Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design

Gillian Smith *Student Member, IEEE*, Jim Whitehead *Senior Member, IEEE*, Michael Mateas

***Abstract*—Tanagra is a mixed-initiative tool for level design, allowing a human and a computer to work together to produce a level for a 2D platformer. An underlying, reactive level generator ensures that all levels created in the environment are playable, and provides the ability for a human designer to rapidly view many different levels that meet their specifications. The human designer can iteratively refine the level by placing and moving level geometry, as well as through directly manipulating the pacing of the level. This paper presents the design environment, its underlying architecture that integrates reactive planning and numerical constraint solving, and an evaluation of Tanagra's expressive range.**

***Index Terms*—Procedural content generation, mixed-initiative, reactive planning, constraint programming, augmented design, level design, games.**

I. Introduction

CREATING a good level is a time-consuming and highly iterative process: the level may start as a simple sketch of the space, which is then filled in with specific geometry. Designers will typically play the level themselves many times before showing it to anyone else, checking that it is playable, engaging, and meets their expectations [1]. Making a change to a small section of a level, such as moving a single critical platform, can have a significant impact on the design and require much of the rest of the level to be modified as well.

Tanagra is a mixed-initiative level design tool, operating in the domain of side-scrolling 2D platformer levels, that incorporates procedural level generation to ease this authoring burden. This paper presents a detailed description of the tool, an architecture for integrating hierarchical reactive planning with numerical constraint solving, and an evaluation of the expressivity of the system by examining the range of content it can create.

The mixed-initiative approach to design, where content is created through iterative cycles between the human designer and a procedural content generator, capitalizes on the strengths of both human and computer designers. Tanagra's underlying level generator is capable of producing many different variations on a level more rapidly than human designers, whose strengths instead lie in creativity and the ability to judge the quality of the generated content. The generator is able to guarantee that all the levels it creates are playable, thus refocusing early playtesting effort from checking that all sections of the level are reachable to exploring how to create fun levels. Also, our game-play centric approach to level representation and content generation opens up possibilities for novel editing operations: in addition to controlling physical properties of the world such as platform placement, the designer is also able to control properties related to potential player behavior by influencing the pacing of the level. In Tanagra, the human designer and procedural generator work together in a collaborative way, each taking turns to build on the work of the other.

A combination of reactive planning and constraint programming allows Tanagra to respond to designer changes in real-time. We use A Behavior Language (ABL) [2] for reactive planning, and Choco [3] for numerical constraint solving. Reactive planning allows for the expression of generator behaviors, such as placing patterns of geometry or altering the pacing of the level, which can be interleaved with a human designer's actions. These behaviors monitor multiple aspects of the generator in parallel, and their hierarchical nature allows for complex geometry patterns to be built up from simpler components. The geometric relationship between level components is expressed as a set of numerical constraints that must be satisfied, thus ensuring that the design tool will never allow for the creation of an unplayable level. This architecture sits atop a rhythm-based representation for levels, where each beat in the rhythm corresponds to a single action taken by the player. This rhythm-based representation is based on the observation that there is a rhythm and pacing of distinct player actions in many kinds of 2D platformers [4,5].

In creating Tanagra we were guided by design principles for intelligent creativity and design support tools [6-9]. There has recently been a call for such intelligent design tools specifically in the domain of games [10,11]. We have been careful to ensure that Tanagra does not push its own agenda on the designer by protecting decisions made by the human so that they cannot be overridden by the system, although it can augment human-placed geometry through the placement of additional level components. Tanagra provides expertise

Manuscript received November 8th, 2010.

G. Smith, and M. Mateas are with the Expressive Intelligence Studio; G. Smith and J. Whitehead are with the Augmented Design Lab, both at the University of California, Santa Cruz, Santa Cruz, CA 95064 USA (phone: 434-242-5448 email: gsmith@soe.ucsc.edu). This work is supported by the National Science Foundation, grant no. 1002852. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

through its ability to ensure that levels are always playable; i.e. every platform and object in the level is reachable, and it is possible to proceed from the left to right side of the level. Tanagra also allows the designer to directly manipulate the pacing of the level by editing the underlying beat structure. It works *with* the iterative design process by supporting new decisions from the designer at any time during creation, and rapidly re-generating sections of a level as needed.

II. Related Work

The earliest uses of procedural content generation in games had the purpose of improving replayability. These games provided more worlds than a human designer could be reasonably expected to author, given both time constraints during design and hardware constraints for storing large, detailed worlds [12,13]. This is still a primary motivation for PCG in modern games such as the *Diablo* series [14], the *Civilization* series [15], and *Dwarf Fortress* [16]. In such games, levels are completely generated by the computer, with little to no creative interplay between human designer and computer generator. Much of the design of levels is implicitly encoded in the generation algorithm itself, with designer input limited to tweaking parameters or restricting the random seeds to ensure that all levels the generator creates are playable, fun, and appropriate. This interaction style can be unintuitive and time consuming, with small shifts in parameters leading to radical changes in the produced content [17]. In contrast, Tanagra provides more direct control over sculpting its generative space to produce a level through editing tools for both precise geometry placement and level pacing.

Another increasingly common use for PCG is personalized content creation, either adapting to a player at runtime [18-21] or learning player preferences offline [22,23]. Many of these techniques involve inferring a model of behavior for specific players by observing actions taken in the game world. The beat representation used for levels in Tanagra is an implicit model of player behavior; interesting future work would be to incorporate different styles of play, perhaps learned from many different players, to encourage designers to include more diversity in their levels.

Author-guided level generation tends to place all authorial control over the generator at the beginning of the process [24-26], occasionally allowing editing after the level is complete [27]. For example, the world builder for *Civilization IV* allows the scenario designer to set certain terrain parameters ahead of time, such as the size of the land masses, distribution of water and land, and climate. After the generator creates the initial world, the designer can modify the terrain according to her own desires. However, there is no way to request another map that respects the changes that the designer has made, or that only a part of the level be regenerated. The mixed-initiative nature of Tanagra means that the designer and computer can collaborate throughout the design process. One important exception is the SketchaWorld project [28], which provides a mixed-initiative authoring environment for virtual worlds, including terrain editing and city building. This project faces many of the same design concerns as Tanagra in determining how best to have designers interact with a PCG system. SketchaWorld focuses on building large-scale virtual worlds made up of largely non-interactive structures, whereas Tanagra focuses on building levels that dictate the core gameplay.

A mixed-initiative approach to level generation requires a new set of techniques, especially given the realtime nature of a design tool. Heavily search-based approaches, such as evolutionary algorithms, have been successful in offline level generation and adaptive content creation between play sessions [22,23,29], but their reliance on an unbounded search process makes them too slow for a realtime tool. An interactive design tool requires the same, if not greater, amount of responsiveness as online PCG techniques. Grammar and rule-based approaches show a great deal of promise in this area [30]; however, existing grammar-based design tools are confined to creating large structures that the player does not interact with. For example, *UnrealFacade* [31] enables procedural design of buildings, but aspects of the design that are heavily tied to core gameplay are still entirely authored by hand [32]. Tanagra use of reactive planning and constraint solving permits a designer to manually and procedurally create levels which focus more on the game's mechanics than its aesthetics, giving greater control over the human-machine design process.

Platformers are well-suited for research in procedural level design due to their relatively simple and well-understood rules but emergently complex level designs: despite the simplicity of their mechanics, there is astonishing variety in levels within the genre [5]. For example, the popular game *Sonic the Hedgehog 2* [33] has only three movement mechanics -- running, jumping, and spin dashing – yet a massive variety in level elements and configurations. There have been a number of 2D platformers released recently with procedurally generated levels. *Canabalt* [34] and *Robot Unicorn Attack* [35] are both heavily rhythm-based games where the player is forced to move forward at a constantly increasing rate, and must precisely time his jumps. These levels are built by fitting together large pre-authored chunks according to simple rules for what chunks can be adjacent to each other [36]. *Spelunky* [37] is a rogue-like platformer where pre-authored pieces are fit together on a grid. *Infinite Mario Bros.*[1] [38] is an open source clone of the *Super Mario World* engine in which levels are procedurally generated with an increasing difficulty. These levels are also built by fitting together hand-authored chunks, although at a smaller scale than *Canabalt*'s, and then scattering the level with a number of enemies proportional to

[1] The *Infinite Mario Bros.* engine was also used in the Mario AI level generation competition held at the Computational Intelligence in Games 2010 (CIG'10) conference (http://www.marioai.org). To our knowledge, this competition was the first of its kind. The competition is a testament to the growing field of PCG and the appropriateness of 2D platformers for investigating procedural level generation.

