User-preference-based automated level generation for platform games

Nick Nygren, Jörg Denzinger, Ben Stephenson, John Aycock

***Abstract*—Level content generation in the genre of platform games, so far, has focused primarily on the typical, tactile-skill based levels with simple left to right movement. In this paper, we present a level generation method that expands these standard kinds of levels by integrating puzzle based content that requires the player to explore the level in order to solve it. With these choices comes the need to elicit what kind of level a user would like to play at the current time, which is also covered by our level generation system.**

I. Introduction

The environment in which both the character(s) controlled by a human player and the non-player characters act in is an important factor for the game playing experience and thus for the success of a game. Many games offer different environments and often even require a player to master one environment in order to get to another. Traditionally, these environments are then called *levels* and for a long time creating levels was considered a task so difficult that human developers had to do it. Even human developers will not create levels that every human player will like: not only are humans different in their personal preferences, but the current mood and situation of a human player also influences what type of level the player wants to play at a particular point in time.

Recently, automated level generation for games has become a research focus, as shown by the Mario AI competition (see [1]). As in this competition, (2D) platform games serve as a convenient experimental evaluation platform in most of the research on this topic. Previous works in the area have concentrated on identifying features of a level that in general are considered important for a level of a game, like Flow and Rhythm (see [2] and [3]). In [4] and [5], presenting choices for exploration has been identified as a possible feature and genre. And [6] focused on adjusting hole width and placement based on player feedback.

In this work, following successes in learning and using user preferences in other areas (as for example recommendation systems, see [7]), we present an automated level generation method for the game of the Mario AI competition. Our method learns the preferences for various features that the user/player currently has, based on having the user play a few short given levels and answering a few questions. We extended the testbed from [1] to allow for more feature variations and create levels in a three step process. First we create a graph representing the future cells for the game that fulfills constraints created out of the observed feature preferences (and the answers to the questions) of the user. This graph is then transformed into a two dimensional level for the game keeping the cell connections from the graph. Finally, the individual cell contents are created, again using constraints and preferences observed from the user (and the user's answers).

In our evaluation, we collected the user preferences from four players with different skills and genre preferences. The levels created for each of the players show substantial differences, aligning them with the collected player expectations.

The authors are with the Department of Computer Science, University of Calgary, 2500 University Drive NW, Calgary, Canada T2N 1N4; (email: {ndnygren,denzinge,bdstephe,aycock}@ucalgary.ca).

II. Our approach to level generation

The Mario AI platform game has the human player navigate through a world of platforms, holes between platforms, portals connecting different parts of a level (essentially being teleporters), and enemies, like "Goombas" and "Spiked Turtles". The player character has to jump from platform to platform or on enemies (to kill them) to pass through a level. The player character can make use of "special effects", like shells and fireballs, and can also collect coins.

In this section, we first present the general steps our system takes to create a level for a particular player. Then we look in more detail at each of these steps.

A. A high-level view on level generation

In general, our approach consists of two phases, the *user preference elicitation* and the *level generation* based on preference parameters. The latter phase consists of three steps, namely creating a cell graph, conversion of the graph into a grid, and finally creating the individual cells making sure that they fit the grid. The second phase can be performed several times using the results from the first phase due to the use of some randomness in the three steps of the second phase. The result of the first phase can change with the mood of the player, so the player can repeat this phase as desired to re-tailor the generated games to their current mood.

We use a combination of multiple choice questions and player performance in short pre-defined levels to get an idea of what kind of level the user would like to play. Two multiple choice questions ask the user about their preferred level of difficulty, and their opinion on different kinds of challenges. The user then plays three pre-defined levels, both to measure their skill level, and to verify the answers provided to the questions. While the user plays the pre-defined levels we count a variety of player behaviors, which we then compact into metrics. Using these metrics allows

978-1-4577-0011-8/11/$26.00 ©2011 IEEE

us to determine four basic parameter values for the player, which are then used during the level generation phase.

For realizing the second phase, we adopt the practice of dividing a level into *cells* from [9]. Cells are subdivisions of the level in which the style of content is locally consistent with regard to some measure, typically rhythm in the literature. In our case, it is a region over which the parameters used for generation do not vary. Instead of using this idea strictly for analysis, we partition the empty space of the level into cells, which are then populated with content.

We start our level generation process by creating a directed graph, which describes the number of cells, and which cells may be accessed from other cells. The structure of this graph will control the large scale complexity of the level, including how often the player's path may diverge and force the player to make a choice, and how much of the content will remain unseen by the player if the most direct route from start to finish is chosen. Figure 1 presents an example of such a graph, with the box indicating the shortest path from start to finish.

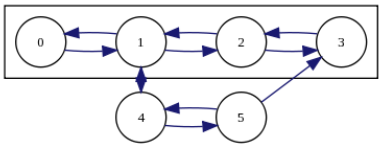


Fig. 1: A sample graph.

