

Tanagra: A Mixed-Initiative Level Design Tool

Gillian Smith, Jim Whitehead, Michael Mateas
Expressive Intelligence Studio
University of California, Santa Cruz
Santa Cruz, CA, USA
{gsmith, ejw, michaelm}@soe.ucsc.edu

ABSTRACT

Tanagra is a prototype mixed-initiative design tool for 2D platformer level design, in which a human and computer can work together to produce a level. The human designer can place constraints on a continuously running level generator, in the form of exact geometry placement and manipulation of the level's pacing. The computer then fills in the rest of the level with geometry that guarantees playability, or informs the designer that there is no level that meets their requirements. This paper presents the design of Tanagra, a discussion of the editing operations it provides to the designer, and an evaluation of the expressivity of its generator.

Categories and Subject Descriptors

I.2.1 [Artificial Intelligence]: Applications and Expert Systems – Games. F.4.1. [Mathematical Logic and Formal Languages] Mathematical Logic – Logic and constraint programming. I.2.8. [Artificial Intelligence] Problem Solving, Control Methods, and Search – Plan execution, formation, and generation.

General Terms

Design, Human Factors

Keywords

Games, level design, procedural content generation, AI-assisted design.

1. INTRODUCTION

Level design is a vital part of the game design process: levels are a “container for gameplay” [3], providing the player with a space to explore the game's mechanics. Creating a good level is a time consuming and 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 a number of times before showing it to anyone else, checking that it is playable, engaging, and meets their expectations [4]. Making a change to a small section of a level, such as moving a single platform, can have a wide impact and require much of the rest of the level to be modified as well. Techniques from procedural level generation offer many opportunities to ease this authoring burden, by having the computer design and modify that which a human would normally painstakingly create. Providing authoring support to level design-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FDG 2010, June 19-21, Monterey, CA, USA
Copyright 2010 ACM 978-1-60558-937-4/10/06... \$10.00

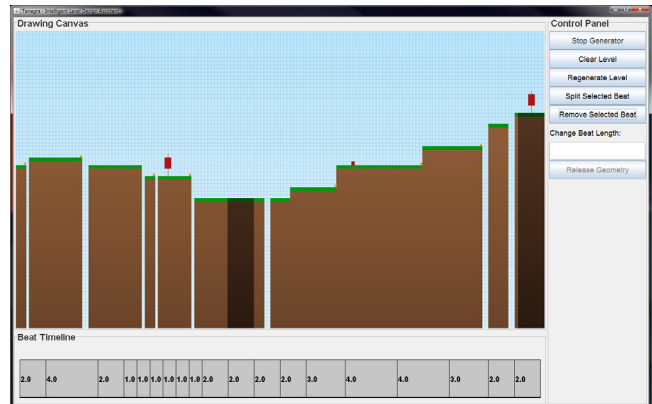


Figure 1. The Tanagra intelligent level design tool. The large area in the upper left is where the level is created. Below that is a beat timeline, allowing manipulation of the pacing of the level. On the right are buttons for editing the level.

ers is especially important now that players, who tend to be inexperienced designers, are increasingly being invited to create their own content for games.

A mixed-initiative approach to level design, where content is created through iterative cycles between the human designer and procedural support, would make it possible for designers to take advantage of the benefits that procedural level generation can offer: rapidly showing alternative designs, regenerating specified portions of a level, and enforcing the playability of any produced levels. Existing techniques in procedural level generation tend to offer little in terms of author guidance other than tweaking parameters in probability distributions, which can lead to unexpected outcomes [20] and do not allow the author to exert enough control over the final design. Consider, for example, our previous work in designer-guided level generation for 2D platformers [26]. We found it was extremely challenging to set fine-grained constraints such as prioritizing a segment of the control line or specifying an order for components to be introduced. It was also impossible to edit the level once it had been created, which drastically reduced the designer's creative control.

This paper presents Tanagra, a prototype mixed-initiative level design tool for 2D platformers (e.g. Super Mario World or Sonic the Hedgehog). The underlying representation for levels in Tanagra is beat-based, which provides a structure for creating levels with a well-defined rhythm [25]. Designers can interact with the tool in two different ways: both by editing the gameplay experience of the level through altering the pacing of the beats, and by manipulating the geometry from which this gameplay emerges. Each beat specifies an interval of play time, during which the player must take some action, such as running or jumping. For example, the designer can draw a few platforms into the editor, and ask Tanagra to generate geometry that will make the level playable but obey

these geometry constraints. The designer could then decide that the beginning of the level should be more fast-paced than the end, and address this by adding new beats to the beginning of the level for which either he or Tanagra can specify geometry. Tanagra is capable of producing a wide range of fully playable levels. A screenshot of a level created in Tanagra is shown in Figure 1. A video of Tanagra in action is also available online¹.

Our work is guided by design principles for intelligent creativity and design support tools [12][14][16][21]. There has recently been a call for such intelligent design tools specifically in the domain of games [11][23]. We have been careful to ensure that Tanagra does not push its own agenda on the designer by ensuring that decisions made by the human cannot be overridden by the system, although it can augment human-placed geometry through the placement of additional level components. Tanagra provides expertise through its ability to ensure that levels are always playable and by allowing the designer to directly manipulate the pacing of the level by editing the underlying beat structure. It also 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.