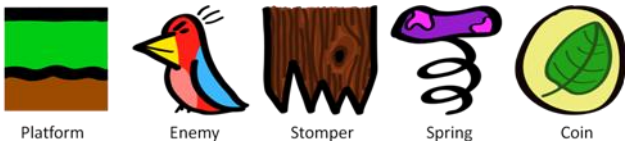

Figure 1. These are the tiles used by Tanagra. Each level component is made up of one or more of these tiles.

the desired difficulty of the level. All of these generation techniques employ relatively large hand-authored sections of a level. These generation techniques work well for their specific game, but cannot easily be applied to other games in the same genre without the significant design burden of authoring large chunks of levels for the generator to use. Spelunky's level generator is especially dependent on its game mechanics, since the player can use tools (that are in-game resources to manage) to modify the level and traverse otherwise impossible terrain. The player's desire to conserve such tools means that many areas of the level will deliberately go unexplored. These aspects of the gameplay mean that there is not a strict requirement for the level generator to ensure that every level it creates is playable, or that every part of the level is guaranteed to be reachable.

Tanagra's level generator differs from these techniques in that the building blocks that it uses are much smaller – a beat encompasses a single player action – and are extensible to different sets of geometry. Furthermore, these building blocks specify patterns for geometry, which may have many different instantiations. Tanagra's level generation technique is built on our previous work in generating levels based on a rhythm that the player feels [26], which in turn is based on Compton & Mateas's work in pattern-based level generation [4].

III. Level Representation

Levels are represented in Tanagra as a series of beats, where each beat corresponds to a single player action. These beats contain level components, also referred to as *geometry*. Supported level components are platforms, gaps, springs, enemies, and stompers. In turn, each of these components is made up of a set of one or more tiles (Figure 1). Tile maps are a common representation for 2D levels; all geometry that is drawn into the level is done at the tile representation layer, which provides a simple interface for creating the level. Tanagra maintains the geometry and beat representations internally. For example, if the designer adds a platform tile adjacent to an existing platform, Tanagra is responsible for ensuring that this tile is grouped into a platform, rather than creating a new platform.

A physics model defines the maximum running speed, maximum jump height, and physics properties of level components, thus guaranteeing that all geometry placed into the level is playable and meets beat duration constraints.

A. Beats

As the building blocks of rhythms, beats are the underlying structure for Tanagra's level generator. A designer can control the pacing of the level by changing their length, or adding, and removing them. A beat represents a single action that is taken by the player, such as jumping or waiting. Its primary role in the level design is to constrain the length of the geometry within it to the distance that can be traversed by the player in the duration of the beat, as calculated from the physics model. Beats are also a convenient way of subdividing the space for the generation algorithm, since geometry for each beat can be generated largely independently.

The action that the player takes can occur at any time between the start and end times of the beat. Each beat has the following properties: constraint variables for the start time, end time, and length of the beat (measured in milliseconds), and knowledge of its preceding and following beats. Beats also keep track of their entry and exit platforms; level playability is guaranteed through beat constraints that match up the exit platform of one beat to the entry platform of its next beat. At any time, the designer can add, remove, or modify a beat, which propagates any changes down to the geometry contained within it.

B. Geometry Pattern Library

Level components are built up into patterns based on the action the player should perform during the associated beat. There is a hierarchy of level patterns, each layer of which builds on the layer below it. At the base of the hierarchy are the following single beat patterns:

- Jumping over a gap from one platform to another
- Jumping to kill an enemy
- Jumping onto a spring
- Waiting before running underneath a stomper

Each of these patterns contains a single user action, and therefore spans a single beat. Gaps can be of variable width, from zero to the maximum length that the player can jump, and variable height, from the maximum height that the player can jump to its opposite value. Examples of each of these patterns are shown in Figure 2. Enemies, springs, and stompers each occupy only one tile, but can have different positions along the platform.

Multi-beat patterns provide further structure to levels and mimic patterns commonly found in 2D platformers. They are composed of the single-beat patterns mentioned above with some additional constraints. The multi-beat patterns that are currently implemented, and the number of beats they span (in

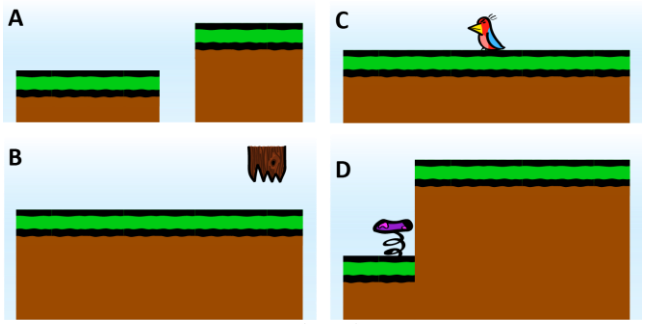

Figure 2. Example instantiations of the four single-beat geometry patterns used in Tanagra. (A) A gap between two platforms, (B) a stomper, (C) an enemy, and (D) a spring to a different platform. Patterns are described and produced using ABL behaviors. Note that there are many different configurations of each pattern; the precise placement of geometry is determined by the constraint solver (Choco).

parentheses), are:

- Gap followed immediately by an enemy (2)
- A valley, consisting of a jump down, an enemy to kill, and then a jump back up (3)
- A staircase, consisting of three gaps in a row, each of them going either entirely up or entirely down (3)
- A mesa, consisting of a jump up, an enemy to kill, and then a jump back down (3)

These more abstract patterns are straightforward to specify due to ABL's hierarchical nature and the separate specification of geometry placement and physics constraints. All of these patterns span consecutive beats, however, this is not a general requirement. For example, it would be possible to specify a more abstract staircase pattern for an entire level, where every other beat contains a jump up over a gap, but the intermediate beats contain randomly selected geometry. In future work we hope to examine these complex patterns in more detail, add more patterns to the library, and allow designers to specify their own patterns.

IV. Design Environment

Figure 3 shows the empty Tanagra environment that designers are first presented with. The main region is the level canvas, a tile-based level editing tool. Below the level canvas is the beat timeline, which is a long grey rectangle subdivided into smaller rectangles which represent beats. The width of each of the beat rectangles corresponds to the length of that beat. On the right of the screen are a number of controls for the designer: zoom controls for the level canvas; a tile selector for placing tiles into the level; controls for pinning, unpinning, and moving level geometry; beat creation, deletion, and resizing controls; and level generator controls.

This section discusses the editing operations available to the designer for both geometry and beat creation and manipulation, and discusses when and why the generator runs.

A. Geometry Editing

The level canvas is the area where both the human designer and computer assistant draw geometry. The canvas, which is initially empty, is made up of a grid of tiles, scaled to fit into the window. The tile-based structure is primarily for ease of design, as tile-based level editors are extremely common for 2D games. However, this structure also provides a reduced search space for the constraint solver, as geometry constraints can be expressed in terms of tiles rather than pixels.

The following geometry editing operations are available to the designer:

- **Drawing Platforms**. A designer can place tiles into the level canvas just as he would for a non-intelligent level editor. When placing platform tiles, Tanagra automatically detects the individual platforms that these tiles create, adding them to the appropriate beat for their position in the level canvas and creating a new beat if necessary.
- **Pin/Unpin Geometry**. Once tiles have been placed into the canvas, geometry components (platforms, enemies, stompers, springs) can be selected and either "pinned" in place so that they stay where they are, or "unpinned" so that the generator can create new geometry in their place.
- **Move Geometry**. Platforms can be selected and moved up and down in the level canvas. When these platforms are moved, the remainder of the level morphs around the newly constrained platform, changing as little as possible.

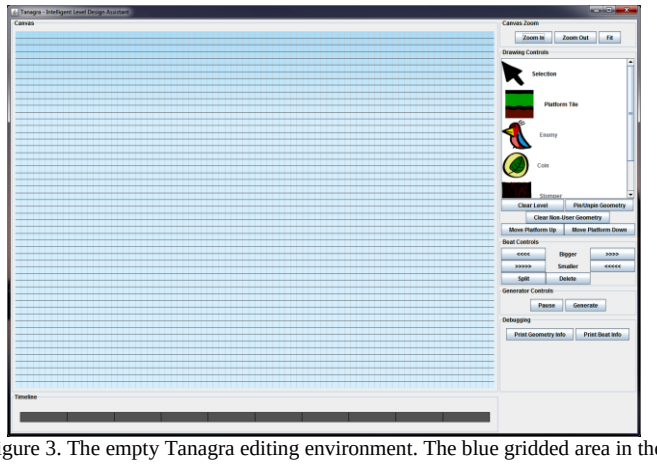

Figure 3. The empty Tanagra editing environment. The blue gridded area in the upper left is the geometry canvas. Below that is the beat timeline. Along the right side of the screen are the editing controls.