In the next step, the graph must be converted into a grid representation, which will decide the geographical size and location of each cell (see Figure 2 for an example). Without the use of a portal system, this conversion is difficult and sometimes impossible. To allow for movement between cells, any two cells which are connected in the graph must share an edge in the grid. For such a grid to exist, it is clear that the graph must be planar, but even some planar graphs do not have solutions. Using *portals* that allow the player to jump between connected cells and thus connect two cells that are geographically separated in the grid not only avoids this problem, it also allows for the construction of more complex levels providing more choices to the player.

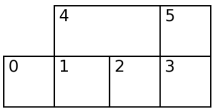


Fig. 2: A grid generated from Figure 1.

In the final step, content is generated for each cell separately, then assembled into a full level according to the requirements represented by the grid. Various requirement constraints are tested and modifications are made as needed to ensure that the level is functional, can be traversed by the player, and will appear to have continuity.

B. Eliciting user preferences

In order to create a level that represents the current preferences of a player, we start by asking the player two pre-game questions. The first lets the player indicate one of three *sub-genres* of the general game that we identified, namely Combat, Flow, and Puzzle. As the name suggests, Combat indicates that the player wants to fight enemies. Flow indicates a wish for terrain that requires tactile skills, and Puzzle requires the player to explore different paths through the level in order to solve it. The second question asks the player to indicate a difficulty level.

The player's self-evaluation is confirmed by a set of trials used to measure the player's abilities in a variety of common situations. These trials are static hand-made levels, which are shorter than a typical level but of much greater difficulty. The trials are intended to be unpassable by the majority of players. This way, the length of the path through the level before the player character is killed provides the most important information. In the following we indicate the distance traveled on the ground by D_G and the distance traveled in the air by D_A. Distances are measured in pixels. For each symbol defined in this section, by adding arguments between 1 and 4 to it we indicate the sum of the particular measure over the indicated trials, e.g. $D_G(1,3)$ is the sum of the ground distances traveled in trials 1 and 3. Another important piece of information extracted from each trial is how fast the player moved, with T_C indicating the time (in seconds) spent by the player, S_G the average speed on ground and S_A the average speed in air. We also use the number F of shots fired, as well as the percentage F_P of fireball hits per shot, the highest number J_B of kills with a single jump and the highest number K_B of kills with a single shell kick. Apart from these measures, additional types of data are gathered specialized to some of the trials.

- **Trial 1**: The player is provided with 2 item boxes, which results in a "fire flower" if both are used together, and many turtle shells. This essentially provides every possible "weapon" available in the game. We count the number K_F of enemies killed by fire in this trial and also the number K_{SH} of enemies killed per shell kick.
- **Trial 2**: The player is provided with only one item box, and no turtle shells. The player faces lines alternating "Goombas" and "Spiked Turtles", which means that half the enemies are effectively invulnerable, so that the player is forced to dodge them. Again, the number of enemies killed is counted. In addition, higher scores are awarded to the players who make fewer jumps with J indicating the total number of jumps and J_C the number of combat jumps.
- **Trial 3**: The player is given no item boxes and no turtle shells and faces a series of jumps of increasing difficulty. The final jumps must be made in a single attempt without losing momentum, otherwise the gap will be too wide to traverse.
- **Trial 4**: The player is given no item boxes and no turtle shells and must choose between a maze of portals which

is designed to disorient, and a combination of difficult jumps and invulnerable enemies which must be dodged. The number P of portal usages is counted. Successfully completing the maze of portals results in a large increase of the the player's puzzle rating.

This data is gathered by an event monitoring system which is similar to the one used in [6].

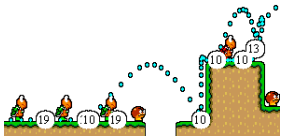


Fig. 3: Player data gathering. Here event 10 represents a jump, 19 an enemy killed by shell, and 13 an enemy killed by a jump.

Every time an image is rendered for the screen, an *At* event is recorded. This marks the player's position with a time stamp (shown in Figure 3 as turquoise dots). In addition to these, other events, such as an enemy being killed, or the player using a portal, are recorded with the position and current system time.

The event counts, together with the path lengths achieved for the trial levels and the movement speeds in these levels, are used to create values for four high-level parameters that are used in the second phase of the level generation. Each of these higher-level parameters is based on a weighted sum of some of the measures described above. The measures and weight (which were determined by doing some initial experiments) are for

- the *jump/flow aspect* $flow_B$: ($S_G(2,3)$, $S_A(2,3)$, $J(2)$, $J_C(2)$, $T_C(2,3)$, $D_G(3)$) with weights (1, 1, 0.1, 0.1, 0.1, 0.00005)
- the *combat aspect* $combat_B$: ($K_F(1)$, $K_{SH}(1)$, $J(2)$, $J_B(2)$, $F(1)$, $F_P(1)$) with weights (0.1, 0.1, 0.1, 0.1, -0.1, 0.01)
- the *puzzle aspect* $puzzle_B$: ($P(4)$) with weight 0.02 or $1 - 0.02P(4)$ if the maze is completed
- the *clutter aspect* $clutter_B$: ($T_C(1,2,3)$, $P(1,2)$, $J_B(1,2,3)$, $K_B(1,2,3)$, $D_A(1,2,3)$) with weights (0.01, 0.01, 0.1, 0.1, 0.00005).

