Toward Supporting Stories with Procedurally Generated Game Worlds

Ken Hartsook, Alexander Zook, Sauvik Das, and Mark O. Riedl

***Abstract*—Computer role playing games engage players through interleaved story and open-ended game play. We present an approach to procedurally generating, rendering, and making playable novel games based on *a priori* unknown story structures. These stories may be authored by humans or by computational story generation systems. Our approach couples player, designer, and algorithm to generate a novel game using preferences for game play style, general design aesthetics, and a novel story structure. Our approach is implemented in GAME FORGE, a system that uses search-based optimization to find and render a novel game world configuration that supports a sequence of plot points plus play style preferences. Additionally, GAME FORGE supports execution of the game through reactive control of game world logic and non-player character behavior.**

I. INTRODUCTION

Computer Role-Playing Games (CRPGs) are a genre of games in which a player enters a virtual game world in the role of a story world character that embarks on a number of quests or missions, ultimately achieving some overarching goal. This genre exemplifies the interleaving of story and open-ended adventuring in games. A canonical CRPG example is Nintendo's Zelda™. CRPGs are content-heavy games, requiring a large world, sophisticated story lines, numerous NPCs, and numerous side-quests. Due to their nature, CRPGs typically have low *replayability* – the replay value drops off significantly after the game play affordances of the story and world are exhausted.

As a consequence of their content-heavy nature, CRPGs are prime candidates for the application of *Procedural Content Generation* (PCG). Procedural content generation is any method that creates game content algorithmically, with or without the involvement of a human designer. There are two broad uses of PCG in games: design time assistance of content creation, and run-time adaptation of gameplay. On one hand, PCG may make the development of content-heavy games such as CRPGs cheaper and easier by semi-automating the construction of some game content. On the other hand, personal information about the player, such as the player's wants, desires, preferences, and abilities – information that cannot be known at design time – can be used to personalize the story and world of the game so as to maximize pleasure and minimize frustration and boredom.

In prior work (cf., [6]), we explored the generation of customized game plots irrespective of the world environment in which they would play out. CRPG game plots were automatically adapted to individual players based on a set of preferences over the types of tasks the player likes to perform in role playing games. However, without a game world, a story cannot be played. Will any world do? We assert that the game world must service the story by providing an environment with the right kinds of places in a reasonable configuration. Since CRPGs also involve adventuring in between significant story events, the world should additionally cater to the player's individual preferences for amount and style of adventuring. Prior work has not addressed the construction and coordination of procedurally generated story and world.

In this paper, we explore the question of how to procedurally generate a complete, executable CRPG, given an *a priori* unknown story and a set of player preferences. The challenge of generating a CRPG, which distinguishes it from many other game genres, resides in the central role of story. Story is a common element in many computer game genres used to establish context, motivate the player to perform certain tasks, and shape the player's overall experience inside the virtual world of the game. Storytelling in computer games can be distinguished from other forms of storytelling by the close relationship between the sequence of narrative events and the virtual world – in games, story is the progression of the player through space [1],[2],[5]. From the perspective of game design, level design shapes the story experience, although game players may not perceive the close relationship between story and space.

We present a technique for automatically generating a fully playable CRPG based on a story – created by a human or computational story generator – and information about the player's play style preferences. The contributions of our work are as follows:

1) A technique for generating and rendering a CRPG game world based on play style preferences and designer constraints.
2) Integration of game plot and game world.
3) A general game execution engine that can puppet non-player characters and other plot-relevant aspects of the game based on parameters of the story and specifics of the world.

Our procedural CRPG generation techniques are implemented in the GAME FORGE system, which is described in the remainder of this paper.

All authors are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332 USA (email: {khartsook3, azook3, sdas7, riedl}@gatech.edu).

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

II. Background and Related Work

Computer role-playing games (CRPGs) employ a complex combination of story and game world in order to orchestrate player experience. The CRPG game playing experience, in its basic form, interleaves periods of time when the player is engaged in plot-centric activities and open-ended adventuring. Periods of engagement with the story typically involve interacting with non-player characters (NPCs), solving puzzles, and battling bosses. In between story plot points, the player is encouraged to engage in open-ended adventuring where he or she can explore the game world, fight randomly encountered enemies, collect items and treasure, advance his or her character's skills and abilities, and go on side-quests – quests that do not significantly progress the main plot line. The game world is integrated with the story to provide concrete experiences and a space for players to engage in adventures.

CRPG players often have particular preferences for both the story they experience in a game and the types of activities they engage in within the game world. Players may prefer to experience particular types of story content, such as rescuing a princess from a dragon or retrieving treasure out of a booby-trapped cave. In addition, players may have different attitudes toward how much time they would like to spend exploring the world, going on side-quests, engaging in combat with random encounters, and acquiring items. Current CRPGs typically handle these diverse interests by providing a "one size fits all" story and world that attempt to provide an optimal experience for all types of player preferences over both story and world. That is, game designers attempt to author a broad range of content that is all included within a single game world and story.

