ensure concepts are introduced at a pace allowing the player to master one before proceeding, as well as roughly showing how the complexity of the game changes at a glance.

Clearly, these concepts are specific to *Refraction*. However, the progression constraints are quite general. Given another game's set of gameplay concepts, we imagine that prerequisites and corequisites will still be meaningful for shaping progression plans. The designer could specify constraints that apply to properties related to categories other than gameplay, such as aesthetics. For example, a designer might want to specify that aesthetic properties like background music and graphical tileset should change in groups, corresponding to movements to different places in the game's fictional world.

Game-Specific System Components
Using these particular properties for level generation and analysis in *Refraction* requires a subtle technical approach. Ensuring that a concept is *required* to solve the level entails checking that *all* possible solutions of the puzzle (there are generally many) use the concept, a computationally difficult problem. For example, in a level in which the player is expected to split a laser in two then later add it back together, careless design might enable the player to use only bending pieces to bypass the splitting and adding altogether. Smith et al. [19] developed a level generator (with *Refraction* as the example application) that can enforce this type of constraint on its outputs. The generator uses *answer set programming* (ASP), a declarative constraint-programming technology, to ensure that specified concepts must be used in all possible solutions. In addition to reusing this generator in our system, we extended it to use the same formal game model to determine if a designer-altered level requires each gameplay concept. This level analyzer is used to update the progression plan after levels are manually edited.

We created a progression plan generator using ASP's ability to solve constrained optimization problems. Progression constraints (e.g., prerequisites) are expressed as soft constraints and manual edits locked by the designer as hard constraints. Soft constraints are implemented with an integer-valued penalty function. The generator searches for plans satisfying all hard constraints with the minimal penalty for soft constraints. We include additional soft constraints that add variation to the plan by penalizing repetition. We implemented (in Javascript) a corresponding analyzer that, given a progression plan, checks for violated constraints.

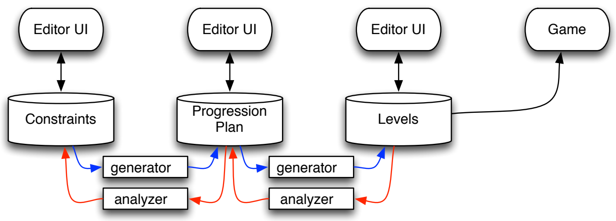

Figure 4. The pieces of our system. Cylinders are models. Ovals represent editing/viewing interfaces that correspond to the four elements of the workflow outlined in Figure 2. The square components are the automated parts that ensure consistency across scales.

SYSTEM DESCRIPTION
In this section we describe in detail each of the system components we built for our *Refraction*-based prototype. For each component, we discuss its role and how our implementation works. Some of our implementations of these components are game-specific, while others could be directly reused for other games. Of course, all parts of the system can be extended or replaced in the interest of better supporting the designer. Figure 4 illustrates how the system components fit together.

Model
The model that the tool manipulates consists of three parts, which directly correspond to the scales of the workflow: the progression constraints, the progression plan, the progression itself (the sequence of concrete level designs). The designer can also output the playable game from the tool.

Working with Progression Plans
The broad-scale iteration loop takes place between manipulating progression constraints and the progression plan. The system has a set of user interface components used to display and manipulate the progression constraints. Figure 5