At the core of Tanagra is a novel procedural level generator that is capable of accepting initial input from a designer and then continuously updating the level when further refinements are made. Our approach integrates a reactive planning language, ABL [15], with a constraint programming and solving library, Choco [5]. ABL chooses the level components that should be placed, while the constraint solver determines their exact location.

This work addresses the following questions:

1. What technical infrastructure can be used to support a mixed-initiative level design tool?
2. What kinds of novel editing operations can a mixed-initiative level design tool provide?
3. How can we measure the expressive range of such a generator, and what is Tanagra’s?

The remainder of this paper discusses related work in both procedural level generation and AI-supported design (Section 2), the implementation details of Tanagra (Sections 3 and 4), and evaluation of our system in terms of use scenarios and analysis of the expressive range of the generator (Sections 5 and 6).

2. RELATED WORK

2.1 Procedural Level Generation

Typically, procedural level generation is used to improve re-playability in games by providing a new scenario each time the player starts a new game. In such games, the levels are completely generated by the computer, with no creative interplay between human designer and computer generator [1][19][22][31]. Much of the design of levels is implicitly encoded in the generation algorithm itself, with designer input limited to tweaking parameters to ensure that all the levels that the generator can create are appropriate. This parameter tweaking can be unintuitive, with small shifts leading to radical changes in the produced content [20]. In contrast, Tanagra focuses on creating a single level at a time, while providing fine-

grained support for the designer in terms of both geometry and beat editing tools.

An increasingly common use for content generation is personalized content creation, either adapting to a player at runtime [2][8][10][17][18] or learning player preferences offline [29]. The goal of Tanagra is to assist a human designer with the creation of a single level at design-time; interesting future work would be to incorporate models of different styles of play to encourage designers to include more diversity in their levels.

Work in author-guided level generation tends to place all of an author’s control at the beginning of the process [9][26][30], and occasionally allows editing after the level is complete [7]. 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 land masses, distribution of water and land, and climate. After the generator creates the initial world, the designer can then modify the terrain according to his 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.

2.1.1 Platformer Generation Techniques

Platformers are well-suited to research in procedural level generation due to their simple rules but surprisingly complex level design [25] and their lasting popularity. Infinite Mario Bros [19] and Spelunky [31] are two examples of released platformers where all levels are procedurally generated. Both games’ generation techniques work by fitting together relatively large hand-authored sections of a level. These generation techniques work well for their specific game, but could not 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 is especially dependent on its game mechanics, since the player can use certain tools to modify the level and traverse otherwise impossible terrain, and the player’s desire to conserve such tools means that many areas of the level will deliberately go unexplored. The building blocks for our levels are much smaller—a beat encompasses a single player action—and are extensible to different sets of geometry. 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 [6].

2.2 AI-Assisted Design Tools

As a mixed-initiative design tool, Tanagra combines the power of procedural content generation with traditional design techniques to improve the human designer’s experience. Lipp et al.’s [13] tool for procedurally-supported building modeling has similar goals to Tanagra, although in the space of 3D modeling for non-interactive structures rather than interactive spaces such as platformer levels.

Other AI-supported design tools include BIPED [24], which allows designers to rapidly specify prototypes of their games and view play traces from both human and computer players. The system’s strength is in allowing a designer to view situations in which their game design “fails” by asking the system to show a playthrough that produces a certain undesirable result. Tanagra does not support any qualitative evaluation of the level, but does make sure that levels are always playable.

¹ <http://users.soe.ucsc.edu/~gsmith/tanagra/fdgdemo.html>

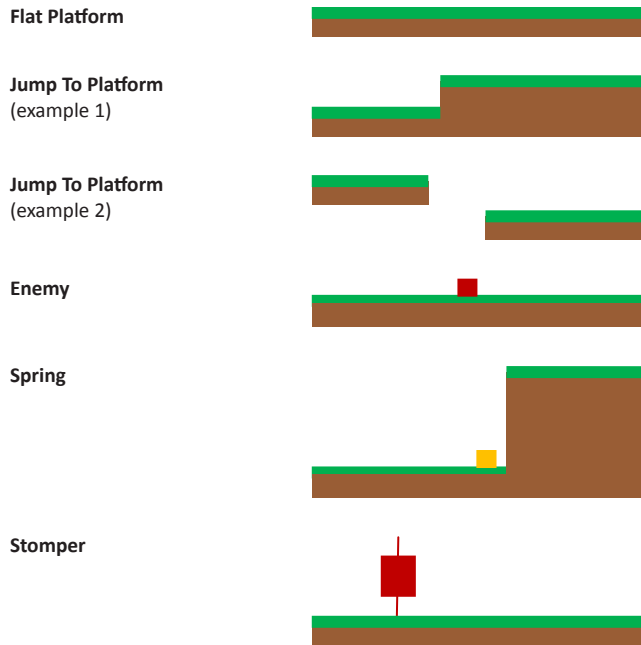


Figure 2. Example instantiations of the different geometry patterns used by Tanagra.