B. Beat Editing

The beat timeline provides a mechanism for editing the pacing of the level by inserting or removing beats and modifying their length. Beat changes prompt Tanagra to make geometry changes, allowing pacing changes without the need to manipulate geometry. The following beat timeline editing operations are available to the designer:

- **Resize Beat**. Change the length of the selected beat, automatically adjusting the length of its neighboring beats. The length change can be made at the beginning or end of the beat.
- **Split Beat**. Add a new beat to the level by splitting the selected beat in half. Any geometry contained in the original beat is retained and re-evaluated to fit the new length constraints. Unless it contains user-placed tiles, the new beat has no geometry in it and can have geometry generated for it.
- **Remove Beat**. Remove the beat and its associated geometry, and change the length of its two neighboring beats to be adjacent to each other.

C. Generator Invocation

The designer may request that the generator run at any time during the editing process by clicking the "Generate" button. This button re-generates level geometry, incorporating any user-created geometry. The designer may also request that geometry be re-created for the selected beat. If a new beat is created during editing and the level already contains geometry, then the generator is called for that new beat to ensure that the level remains playable.

The level "re-solves" (i.e. the constraint solver is called again) whenever a change has been made that potentially alters the playability of the level. For example, if a platform is

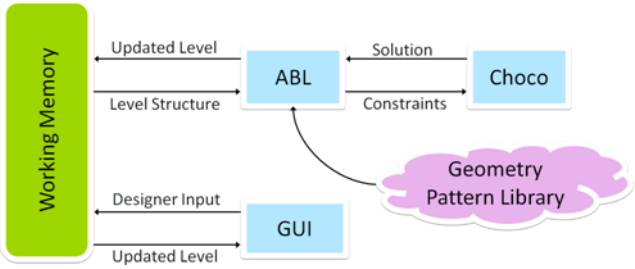

Figure 4. Tanagra is made up of three main components: the GUI, an ABL agent, and the Choco constraint solver. The GUI and ABL communicate through working memory. ABL posts constraints to Choco and determines when the solver should be called; Choco responds with either a potential solution or a notification that no solution exists. A library of geometry patterns are specified using ABL behaviors.

moved then it is important to ensure that move did not make the level unplayable, e.g. if the platform movement caused a gap to be too high to cross. Re-solving the level can cause the generator to be invoked, if no solution can be found by keeping all the geometry patterns in place. This use of the generator is discussed further in Section VIII.

V. Tanagra System Overview

Tanagra integrates reactive planning and numerical constraint solving to perform level generation and enable the editing operations described above. The level generator fulfills the following requirements:

1. Autonomously create levels in the absence of designer input.
2. Respond to designer input in the form of placing and moving geometry.
3. Respond to designer input in the form of modifying the beat timeline.
4. Ensure that all levels are playable.

Figure 4 is a general architecture diagram for Tanagra, showing how the components of the system interact with each other. We use the reactive planning language ABL (A Behavior Language) [2] to respond to designer input, choose the geometry that should be placed for each beat, and communicate with Choco. Choco [3] is the constraint solving library used to specify and solve constraints on the placement of different level components. The Tanagra ABL agent can be imagined as a subordinate assistant to the primary level designer. It can suggest different potential designs based on what has been done so far, and obeys the primary designer's commands for geometry or pacing changes. This ABL "assistant" has a colleague, Choco, which determines the precise physical placement of components in the level, and is responsible for reporting to the designer if his changes lead to an unplayable level.

Tanagra works quickly enough to permit rapid reaction to designer input. It can generate parts of levels in response to a change made by the designer, or can regenerate the entire level on demand, while respecting any constraints placed by the human designer. We have found it useful to keep separate the choice of a geometry pattern (using ABL) from the instantiation of that pattern (using Choco), as the precise placement of level components is influenced by surrounding

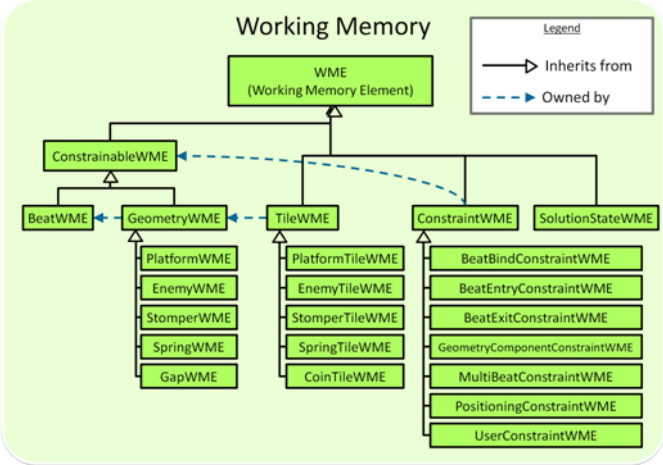

Figure 5. The hierarchy of working memory elements used to represent levels in Tanagra. Solid, black arrows denote inheritance; dashed blue arrows denote ownership. For example, *ConstraintWME*s have multiple owners, each of which must be a *ConstrainableWME*.

geometry. This means that the placement of components in one beat may be able to change based on the placement of level components in a later beat, while still maintaining the same geometry pattern in both beats.

Many different configurations of component placement meet the same geometry pattern. For example, a jump to a different platform could have a short initial platform and a long later platform, or vice versa. The search for a valid level occurs in two stages: ABL searches at the structural, pattern level, and Choco searches for a valid, numerically-parameterized instantiation. The pattern abstraction also permits adding new kinds of design patterns easily, as instead of specifying all possible combinations of geometry components, we can instead specify rules for the construction of the pattern.

This section provides an overview of how ABL and Choco each work, their responsibilities in Tanagra, and how they communicate with each other. Sections VI and VII provide a more detailed explanation of how different generation and editing operations are implemented.

A. *Reactive Planning with ABL*

ABL is a Java-based reactive planning language created by Mateas & Stern for use in creating believable agents that can rapidly react to a changing world state. Reactive planning, as a paradigm, is focused on domains where the world can change out from underneath an agent's own actions: this is exactly the situation in mixed-initiative level generation, where the designer makes manual edits that warrant a response from Tanagra. The world state is communicated using a blackboard architecture with Working Memory Elements (WMEs), which represent facts that the automated design agent knows and reasons about. These facts can be asserted or retracted from the agent's memory by either the human or the computer. Every object in the world that the Tanagra ABL agent must interact with is stored as a WME: Figure 5 shows the inheritance hierarchy for all WMEs used in Tanagra.

ABL agents are authored as a set of hierarchical *behaviors* that can be performed towards *goals*, where each of these

```
//Use a platform that already has this beat and these properties
sequential behavior placePlatform(BeatWME beat, boolean entry,
                                  boolean exit)
{
    precondition {
        plat = (PlatformWME owner==beat isEntry==entry isExit==exit)
    }
    mental_act {
        beat.addGeometry(plat);          //add geometry to beat
        plat.setReadyForSolving(true);   //platform ready to solve
    }
}

//Worst case scenario: make a new platform
sequential behavior placePlatform(BeatWME beat, boolean entry,
                                  boolean exit)
{
    specificity -1; //choose this behavior last
    precondition {
        !(PlatformWME owner==beat isEntry==entry isExit==exit)
        (beat.numPlatforms() < 2)
    }
    mental_act {
        //create a platform with desired properties and add to beat
        PlatformWME plat = new PlatforMWME(beat, entry, exit);
        beat.addGeometry(plat);

        //add the new platform to working memory
        BehavingEntity.getBehavingEntity().addWME(plat);
    }
}
```

Listing 1. Two different behavior definitions for placing a platform into a specific beat, given that the platform must have the specified entrance and exit properties in relation to the beat.

behaviors can have a number of subgoals, similar to the behavior and goal relationship in a hierarchical task network. These behaviors can operate either in sequence or in parallel. Behaviors "ground out" in direct actions that should be taken in the world. In Tanagra's case, these actions are expressed in *mental acts*, which are written as normal, sequential Java code. There are typically several behaviors that can fulfill a particular goal. A particular behavior is chosen to fulfill the goal based on whether or not its precondition is true, and what internal priority (*specificity*) is given for it. If a behavior fails for any reason, the next most suitable behavior is chosen to fulfill the goal.[2]

For example, Listing 1 shows the ABL code for two out of the five different placePlatform behaviors used in Tanagra's geometry creation. The first of these behaviors handles the situation in which there is already a PlatformWME whose beat owner, entry, and exit conditions exactly match those specified in the behavior's arguments. This platform is bound to the variable plat. If the precondition succeeds, then plat is added to the beat's geometry set and is flagged to be ready for solving by Choco. If this behavior's precondition fails, then it is still possible for the second behavior to execute for the goal. The second behavior handles the case where there is no PlatformWME that fulfills the stated requirements, and less than two platforms already in the beat. In this case, a new platform is created (which, by default, is ready for solving)

[2] For a more complete description of ABL's semantics, inner workings, and design idioms we refer the reader to other literature on the topic [41,42].