These base values for the parameters are then modified by adding a Δ_d that is based on the player's indicated difficulty preference ($\Delta_d = 0.25$ for easy, 0.5 for medium, 1 for hard and 1.25 for impossible). For the final value of $flow$, we also add a δ_f based on the player's answer to the genre question ($\delta_f = 0.4$ for flow as genre, -0.4 for combat and -0.2 for puzzle) and then scale the combined value into the interval [0,1]. For the final value of $combat$, we add a δ_c ($\delta_c = -0.4$ for genre flow, 0.4 for combat and -0.2 for puzzle) and scale; for the final value of $puzzle$, we add a δ_p ($\delta_p = 0$ for flow and combat and 0.4 for puzzle) and scale. The final value of $clutter$ is just scaled.

C. Graph Generation

The first step in actually creating a level using our method is to create a graph representing a given number, namely $8 \cdot puzzle + 7$, of cells (as nodes) and the intended connections between those cells. For a graph to be acceptable (feasible) for the next steps, we require it to fulfill a set of given (feasibility) constraints. This still leaves quite a number of possible graphs and in order to create a "good" graph, we are using a feasible-infeasible two-population genetic algorithm (as used in [10], based on [11]) to search for a graph that is "good" with regard to a given fitness function.

In the following, we will first describe the genetic algorithm, then look at the fitness function and finally present the feasibility constraints.

1) Searching for a good graph: A feasible-infeasible two-population genetic algorithm is an augmented genetic algorithm, which is able to solve problems that require fulfillment of a set of hard constraints but also come with the need to optimize a given goal function. To preserve genetic diversity, two populations are maintained. The first, Feasible, contains those candidates which satisfy the constraints, the second, Infeasible, is for those that do not. Candidates are generated from both populations in the usual way, evaluated by the same fitness function. Each candidate of each new generation is evaluated at the time of its creation and placed in the appropriate population based on the constraints. The genetic diversity that would be lost if the infeasible populations were simply discarded is preserved and continues to be optimized.

An individual in our genetic algorithm is the Boolean array indicating the adjacency matrix of the graph represented as a single Boolean vector. As usual, the first generation is created by creating random individuals. New generations are generated from previous ones using the following genetic operators: To create a new individual using *crossover*, two candidate individuals are chosen from the previous generation. At a random point in the individuals, both candidates are divided and the new individual is constructed by appending the left portion of one candidate with the right portion of the other. After a new candidate individual is constructed by a crossover, at each index of the individual there is a very small chance that the value will be mutated. Since the individual is a Boolean array, this means setting the value to its negation.

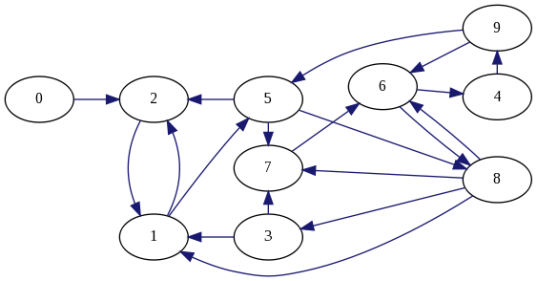


Fig. 4: A generated graph example.

2) The fitness function: The fitness function evaluates the individuals and is a weighted sum of the following four measures:

- **Average Outgoing Edges** - $E_O()$: Outgoing edges are simply counted and averaged. Figure 4 has an average of 1.9. This provides a good description of the complexity

of a graph: the more choices presented to the player at any given location, the more likely they will be to make the wrong choice. The absolute difference $E_O()$ between this average and a target value elicited from the player, namely $1 + 2 \cdot puzzle$, is used in the fitness measure and assigned a negative weight.

- **Start Node Connections** - $E_S()$: The number of outgoing connections from the start point is monitored separately. This is ideally low or exactly one, to match the Mario style. This is ignored if there is exactly one exit from the start node, otherwise it is assigned a negative weight.
- **End Node Connections** - $E_E()$: This is similar to the Start Node Connections, except it is used in the fitness measure to keep incoming connections to the end node as close to one as possible.
- **Path Length** - $L()$: The length of the shortest path from the first node to the last (or 0 if the path does not exist), illustrated in Figure 5, is assigned positive weight in the fitness function and has many desirable effects if used as a measure. Graphs with longer path lengths will be more likely to be planar, and then easier to convert to a grid. This also prevents the start and finish of the level from being placed side by side. Finally, longer path lengths also make it more likely for the graph to satisfy the feasibility constraints, creating a trend across generations of individuals moving from Infeasible to Feasible.