Research on procedural generation of computer game content has become popular in recent years. *Search-based procedural content generation* (SBPCG) involves the use of search-based algorithms to explore a space of game content for those meeting a set of evaluation criteria [13]. Examples of content generated using SBPCG include: decorations of virtual environments such as trees and other vegetation; landscapes and 3D environmental geometry; weapons; opponents; game levels; and quests [13],[16]. Work on SBPCG has focused mainly on non-story-driven game genres such as platformers, racing games, and shooters. Moreover, a majority of SBPCG work has focus on generation of a single aspect of the game, rather than integrating multiple aspects for a full game.

Game level generation is the procedural construction of an environment, or space, through which the player must navigate. Level generation is most commonly used in non-story-driven games where the goal is to reach the end of the environment (cf., [9],[10],[11],[12]). In games in which story is analogous to movement through space, level generation may result in the creation of story by constraining the way players move through an environment [2],[4]. These simulation-based methods, however, do not support the orchestration of a story as complicated as those in CRPGs and also typically do not incorporate player preferences into the world model.

Others have emphasized methods for incorporating given world elements into the generation process. Tutenel et al. [14] use semantic information and a constraint-based layout solver to create game worlds meeting designer aesthetics. We have employed elements similar to layout systems in our use of probabilistic distributions tied to particular types of environments while employing a greedy search algorithm for world layout. There is a trade-off between constraints, which require knowledge engineering effort, and algorithmic flexibility.

Dormans [3] has addressed the bipartite nature of story and space. This work uses a context-free shape grammar to arrange a cave to support an individual mission (story). In contrast, we employ an optimization process to avoid the problems of over-generation associated with context-free grammars and to incorporate player preferences.

Many efforts have recently aimed to incorporate player preferences in the field known as *experience-driven procedural content generation* [16]. Within this framework our system uses a subjective player experience model by directly asking about player preferences. Stories and worlds are evaluated directly, mapping given content and structures to quality values based on the player model. Our content representation involves multiple levels of abstraction and uses a non-exhaustive search process for generation. Currently Game Forge does not provide a closed loop where feedback at either the story or world levels influences subsequent rounds of content generation.

III. Story Representation

Game Forge takes story content produced by a user, computational story generation system, or other means and builds a world that supports the story. The story must be provided as a list of plot points. A *plot point* is a high-level specification of a period of time with a semantic and recognizable meaning. Examples include fighting and killing enemies, buying and selling objects, conversing with NPCs, engaging in social activities with NPCs (e.g., marrying a princess), and solving puzzles. The type of plot point is parameterized to reference NPCs and locations appropriate to the story. Each plot point type also includes a reactive script that provides reactive execution logic to control NPCs and modify the game world as appropriate to the plot point. See Section V for more information about reactive scripts.

Plot points reference places by name where the action is to occur. Named places are of a particular type, e.g., "palace" might refer to a particular instantiation of a location of type "castle". As with plot point types, place types are pre-defined. It is important to note that, although plot points reference named places, the story structure itself does not contain information about the spatial layout of these locations; it is the responsibility of Game Forge to automatically determine the spatial layout of the world.

In addition to a sequence of plot points, Game Forge also

1. **Take** (paladin, water-bucket, palace)
2. **Kill** (paladin, baba-yaga, water-bucket, graveyard1)
3. **Drop** (baba-yaga, ruby-slippers, graveyard1)
4. **Take** (paladin, shoes, graveyard1)
5. **Gain-Trust** (paladin, king-alfred, shoes, palace)
6. **Tell-About** (king-alfred, treasure, treasure-cave, paladin)
7. **Take** (paladin, treasure, treasure-cave)
8. **Trap-Closes** (paladin, treasure-cave)
9. **Solve-Puzzle** (paladin, treasure-cave)
10. **Trap-Opens** (paladin, treasure-cave)

Hero (paladin), NPC (baba-yaga), NPC (king-alfred), Place (palace), Place (graveyard1), Place (treasure-cave), Thing (water-bucket), Thing (treasure), Thing (ruby-slippers), Type (baba-yaga, witch), Type (king-alfred, king), Type (palace, castle), Type (graveyard1, graveyard), Type (treasure-cave, cave), Type (water-bucket, bucket), Type (ruby-slippers, shoes), Type (treasure, gold), Evil (baba-yaga) …

Fig. 1. A simple story represented as a list of plot points (top) and an initial state (bottom).

requires an initial state: a list of propositions that describe story-specific details about the story world. The initial state includes information about NPCs, objects, and places that are referenced by the plot points. For example, information about NPCs includes their names, character classes, and other attributes. The initial state provides the type of each of the referenced places as a set of propositions. Proposition types are general and pre-specified. Figure 1 shows a simple example story with its corresponding initial state.