QuestBrowser [28] is a brainstorming tool for quest designers, intended to show different potential solutions that a player might think of, to better inform the design. Brainstorming is an important aspect to creativity support [27], which Tanagra fulfills by providing the ability to rapidly regenerate levels.

3. LEVEL REPRESENTATION

Tanagra represents levels as a set of beats, each of which contains level components, collectively referred to as “geometry”. The supported level components are platforms, gaps, springs, enemies, and stompers. A physics model that defines the maximum running speed and maximum jump height for the avatar guarantees that all geometry placed into the level is playable and meets beat duration constraints. Tanagra focuses on structural properties of the level, displaying platforms and other level components as differently colored rectangles.

3.1 Beats

As the building blocks of rhythms, beats are the underlying structure for Tanagra’s level generator. A beat’s primary role is to constrain the length of the geometry within it to the distance that can be traversed by the player in the duration of that beat, as calculated from the physics model. Beats are also a convenient way of subdividing the space for the generation algorithm, as geometry for each beat can be generated separately.

Beats encapsulate a single player action, such as a jump, which can occur at any time between the start and end times of the beat. Each beat has the following properties: a start time, an end time, a collection of level components associated with the beat, and knowledge of its preceding and following beats. At any time, the designer can add, remove, or modify a beat, which propagates any changes down to the geometry contained within it.

3.2 Geometry

Level components are built up into patterns based on the action the player should perform during the associated beat. These patterns are:

- Running along a long platform
- Jumping to a second platform (with or without a gap)
- Jumping to kill an enemy
- Jumping onto a spring
- Waiting before running underneath a stomper

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 are always the same size, but can have different positions along the platform.

Geometry patterns also have a defined player entrance and exit point, which can be tied together by the geometry generator such that the exit point of geometry in beat n is equivalent to the entrance point of geometry in beat $n+1$.

4. DESIGN ENVIRONMENT

Tanagra can give suggestions for level designs to the human designer, ensure that all levels it creates are playable, and supports editing the level at both the geometry layer and the beat layer to modify pacing. When Tanagra starts, the designer can specify the length of the beat timeline and number of beats in it, and they are presented with an empty geometry canvas. A control panel on the right side of the design environment allows the designer to insert and remove beats, change the length of a beat, select geometry to be locked in place or moved in the canvas, and request a newly generated level that respects any changes the human has made.

4.1 The Geometry Canvas

The geometry canvas is the place where both the human and computer draw geometry. The canvas, which is initially empty, is made up of a grid of tiles that are 5 units wide and 5 units tall. This grid serves two purposes: to reduce the search space for the constraint solver, and for ease of adding tile-based art assets at a later time.

The designer can choose to draw platforms associated with specific beats to guide the generator. Each beat has an area allotted to it that spans the maximum distance the player can cover in the length of the beat. Drawn platforms must fit into the area that is allotted to the beat, based on its start position and length. Designers can assign geometry to as many beats as they wish before starting the generator. Once the computer has created geometry, the designer can select that geometry to be “glued” in place or moved around. All designer-specified geometry is respected by the generator and cannot be moved, unless the designer chooses to “release” that geometry back to the generator. The following geometry canvas editing operations are available to the designer:

- *Draw platform*: draw a platform associated with a beat.
- *Remove platform*: delete geometry, either user-created or generator-created, and request new geometry in its place.
- *Glue geometry*: select geometry that the computer has created

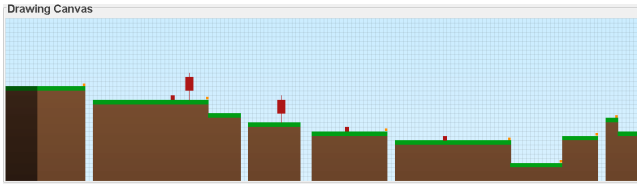


Figure 3. Tanagra's drawing canvas, containing a level created by cooperation between the human designer and procedural generator.

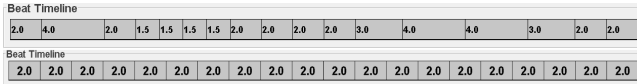


Figure 4. Top: the timeline used in creating the level shown above. Bottom: the original timeline that was presented to the designer.

and have it be treated as user-created from now on.

- *Replace geometry*: draw new geometry for a beat, replacing what was previously used for that beat.

An example of a simple level with mixed human- and computer-created geometry is shown in Figure 3. Note that human-created geometry is shaded darker than computer-created geometry.

4.2 The Beat Timeline

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 the generator to make geometry changes, allowing pacing changes without the need to manipulate geometry. The following beat timeline editing operations are available to the designer:

- *Remove beat*: remove the beat and its associated geometry by merging its two neighboring beats.
- *Split beat*: divide the selected beat in half, moving all of the existing geometry to one of these halves and generating new geometry for the other half.
- *Resize beat*: change the length of the beat, adjusting the next and previous beats to compensate for the length change.