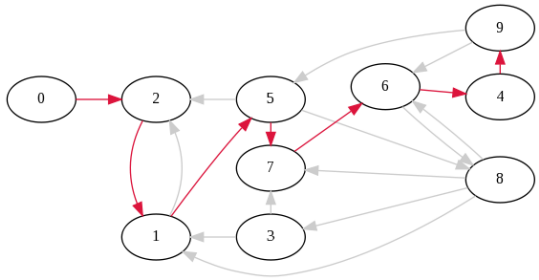


Fig. 5: The shortest path from node 0 to node 9 in Figure 4.

Then the total fitness of a level graph x is evaluated as:

$$F(x) = w_1 \cdot E_O(x) + w_2 \cdot E_S(x) + w_3 \cdot E_E(x) + w_4 \cdot L(x)$$

where $(w_1, w_2, w_3, w_4) = (-1, -1, -1, 1)$ are weights that are system parameters.

3) Feasibility constraints: As stated above, only feasible graphs can be used in the following steps of our level generation method. Feasibility is defined by fulfilling the following constraints:

- **Path Length** - $C_L()$: To make absolutely sure that the start and finish nodes are not adjacent we also use path length to formulate a feasibility constraint. $C_L()$ returns false, if the shortest path from start to finish traverses fewer nodes than a player defined limit, namely a fraction of graph size equaling $1.0 - 0.3 \cdot puzzle$, otherwise it returns true.
- **Forward Reachability** - $C_F()$: This constraint returns false, if a traversal of the graph, starting at node 0, as shown in Figure 6, results in any node not being visited. This ensures that it is possible to reach all content in the level.
- **Backward Reachability** - $C_B()$: This constraint returns false, if a traversal of the graph, starting at the final node, with the direction of every edge reversed, as shown in Figure 7, results in any node not being visited. This constraint ensures that there are no "traps", where the player would be forced to fail and restart the level. Although there may still be dead ends, the player has the option to backtrack. Combined with Forward Reachability, for every node there will be a path from start to finish that passes through it.

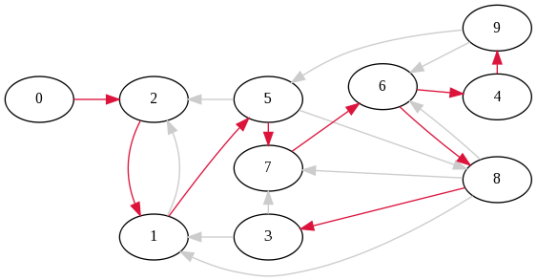


Fig. 6: The forward traversal of the graph in Fig. 4.

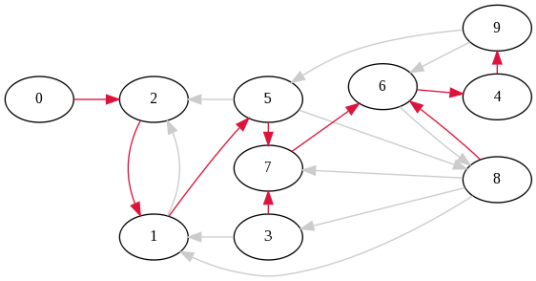


Fig. 7: The backward traversal of the graph in Fig. 4.

With these constraints, every graph x, for which the expression

$$C(x) = C_L(x) \ \wedge \ C_F(x) \ \wedge \ C_B(x)$$

is true is placed in the Feasible population, and all other graphs are placed in the Infeasible population.

D. Grid Generation

The best (feasible) individual (graph) found by the genetic algorithm from the last step is used as the input to the second step of our method, the grid generation step. Prior to beginning the placement of cells on the grid, the cells (nodes) in the graph are put in a *placement order*. This order is created by performing a basic breadth-first traversal of the graph. Then cells are added to the grid one at a time in this order. This allows the already-placed cells and remaining cells to be quickly distinguished. Since there are usually several different possibilities where a cell can be placed in the grid (see Figure 8), the grid generation is essentially an

or-tree based search. At any given depth in this search tree, the next cell to be added is the cell named at the same depth in the placement order.

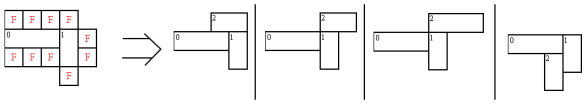


Fig. 8: Some candidate positions generated for cell 2.

The placement search begins with cell 0 at the leftmost edge of the grid, centered vertically. From there, the *fringe* is identified (see Figure 9). The fringe are the locations on the grid which are empty, but directly adjacent to some space that is not empty. From each fringe location found, and each of the directions (up, down, left, right), many candidate placements are generated (Figure 8). These begin at a length of 1 and increase in length by one unit until one of the candidates

- overlaps another cell,
- reaches the edge of the grid, or
- reaches the maximum cell size.