Our story representation is consistent with AI planning-based story generation systems such as [6] and [8] that either generates plots from scratch or adapt novel plots from existing plots. However, the representation is simple enough that humans – with the assistance with authoring tools – can also author their own stories by assembling and parameterizing known plot point types.

GAME FORGE was implemented to create game worlds corresponding to stories generated by the game plot adaptation system described by Li and Riedl [6]. The game plot adaptation algorithm takes an existing hand-authored game plot – represented as a partial-order plan [15] – and a set of preferences about the types of things the player likes to do in CRPGs and searches for a sequence of changes that transforms the original game plot into a new game plot that meets the player's preference specifications. The game plot adaptation algorithm is responsible for adding and removing plot points until success criteria are met. Once the search is complete, a potentially novel story structure may exist. This story structures is converted into the GAME FORGE story representation (the translation from partial-order plan to our story representation is trivial and straight-forward), and sent to GAME FORGE to render and execute the game.

Note that the GAME FORGE story representation currently only handles linear stories. Computer games typically have a single main storyline that constitutes the set of plot points that are necessary and sufficient for completion of the game. GAME FORGE currently only concerns itself with this main storyline. CRPGs, however, often utilize side-quests, plot points that are optional and do not causally link back in with the plot points in the main storyline. GAME FORGE supports side-quests by generating portions of the game world that branch off from essential parts of the world based on the player's stated preference for adventuring. These branching spaces could be areas of the world where side-quests unfold. GAME FORGE, however, does not generate the story content of the side-quests; the creation of the story content for a side-quest is the responsibility of an external agent and may occur before game world generation – providing input into the required size and shape of the world – or after game world generation – to utilize optional portions of the world.

IV. GAME WORLD GENERATION

We solve the problem of automatically designing, building, and rendering a completely functional game world for a story. Recall that CRPGs interleave plot points and open-ended game play; the game world to be generated must ensure a coherent sequence of plot points are encountered in the world. In the process of generating the world, our approach also attempts to incorporate preferences for game play style into the configuration of the world. The problem can be specified as follows: given a list of plot points of known types, referencing locations of known types, generate a game world that allows a linear progression through the plot points and supports user play style preferences. GAME FORGE is specifically targeted to CRPGs, but demonstrates how other heavily story dependent game genres can be supported.

To map from story to space, GAME FORGE utilizes a metaphor of *islands* and *bridges*. Islands are areas where critical plot points occur. Each event in the plot is associated with a location. For example, a story may involve plot points located in a castle, graveyard, and a cave. The game world generator parses the generated plot and extracts a sequence of locations, each of which becomes an island. For example, the story in Figure 1 plays out in three locations: a castle (plot points 1, 5 and 6), a graveyard (plot points 2 through 4), and a cave (plot points 7 through 10). Bridges are areas of the world between islands where non-plot-specific game play occurs, such as random encounters with enemies and discovery of treasure. Bridges can branch, meaning there can be areas that the player does not necessarily need to visit in the course of the story. The length of bridges and the branching factor of bridges are dictated by a player model.

In addition to the hard story constraint that there are specific islands that appear in the world in a specific sequence, we treat the generation of the game world environment as an optimization process, balancing two competing sets of requirements:

1) **Game world model:** Captures the designer-specified believable transitions between environment types as the player moves through the game world. For example, one would not expect to step directly from a castle into a mountain lair without first traversing through mountain terrain.
2) **Player model:** Captures player-indicated play style preferences. Play style preference is represented by a

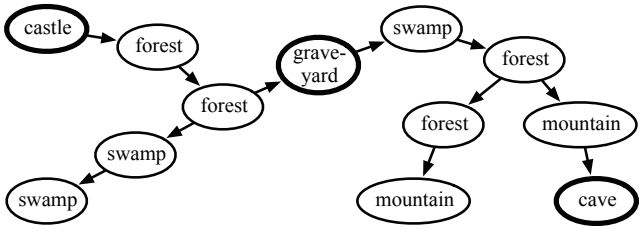

Fig. 2. An example space tree. Islands are denoted by bold outlines.