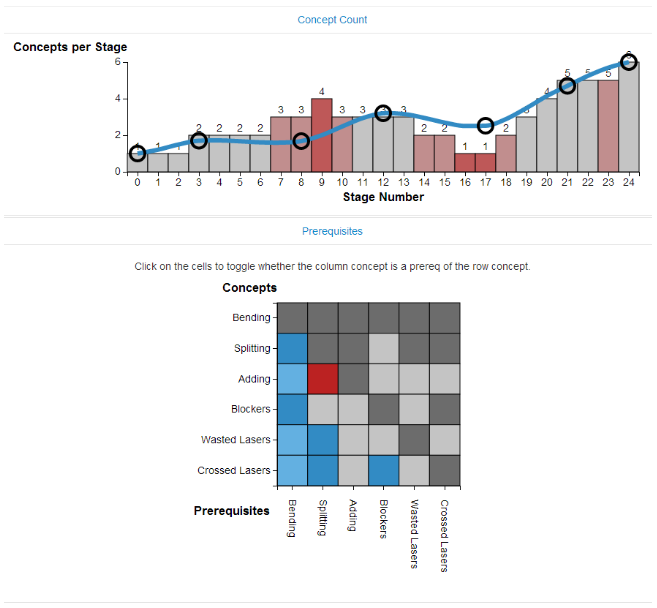

Figure 5. View of the two types of constraint editors in our implementation, which allow manipulation of the various constraint parameters. Each interface type is used for multiple constraint types, e.g., the grid is used for both prerequisites and corequisites. If the current progression plan violates these constraints, the violations are displayed in red. On top, the *concept count* graph shows us that several levels have too many or too few concepts. The blue curve is the target, and the bars are the values of the current plan. On bottom, the red cell of the *prerequisite* chart shows us that the "adding" concept is erroneously introduced before the "splitting" concept. Blue indicates an active constraint (dark is manual, light is inferred via transitivity), while dark gray indicates an unselectable constraint (e.g., two concepts may not be mutual prerequisites).

shows the set used in our implementation. These components are responsible both for letting the designer adjust the constraint parameters and showing whether the current progression plan violates these constraints. As the designer changes the constraint values, the component interfaces update to display whether the current progression plan satisfies the new constraints.

The primary component for manipulating the progression plan is the *progression plan editor*, shown at the top of Figure 3. For our implementation, this consists of a grid showing level properties of each level, where the cells are the current value of the property for that level. The user can directly manipulate this grid to change level properties. As the designer edits the plan, the *progression analysis* component checks whether the constraints are still satisfied. This is then reflected in the constraint editors in real time.

The designer manually manipulates the constraints and progression plan in order to produce a plan that matches their intent. However, the *progression generation* component can be used for rapidly populating the plan and sampling alternatives. When manually activated by the designer, the progression generator creates a new progression plan (overwriting any existing) that best satisfies the current constraints.[8] In order to prevent any manual edits from being overwritten by generation, our system allows the designer to "lock" a particular level. The generators will not modify these levels or their corresponding entries in the property matrix.

Thus, the full iteration loop for this scale is: set values of the constraints, optionally generate a new progression, manually edit the progression, check whether these edits violate any new constraints or unmodeled design criteria, repeat.

Working with Levels

The narrow-scale iteration loop involves manipulating the progression plan and the levels themselves. The structure of components at this scale corresponds with the structure above. First is the previously described progression plan editor. Like the constraint editors, manipulation of the progression plan does not immediately change the levels below.

The *level editor* is used for editing individual levels. Because levels are always game-specific, a single-level editor must be supplied by the game's developers. Developers must also provide a *level analyzer* that computes the relevant design properties for each level. The difficulty of producing such an analyzer scales with the subtlety of the properties checked: determining if a level uses a particular piece or is set to use a particular graphical tileset is easy, but determining the level of strategy needed to complete the level is harder. When the designer edits a level, the level analyzer is used to immediately reflect the changes in the progression plan. These changes are propagated further up to the progression constraints.

For the purposes of automation, the developers may optionally provide a *level generator*. Though the designer could manually edit levels, certain games, including *Refraction*, have generators that divert significant burden from the designer. The level generator, if it exists, can be triggered by the designer to create levels that match the properties described in the current plan. Any levels that are "locked" are not overwritten, so manually-tuned levels can be preserved.

This iteration loop corresponds to the one above: edit the progression plan, optionally generate levels, manually edit levels, check how the progression (and the progression constraints) changed based on these edits, repeat.

Testing the Game

The final component of the system allows for playtesting the levels directly. Thus the game must have an interface that accepts new level progressions to be played immediately.