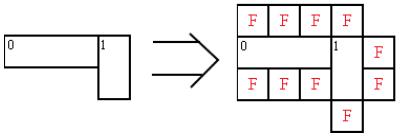


Fig. 9: The identification of fringe locations.

From here, these steps are repeated until every node is added to the grid, and a successful leaf node of the search tree is created. Unfortunately, a search tree constructed in this way may have a branching factor of over 100, so bounds need to be placed on the search. These bounds are described next.

When cells connected in the graph are also connected in the grid, then *conventional travel* is possible for the player. If they are not adjacent, they need to be connected in the game by a portal, meaning the player must use *portal travel* (see Figure 10). The fraction of graph connections which allow conventional travel is referred to as the conventional travel proportion p_{ct}. The fraction for portal travel is the portal travel proportion p_{pt} with $p_{pt} = 1 - p_{ct}$. We can control the number of portals in a level by setting a target, t, between 0 and 1, for p_{ct} and directing the search toward this value. t is determined via the user preferences as $t = 1.0 - 0.25 \cdot puzzle$. A small interval $[t - \delta, t + \delta]$ is chosen (with $\delta = 0$ initially, increasing if no solution is found) and the search discards those grids with p_{ct} outside of this interval.

Naturally, during the search process we have to predict the value of p_{ct} for a partially formed grid. For this, we sum up the instances of portal travel and conventional travel as usual, then referring back to the graph it can be determined how many connections are still to be added. If each of the remaining connections is assumed to be an instance of conventional travel, this quantity represents the largest possible p_{ct} of any child candidate of this grid; this value will be referred to as Max_{ct}. Assuming these connections are instances of portal travel results in the minimum min_{ct} (see Figure 11). Due to the definition of these quantities, their change will be conveniently monotonic as cells are added to the grid, and will be equal when the last cell is added.

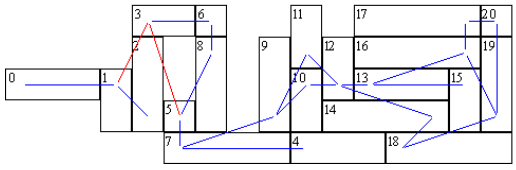


Fig. 10: Instances of conventional travel are shown in blue, portal travel in red.

If there is no overlap between the interval $[min_{ct}, Max_{ct}]$ and the interval $[t-\delta, t+\delta]$, it is clear that no child candidate of the given grid will fall within the interval, so this branch of the search tree will be ignored. An example, with overlap, is illustrated in Fig 11. Examining this condition at every node drastically reduces the search space, and with a well chosen δ, almost the entire search space will be ignored.

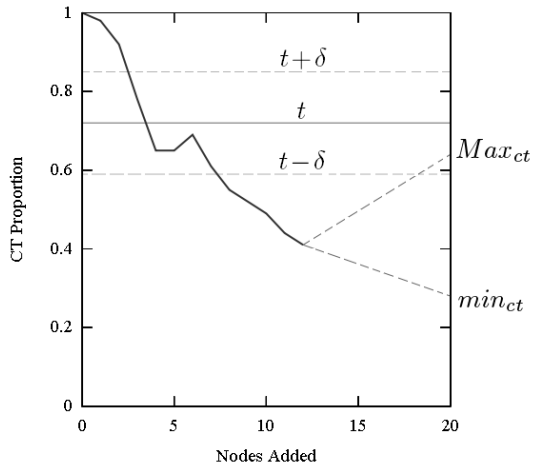


Fig. 11: The change in p_{ct} as nodes are added.

E. Cell Content Generation

With the grid created by the last step, the remaining task is to fill the cells in the grid with content. This is achieved for a cell by beginning with an empty space and then iteratively adding components, such as enemies, platforms, or coins, or removing all components in a small rectangular region (see Figure 12), and evaluating the cell at each step. This is again a search process where, in each step, several possible candidates are created that add to the already existing cell a new component out of a group of different types. If the type is "enemy", a subtype is chosen: Red Turtle, Goomba,

etc. If it is "platform" or "string of coins", a width for this component is also chosen. The candidate cell is then created as a copy of the parent cell, with the new component added in an appropriate position.

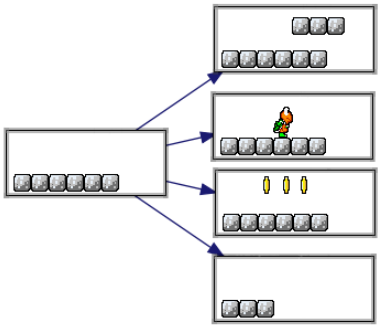


Fig. 12: Generating candidate cells by adding new components.