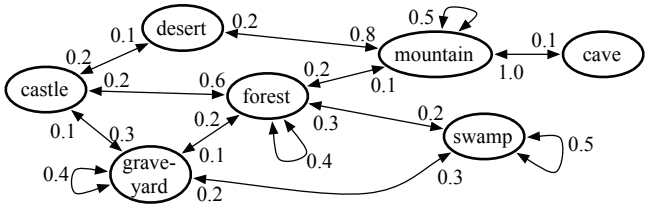

Fig. 3. An example of an environment transition graph.

number of parameters capturing average bridge length (correlating to overall size of the world), linearity of the world (how often there are sidepaths), likelihood of random enemy encounters, and likelihood of finding treasure. The world generation process only uses the parameters associated with bridge length and side paths; the others control gameplay execution.

The game world model is provided by the designer to capture his or her intuitions about believable game worlds. The player model is currently provided directly by the player who answers some simple questions about how much adventuring he or she likes to perform, as well as how much combat and treasure to incorporate. The plot points, manifesting as islands, act as a third, implicit set of requirements. GAME FORGE generates the game world through the use of a genetic algorithm (GA) guided by a fitness function incorporating the above requirements.

A. Genotype Representation

We represent a game world genotype as a space tree, a tree data structure in which each node represents a portion of the world. Each node has an environment type (e.g., castle, field, forest, mountain) that determines what that region of the world should look like in terms of the type of visual qualities, graphical objects and decorations to be found in that region. Space tree nodes also record whether a region is an island or a bridge node. Figure 2 shows an example space tree with islands that correspond to the story shown in Figure 1. Island nodes will always occur along a single path through the tree and islands will always be encountered in the order they are first referenced in the input story.

B. Fitness Function

We employ evaluation criteria based on a game world model and player model. The game world model constrains generation to more natural worlds, while the player model guides generation towards the expressed preferences of a given player. The game world model evaluates the adjacency of environment types along bridges and islands based on an environment transition graph, in which nodes specify types of environments and links indicate the designer-specified probability for two environment types to be adjacent. For example, there is a high probability of a cave being adjacent to a mountain, but a low probability for a forest to be adjacent to a mountain. Figure 3 shows an example of an environment transition graph. Sequences of bridges that more closely match the model are rewarded, while those that incorporate more divergent connections are penalized. Note that while there can be zero probability of two environment types being adjacent according to the transition graph, it is possible for regions of those types to be adjacent in the space tree – it scores very poorly. The game world model encourages game worlds that are believable according to design principles, regardless of personal player preferences.

The player model, on the other hand, drives generation toward game worlds that match expressed player preferences. Bridge length and "branchiness" of the space tree are both evaluated according to this model. The player model specifies a range of bridge lengths desirable to the player, allowing for more densely placed islands (and thus plot points), or longer sojourns between regions encouraging more adventuring through exploration and time spent being lost. Branchiness controls the likelihood of sidepath generation. Players that prefer greater branchiness will be more likely to have side paths from the main course of the plot, allowing exploration to seek additional battles or rewards as any time the player is on a bridge area there is a probability of random encounter with an enemy. The player model skews game world generation towards a desired level of content density and linearity, capturing the unique preferences of individual players.

The fitness of a space tree is the linear sum of feature penalties. The features are: average bridge length, average length of sidepaths, average number of sidepaths per bridge, total number of sidepaths, total number of nodes, environment transition probability, and environment transition variance. Each feature is the *penalty* as distance from a target value. Target values for bridge length and sidepath features are derived from a few simple questions to the player at initialization time. The penalty for environment transition probability is computed by comparing the actual environment transition probability sampled from the space tree to values in the environment transition graph. Because the environment transition probability feature leads the GA to predictable transitions, environment transition variance forces the system to consider transitions that vary from the ideal.

C. Genetic Algorithm Implementation

Our GA starts with an initial population of randomly generated candidate space trees; each is created from a list of islands and pseudo-randomly constructed bridges between each adjacent pair of islands based on the player model and world model. Our GA halts when any individual's fitness is less than a given percentage of the total theoretical possible

```
Let Stock ← Generate initial population
While no individual in Stock is less than or equal to threshold do
   Let Children ← Copy individuals from Stock
   Determine temperature buckets for individuals in Children:
      (10%ile, 25%ile, 50%ile, rest)
   Replace 10% least fit in Children with new individuals
   Foreach individual i in Children do
      If i is chosen according to crossover probability do
         Let j ← Randomly select individual with same temp.
         i,j ← Crossover(i, j)
      End if
      i ← Mutate(i)
   End Foreach
   Survivors ← ∅
   Foreach individual i in Children do
      Let j ← parent of i in Stock
      Survivors ← Survivors ∪ {pick i or j based on fitness}
   End Foreach
   Stock ← Survivors
End While
```

Fig 4. The game world generation algorithm.

penalty. The success threshold can be raised or lowered to increase or decrease the tolerance for variability and randomness; we have experimented with 5% and 7.5% thresholds. Our GA also halts if more than 100 generations pass without significant improvement of the fitness of the best space tree.