EVALUATION

Our implementation of the prototype system was intended for internal use by the authors. We evaluate the general architecture and our game-specific implementation through a set of case studies, exploring new design techniques that our system enables. The first case study requires the game-specific level generator, but the others do not require such technology.

[8]As there are often many progression plans that achieve an equivalent score with respect to the stated constraints, we have configured the progression generator to make liberal use of randomness in the interest of exploring diverse solutions.

Rapid Exploration at a Broad Scale

Assuming the system has a level generator, this tool enables game designers to rapidly explore different progression plans and test the plans in playable games. For example, consider a scenario for *Refraction* early in the design phase of a new progression for a new, younger audience. We do not know how this set of players will respond, so we wish to rapidly explore and playtest several possible progressions. We might want to remove "addition pieces" from the game entirely and avoid introducing the "crossed laser" concept (see Table 1) until late in the progression. Our tool allows the designer to quickly sketch these modifications into the progression plan, generate a new set of levels satisfying the plan, and test these levels immediately. While these generated levels may lack some of the unmodeled properties the designer desires, this is acceptable at an early stage of rapid exploration. The designer also has the ability to edit the generated levels to suit their needs, which can be easier than creating new levels from scratch. Previously, creating even an initial progression with the key gameplay concepts removed would have entailed massive effort before the first round of tests.

This rapid exploration applies at the broader constraint scale as well. Suppose in *Refraction* we wish to add an additional prerequisite constraint that "laser crossing" must come after introducing "splitting pieces." The tool can automatically create a progression plan in conformance with this constraint. It then populates a sequence of levels compatible with that plan, allowing a game satisfying the new constraint to be played with minimal designer effort. Without our system, the designer would have to first manually evaluate where the existing plan failed, devise a new plan for an adjusted progression (in notes outside of the editor), then create several new levels, either by hand or by manually configuring the level generator. We expect this reduction in friction to enhance rapid prototyping of progressions.

Automatic Detection of Problems during Iteration

As discussed, levels and progressions undergo a great deal of iteration during the development process. One of the dangers of changes during iteration is breaking previously expressed planning intents, both global progression constraints and level property constraints.

Consider an example in *Refraction* where, late in the development process, the designer wishes to make a minor adjustment to the mathematical portions of the fifth level of the progression. They exchange a "splitter piece" with two outputs for one with three outputs. However, this extra output introduces what we call a "wasted laser," or a laser that is not used in the solution. 'Wasted lasers" is one of the concepts controlled in the system and is constrained via the prerequisite constraint to first appear after another concept, "crossed lasers," which has not been introduced by this level. This could be corrected by adding additional pieces (a new target to absorb the extra laser), but may go unnoticed by the designer. However, upon saving the change in our tool, the system will automatically update the progression plan to reflect that wasted lasers appear in this early level. Since a prerequisite constraint is now violated, the constraint editor highlights this problem to notify the designer. The designer may then take steps to fix it: one option is moderately editing the offending level. If the designer prefers to keep the level as-is, they can generate a new plan for surrounding levels or remove this prerequisite constraint. We believe that this feature can make managing large design projects easier by notifying the designer of automatically detected problems.

Global Optimization of Progression Properties

A designer often needs the progression to satisfy many complex time-varying properties simultaneously. For large progressions this task is difficult to do manually. For example, suppose a designer for *Refraction* is trying to control the pacing of the progression. They are interested in controlling aggregate properties: the number of concepts per level, and the rate at which new concepts are introduced during the progression. The designer wants new concepts to be introduced at an even rate, but they want the number of concepts per level to vary significantly between levels. These constraints are modeled in our system, so the designer may directly edit curves (see Figure 5) to control the constraints. Our system's built-in progression generator can find the closest feasible solution through global optimization. As the designer refines these curves, they can quickly see possible solutions and adjust parameters appropriately. The designer may also rely on the fact that all design changes that were manually introduced can be preserved through the optimization process. We expect this to ease the process of creating progression plans that satisfy several complex constraints.