The search halts when a candidate cell meets three criteria. First, it must be possible to traverse the entire width of the cell. Second, all outgoing portals must be reachable, and third, the cell must score sufficiently high in an evaluation. Cells are scored based on quantities derived from an *accessibility graph* which describes the available paths that a player and enemies may take through the level. It is generated by coarse-grained traversal of the level (equating possible positions with entire blocks). The quantities derived from this fall into two categories: paths and regions. Paths represent stylized player behavior, such as making the fewest jumps possible, or touching the ground as seldom as possible. From these we estimate the path the player is likely to use, identify holes, and determine the total difficulty of the jumps.

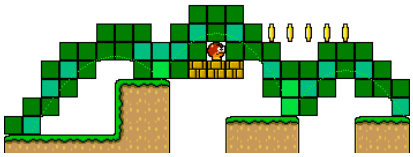


Fig. 13: The path through the cell using the fewest jumps.

The regions are based on both the player accessibility graph and the enemies' accessibility graph. The cell is divided into safe zones and enemy-controlled areas. This is used to measure and control the total difficulty of the cell as it corresponds to the presence of enemies.

A total of 38 metrics are derived from the paths and regions, and they can be classified as follows:

- **Path Length** includes: path with fewest jumps, the coins collected in this path (modified by $clutter$, see below), hole count, etc. There are seven total metrics of this type.
- **Space Composition** includes: width and height which are reachable by the player, and by enemies. It also includes the percentage of each category described in Figure 14. $clutter$ is applied to the weights of safe-zones and combat-zones. There are 15 metrics of this type.
- **Difficulty**: There are two metrics in this category, Jump Difficulty and Enemy Difficulty. Jump difficulty is calculated, for each hole, as the Euclidean distance from the launch point to the landing point, subtracting the excess width of the launch and landing zones (see Figure 15).

  Enemy Difficulty is calculated for each enemy separately and combined into a single value. For each enemy, its type and the portion of the cell which can be reached by it are taken into account.

  These values are directed towards targets, which are calculated from $flow$ and $combat$ below.
- **Ratios** of the above metrics, such as blocks reachable to width, or coins in a path to coins total. We used 14 ratios of metrics.

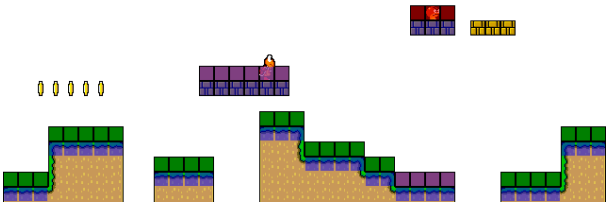


Fig. 14: Regions identified within a cell. **Green**:Safe, **Red**:EnemyOnly, **Purple**:Combat

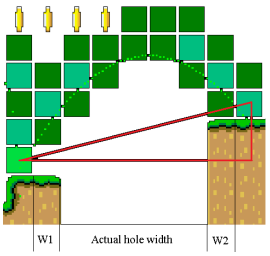


Fig. 15: The difficulty calculated for a single hole.

All these metrics are combined using a weighted sum, which is then adjusted using the $flow$, $combat$, and $clutter$ aspect values for the particular player:

$$score = \sum_{i=1}^{38} w_i \cdot m_i - 20|t_J - m_J| - 20|t_E - m_E| + (clutter) \sum_{i \in C} (v_i) m_i,$$

where C is the set of metrics we want to be modified by $clutter$ (as indicated above), each w_i and v_i represents a weight, and each m_i represents a metric. $t_J = 10 \cdot flow$ is the target Jump Difficulty and $t_E = 100 \cdot combat$ is the target Enemy Difficulty. A threshold for $score$ is given to the system as a system parameter, currently 300.

III. Evaluation

Our level generation method aims at creating a level for a particular person respecting the current moods and wishes of this person. Therefore we have chosen a show-and-tell-like approach to evaluate the method, where we point out parts of created levels that reflect the indicated wishes of the respective players. Fortunately, the skills and wishes of the authors of this paper are diverse enough to allow for such an approach, as indicated by Table I.

	User 1	User 2	User 3	User 4
$T_C(1)$	24	10	23	28
$D_G(1) + D_A(1)$	1539	188	1760	785
$T_C(2)$	21	15	13	20
$D_G(2) + D_A(2)$	1550	314	1251	1015
$T_C(3)$	12	12	10	14
$D_G(3) + D_A(3)$	1503	298	1567	891
$T_C(4)$	36	21	20	127
$D_G(4) + D_A(4)$	3391	390	5019	12070
$K_F + K_{SH}$	21	0	19	1
P	3	0	9	17
Chosen Difficulty	Hard	Easy	Hard	Medium
Chosen Genre	none	puzzle	puzzle	none

TABLE I: A brief outline of the player data gathered in the trials.