To perform a mutation, we randomly choose a node in the given candidate tree to mutate. We then randomly select one of three possible mutations and attempt to mutate the chosen node. If the mutation is successful, we return; otherwise, we chose another node randomly and repeat the process until a mutation succeeds. The possible mutations are:

1) **Addition:** Nodes can be added to a tree to create side-paths (branching) or extend the length of an existing path (extension).
2) **Deletion:** Nodes can be deleted by removing them from their parent's child list and making the deleted node's children refer to its parent as their parent. Deletion fails if the node is an island or the parent of the deleted node has more than two children (as described below this constraint increases the probability that the space tree can be mapped to a 2D world).
3) **Environment Type Change:** The environment of a non-island region can be mutated to any other viable environment, according to the environment transition graph. This may fail if no alternative environment types are possible.

We also use a means of crossover breeding. All space trees share a common set of island nodes in the same order, while differing in the bridges between these islands. In crossover, we randomly select two candidate space trees and randomly exchange some number of bridge nodes between them. The GA pseudocode is shown in Figure 4.

We employ three techniques to maintain diversity in the candidate population while driving the process towards improvement: tournament selection, temperature buckets, and bottom culling. In tournament selection, a space tree and its mutation are compared using the fitness function, with the more fit solution kept for the next generation. That is, each member of the subsequent population must be more fit that its parent. Temperature buckets divide candidate space trees into groups based on their fitness ranks. Temperature is used to describe the mutation rate – higher temperature indicating a greater probability of mutation – and to segregate the population to protect low-fitness solutions. Higher quality solutions have lower temperatures, allowing fine-tuning of high quality game worlds while exploring more broadly using low quality solutions at higher temperatures. Bottom culling involves removing the bottom 10% of the candidate population each iteration and replaces them with randomly generated trees. The cull-and-replace promotes diversity in the stock. Newly generated candidates are placed in the highest temperature bucket where they will be competitive with other candidates in the bucket, yielding offspring that bubble up to lower temperature buckets and thus exploring new parts of the search space.

D. Graphical Realization: From Genotype to Phenotype

The genotype of a game world is the space tree that describes the regions of the world, their environment types, and their adjacency relationships. The phenotype manifestation is a graphically realized virtual world that can be navigated by the player's avatar. Our graphical realization process creates the visual representation of the world that the player will directly interact with. Our graphical realization process builds a 2D world reminiscent of classical CRPG games such as Zelda™. While there are many ways to realize the world, we focus on 2D as a means of demonstrating the general approach of preference plus design constraints in search-based optimization.

For our implementation of GAME FORGE, two issues must be addressed for graphical realization. First, the graph must be mapped to a 2D grid. Second, the grid must be tiled. That is, each grid space receives a graphically rendered tile. The virtual world is a grid where each grid location contains one tile, a 40 x 40 square of pixels. To make graphical realization easier, we hierarchically segment the tiles according to screens and regions. A screen consists of 34 x 16 tiles and is the largest set of tiles that can displayed at once. A region is 3 x 3 screens; there is one space tree node per region.

A recursive backtracking algorithm maps the space tree to regions on the grid. We use a depth-first traversal of the tree, placing each child adjacent to its parent on a grid. In order to prevent an algorithmic bias toward growing the world in a certain direction (e.g. from left to right), our algorithm randomizes the order of cardinal directions it attempts to place each child. To minimize the likelihood that there is no mapping solution because nodes must be mapped to the same region on the grid, we enforce a constraint that nodes have no more than two children, for a total of three adjacent nodes. When there is no mapping solution, we discard the space tree and return to the GA to continue searching for a new optimal space tree configuration. Figure 5 depicts the

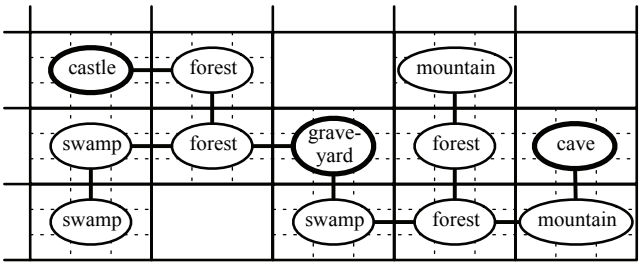

Fig. 5. A space tree mapped to a grid. Regions are solid squares and screens are dashed squares. Space tree nodes are ovals with islands marked in bold.

space tree from Figure 2 mapped to the virtual world's grid.

Once each node in the space tree has been assigned a region on the grid, the module begins graphical instantiation of the world. Each node from the space tree has an environment type, which determines what "decorations" will be placed. Decorations are graphical assets that overlay tiles and visually depict the environment type. For example, a forest environment type will be decorated with trees. The default behavior of GAME FORGE is to distribute decorations based on a Gaussian distribution centered on the center point of a region. To avoid discrete transitions between regions of different environment types, the distribution function extends beyond the boundaries of the region. The net effect is that the regions appear to blend together; decorations can appear in adjacent regions of other environment types.