DISCUSSION

We believe that ideas from this system can be applied to a wide range of games. Some components are game-specific, such as the level editor and generator. The others, particularly those at the broader scale of progression constraints and plans, can be reused between games; the pairing of the our constraint editing interfaces with a generic constrained optimization tool on the back end are generic. The modularity of our framework allows it to be extended for new games and for additional functionality. In this section we discuss some of the limitations and some of the possible ways in which the system could be extended. We then discuss how these ideas might be explored in other, non-game domains.

Limitations and Extensions

We have defined progression plans as a sequence of level properties. In our implementation, these properties are binary values indicating whether a level contains a particular concept, but more nuance is certainly possible. For example, we may be concerned with *how many* bending pieces appear or the overall size of the player's solution. Having numeric knobs instead of binary values would enable the designer to specify even more precise progression plans. The system is also extendable to address some aesthetic concerns; for example, we might give the designer control of the average distance between pieces to allow them to create dense or open-feeling levels. We have already explored constraints on symmetry, balancing, and packing for *Refraction* levels, but elided those properties from the prototype presented here.

These new constraints are significantly more detailed than the current set and will require new display mechanisms to make them easy to see and manipulate.

Our prototype supports only a linear progression, limiting its application to some games. For example, in Nintendo's *Super Metroid*, the player explores a large, nonlinear open world. The player finds "power ups" in this world that unlock new abilities, allowing the player to explore previously inaccessible locations. Therefore, the "progression" is defined over which abilities the player has unlocked and which mechanics they understand how to use, rather than a single number describing how many levels they have completed. While extensions to the system to support other progression structures are plausible, most require finding a natural visual depiction for such structures. Currently, our ability to generate progressions and levels under hard and soft constraints exceeds our ability to present useful interfaces that would allow a designer to guide these generators in directions that would satisfy some external intent, so more research must be done to explore how to depict and interactively manipulate these more abstract progression mechanisms.

Refraction is a game of relatively small scope, but we believe this system could be used for much larger games. To remain tractable, one strategy would be to only model the most salient properties of levels and solutions. If complex levels can be created by connecting detailed tiles (e.g., pre-authored chunks of terrain in an 3D adventure game), then the representation used in progression planning can remain discrete and visually manageable. Another strategy would be modeling at additional scales. For *Refraction* it was feasible to work at two scales because puzzles are short enough to be treated as atomic objects. However, many games have very large "levels," such as the single-player campaign missions of real time strategy games like Blizzard Entertainment's *Starcraft*. While designers may wish to model progressions over the entire game's missions, the levels themselves are quite long, and the designer may wish to model an "intra-level" progression of a single level, or perhaps even modeling the progression over individual encounters within a level. Such progressions could be modeled by hierarchically nested progression diagrams: scales for each encounter, mission, and overall game.

We have supplied a small example of possible progression constraints, but they can be extended or new ones introduced. Luckily, many progression constraints apply generally to a broad class of games and so can be reused. This is in contrast with level constraints, which are almost always game-specific. For example, many games are concerned with the order mechanics are introduced, so our *prerequisite* constraint can be used in those games. Likewise, many games require care with pacing, so a constraint dealing with how long concepts should be practiced after introduction before moving on to new concepts would be generally useful.

The existing user interfaces for constraint editing can be repurposed for different kinds of constraints. For example, in addition to prerequisites and corequisites, we might use the same interface for constraints such as: *mutual exclusion*, where concepts A and B shall not show up in the same level; *sequential exclusion*, where if A was used in a previous level, B cannot be used in the current level; or *sequential introduction*, where A must be introduced immediately preceding B.

Similarly, the spline editing interface can be reused for many designer-specified functions. For example, we can model a sense of "scale" for *Refraction* levels by measuring the minimal number of pieces required to complete the level. Our *concept count* constraint gives a very primitive idea of level intensity, and as long as a designer can usefully employ edits of this curve in place of lower-level tweaks, it is valuable. However, it would be desirable to use a function more representative of real-world difficulty and player experience. Backing the spline editor with more sophisticated functions, such as a learning rate derived from an externally-validated learner model, is an obvious and enticing avenue of future work.