The speed with which a player moves indicates experience with the game, particularly with the flow aspect, and is much higher for both User 1 and User 3. The same two players demonstrated confidence in dispatching enemies. Only User 3 completed the fourth trial through use of portals, resulting in a much higher portal score. In the case of Users 2 and 4, who moved slower, covered less ground and killed fewer enemies, the clutter rating has been increased. This adds more coins and items, common to the easier levels.

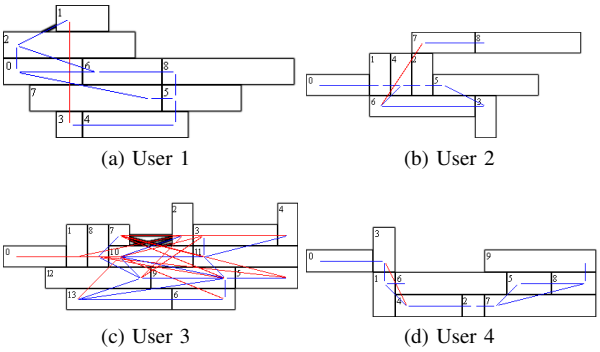
Using our definitions of the high-level parameters from Section II-B, we get the values of these parameters in Table II.

	User 1	User 2	User 3	User 4
flow	0.779	0.0	0.923	0.075
combat	1.0	0.175	0.7	0.325
puzzle	0.29	0.28	0.997	0.393
clutter	0.156	0.912	0.091	0.606

TABLE II: The parameters supplied to the level generator for each player.

Since all users had comparable *puzzle* values, the grid layouts of each level are of similar complexity (Fig. 16), with the one exception being User 3. While User 4 made more extensive use of the portals in the trial, the puzzle genre was not selected as a preference so the final *puzzle* aspect was similar to the players that selected it, but did not use the portals in the trial.

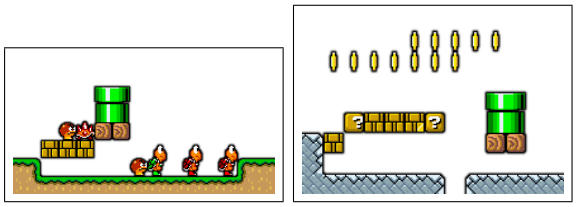
In terms of the combat aspect, the full range of possibilities is represented in the levels (as illustrated in Figure 17). The level generated for User 1 had an average of eight enemies per cell. This is a constant attack with no fewer than two enemies on screen at a time. For User 2, however, there was only one enemy per cell on average, and some cells contained none. User 3 had an average of four enemies per cell, User 4 had three.



(a) User 1 (b) User 2

(c) User 3 (d) User 4

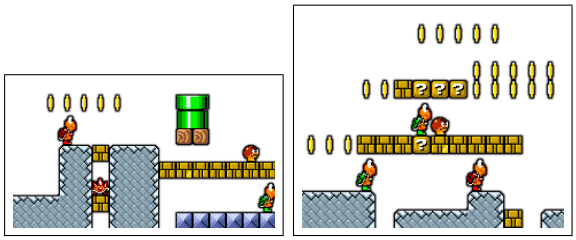
Fig. 16: The associated grids of the levels generated for each player.



(a) User 1 (b) User 2

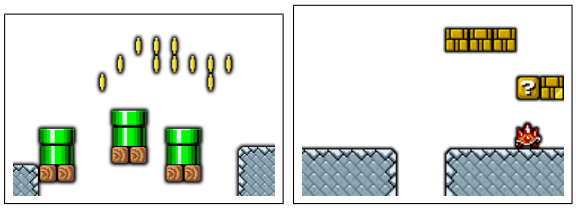
Fig. 17: Extreme cases with respect to *combat*.

While Fig. 17 also displays the difference made by the *clutter* aspect, Fig. 18 is a better example. Even in the case of near zero clutter coins will appear, but the large clusters that User 4 received would never appear for User 3.



(a) User 3 (b) User 4

Fig. 18: Extreme cases with respect to *clutter*.



(a) User 3 (b) User 4

Fig. 19: Extreme cases with respect to *flow*.

As with coins, holes will appear in every level regardless of supplied parameters. What does vary with respect to *flow* is the size and frequency of these holes. The hole shown in Figure 19b is the largest in the level, and the only hole in that cell. In Figure 19a, holes are nearly everywhere except where they would allow a "shortcut" between two cells which are intended to be separate.

IV. Related Work

The majority of work on the subject of procedural content generation for platform games has been focused on Flow and Rhythm. Cskszentmihlyi's concept of Flow [2] has been a popular subject in games, and very popular in platform games, which are based primarily on tactile skill, quick reflexes and timing, and Rhythm has been the preferred method for creating the flow experience. This has been explored very thoroughly in [3], [14], and [12], and is not a big focus in our work (except that we naturally use this aspect as one of the subgenres). See also [15] for an overview of systems using procedural content generation.