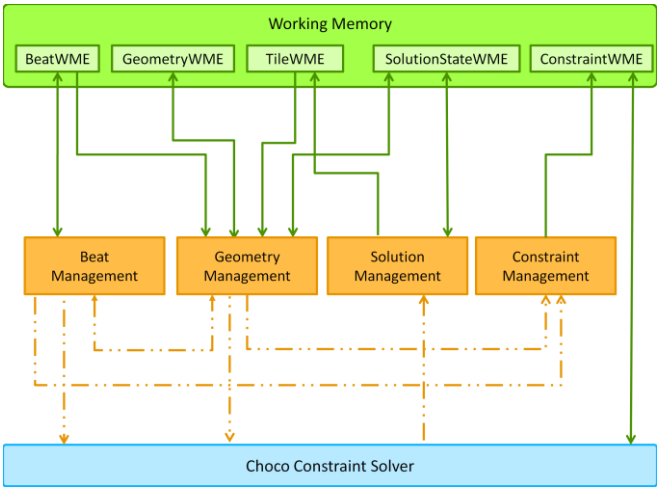

Figure 6. This diagram shows the communication channels between working memory, ABL, and Choco, and dependencies between different managers within ABL. Green solid lines denote WME reading and writing, and orange dashed lines denote behavior and function calling.

and added to working memory. Similar behaviors exist for placing gaps, enemies, stompers, and springs into the level, as well as for adding and removing constraints and handling beat changes.

Tanagra uses a number of parallel behaviors called *managers* which wait for a change in a certain aspect of the world state and then initiate behaviors that react to that change. Much of the intelligence behind Tanagra comes from these managers working towards different, but related, goals in concert. For example, consider a scenario in which a designer adds a tile to an existing partial level. The ABL behavior that reacts to this change may add or extend an existing piece of geometry, which in turn modifies the length of the beat containing the tile. The change also mandates the need for further constraints to be placed on the level. All these changes may also lead to a need for different geometry to be generated for the remainder of the level, to maintain its playability. Each of these scenarios is handled by separate managers, which are specified independently from each other.

There are four different categories of managers. Beat management and geometry management handle changes made to beats and geometry, respectively. In parallel, the solution managers addresses solution management and assignment, and the constraint managers handle adding constraints to the level as necessary. Each manager has a different priority; for example, beat management has a higher priority than geometry management, since the geometry that can be placed is dependent on its beat's length being predetermined.

Figure 6 shows the communication model for Tanagra between working memory, ABL managers, and the Choco constraint solver.

B. Numerical Constraint Solving with Choco

Choco is a Java library for modeling and solving numerical constraint satisfaction problems. Constraints can be expressed on variables that are boolean, integer, or real numbers. These constraints are then read by Choco's constraint solver, which

finds a value for each variable such that all constraints are satisfied.

We use Choco to model constraints for the geometric relationships within and between level components in geometry patterns, and also for the relationship between each geometry pattern and its associated beat. These constraints are designed to ensure that the level is playable by taking into account a simple avatar physics model. Tanagra is aware of the player's movement speed, initial jump velocity, and the impact that springs have on this jump velocity.

Each type of level component has a number of constraint variables associated with it. A platform's variables are a start and end position and a width. Enemies, stompers, and springs have an x and y coordinate. Gaps have a width and height. Each component also has a length variable associated with it, which states the amount of time the player takes to interact with that component in an ideal playthrough – i.e. a situation in which the player character takes minimal time to traverse the level. This allows a single numerical constraint – that the sum of the lengths of all components equals the length of the beat they inhabit – to maintain the desired length and pacing of the level.

All constraints store a set of their "owners", which are all *ConstrainableWME*s (i.e. beats and geometry components). This ownership property makes it straightforward to remove constraints whenever geometry or beats are modified or deleted. As seen in Figure 5, there are a number of different kinds of constraints, each of which serves a different purpose in ensuring the playability of levels. Constraints are differentiated to ensure that some of them (such as those that specify internal geometry constraints) are never retracted during solving, and others (such as those that force geometry to a specific position) are retracted at the appropriate time. The *ConstraintWME*s below encode facts about the structure of the level:

- **BeatBindConstraintWME.** This is a constraint that binds the exit point of one beat to the entry point of the next beat, and vice versa. It ensures that geometry between beats "lines up".
- **BeatEntryConstraintWME, BeatExitConstraintWME.** These constraints bind the start point of the entry platform for a beat to the entry point of that beat, and the end point of the exit platform for a beat to the exit point of that beat.
- **GeometryComponentConstraintWME.** This is a constraint internal to specific geometry. For example, the expression that the width of a platform is equal to its ending x position minus its starting x position.
- **MultibeatConstraintWME.** This is a constraint that is applied to geometry within patterns that span multiple beats. For example, the staircase pattern consists of three gaps between platforms. The multibeat constraints for these gaps ensure that each gap has identical width and height.
- **PositioningConstraintWME.** This is a constraint that is placed on every geometry component when a solution is found, that forces geometry to stay in the same position whenever possible. Positioning constraints are frequently added and removed during the geometry search process.
- **UserConstraintWME.** A constraint placed on geometry by the user, either when drawing platforms into the canvas or by pinning geometry in place. User constraints are never removed unless the designer unpins or removes the constraints' owner.

Choco is called each time the generator places geometry for a beat; typically, solutions are found within 5 milliseconds, on a 3GHz Intel Core 2 Duo. It can take significantly longer to exhaustively determine that there is no solution, so Tanagra stops Choco from searching after 50 milliseconds, since it is unlikely to find a solution after that point. Choco is set to attempt a random integer value in the domain (post-propagation) of the variable, allowing it to create a variety of levels even when using the same geometry patterns.

VI. Geometry Management and Generation

The majority of ABL behaviors in Tanagra deal with managing user-provided geometry and generating new geometry. These behaviors can be grouped into three major categories: expression and implementation of hierarchical geometry patterns, creating level components from user-placed tiles, and incorporating user-created geometry into patterns. This section will discuss in detail how ABL and Choco interact to perform these tasks.

A. Hierarchical Geometry Patterns

At its base, a geometry pattern consists of a set of level components and constraints that are asserted on these components. Geometry patterns can contain other geometry patterns, and optionally additional constraints on components that exist between patterns. Once a pattern has been assigned to a beat, the solver ensures that at least one instantiation of the pattern still allows the level to be playable.

Tanagra's representation for levels mandates that no beat may contain more than two platforms, since each beat may contain only a single player action, such as jumping or waiting. An enemy or a stomper pattern uses a single platform; a gap or spring pattern uses two platforms. Platforms are labeled by whether or not they are an entry or exit platform for a beat. Playability is guaranteed by constraining the exit platform of one beat to have an identical vertical coordinate to the entry platform of its neighboring beat. While the level may still be playable if the platforms had positions that differed by only one or two tiles, this difference would constitute a separate player action (i.e. a jump between platforms) which would belong in a separate beat.

Figure 7 shows a diagram representing a partially specified level; Table 1 describes its constraints. In this example, the first three beats in the level are assigned the *valley* pattern. This pattern consists of a jump down over a gap, followed by a platform with an enemy on it, and finally a jump over a gap, consuming three beats in total. In addition to the internal constraints for each geometry pattern, the *valley* pattern also imposes the constraint that the gaps in the first and third beats should have values that are consistent with building a valley: the first gap should cause the player to jump down to the platform, and the second gap should have an equal height to

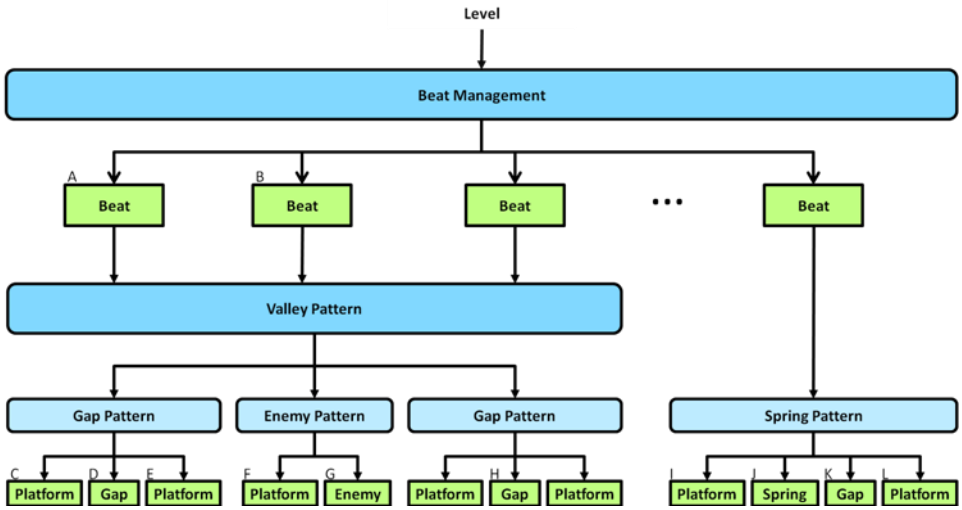

Figure 7. A diagram representing a potential partial level during level generation. Beats A – C are consumed by the Valley Pattern, which in turn is a gap pattern, enemy pattern, and another gap pattern. Beat D is assigned the Spring pattern, which places two platforms, a spring, and a gap. Blue, rounded boxes denote ABL behaviors. Green boxes are WMEs. Not represented here are the tiles that make up each level component. Table 1 provides examples of the constraints imposed between each level component.

the first but in the opposite direction. The final beat of the partial example contains the spring pattern, which consists of two platforms, a gap between them, and a spring that sits on the first platform for the player to jump on. This pattern has all the constraints that a gap pattern has, but also the constraint that the spring must sit on the end of the first platform. The spring modifies the domain values of the gap to support a larger height, according to the physics model for the game. Playability is ensured within beats by the constraints for each pattern; it is ensured across the entire level by matching up exit and entry platforms between beats.