Figure 4 shows the original timeline and a modified beat timeline.

5. GENERATION TECHNIQUE

Tanagra's 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.

To meet these goals, Tanagra employs a reactive planning language, ABL [15], to respond to user input and choose the geometry that should be placed for each beat. A constraint programming library, Choco [5], is used to specify and solve constraints on the placement of different level components. A diagram of Tanagra's system architecture is shown in Figure 5. Pseudocode for how ABL handles geometry generation and response to designer input for a

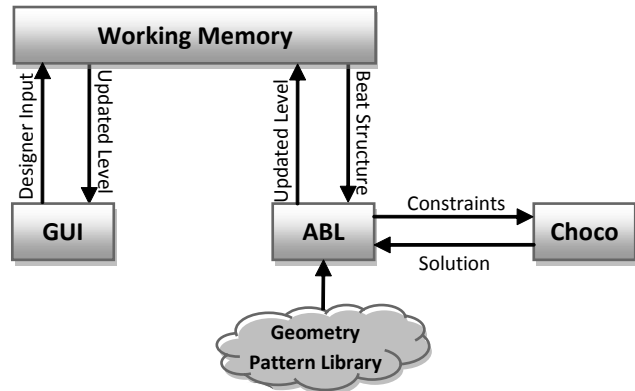


Figure 5. Tanagra is made up of three components: the GUI, an ABL agent, and the Choco constraint solver. The GUI and ABL communicate through working memory.

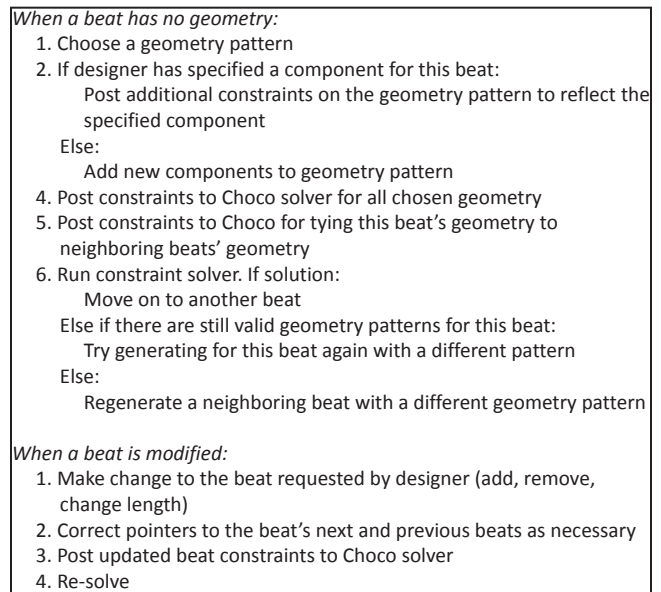


Figure 6. Algorithm for Tanagra's level generator as applied to a single beat.

single beat is shown in Figure 6.

Tanagra works quickly enough to permit rapid reaction to user input. It will continuously generate geometry until a solution is found, and regenerate levels either in response to a change made by the designer, or on demand. 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 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. There are also many different configurations of component placement that 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. To enumerate and search through each allowed instantiation of a geometry pattern would be inefficient and tedious; instead, the search can be split into two stages: ABL searches at the pattern level, while Choco searches for a valid

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.

5.1 Constraint Solving

Choco is a Java library for constraint satisfaction problems. The position and dimension for every level component are expressed as Choco integer variables, where the domain for each variable is the size of the canvas they are placed in. For example, a platform has 6 constraint variables: `startX`, `endX`, `startY`, `endY`, `width`, and `height`, and begins with the constraints:

```
startX + width = endX; startY + height = endY
```

While `width` and `height` are obviously not necessary for finding a solution, we find it useful to include them both for ease in specifying other constraints, and for pruning Choco's search space: frequently, the width of a platform is more constrained than its beginning or end positions, and so a good place to start Choco's search is with one of these variables that has very few legal values.

Choco is responsible for ensuring that the level is playable through solving constraints for:

- Matching the end of one beat's geometry with the beginning of its following beat's geometry
- The length of the sum of all geometry components in a beat being equal to the distance that should be covered in that amount of time
- Individual geometry elements having an appropriate position and dimension; for example, a gap should separate the ending and beginning points of two platforms by its width and height, and platforms should never overlap

Consider the example of a gap separating two platforms. A gap has two constraint variables: `width` and `height`, and knows which two platforms it is separating. Both `width` and `height` are constrained by the distance the avatar can jump in the length of time the gap encompasses, as determined from the physics model. For two platforms, `p1` and `p2`, and a gap, `g`:

```
p1.endX + g.width = p2.startX;  
p1.endY + g.height = p2.startY;
```

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 itself from searching after 500 milliseconds, since it is unlikely to find a solution after that point. The solution space is pruned by requiring all position variables to be a multiple of 5, to match the grid representation, anchoring the first geometry component placed in the grid to have an x value of 0, and only placing geometry for a beat when a neighboring beat already has geometry in it.