Our system has basic support for preserving manual edits by "locking" entire levels as fixed with respect to the generators, but better methods could be applied to make the tool more useful. An obvious first step is allowing more fine-grained locking by locking only particular concepts: for example, the designer may want to ensure that splitting shows up in levels 2–5 but wants to allow the generator to make other decisions for those levels. Allowing the generators to reorder existing levels or minimally modify levels while preserving designer-specified features may also make the locking tool significantly more useful.

Finally, the generators and analyzers can be replaced as technology improves. Depending on the game, many existing level generation technologies would fit in the system. While answer set programming fit our game well for both level and progression plan generation, we could explore many other optimization or planning techniques. For example, for level generation, a game with continuous physics like Rovio Entertainment's *Angry Birds* might use a generate-and-test system that uses the game's internal physics engine. Many relatively simple techniques, such as making an approximate query against a database of previously authored and annotated plans and levels, are likely to be fruitful as well.

Application to Other Domains

We believe these ideas can be explored in other domains beyond games. Here we propose some potential applications of this system, looking at two other domains in closer detail.

Example Domain: Teaching Programming

As one example, we consider an interactive application that teaches programming, similar to *Codecademy*[9]. "Levels" in this domain are individual programming exercises. We speculate a designer may be concerned with properties such as which programming concepts (e.g., recursion, conditionals) are explained or introduced in the exercise, or which concepts the user is assumed to already know. Another may be the complexity of the exercise, either by number of lines of code required or whether certain library function calls are required. Other properties are subjective in nature, such as the

[9]http://www.codecademy.com

directness of suggestions given to find the solution, the narrative used to give context to the user, or the visual layout of the exercise in the application and access to related examples.

A designer may wish to ensure several global progression constraints over the exercises. Concepts should not be used until an exercise where they are intentionally introduced. Complexity should vary between exercises to prevent frustration or boredom. Concepts are revisited to give users sufficient practice, and should be recombined with other concepts; for example, designers may wish for each concept to be used with at least three other concepts.

This problem domain shares many of the characteristics of our game design problem (indeed, many of these concerns map exactly into Cook's discussion of skill atoms) so it should likely benefit from using a progression design tool. As most of our tool components can be directly reused, the designers need only supply an editor for exercises and tools to compute properties of crafted puzzles. For the computationally complex task of determining if there are alternate solutions to a programming task that avoid the use of certain concepts, program synthesis techniques could be used to search the constrained space of small programs in a language that fit a specification [3].

Example Domain: High-School Algebra Problems

We consider creating a homework assignment containing a progression of algebra problems. There are several level properties a curriculum designer may wish to control, some modelable, others subjective: which algebraic concepts are used (e.g., factoring, cancellation, distribution), how many steps a particular problem takes to solve, or whether the problem resembles in-class examples.

We consider several progression constraints a designer might wish to enforce, such as the partial ordering of concepts (e.g., properties of roots before properties of logarithms). Designers may want to introduce new concepts in isolation to allow students to master them before combining them with others. They may also wish to ensure the supporting text for each problem is thematically coherent with nearby problems.

There is existing work moving towards being able to analyze and generate such math problems automatically. For instance, Andersen et al. [1] created a framework capable of automatically generating example problems and progressions for elementary and middle school mathematics. Given a hand-authored problem, their system can report which pathway through a mental algorithm the student is likely to take, thus allowing a metric for analysis and synthesis of problems, which is also suitable for our progression designer.

More generally, we suspect that the progression designer could be useful for analysis of the entire math or science curricula through 13 years of K–12 education, or design of a problem-based learning course.