Environment types sometimes require specific configurations of decorations that are not supported by a simple Gaussian distribution. For instance, a castle would ideally have semi-orderly rows of buildings surrounded by a defensive perimeter of towers. For certain environment types, we use custom distributions. Designers may provide custom distributions that are loaded in the form of bitmaps, where distinct decoration types can have different distributions determined by color intensity. This provides greater levels of authorial guidance to the designer without requiring an exact solution for any environment type to be hard-coded into the game engine. Figure 6 shows examples of the population of different types of environments. The top image shows a forest environment type, populated by a Gaussian distribution; note the blending of decorations from the adjacent swamp above this forest. The middle and bottom images show a castle environment type and the custom distribution bitmap used to generate it; the distribution of buildings is in red, towers in blue, and pavement stones in green.

The graphical instantiation is finished by drawing boundaries around the traversable regions to prevent the player character from walking off of mapped territory or in between adjacent grid positions, which are not connected in the space tree. Boundary walls are created by computing a fractal line connected to other boundary lines; non-passable tiles depicting water are placed along the fractal line.

E. Results

Figure 7 shows a world generated for a plot with three

Fig. 6. Screenshots of forest (top) and castle (middle) regions. The castle is a result of custom distributions over buildings and towers provided by a distribution bitmap (bottom).

islands: castle, graveyard, and cave. Figure 8 shows the game being played. Figure 9 shows some of the range of virtual spaces generated for the same story with two very different play styles: large, branchy worlds supporting adventuring between plot points (top), and smaller, less branchy worlds supporting a more direct progression from plot point to plot point (bottom).

Figure 10 shows the learning curves for three problem configurations expressing different player preferences. Each learning curve is the fitness of the best individual in the population, averaged over 50 runs. The blue solid line shows the average evolution of large, branchy worlds that support a lot of adventuring. The red dashed line shows the average evolution small, non-branchy worlds. The orange dotted line shows the evolution of worlds with moderate size and moderate branching. Recall that our GA minimizes penalty. The horizontal lines show the 7.5% and 5% penalty thresholds, respectively. Each configuration has different thresholds due to differences in maximum possible penalty.

After 100 generations, the moderate world population reaches lower fitness levels those for large branchy worlds or small non-branchy worlds (see Figure 10). For extremely large worlds and extremely small worlds, several features in

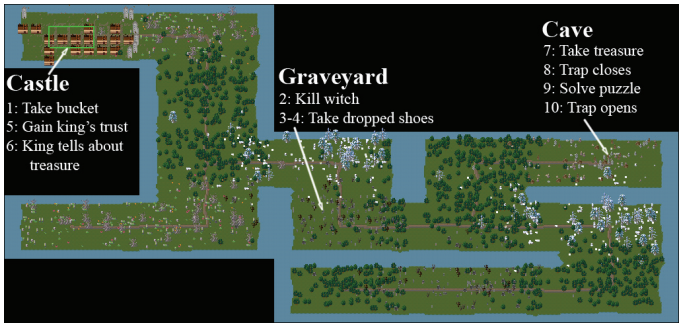

Fig. 7. A game world generated for the story from Figure 1, involving plot points that take place in a castle, graveyard, and cave. Plot points are listed.

Fig. 8. A shot of the game being played. The player is about to encounter the king in the castle island.