TABLE 1. A REPRESENTATIVE SAMPLE OF THE CONSTRAINTS IMPOSED ON THE EXAMPLE LEVEL STRUCTURE SHOWN IN FIGURE 7.

Cause	Constrained	Constraint equation
Beat Management	A, B	A.exitPointX = B.entryPointX A.exitPointY = B.entryPointY
Beat Management Gap Pattern (1)	A, E	A.exitPointX = E.endX A.exitPointY = E.posY
Beat Management Enemy Pattern	B, F	B.entryPointX = F.startX B.entryPointY = F.posY
Beat Management Gap Pattern (1)	A, C, D, E	C.length + D.length + E.length = A.length
Valley Pattern	D, H	D.width = H.width D.height = -1*H.height D.height > 0
Gap Pattern (1)	C, D, E	C.endX + D.width = E.startX C.posY + D.height = E.posY
Enemy Pattern	F, G	G.posY = F.posY - 1 F.startX <= G.posX <= F.endX
Spring Pattern	I, K, L I, J	I.endX + K.width = L.startX J.posY = I.posY - 1 J.posX = I.endX
J (Spring)	K	K domain change due to J's physics modification

B. *Creating User Geometry from Tiles*

The designer interacts with the level canvas by placing individual tiles into it. Since all level generation and playability verification is performed on level components rather than individual tiles, Tanagra must determine the appropriate level components based on the tiles that it detects such that Tanagra's internal representation for levels is not violated. This geometry is then incorporated into geometry patterns as appropriate.

Platforms, like any other geometry, are "owned" by a particular beat. While handling platform placement from tiles, it is also important to determine if the placed platform is an entry platform, exit platform, both, or neither, to assist with the pattern assignment described above. Also, constraints must be added to the system to ensure that the newly created platform stays in the place indicated by the designer.

This entire process of creating platforms from tiles is accomplished in two stages, using two different managers: the first, higher priority, manager is responsible for constructing platforms from tiles, and the second is responsible for placing constraints on the platform and determining entrance and exit properties. When creating a platform from tiles, ABL determines if the tile belongs to a platform that isn't already in working memory, should be added to an existing platform, or should cause the merging of two neighboring platforms. Beat ownership and entry/exit determination are calculated by the position of the tile: from this position, and an understanding of the movement speed of the avatar, we can calculate the beat that the platform must belong to. A newly created platform is given constraints such that the tiles placed by the designer will always be contained in the platform, but the platform may eventually contain additional tiles. This allows platforms to extend to fit whichever geometry pattern they are assigned to.

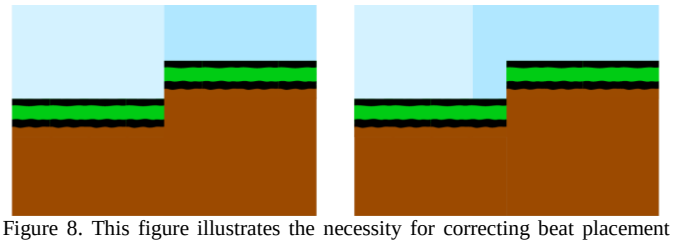

Figure 8. This figure illustrates the necessity for correcting beat placement based on geometry placement. The blue shaded boxes denote beat boundaries. If the designer draws geometry as depicted on the left, there can never be a solution to the level as the platforms do not line up at the beat boundary. However, this geometry placement should clearly be playable. The right side shows how Tanagra corrects beat boundaries to address this issue.

C. *Incorporating User-Created Geometry into Patterns*

User-created geometry is incorporated into patterns through additional behaviors that check to see if a certain geometry component already exists in a beat before creating new geometry. Because constraints are expressed on individual geometry components and beats, rather than on the pattern as a whole, it is possible to simply swap out the existing platform for the platform that otherwise would have been generated, and have the constraints expressed as normal. The additional user constraints placed during platform inference remain associated with the platform, and any additional constraints placed on the platform are redundant.

VII. Beat Management

There are four main design concerns that Tanagra must be able to reason about with regard to the beat timeline: handling responses to the designer making a change to the length of a beat, splitting a beat to create a new one, deleting a beat, or adding too many platforms to a single beat. Each of these are resolved by separate ABL managers. The first three managers work simply by monitoring for a flag that is set on a beat when the designer initiates one of these changes. When the flag appears, a new behavior is subgoaled to handle the necessary changes.

The *TooManyPlatformsManager* handles a slightly more subtle problem that can arise from a designer placing geometry into the level canvas. Tanagra begins with a default number of beats in the timeline, but the designer is capable of freely drawing platforms wherever (and however frequently) she wishes. Recall, however, that a single beat is only ever allowed to contain at most two platforms, which are the beat's entry and exit platforms. Therefore, when a designer places more than two platforms into a beat, Tanagra must respond by splitting beats in an appropriate location. An associated manager, the *PlatformMatchupManager*, handles a related problem: if the designer draws two platforms at different vertical positions, but whose end-points match up to the same beat boundary, Tanagra should be able to move the beat boundary in attempting to ensure level playability (Figure 8). These two managers constantly monitor for platforms that violate these rules, and raise the beat length change flag or beat split flag as appropriate.

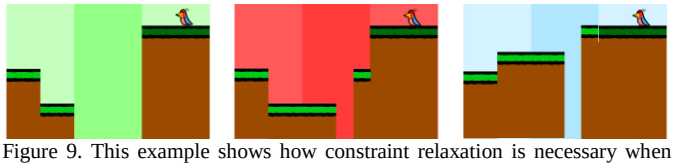

Figure 9. This example shows how constraint relaxation is necessary when creating levels. On the left, a level with a user-specified platform on the right is being generated. A gap pattern has been chosen for the first beat, and Choco has instantiated this pattern as a jump down. In the middle example, the second beat also has a gap pattern, but using the existing constraints for the first beat, the level is unplayable. The right example shows how relaxing the positioning constraints for the first beat renders the level playable, without changing any of the geometry patterns.

Figure 10. A small example illustrating the need for geometry search. On the left, a partial level is being solved, where existing geometry in the first and third beats is pinned in place by the user. A geometry pattern must be selected for the middle beat. In the middle level, Tanagra selected the stomper pattern to fill the middle beat, which leads to an unsolvable level, as the exit platform and entry platforms for beats two and three can never line up. On the right, Tanagra has instead selected the spring pattern, which leads to a playable level segment.

VIII. Constraint Solving and Search

The motivating force behind Tanagra's solving and search process is to minimize the number of global changes that must be made to a level in order to resolve small, locally made changes from either the human or computer designer. This is essential to providing the designer with an editing environment in which the response to any changes made to the level continue to ensure playability while leaving the level as similar as possible to its prior state. However, the designer should still be able to request a complete re-generation of the level at any time, in order to see different potential levels given the current constraints. This motivation leads to a two-stage solution and search process whenever a change is made to the level. The first stage iteratively relaxes constraints on the level; if necessary, the second stage iteratively re-generates small sections of the level until a solution is found.

Recall that ABL assigns positioning constraints to level components upon solving, to minimize the changes made to the level when new geometry is generated. These constraints must often be partially relaxed when new geometry is added (Figure 9). There are also many situations in which the first geometry pattern selected for a beat would be invalid. For example, consider a scenario in which the designer has drawn two long platforms that are separated by a single beat. These platforms have different y values, so the connecting geometry for the middle beat could not be another single platform, as the endpoints would not line up. However, using a different geometry pattern, such as a spring, may complete the level geometry to make a playable level. Figure 10 illustrates this scenario.

These two scenarios lead to the need for multi-stage constraint solving on Choco's side, and backtracking on ABL's side.