RELATED WORK

There is a long history of creating design tools in HCI research. Many tools have been created to support rapid exploration and prototyping at an early stage. Sketch-based tools such as Silk [13] and Denim [14] allow the designer to sketch interfaces with a stylus. Suede [12] explored the rapid creation of prototypes of speech-based user interfaces using wizard-of-oz techniques. Similar techniques could be used to prototype progression design tools in domains where automation similar to that which we built on is not yet available. In this case, the ability to sketch in broad-scale properties of a progression (e.g. a target pacing curve) and sample alternative plans is still useful even if per-level design is done by hand.

Systems like d.tools [10] support an iteration loop of creating, testing and analyzing interface prototypes. Because user testing and iteration is critical when creating game levels and progressions, our system also aims to support these kinds of iteration loops, but a manner specific to our domain.

Juxtapose [11] and Side Views [23] are examples that allow the user to quickly explore alternatives to the current design. Our tools tries to support this though sampling randomly generated alternatives under constraints, though adopting a literal side-by-side view could be more effective.

Much work has been done using constraint solvers in user interfaces, such as Cassowary [2] and the engine in Amulet [15]. In contrast to familiar applications of geometric layout constraints on interface elements, many of our constraints are much more abstract in nature, such as the nonexistence of shortcut solutions for a level that is supposed to introduce a new gameplay concept. Even so, we strongly separate the definition of the constraint (as driven by the interface of the design tool) from the back-end search technology used to find satisfying solutions. Creating generators and analyzers for *Refraction* involved setting up constraints and passing these to a domain-independent solver from the Potassco project [8].

There is also a rich history of work in mixed-initiative planning and collaboration tools. COLLAGEN increased the ease with which users could create mixed-initiative planning tools [16]. The framework specifies an interface with goals, recipes, and steps, to be implemented by domain-specific planners. COLLAGEN would then provide a dialogue system capable of conversing with the end-user, using the underlying planner. More generally, the SHARED-PLAN architecture underlying COLLAGEN allows computer and human agents to collaborate together in groups to satisfy goals, such as planning or interface design [9]. Our tool is more visually oriented, rather than relying on dialogue as a method of interaction; in addition, we optimize directly while planning subject to constraints instead of requiring recipes to search for solutions.

Mixed-initiative planners have also been used in other domains. NASA's MAPGEN is actively used to create daily activity plans for Mars rovers [4]. OZONE was designed to be a mixed-initiative constraint-based planning framework with pluggable components and was used to plan military resource allocation and transport tasks [22]. Our domain is different in that designers must both create plans and levels.

CONCLUSION

In this paper, we have identified the potential utility of mixed-initiative progression design tools for games, described how such systems could work, and created an implementation to be used with our own deployed game. Progressions and levels are both very difficult to design, and it is arduous to juggle *all* design considerations while working at multiple scales. For several types of games, many of these considerations can be formally modeled, so we can use computation to automate portions of the process. At the same time, many design concerns are completely subjective, so there is a strong need to retain human involvement in the creation of final progression plans and levels for deployment. We expect that game progression design tools can significantly enhance the game creation process. There are many other domains where interactive experiences are composed of scaffolded episodes. Practically all educational and general training environments fit this paradigm to some extent. We describe how given a breakdown of key concepts and the ability to automatically generate levels from parametric specification other domains directly map to our progression design process.

There are several areas of future work, in addition to potential extensions mentioned earlier. Experience with this prototype has already prompted a number of game-specific and general directions to explore next. For example, in addition to adding more details to the progression plan (properties and curves), we want to investigate how concrete patterns discovered in the level editors (such as a commonly used cluster of pieces) can be upgraded into plan-scale properties without additional programming. Meanwhile, broader user studies are required to determine the effectiveness of this model before we can make progress on deploying progression design tools for an audience beyond experienced level designers. More study is needed to discover how well this system applies to a wider class of games. Finally, we propose exploring these ideas in other, non-game domains.

ACKNOWLEDGMENTS

This work was supported by the University of Washington Center for Game Science, DARPA grant FA8750-11-2-0102, the Bill and Melinda Gates Foundation, the William and Flora Hewlett Foundation, and an NSF Graduate Research Fellowship under Grant No. DGE-0718124.