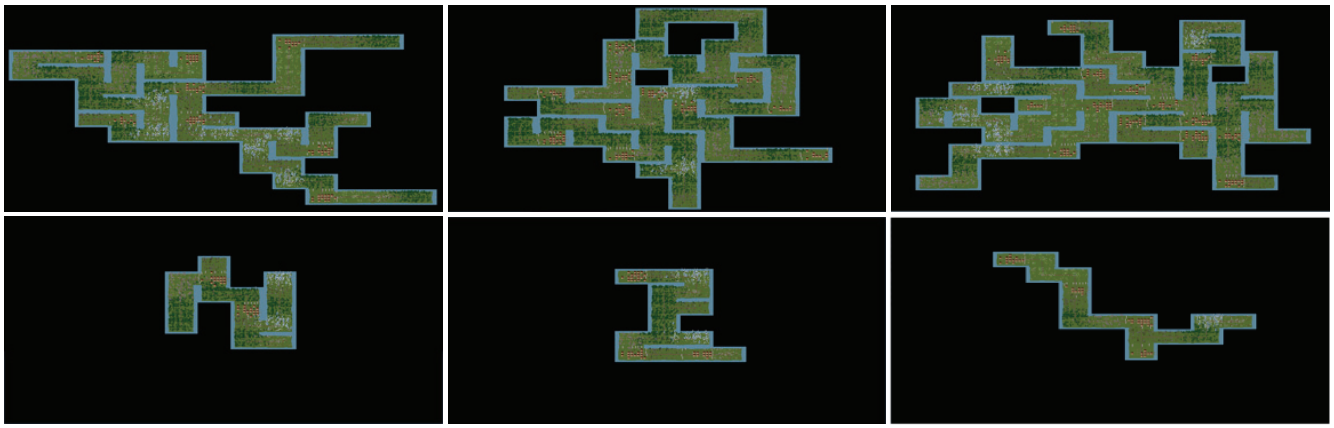

Fig. 9. Example worlds generated for the same plot. The top row was generated with parameters set for a larger world with greater branching. The bottom row was generated with parameters set for a smaller world and little branching.

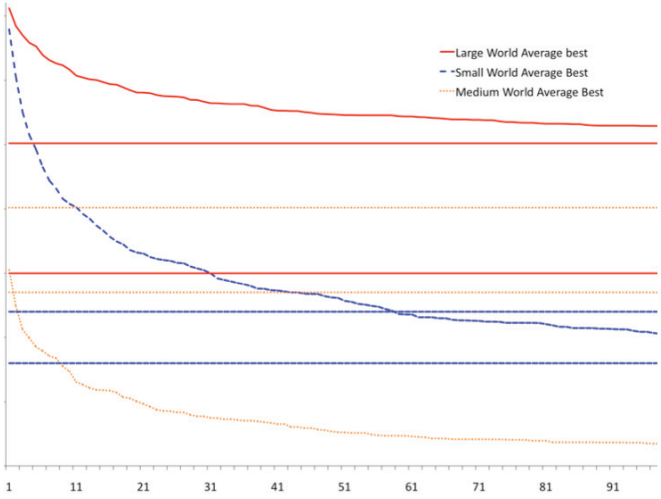

Fig. 10. Fitness of best individual per generation for three different configurations of the player model: a larger world with a lot of branching, a smaller world with little branching, and a moderate sized world with moderate branching. Horizontal lines show the 7.5% and 5% penalty thresholds for each problem configuration.

the fitness function conflict. In large, branchy worlds, we find that as the algorithm adds more nodes to expand the world, the space tree's environment transition variance penalty is likely to increase, making it difficult to minimze penalties for both. For small worlds, the algorithm stagnates when trying to create additional side paths along short bridges. Optimizing the environment transition variance for small worlds also seems to be a difficult task given the imposition of immutable environments (e.g., the islands). Future work is needed to determine the impact of high penalty values on human players perception of the world and enjoyment with different play style preferences.

V. Story Execution

After construction of the game world, the game must be playable without further modification by the user. There are two issues that must be addressed: (a) the world must be populated with NPCs, and (b) the NPCs must act out the story, which was not known prior to execution.

Population of the world with NPCs and items begins by parsing the input story structure. Character type information is used to select the appropriate avatar for each character. In our current system, avatars are simple animated sprites. We use a simple, easily extensible meta-data map from symbolic descriptors to art assets. The same process is used to handle items (e.g., the bucket, shoes, and treasure). The location at which each NPC and item appears is determined by parsing the story structure, looking for the location of the event in which the NPC or item is first referenced (always an island). Thus, the coordinate position at which to instantiate the NPC or item is determined by locating the island in the space tree and determining the center coordinates in Cartesian space with which the island node correlates. See Figure 8 for a

screen shot of the player about to interact with an NPC.

NPC and item behavior in the world is the result of executing *reactive scripts*. A reactive script is a tree of behaviors implemented in the ABL (A Behavior Language) reactive planning language [7]. While many game scripting systems utilize complicated IF-THEN-ELSE structures to manage their control flow, ABL uses hierarchically arranged behaviors defined with conditions and priorities. Each behavior encapsulates the logic to complete a specific goal under a specific world condition; alternative behaviors provide robust execution when game world conditions vary.

Story execution activates each plot point in a totally ordered sequence. Each plot point is associated with a reactive script that puppets NPCs and items, a technique similar to the *action classes* [17] approach. The reactive scripts send instructions to the game engine to modify the game environment and manipulate NPCs and items. The reactive scripts further monitor player actions and the game world state changes through sensors. When a reactive script terminates, GAME FORGE loads the next plot point and triggers the corresponding reactive script.