A. Constraint Solving

ABL calls the constraint solver by subgoaling the *Solve* behavior. This behavior takes as a parameter a list of *ConstrainableWMEs* whose positioning constraints should be ignored, called an "ignore set". *Solve* is subgoaled after each attempt to place a new geometry pattern during geometry generation, and after modifying the beat time through adding, deleting, or changing the length of a beat. The list of *ConstrainableWME*s initially consists only of the new geometry components added to the level, or the beats affected by the timeline change. The solver iterates over every *ConstraintWME* in working memory, excluding only positioning constraints belonging to objects in the ignore set. If a solution is not found, then positioning constraints are slowly relaxed outward from the original ignore set. On each attempt to solve, the solver finds the ignore set's "frontier" beats (i.e. those that are in the set but whose neighbors are not) and randomly chooses one of these frontier beats to add to the ignore set. Constraints imposed on geometry by the user are never added to the ignore set. When the frontier meets user-specified geometry or the edge of the level, there is no solution for this particular configuration of geometry, and ABL must search for a solution (Section VIII.B). By running the solver so many times just for a single geometry pattern placement, this algorithm does incur a high solving cost, but provides the benefit of minimizing global changes. Even with the solver running so frequently in the worst case scenario, Tanagra still can find valid levels at an interactive rate.

B. Searching for a Solution

If the solver does not find a solution, then we can be sure that the last change made to the level has resulted in making it unplayable. There are two potential causes for this:

1) The concrete geometry corresponding to the most recent change leads to the level being unplayable, and should be replaced.

To address this issue, Tanagra maintains a solution state. This state consists of the geometry patterns that are being used at any given time, indexed by the beat they belong to, and a set of states that are known to be invalid. Whenever the solver runs, a *CleanupSolver* behavior is subgoaled. This behavior can do four different things, depending on the solution state. If the solver succeeds, then the *CleanupSolver* behavior simply records a success. However, if the solver fails, then the behavior must record the state currently being attempted as invalid and all non-user geometry and constraints should be removed from the beat. If there are remaining geometry patterns to be tried for a beat, then the cleanup behavior is complete, and the *MissingGeometryManager* will take care of attempting a new pattern. If there are no remaining geometry patterns for the beat, then non-user geometry and constraints are removed from both that particular beat and its neighbors, as the solver's failure signifies a more global problem with geometry placement. This search continues until either a solution is found, or it is determined that the failure to solve is due to a different cause:

2) The player has imposed constraints that contradict each other, or drawn geometry that can never lead to a solvable level.

Due to the requirement that Tanagra respect every constraint that the designer places on a level, it is possible that the designer will place constraints that conflict with each other. This is most common during geometry placement, by placing platform tiles that, no matter the intervening geometry, would never be reachable from each other given the game's mechanics. In this scenario, Tanagra changes the background of the level canvas from pale blue to red, denoting the lack of a solution. From here, the designer can begin manually removing offending geometry.

IX. Expressivity

As a level generator that must support designers, it is especially important that Tanagra can create a wide range of levels. While it is easy to show that Tanagra can create a large quantity of levels, we feel it is more interesting to examine the qualities of the levels that are produced, and compare how similar they are to each other. In this section, an analysis of Tanagra's *expressive range* is presented, using a similar technique[3] to that defined in our previous work in analyzing and visualizing the expressive range of a level generator [39]. Two different metrics are used for comparing levels: the linearity of a level, and how lenient the level is towards the player. These metrics are then applied to levels that are entirely procedurally generated using the default beat timeline.

Linearity. The first metric for evaluating Tanagra's expressivity is evaluating the "profile" of produced levels. We do this by fitting a line to the produced geometry, and determining how well the geometry fits that line. The goal here is not to determine what exactly the line is, but rather to understand Tanagra's ability to produce levels that range between highly linear and highly non-linear. The linearity of a level is measured by performing linear regression on the level, taking the center-points of each platform as a data point. Each level is then scored by taking the sum of the absolute values of the distance from each platform midpoint to its expected value on the line, and dividing by the total number of points. Results are normalized to a [0, 1] scale, with 0 being highly non-linear and 1 being highly linear.

Leniency. While we hesitate to classify the difficulty of Tanagra's levels, as such a measure would be a) subjective, and b) highly dependent on the ordering of geometry, we can classify how "lenient" each geometry pattern is to the player. Clearly, a level made up of mostly gaps and enemies is far less forgiving than a level with no gaps and a number of long, flat platforms. To measure this, we define a leniency score that is the weighted sum of chosen geometry patterns in the level, normalized by the number of beats in the level. A score of 1 describes a highly lenient level, 0 is a non-lenient level. Scores for each pattern are as follows:

- Gap (width > 0): 0.0
- Enemy: 0.0

[3] Note that the details of the metrics themselves have changed from this prior work. Scales are now reset such that the higher the score for a metric, the better the level fits its metric. Leniency is now also on a [0, 1] scale.

- Stomper: 0.5
- Spring: 0.5
- Gap (width = 0): 1.0

These two metrics allow us to compare produced levels and describe Tanagra's expressive range by categorizing a representative sample of levels it produces into bins based on their linearity and leniency scores. Figure 11 shows the expressive range of the generator when it runs without any user constraints. Each hexagon is colored to indicate the number of generated levels that have the corresponding linearity and leniency scores. This data is collected by generating 10,000 levels without any user-placed geometry or modifications to the beat timeline. Each of these 10,000 levels is unique, although there are only 7,201 unique pattern combinations in that set; i.e. there are some levels that have an identical combination of patterns but different geometry positioning. Repetition in levels at the pattern layer is largely due to the influence of multi-beat patterns: they have an equal likelihood for selection as a single-beat pattern, and restrict the other patterns that can be chosen.

This expressive range is slightly skewed towards creating linear, non-lenient levels, with a peak in the graph at linearity=0.65 and leniency=0.3This graph does imply a limit to the expressivity of the generator: not many generated levels are both highly non-linear and highly lenient. However, the number of unique levels and level patterns implies that there is a reasonable amount of variation within each of the bins shown in the graph. Since a designer will be constructing a single level, it is encouraging to see that Tanagra provides good coverage across both linearity and leniency. Figure 12 shows a level that fits into the largest bin in Tanagra's expressive range. We expect that more sophisticated patterns of geometry, in terms of both new geometry components and more hierarchical patterns, would improve the expressivity of Tanagra a great deal, and hope to investigate this in future work.

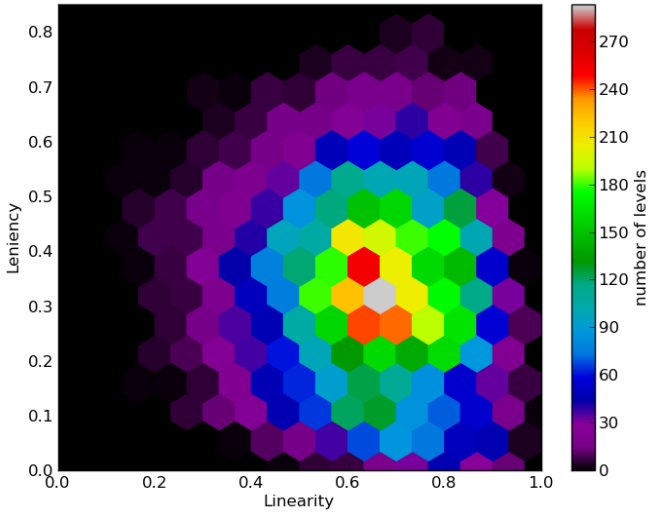

Figure 11. Tanagra's expressive range, measured on the linearity and leniency of generated levels. 10,000 unique levels were generated and graded by these metrics. In this visualization, each hexagon corresponds to a bin containing levels and is colored by the size of each bin according to the scale at right.

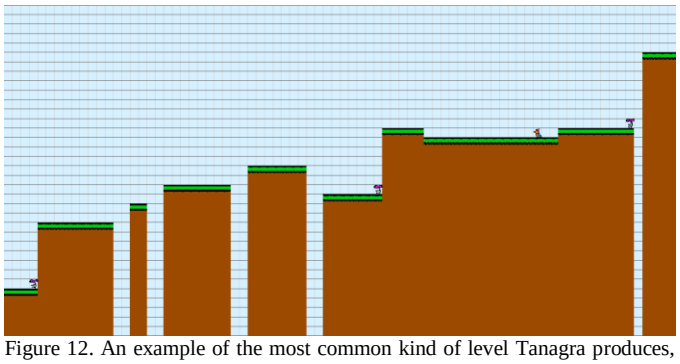

Figure 12. An example of the most common kind of level Tanagra produces, according to expressive range analysis shown above.

X. Use Scenario

This section presents a detailed use scenario, showcasing Tanagra's key abilities: auto-filling geometry, brainstorming level ideas, and manipulating level pacing. Figure 13 illustrates this scenario with screenshots taken at key moments. A video showing Tanagra in action is also available online[4].