REFERENCES

1. Andersen, E., Gulwani, S., and Popovic, Z. A trace-based framework for analyzing and synthesizing educational progressions. CHI (2013).
2. Badros, G. J., Borning, A., and Stuckey, P. J. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact. 8*, 4 (Dec. 2001), 267–306.
3. Basin, D., Deville, Y., Flener, P., Hamfelt, A., and Nilsson, J. F. Synthesis of programs in computational logic. In *PROGRAM DEVELOPMENT IN COMPUTATIONAL LOGIC*, Springer (2004), 30–65.
4. Bresina, J. L., Jónsson, A. K., Morris, P. H., and Rajan, K. Mixed-initiative planning in mapgen: Capabilities and shortcomings. In *Proceedings of the ICAPS-05 Workshop on Mixed-initiative Planning and Scheduling, Monterey, CA*, Citeseer (2005), 54–61.
5. Chen, J. Flow in games (and everything else). *Communications of the ACM 50*, 4 (2007), 31–34.
6. Cook, D. The chemistry of game design. *Gamasutra* (2007).
7. Csikszentmihalyi, M. *Flow: The Psychology of Optimal Experience*. Harper & Row Publishers, Inc., 1990.
8. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. Potassco: The potsdam answer set solving collection. *AI Communications 24*, 2 (2011), 107–124.
9. Grosz, B. J., Hunsberger, L., and Kraus, S. Planning and acting together. *AI Magazine 20* (1999), 23–34.
10. Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., and Gee, J. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, UIST '06, ACM (2006), 299–308.
11. Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S. R. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, ACM (2008), 91–100.
12. Klemmer, S. R., Sinha, A. K., Chen, J., Landay, J. A., Aboobaker, N., and Wang, A. Suede: a wizard of oz prototyping tool for speech user interfaces. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, UIST '00, ACM (2000), 1–10.
13. Landay, J. A. Silk: sketching interfaces like krazy. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, ACM (1996), 398–399.
14. Lin, J., Newman, M. W., Hong, J. I., and Landay, J. A. Denim: finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '00, ACM (2000), 510–517.
15. Myers, B., McDaniel, R., Miller, R., Ferrency, A., Faulring, A., Kyle, B., Mickish, A., Klimovitski, A., and Doane, P. The amulet environment: new models for effective user interface software development. *Software Engineering, IEEE Transactions on 23*, 6 (1997), 347–365.
16. Rich, C., and Sidner, C. Collagen: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction 8*, 3-4 (1998), 315–350.
17. Schanda, F., and Brain, M. Diorama. **`http://warzone2100.org.uk/`**, Apr. 2013.
18. Smelik, R., Tutenel, T., de Kraker, K. J., and Bidarra, R. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, ACM (2010), 2:1–2:8.
19. Smith, A., Butler, E., and Popović, Z. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG '13: Proceedings of the Eighth International Conference on the Foundations of Digital Games* (2013).
20. Smith, A. M., Andersen, E., Mateas, M., and Popović, Z. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, ACM (2012), 156–163.
21. Smith, G., Whitehead, J., and Mateas, M. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on 3*, 3 (2011), 201–215.
22. Smith, S., Lassila, O., and Becker, M. Configurable, mixed-initiative systems for planning and scheduling. In *Advanced Planning*, AAAI Press (1996), 235–241.
23. Terry, M., and Mynatt, E. D. Side views: persistent, on-demand previews for open-ended tasks. In *Proceedings of the 15th annual ACM symposium on User interface software and technology*, UIST '02, ACM (2002), 71–80.
24. Togelius, J., Yannakakis, G., Stanley, K., and Browne, C. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on 3*, 3 (2011), 172–186.
25. Zook, A., Lee-Urban, S., Riedl, M. O., Holden, H. K., Sottilare, R. A., and Brawner, K. W. Automated scenario generation: toward tailored and optimized military training in virtual environments. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, ACM (2012), 164–171.