To avoid always getting the same values for position and dimension of components in an instantiation of a geometry pattern, Choco is instructed to use a random value in the domain of the variable each time it has to make a decision on variable assignment. Once the pattern has been instantiated, however, we want to keep the same values for as long as possible, so that any changes made by the

designer are as local as possible. For example, extending the length of a beat should not result in an entirely different level, but just a minor change to the affected beats. To accomplish this, Tanagra sets additional constraints on the solver, that the value of a variable should be the same as the last time the solver was run. These constraints are then relaxed if no solution can be found.

5.2 Reactive Planning

A Behavior Language (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 [15]. ABL defines agents as a set of behaviors (goals) that can be performed, where each of these behaviors can have a number of subgoals that can operate either in sequence or in parallel. Behavior selection is determined by fulfillment of preconditions and the behavior's priority.

The world state is communicated to the agent through Working Memory Elements (WMEs). Behaviors can request information from, update, or poll for changes in any WMEs that are registered to working memory. These WMEs are also accessible to the application; for example, each beat in Tanagra is stored as a `BeatWME`, that can be accessed by both ABL and the GUI.

ABL behaviors specify how the system should react to user input, and rules for what geometry patterns can be chosen for beats. Its responsibilities are:

- *Beat management*: making sure that all beats have correctly assigned preceding and following beats, and that changes to beats result in changes to geometry.
- *Creating geometry*: if a beat has no geometry pattern associated with it, ABL will choose a pattern for the beat.
- *Responding to user input*: any change made to the level is detected by the ABL agent and the appropriate response is given.
- *Posting constraints*: aside from constraints internal to level components, which are typically known when that component is created, ABL is responsible for posting all constraints to the Choco constraint solver.

How ABL handles these responsibilities is described in the following sections.

5.3 Beat Management

ABL is responsible for maintaining the accuracy of the beat timeline, by making sure that each beat is pointing to the correct preceding and following beats, and that the length of the beat is correct. This is accomplished by storing each beat as a `BeatWME`, thus making it accessible to ABL. There are then managers for each action that can be performed on a beat, and for ensuring that the beat knows about its neighbors. These managers constantly poll working memory, looking for `BeatWMEs` that have been modified or do not have pointers to their next or previous geometry assigned correctly.

5.4 Geometry Creation

There are three different kinds of geometry management that ABL performs: assigning geometry to a beat with no geometry associated with it, assigning geometry to a beat with user-created geometry associated with it, and modifying geometry when its associated

beat has been modified.

If ABL encounters a BeatWME with no geometry associated with it, it can choose to place any of the geometry patterns described earlier. ABL creates all of the level components required for the geometry pattern; for example, if “jump to another platform” was chosen, it would create two Platforms and a Gap. Each of these level components is itself a WME. ABL adds these WMEs to working memory and posts the appropriate constraints to the Choco physics model. Finally, after all of the level components have been placed, ABL calls on Choco to solve the constraint problem. If a solution is found, the process is complete. If not, a different geometry pattern is tried. If all geometry patterns fail, the generator backtracks, as described below.

When the designer adds a platform to the geometry canvas, he adds it to a specific beat. If ABL detects it is assigning a pattern to a beat that already has a designer-specified platform, it will incorporate this platform into each pattern that it attempts. If the designer attempts to place geometry “illegally” (for example, by drawing a platform on top of another one), Tanagra determines that no solution is available and waits for the designer to edit the level such that a solution can be found.

When a beat is removed or has its length changed, all geometry associated with the affected beat is removed, and ABL will reassign new geometry to the new beats. If the beat’s length was changed, ABL will first attempt the geometry that had been previously assigned to it.

5.5 Searching for a Solution

There are many cases in which the geometry pattern selected for a beat will be invalid. For example, consider a scenario in which the designer has drawn two long platforms that are separated by a beat. These platforms have different y values, so clearly the connecting geometry for the middle beat could not be another long platform, as the endpoints would not line up. However, a different geometry pattern, such as a jump to a different platform, may solve this scenario.

There are many scenarios in which no geometry pattern for a given beat can produce a playable level; in these cases, ABL must backtrack and try a different geometry pattern for other beats, in order to find a correct combination of geometry patterns.

There are three ways in which ABL will backtrack for a given beat, in order:

1. Relaxing absolute positioning constraints on other geometry in the level, which allows the rest of the level to “flow” around the newly chosen geometry.
2. Choose a different geometry pattern, marking the currently selected pattern as invalid.
3. Ignore all chosen geometry for the current beat, and choose new geometry for either the next or previous beat. Mark the pattern of currently chosen geometry as invalid regardless of the geometry used for the current beat.