Tanagra can generate levels with or without initial designer guidance. Figure 13(A) shows an example level generated without any input, using the default beat timeline, while (B) shows a level with partially designer-specified geometry, which altered the beat timeline to ensure that there are never more than two platforms per beat. Tanagra can rapidly regenerate the level to show different variations that meet the same requirements (Figure 13C, D). Geometry for a beat can be moved around within that beat at any time; for example, Figure 13E shows that the level has been modified to have the closing platform moved significantly higher. Notice that only the geometry at the end of the level has changed to accommodate this alteration. Tanagra also supports editing the beat structure itself, in addition to manipulating geometry. In Figure 13F, the designer has significantly changed the pacing of the level by adding a number of beats to the middle and deleting beats at the beginning and end of the level. The designer then extends the lengths of the middle beats slightly in Figure 13G. Finally, Tanagra is capable of re-generating geometry for specific beats rather than the entire level. The designer completes the level in Figure 13H by regenerating geometry for the third beat and second to last beat.

XI. Discussion and Future Work

Tanagra is a mixed-initiative level design tool for 2D platformers that supports a human designer through procedurally generating new content on demand, verifying the playability of levels, and allowing the designer to edit the pacing of the level without needing to manipulate geometry. An important motivation in the design of Tanagra was the desire to focus on how the mechanics of the game relate to designing levels, and being able to reason about the gameplay that levels provide. This article has presented Tanagra's level representation, how it integrates reactive planning and numerical constraint solving, an evaluation of its expressive range, and a detailed example of how it can be used.

[4] http://users.soe.ucsc.edu/~gsmith/tanagra/v2_demo/demo.htm

Tanagra is itself part of a larger intended system that will support more editing operations and different views on the level. We envision a final design tool that provides an ability to directly edit entire level paths, which are made up of linear sections built with Tanagra, without needing to focus on fine-grained geometry details. To reach this goal, there are a number of interesting questions that must be addressed.

A major issue that must be addressed is that of how to handle conflicting user constraints; for example, if the designer carefully crafts two different sections of a level that has no valid connecting geometry, Tanagra currently merely informs the designer there is no solution. However, a better solution may be to suggest levels that incorporate each section in different locations, or relax constraints on one part of the level but not on another part. It can also be difficult to determine the "correct" action to take in certain situations; for example, when splitting a beat in half, how should any existing geometry for that beat be divided up? We propose to address these issues in future work, which will examine ways to detect and present meaningfully different design variants.

Related to this, there is plenty of further work to be done in improving the expressivity of Tanagra and ways to measure it. The current expressive range of Tanagra is fairly large with only a few different geometry patterns; we hope to add more geometry patterns that correspond to different kinds of player actions. For example, the canonical loop-de-loops from *Sonic the Hedgehog 2*, gaps that require double-jumping, and obstacles that involve wall-jumping would be interesting components to add and could greatly expand the number and variety of levels that can be produced. Adding these patterns would require creating new ABL behaviors for pattern placement and responses to geometry, and associated constraints that reflect a more sophisticated physics model. It would also be interesting to explore ways for designers to specify their own geometry patterns and constraints without needing to edit ABL or Choco code directly.

There is also work to be done in determining additional metrics for levels to be used in expressivity analysis, and in presenting that information to the designer. A designer's interactions with Tanagra can be visualized as iteratively sculpting the generative space: for example, requiring that a level have more enemies would shift the shape of the expressive range graph towards the non-lenient end of the y-axis. This information could be useful feedback to novice designers, who may be unaware of how their changes are impacting the experience the player will have with the level.

Other future work with Tanagra involves running user studies to study how people interact with an intelligent level design assistant, and a more quantitative study to examine how Tanagra assists designers in effectively and efficiently creating a variety of levels.

Finally, we intend to explore how the combination of reactive planning and constraint solving can apply to genres other than side-scrolling platformers. We believe the technique of separating abstract geometry patterns from physical, numerical constraints could work well in genres such as first-person shooters, which employ common design patterns such as chokepoints and arenas [40]. We expect that the representation for such levels would differ quite substantially, however. The beat timeline offers a powerful editing mechanism for controlling the pacing of a 2D platformer level, but is less appropriate for games that rely more on tension, flow, and story. For example, an alternative representation for a role-playing game might be a graph describing the structure of quests. There are also additional complexities arising from choices that the player could make, from character ability customization to different paths taken through a level.

This paper has presented Tanagra, a level design tool for 2D platformers that allows a human designer to interact seamlessly with a procedural content generator. This mixed-initiative approach offers a way to reduce authorial burden in level design while still allowing human designers to exercise their creativity and aesthetic judgment.

REFERENCES

[1] J. Novak and T. Castillo, *Game Development Essentials: Game Level Design*, Delmar Cengage Learning, 2008, p. 320.

[2] M. Mateas and A. Stern, "A behavior language for story-based believable agents," *IEEE Intelligent Systems*, vol.17, iss.4, 2002, pp. 39-47.

[3] Choco Team, "Choco: an Open Source Java Constraint Programming Library," *White Paper, 14th International Conference on Principles and Practice of Constraint Programming, CPAI08 Competition*, Sydney, Australia: 2008.

[4] K. Compton and M. Mateas, "Procedural Level Design for Platform Games," *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE06)*, Palo Alto, CA: 2006.

[5] G. Smith, M. Cha, and J. Whitehead, "A Framework for Analysis of 2D Platformer Levels," *Proceedings of ACM SIGGRAPH Sandbox Symposium 2008*, Los Angeles, CA: 2008.

[6] B. Lawson and S.M. Loke, "Computers, words and pictures," *Design Studies*, vol.18, iss.2, 1997, pp. 171-183.

[7] T. Lubart, "How can computers be partners in the creative process: Classification and commentary on the Special Issue," *International Journal of Human-Computer Studies*, vol.63, iss.4-5, Oct. 2005, pp. 365-369.

[8] N. Negroponte, "Soft Architecture Machines," *The New Media Reader*, N. Wardrip-Fruin and N. Montfort, eds., Cambridge, MA, USA: The MIT Press, 2003, pp. 353-366.

[9] M. Resnick, B. Myers, K. Nakakoji, B. Shneiderman, R. Pausch, T. Selker, and M. Eisenberg, "Design Principles for Tools to Support Creative Thinking," *NSF Workshop Report on Creativity Support Tools*, Washington, DC: 2005.

[10] D. Isla, "Next-Gen Content Creation for Next-Gen AI," *Invited Talk: 2009 International Conference on the Foundations of Digital Games (FDG 2009)*, Orlando, FL: 2009.

[11] C. Satchell, "Evolution of the Medium - Positioning for the Future of Gaming," *Keynote: 2009 International Conference on the Foundations of Digital Games (FDG 2009)*, Orlando, FL: 2009.

[12] M. Toy, G. Wichman, K. Arnold, and J. Lane, *Rogue*, 1980.

[13] D. Braben and I. Bell, *Elite*, Acornsoft, 1984.

[14] Blizzard North, *Diablo*, Blizzard Entertainment, 1997.

[15] S. Meier, *Civilization*, MicroProse, 1991.

[16] T. Adams, *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress*, Bay 12 Games, 2006.

[17] J. Regier and R. Gresko, "Random Asset Generation in Diablo 3," *Invited Talk, UC Santa Cruz*, Oct. 2009.

[18] M. Booth, "The AI Systems of Left 4 Dead," *Keynote, Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE09)*, Palo Alto, CA: 2009.

[19] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: A Model for Dynamic Level Generation," *Proceedings of the Sixth*

Artificial Intelligence in Interactive Digital Entertainment Conference (AIIDE10), Palo Alto, CA: 2010.

[20] E.J. Hastings, R.K. Guha, and K.O. Stanley, "Automatic Content Generation in the Galactic Arms Race Video Game," *IEEE Transactions on Computational Intelligence and AI in Games*, vol.1, iss.4, 2009, pp. 245-263.

[21] M. Nitsche, C. Ashmore, W. Hankinson, R. Fitzpatrick, J. Kelly, and K. Margenau, "Designing Procedural Game Spaces: A Case Study," *Proceedings of FuturePlay 2006*, London, Ontario: 2006.

[22] J. Togelius, R. De Nardi, and S.M. Lucas, "Towards automatic personalised content creation for racing games," *IEEE Symposium on Computational Intelligence and Games 2007 (CIG07)*, Honolulu, HI: 2007, pp. 252-259.

[23] N. Shaker, G.N. Yannakakis, and J. Togelius, "Towards Automatic Personalized Content Generation for Platform Games," *Proceedings of the Sixth Artificial Intelligence in Interactive Digital Entertainment Conference (AIIDE10)*, Palo Alto, CA: 2010.

[24] K. Hullett and M. Mateas, "Scenario Generation for Emergency Rescue Training Games," *Proceedings of the 2009 International Conference on the Foundations of Digital Games (FDG 2009)*, Orlando, FL: 2009.