Plot points are parameterized to account for different narrative needs. When plot points execute, they pass their parameters on to the reactive scripts. These parameters not only determine which NPC or item performs the associated behavior, but causes different branches through the ABL behavior tree. Thus, the content and role of reactive plot scripts can vary with the nature of the type of the plot point. For instance, the "marry" script decorates a location with wedding items, adds guests in tuxes, controls the march of the bride down the aisle, and executes the dialogue of the NPC performing the ceremony. Conversely, a reactive plot script can allow for significant player interaction: the conditions in ABL behaviors coupled with sensors can implement dialogue trees, and a plot point representing a puzzle can directly change environmental conditions as the player manipulates levers and objects.

VI. CONCLUSIONS

GAME FORGE is a system that uses artificial intelligence techniques to integrate story and world in CRPG games. This process balances story requirements, designer control – in the form of insights about good arrangements of environments – and player preferences. The result is an ability to produce a game world that is both functional and favors a particular individual's play style.

Using a genetic algorithm and the metaphor of islands and bridges, GAME FORGE solves the problem of finding an optimal game world configuration that supports a given story structure that it has never seen before. The genetic algorithm attempts to balance player preferences with designer-specified information about appropriate world environment transitions. With a player model, GAME FORGE is able to incorporate individual player differences with respect to adventuring activities that occur between plot points. The designer-specified information constrains against non-aesthetic environment transitions, maximizing the believability and coherence of the game world. After the game world is generated, GAME FORGE is able to use information from the story, combined with a library of reactive plot scripts, to execute the story without further human effort.

While GAME FORGE specifically applies to Computer Role Playing Games, with a specific focus on 2D Zelda™-like games, we believe that the techniques used by GAME FORGE can extrapolate to other story-based game genres. When coupled with automated story generation systems such as that in [6], we believe that GAME FORGE demonstrates the potential for semi- or fully-automated generation and execution of highly story-dependent computer games.

REFERENCES

[1] E. Aarseth, "Beyond the frontier: quest games as postnarrative discourse," in *Narrative Across Media: The Language of Storytelling*, M.-L. Ryan, Ed. Lincoln: University of Nebraska Press, 2004.

[2] C. Ashmore and M. Nitsche, "The quest in a generated world," in *Proc. Annual Conf. of the Digital Games Research Association*, 2007.

[3] J. Dormans, "Adventures in level design: generating missions and spaces for action adventure games," in *Proc. Workshop on Procedural Content Generation in Games*, 2010.

[4] K. Hullett and M. Mateas, "Scenario generation for emergency rescue training games," in *Proc. 4th International Conf. on the Foundations of Digital Games*, 2009.

[5] H. Jenkins, "Game design as narrative architecture," in *First Person: New Media as Story, Performance, Game*, N. Wardrip-Fruin & P. Harrigan, Eds. Cambridge: MIT Press, 2004.

[6] B. Li and M.O. Riedl, "An offline planning approach to game plotline adaptation," in *Proc. 6th Annual Conf. on Artificial Intelligence for Interactive Digital Entertainment*, 2010.

[7] M. Mateas and A. Stern, "A Behavior Language: joint action and behavioral idioms," in *Life-like Characters: Tools, Affective Functions and Applications*, H. Prendinger and M. Ishizuka, Eds. Springer, 2004.

[8] M.O. Riedl and R.M. Young, "Narrative planning: balancing plot and character," *J. Artificial Intelligence Research*, vol. 39, pp. 217-268, 2010.

[9] N. Shaker, G. Yannakakis, and J. Togelius, "Towards automatic personalized content generation for platform games," in *Proc. 6th Annual Conf. on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[10] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2D platformers," in *Proc. 4th International Conf. on Foundations of Digital Games*, 2009.

[11] N. Sorenson and P. Pasquier, "Towards a generic framework for automated video game level creation," in *Proc. of the International Conf. on Evolutionary Computing in Games*, 2010.

[12] J. Togelius, R. De Nardi, and S. Lucas, "Towards automatic personalised content creation for racing games," in *Proc. IEEE Symposium on Computational Intelligence and Games*, 2007.

[13] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, "Search-based procedural content generation," in *Proc. 2nd European Event on Bio-inspired Algorithms in Games*, 2010.

[14] T. Tutenel, R.M. Smelik, R. Bidarra, and K.J. de Kraker, "Using semantics to improve the design of game worlds," in *Proc. 5th Annual Conf. on Artificial Intelligence for Interactive Digital Entertainment*, 2009.

[15] D. Weld, "An introduction to least commitment planning," *AI Magazine*, vol. 15, pp. 27-61, 1994.

[16] G. Yannakakis and J. Togelius, "Experience-driven procedural content generation," *IEEE Trans. on Affective Computing*, to be published.

[17] R.M. Young, M.O. Riedl, M. Branly, A. Jhala, R.J. Martin, abd C.J. Saretto, "An architecture for integrating plan-based behavior generation with interactive game environments," *J. Game Development*, vol. 1, pp. 51-70, 2004.