6. USE SCENARIO

This section presents a detailed use scenario, showcasing Tanagra’s key abilities: auto-filling geometry, brainstorming level ideas, and

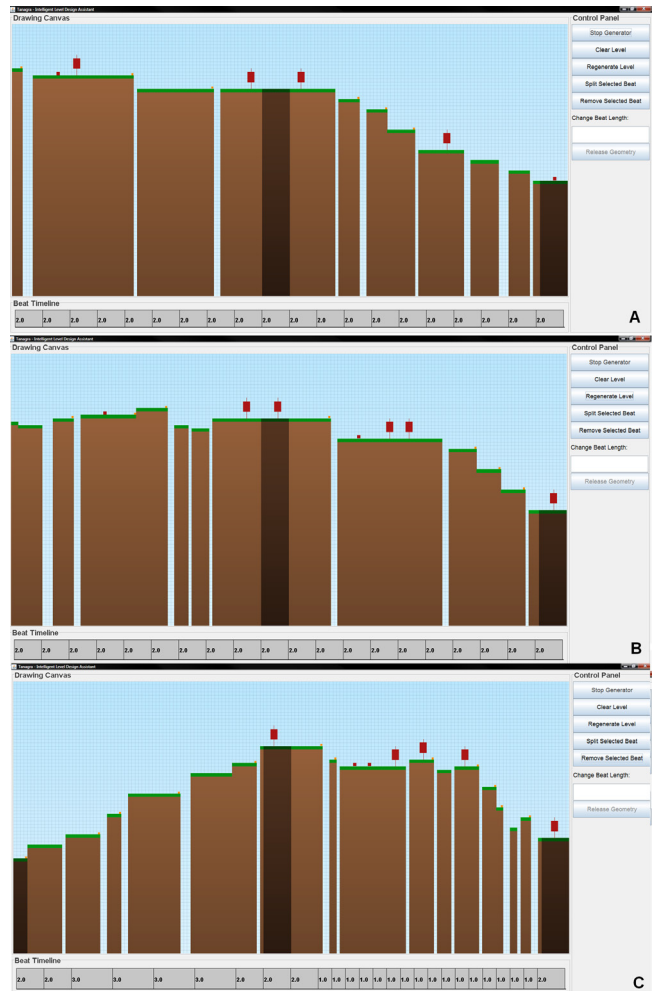


Figure 7. A use scenario for Tanagra. A and B show different levels generated for the same designer input. C shows a new platform drawn into the first beat, and pacing changes at both the beginning and end of the level.

manipulating level pacing.

The designer can provide initial guidance to Tanagra by placing initial platforms in the geometry canvas. In this example, we place two platforms in the level: one at the mid-point, and one at the end. The generator then fills in geometry for the rest of the level that matches the initial specifications (Figure 7A).

Tanagra can rapidly regenerate the level to show different variations that meet the same requirements. Figure 7B shows one such alternative level.

Geometry for a beat can be replaced with user-drawn geometry at any time, and Tanagra will regenerate surrounding geometry to accommodate the change. In Figure 7C, the designer has decided to move the beginning of the level lower down, by drawing a different platform in the first beat.

Finally, Tanagra supports editing the beat structure itself, in addition to manipulating raw geometry. In Figure 7C, the designer has also slowed down the pacing of the first half of the level, and significantly increased the pacing of the second half.

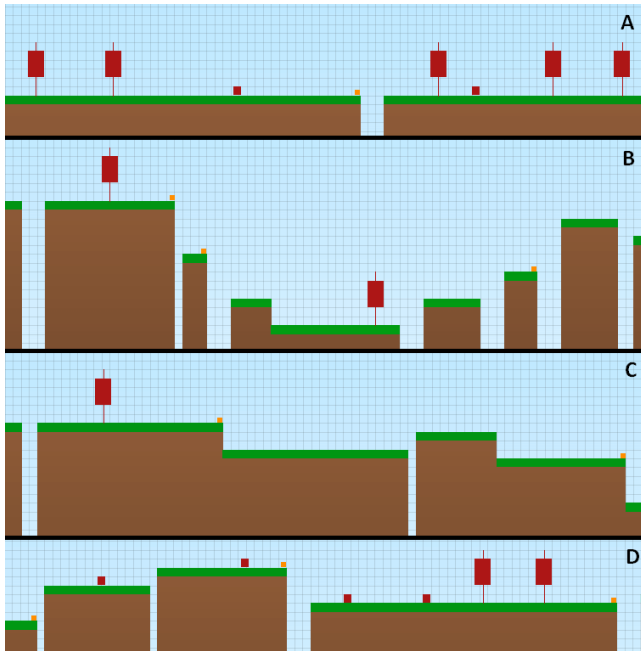


Figure 8. Examples of levels at the extrema of Tanagra’s expressive range. (A) A highly linear level, score=0.0 (B) A highly non-linear level, score=1.0 (C) A highly lenient level, score = -0.8 (D) A highly non-lenient level, score=1.0

7. GENERATOR 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. To this end, we have created two different metrics for comparing levels: the linearity of a level, and how lenient the level is towards the player. We then apply these metrics to levels that are entirely procedurally generated.

7.1 Linearity

The first method 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. Examples of levels that fall at the extremes of this scale are shown in Figure 8A and Figure 8B. 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. We then score each level by taking the sum of the absolute values of the distance from each platform midpoint to its expected value on the line, and divide by the total number of points. Results are normalized to a [0, 1] scale, with 0 being highly linear and 1 being highly non-linear. A graph showing the distribution of linearity scores across 500 generated levels is shown in Figure 9A.