[25] T. Tutenel, R.M. Smelik, R. Bidarra, and K.J. de Kraker, "Using Semantics to Improve the Design of Game Worlds," *Proceedings of the Fifth Artificial Intelligence in Interactive Digital Entertainment Conference (AIIDE09)*, Palo Alto, CA: 2009.

[26] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-Based Level Generation for 2D Platformers," *Proceedings of the 2009 International Conference on the Foundations of Digital Games (FDG 2009)*, Orlando, FL: 2009.

[27] Firaxis Games, *Civilization IV*, 2K Games, 2005.

[28] R.M. Smelik, T. Tutenel, K.J. de Kraker, and R. Bidarra, "Integrating Procedural Generation and Manual Editing of Virtual Worlds," *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (co-located with FDG 2010)*, Monterey, CA: 2010.

[29] J. Togelius, M. Preuss, and G.N. Yannakakis, "Towards multiobjective procedural map generation," *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (co-located with FDG 2010)*, Monterey, CA: 2010.

[30] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural Modeling of Buildings," *ACM Transactions on Graphics*, vol.25, iss.3, 2006, pp. 614-623.

[31] Epic Games, "Unreal Development Kit," 2010.

[32] J. Golding, *Building Blocks: Artist-Driven Procedural Buildings*, Game Developers Conference, 2010.

[33] Sonic Team, *Sonic the Hedgehog 2*, SEGA, 1992.

[34] A. Saltsman, *Canabalt*, http://www.adamatomic.com/canabalt/, 2009.

[35] [adult swim] games, *Robot Unicorn Attack*, 2010.

[36] A. Saltsman, "Tuning Canabalt," *Gamasutra [Online]*, Sep. 2010.

[37] D. Yu, *Spelunky*, http://www.spelunkyworld.com/, 2009.

[38] M. Persson, *Infinite Mario Bros!*, http://www.mojang.com/notch/mario/, 2008.

[39] G. Smith and J. Whitehead, "Analyzing the Expressive Range of a Level Generator," *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (co-located with FDG 2010)*, Monterey, CA: 2010.

[40] K. Hullett and J. Whitehead, "Design Patterns in FPS Levels," *Proceedings of the 2010 International Conference on the Foundations of Digital Games (FDG 2010)*, Monterey, CA: 2010.

[41] M. Mateas, "A Behavior Language," *Interactive Drama, Art and Artificial Intelligence [PhD Dissertation]*, Carnegie Mellon University, School of Computer Science: 2002, pp. 66-100.

[42] B. Weber, P. Mawhorter, M. Mateas, and A. Jhala, "Reactive Planning Idioms for Multi-Scale Game AI," *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG10)*, Copenhagen, Denmark: 2010.

Gillian Smith is a PhD candidate at the University of California, Santa Cruz. She received a B.S. in Computer Science in 2006 from the University of Virginia (Charlottesville, Virginia, USA), and in 2009 an M.S. in Computer Science from University of California, Santa Cruz (Santa Cruz, California, USA).

She teaches game design and programming in the UC Santa Cruz COSMOS summer program for high school students. Her research interests include procedural content generation and mixed-initiative design tools.

Ms. Smith is a student member of the IEEE, IEEE Women In Engineering, ACM, AAAI, and IGDA.

Jim Whitehead is an associate professor in the Computer Science department at UC Santa Cruz. He received his Ph. D. in Information and Computer Science in 2000 from UC Irvine (Irvine, California, USA).

He was an active participant in the creation of the Computer Science: Computer Game Design major at UC Santa Cruz in 2006. His research interests include software evolution, software bug prediction, procedural content generation, and augmented design.

Prof. Whitehead is a Senior Member of the IEEE, and member of ACM, and IGDA. He is the founder and chair of the Society for the Advancement of the Science of Digital Games (SASDG).

Michael Mateas is an associate professor in the Computer Science department at UC Santa Cruz, where he holds the MacArthur Endowed Chair. He received his Ph. D. in Computer Science from Carnegie Mellon University (Pittsburg, PA, USA) in 2002. His research in AI-based art and entertainment combines science, engineering, and design to push the frontiers of interactive entertainment.

He founded and co-directs the Expressive Intelligence Studio at UC Santa Cruz, which has ongoing projects in autonomous characters, interactive storytelling, game design support systems, procedural content generation, automated game design, and learning AI from datamining gameplay traces.

With Andrew Stern, Prof. Mateas released Façade, the world's first AI-based interactive drama. Façade has received significant attention, including top honors at the Slamdance independent game festival.

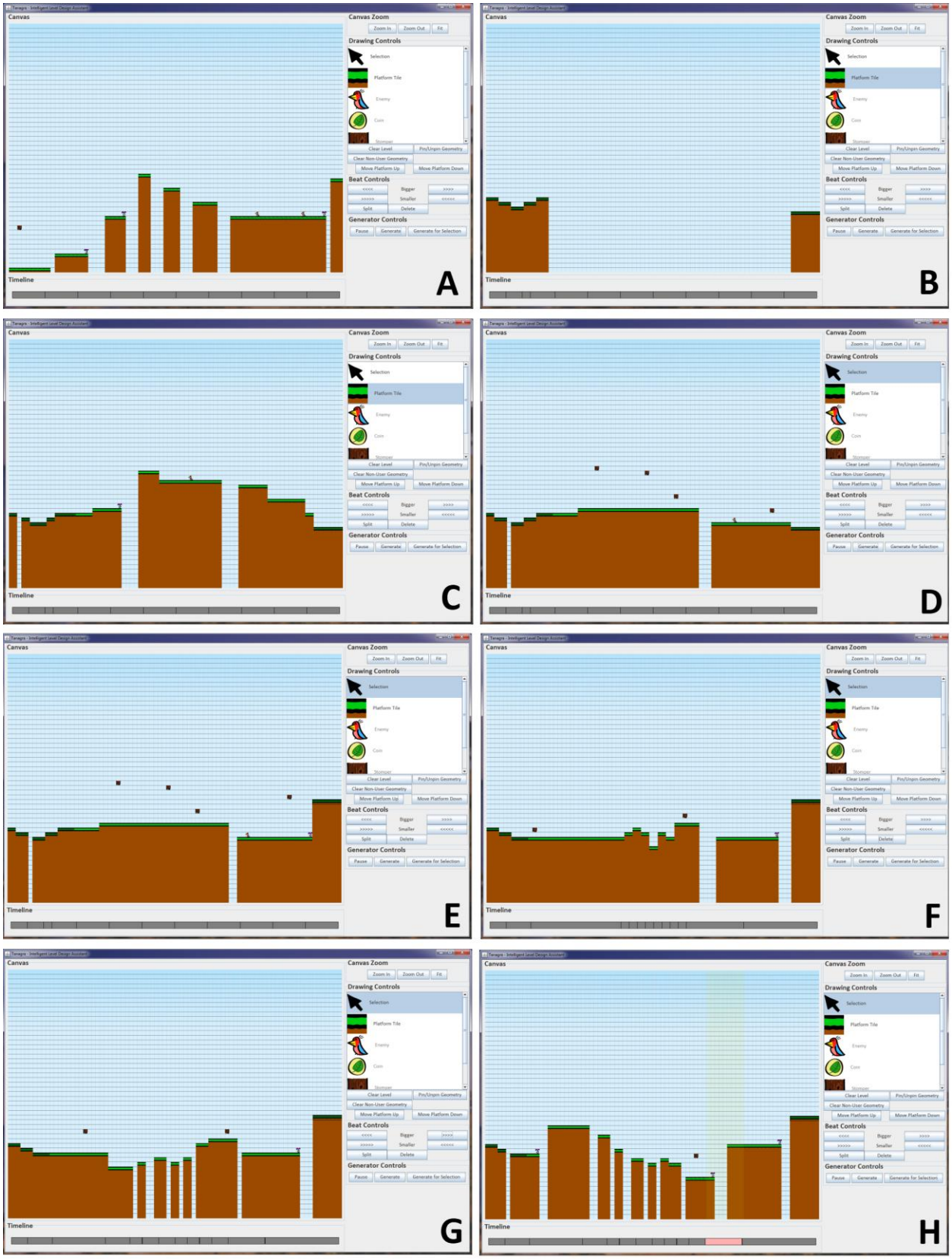

Figure 13. These screenshots show Tanagra in different stages of a level being edited. (A) A purely procedurally generated level with the default beat timeline. (B) The level designer has cleared the level and drawn in platforms; notice that the beat timeline has been automatically adjusted to reflect the drawn platforms. (C, D) Two different levels are generated to fill between the drawn geometry. (E) The last platform has been moved up, with the rest of the level held as constant as possible. (F) New beats have been added to the middle of the level; beats have been deleted from the beginning and end to slow the pacing. (G) Beat lengths are changed to make the middle of the level slightly less tightly packed. (H) Geometry has been re-generated specifically for the third beat and the second to last beat.