We have instead built a PCG system based on two other ideas: that the content will adapt to the user who is playing the game, and that the levels will (optionally) have multiple paths. The cell-based decomposition of a level is mentioned in [13] and discussed in great detail in [9]. The use of cell-based level generation is also successfully used in [12]. Although the possibly of using this for multi-path levels is mentioned in all three papers, it was never fully implemented, in contrast to our system. One system [16] does generate results which, through their complexity, have multiple paths, at least in a local sense. These paths create very interesting levels, but without the intention of having multiple paths, so the choice made by the player of which path to take has only effects on the scale of a single cell, not the entire level (which is what we aimed at).

All the papers mentioned so far do not deal with eliciting user preferences. To our knowledge, the only work that focuses on getting input for level design from users/players is [6]. The event-based user data gathering system we have used is in fact very similar to the one used in [6] and naturally inspired by it. But the aim of [6] was to use the different kinds of metrics produced by the data gathering system together with player classifications to identify metrics and values for these metrics that indicate levels that are fun to play for (nearly) every player, in contrast to our approach that focuses on a single player and creating a level that this single player likes.

V. Conclusion

We presented a method for creating levels for platform games based on eliciting the preferences of a particular player and allowing for different subgenres of platform games. The three step process of creating levels allows for making use of the player preferences on different levels of abstraction and also allows for focusing the search on particular aspects of the game only, which – together with appropriate constraints – keeps the search spaces manageable. In our evaluations, the created levels reflected the wishes of the players they were created for.

We think that the general three step process of level generation together with user preference elicitation using both questions and small trial levels, can also be applied to other platform games and even to other game genres that require generation of one or several levels/worlds. Consequently, among our future plans is applying our general ideas to other games. But we also see potential for improvements of the details of our method, like reducing the number and complexity of trials by using some of the results of [6], allowing us to focus on game features for which different players have different opinions. Also, additional evaluations using a large group of players with appropriate questionnaires might indicate possible improvements.

Acknowledgments

The fourth author's research is supported in part by the Natural Sciences and Engineering Research Council of Canada. The first author was supported in part by a PURE grant.

References

[1] J. Togelius, S. Karakovskiy, and R. Baumgarten: The 2009 Mario AI Competition, Proc. CEC 2010, Barcelona, 2010.
[2] M. Csikszentmihalyi: Flow: the psychology of optimal experience, Harper Perennial, 1990.
[3] G. Smith, M. Treanor, J. Whitehead, and M. Mateas: Rhythm-based level generation for 2d platformers, Proc. 4th Intern. Conf. on Found. of Dig. Games, Port Canaveral, 2009, pp. 175–182.
[4] R. Koster: A Theory of Fun for Game Design, Paraglyph Press, 2005.
[5] J. Juul: Fear of Failing? The Many Meanings of Difficulty, in The Video Game Theory Reader 2, Routledge, 2009, pp. 237–252.
[6] C. Pedersen, J. Togelius, G. Yannakakis: Modeling Player Experience in Super Mario Bros, Proc. CIG-09, Milano, 2009, pp. 132–139.
[7] K.-Y. Jung: User Preference Through Learning User Profile for Ubiquitous Recommendation Systems, Proc. KES 2006, Springer LNAI 4251, 2006, pp. 163–170.
[8] N. Shaker, G .Yannakakis, J. Togelius: Towards automatic personalized content generation for platform games, Proc. 6th AIIDE, Palo Alto, 2010, pp. 63–68.
[9] G. Smith, M. Cha, J. Whitehead: A framework for analysis of 2D platformer levels, Proc. ACM SIGGRAPH Symposium on Video games, Los Angeles, 2008, pp. 75–80.
[10] N. Sorenson, P. Pasquier: Towards a Generic Framework for Automated Video Game Level Creation, Proc. of EvoApplications 2010, Istanbul, 2010, pp. 130–139.
[11] S.O. Kimbrough, M. Lu, D.H. Wood, D.J. Wu: Exploring a two-market genetic algorithm, Proc. GECCO'02, San Francisco, 2002, 415–422.
[12] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, M. Cha: Launchpad: A Rhythm-Based Level Generator for 2D Platformers, to appear in IEEE Trans. Computational Intelligence and AI in Games, 2011.
[13] K. Compton, M. Mateas: Procedural Level Design for Platform Games, Proc. 2nd AIIDE, Stanford, 2006, pp. 109–111.
[14] N. Sorenson, P. Pasquier: The Evolution of Fun: Automatic Level Design through Challenge Modeling, Proc. First International Conference on Computational Creativity (ICCCX), Lisbon, 2010, pp. 258–267.
[15] G. Yannakakis, J. Togelius: Experience-driven Procedural Content Generation, to appear in IEEE Trans. Affective Computing, 2011.
[16] P. Mawhorter, M. Mateas: Procedural level generation using occupancy-regulated extension, Proc. CIG 2010, Copenhagen, 2010, pp. 351–358.