Tanagra is slightly biased towards creating more linear levels; we hypothesize that this is because there are more instantiations of geometry patterns that have no height variation. Flat platforms, enemies, and stompers all appear on flat platforms, while gaps can have a positive, negative, or zero height.

7.2 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. Scores are as follows: flat platform: -2, jump to platform with no gap: -1, jump to platform with gap, enemies, or stompers: 1. The score is normalized for the number of beats in the level, with a maximum value of 1 and a minimum value of -2. However, very few levels score lower than -1. Figure 8C and Figure 8D shows examples of levels at the extremes of the leniency score. Notice that the lenient level has very few gaps or enemies, whereas the non-lenient level has many enemies and stompers. A graph showing the distribution of leniency scores is shown in Figure 9B.

Again, Tanagra seems slightly biased towards one side of this range, creating more unforgiving levels than lenient ones. This is because there are more geometry patterns that contain a level component classified as unforgiving: enemies, stompers, and gaps can all occur quite frequently.

7.3 Expressive Range

With two different axes on which to compare generated levels, Tanagra’s expressive range is the distribution of levels within the rect-

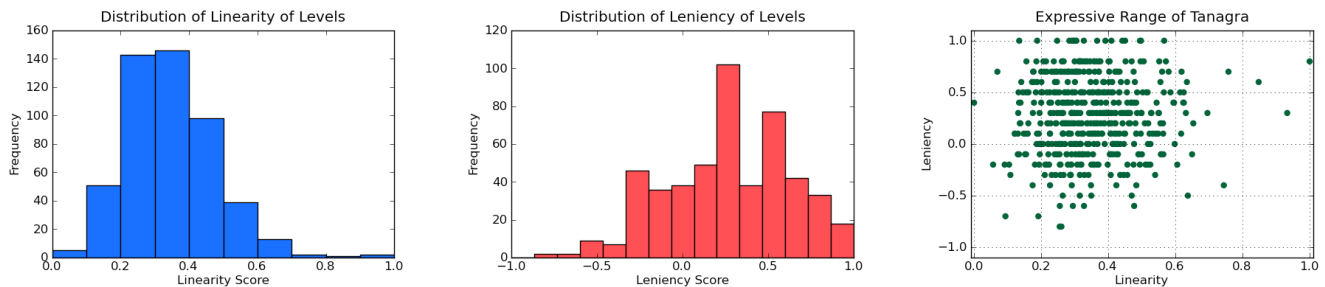


Figure 9. Graphs showing the expressive range of Tanagra. The first shows the distribution of levels with specific linearity scores, the second the distribution of levels with specific leniency scores, and the third shows the combined distribution of levels with linearity on the x-axis and leniency on the y-axis.

angle defined by these axes. Figure 9C shows a plot of each of the generated levels within this range, where each dot corresponds to a single generated level. Although heavily biased towards creating linear levels, Tanagra is clearly capable of generating levels with varying linearity and leniency, and there is no evident correlation between these two measures.

8. 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. This paper presented Tanagra's design, as well as examples of how it can be used and a method for measuring the expressive range of the levels it can produce.

Tanagra is still in a prototype stage, and is itself part of a larger intended system that will support more editing operations and different views on the level. We envision a final system that uses Tanagra as a subcomponent, with other components being a difficulty analyzer, and a way to easily view and edit entire level paths without needing to focus on fine-grained geometry details.

There is work to be done in easily expressing different geometry patterns; for example, Sonic the Hedgehog has long spirals and loop-de-loops that are not currently supported by Tanagra. There are also interesting issues to address in terms of an appropriate interface for a mixed-initiative level design tool. It can be difficult to determine the "correct" action to take in certain situations; for example, when splitting a beat in half, how should the existing geometry for that beat be divided up?

More broadly, there is a need for a standard way to measure the expressive range of a procedural content generation system, whether it is entirely autonomous or mixed-initiative. This paper presents a step in the right direction, by classifying Tanagra's range along two different measures of the level. We anticipate performing future work in defining new metrics for comparing levels.

REFERENCES

[1] Bay 12 Games. 2006. Dwarf Fortress (PC Game).
 [2] Booth, M. 2009. The AI Systems of Left 4 Dead. Keynote, Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '09). Stanford, CA. October 14 – 16, 2009.
 [3] Byrne, E. 2005. *Game Level Design*. Charles River Media.
 [4] Castillo, T. and Novak, J. 2008. *Game Development Essentials: Game Level Design*. Delmar Cengage Learning.
 [5] Choco Team. 2008. Choco: an Open Source Java Constraint Programming Library. White Paper, CPAI08 Competition. <http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf>
 [6] Compton, K. and Mateas, M. 2006. Procedural Level Design for Platform Games. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '06)*. Stanford, CA. 2006.
 [7] Firaxis Games. 2005. Sid Meier's Civilization IV (PC Game).
 [8] Hastings, E., Guha, R., and Stanley, K.O. 2009. Evolving Content in the Galactic Arms Race Video Game. In *Proc. of the IEEE Symposium on Computational Intelligence and Games (CIG '09)*. Milano, Italy. September 7 – 10, 2009.
 [9] Hullett, K. and Mateas, M. 2009. Scenario Generation for Emergency Rescue Training Games. In *Proceedings of the 4th International Conference on Foundations of Digital Games*. Orlando, FL. April 26 – 30, 2009.
 [10] Hunnicke, R. and Chapman, V. 2004. AI for Dynamic Difficulty Adjustment in Games. Papers from the 2004 AAAI Workshop on Challenges in

Game Artificial Intelligence. Technical Report WS-04-04, The AAAI Press, Menlo Park, CA.
 [11] Isla, D. 2009. Invited Talk: Next-Gen Content Creation for Next-Gen AI. Fourth International Conference on the Foundations of Digital Games. Orlando, FL. April 26-30, 2009.
 [12] Lawson, B. and Loke, S.M. 1997. Computers, Words, and Pictures. *Design Studies*, vol.18, no.2, pp171-183. 1997.
 [13] Lipp, M., Wonka, P., and Wimmer, M. 2008. Interactive Visual Editing of Grammars for Procedural Architecture. In *Proc. of the Intl. Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '08)*. Los Angeles, CA. August 11-15, 2008.
 [14] Lubart, T. 2005. How Can Computers be Partners in the Creative Process: Classification and Commentary on the Special Issue. *International Journal of Man-Machine Studies*, vol.63, no.4-5, pp365-369, 2005.
 [15] Mateas, M. and Stern, A. 2002. A Behavior Language for Story-Based Believable Agents. *IEEE Intelligent Systems*, pp39-47. July/August, 2002.
 [16] Negroponte, N. 2003. Soft Architecture Machines. In *The New Media Reader*. Noah Wardrip-Fruin and Nick Montfort, Eds. Chapter 23, pp353-366. The MIT Press, Cambridge, MA. 2003.
 [17] Nitsche, M., Ashmore, C., Hankinson, W., Fitzpatrick, R., Kelly, J., and Margenau, K. 2006. Designing Procedural Game Spaces: A Case Study. In *Proceedings of FuturePlay 2006*. London, Ontario. October 10 – 12, 2006.
 [18] Pedersen, C., Togelius, T., and Yannakakis, G. 2009. Modeling Player Experience in Super Mario Bros. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG '09)*. Milano, Italy. September 7 – 10, 2009.
 [19] Persson, Marcus. 2008. Infinite Mario Bros! (Online Game). Last Accessed: December 2008. <http://www.mojang.com/notch/mario>
 [20] Regier, J. and Gresko, R.. 2009. "Random Asset Generation in Diablo 3", Invited Talk, UC Santa Cruz. October 30, 2009.
 [21] Resnick, M. et al. 2005. Design Principles for Tools to Support Creative Thinking. Technical Report: *NSF Workshop Report on Creativity Support Tools*. Washington, DC. 2005.
 [22] Rogue Basin. Articles on Implementation Techniques [Online]. Last Accessed: July 2009. <http://roguebasin.roguelikedev.com/index.php?title=Articles#Implementation>
 [23] Satchell, C. 2009. Keynote: Evolution of the Medium – Positioning for the Future of Gaming. Fourth International Conference on the Foundations of Digital Games. Orlando, FL. April 26-30, 2009.
 [24] Smith, A.M., Nelson, M.J., and Mateas, M. 2009. Prototyping Games with BIPED. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '09)*. Stanford, CA. October 14-16, 2009.
 [25] Smith, G., Cha, M., and Whitehead, J. 2008. A Framework for Analysis of 2D Platformer Levels. In *Proceedings of the ACM SIGGRAPH Sandbox Symposium 2008*. Los Angeles, CA. August 9 – 10, 2008.
 [26] Smith, G., Treanor, M., Whitehead, J., and Mateas, M. 2009. Rhythm-based Level Generation for 2D Platformers. In *Proc. of the 4th Int'l Conference on Foundations of Digital Games*. Orlando, FL. April 26 – 30, 2009.
 [27] Sternberg, Robert J. 1999. Enhancing Creativity, *Handbook of Creativity*. pp 401 – 402. Cambridge University Press.
 [28] Sullivan, A., Mateas, M., and Wardrip-Fruin, N. 2009. QuestBrowser: Making Quests Playable with Computer-Assisted Design. In *Proceedings of Digital Arts and Culture 2009 (DAC '09)*. Irvine, CA. December 12-15, 2009.
 [29] Togelius, J., De Nardi, R., and Lucas, S. 2007. Towards Automatic Personalised Content Creation for Racing Games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG '09)*. Honolulu, HI. 2007.
 [30] Tutenel, T., Smelik, R., Bidarra, R., and Jan De Kraker, K. 2009. Using Semantics to Improve the Design of Game Worlds. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '09)*. Stanford, CA. October 14 – 16, 2009.
 [31] Yu, D. 2009. Spelunky (PC Game).