Smart Terrain Causality Chains for Adventure-Game Puzzle Generation

Isaac Dart and Mark J. Nelson

***Abstract*—Adventure videogames have the player assume the role of protagonist in an interactive story, which is primarily driven by exploration and puzzle-solving. A major drawback with this genre is minimal replayability, since the player has already seen what there is to explore, and knows how to solve the puzzles. We propose a technique to generate variations on puzzles that fit in the same location in the original story, and therefore don't require fully procedural story generation. We keep a database of smart terrain items, which can have effects on other items. Puzzles are generated by taking advantage of a duality between puzzle-solving and generation. Once we build *smart terrain causality chains* (STCCs) of puzzle solutions, a puzzle known to be solvable can be generated by simply inserting the items contained in a causality chain into the environment. We demonstrate this technique in an experimental videogame, *Space Dust*, which shows that even a very short adventure game can produce multiple interesting playthroughs when STCC-based puzzle generation is added.**

I. Introduction

A major criticism of the traditional adventure-game genre has been that once a player completes a game, there is little or no replayability. This is due to the progressive, story-driven nature of the genre, in which a player uncovers how to progress through an environment, solving puzzles and watching the story unfold along the way. Once the progression has been uncovered, there is minimal interest in uncovering the same thing again.

One approach to increasing replayability could be to make the story less linear and more dynamic. In recent years, some adventure games such as *Heavy Rain* (Quantic Dream, 2010) have attempted to overcome the replayability issue by introducing branching storylines, where a player's choices influence which forks are taken. The motivation for branching stories is usually to increase *agency*—making the player feel that their decisions really impact the story—rather than specifically replayability. Nonetheless, branching stories in adventure games do add a certain kind of replayability, since the player can take a different fork the next time through, and experience a new portion of the story, which will contain new challenges not seen in the first playthrough.

A problem with branching stories is that considerable additional authoring and engineering effort is needed: each of those branches, and all the locations, characters, and art assets needed for them, must be created and tested. Procedurally generating those story branches is a possible medium- to long-term solution, but involves solving a mixture of content-generation problems, including the notoriously difficult problem of story generation.

Here we propose a different approach: replayability *without* high-level narrative variation, by procedurally generating puzzles that reuse existing locations and fit into the existing narrative progression. Despite retaining the same storyline and content, this will nonetheless produce new challenges when replayed; the player cannot just rush through doing exactly the same thing as on a previous playthrough.

We aim for a near-term, practical, "drop-in" solution to adding procedural variation within adventure games. We can observe a common pattern in adventure games where portions of the story have as a bottleneck an objective the player must reach to progress, and a puzzle they must solve to reach the objective, often by using several items. A simple example might be to open a treasure chest; the puzzle is finding and using the key. For our purposes, each challenge is made up of game props which, when used in the correct way, solve the challenge. It is at this level that we are able to introduce emergence and re-playability into adventure games without requiring changes in the high-level content: by retaining the same objective of a particular challenge, but varying the available props and their usage, we can replace a puzzle with another one that serves an equivalent role in the story progression, but is not the same as on previous playthroughs.

II. Smart terrain causality chains

Items in an adventure game can be viewed from the perspective of "smart terrain", which characterises items by what problems they can solve for the player. The term was popularised by game designer Will Wright; in Wright's *The Sims*, "objects on the terrain broadcast what they offer to a Sim character... for example, a refrigerator might broadcast the fact that it can satisfy hunger" [1]. Pieces of smart terrain, like the related computer-graphics concept of "smart objects", can be queried for a list of actions a particular object can perform, relative to some semantics of a given virtual world [2]. Objects may also broadcast and react without direct player involvement; for example, an oven in *The Sims* offers the player a list of explicit actions the oven can be used for (*e.g.*, bake another item), and also contains actions not shown to the player, such as catching on fire if an item is left too long.

In an adventure-game puzzle, what an item does for the player is one of two things. Either it solves the puzzle directly, allowing the player to reach the objective, or it produces

The authors are with the Center for Computer Games Research at ITU Copenhagen. email: isaac.dart@gmail.com, mjas@itu.dk

978-1-4673-1194-6/12/$31.00 ©2012 IEEE

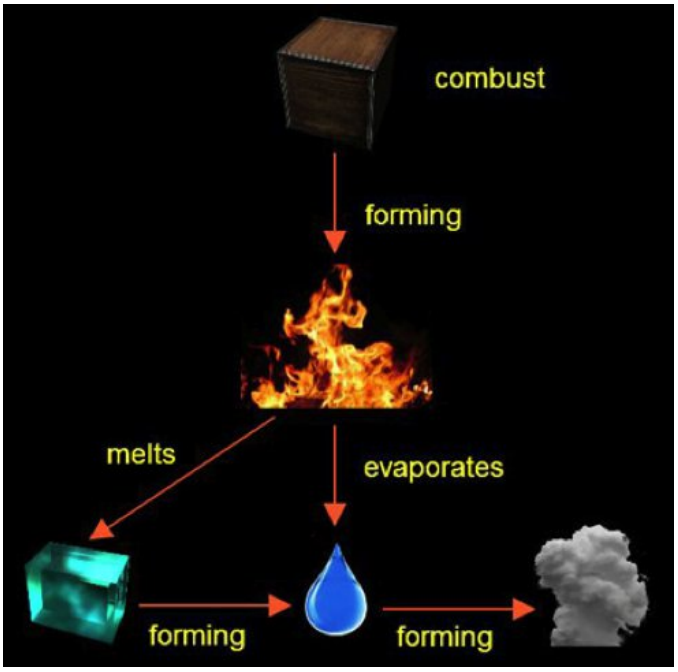


Fig. 1: An example smart terrain causality chain (STCC). In this case, the puzzle starts with two smart-terrain objects, a wooden crate and an ice cube. The outcome is steam, and the chain encodes the causal relationships to get from the starting objects to steam.

an effect on another item, which will allow another item to (eventually) get the player to the objective. Therefore we can think of any puzzle solution as a sequence of items; this is the smart-terrain shorthand for viewing a puzzle solution as the sequence of actions that the player can take by using those items.

A *smart terrain causality chain* (STCC) is a directed graph specifying the causal dependencies between the smart terrain objects present in a scene and the objective of the puzzle. Figure 1 shows an example causal chain, one that produces steam from starting items wood and ice (by burning the wood, which produces fire, which melts the ice, which is then evaporated by the fire).

In *Space Dust*, many of the items provide the player with an "actions" menu when selected. These active actions could be compared to *The Sims'* oven in displaying a list of actions to the player. On the other hand, passive actions are those which occur without player input, such as a "heat" instance notifying nearby smart terrain to increase in temperature.

A. Smart terrain

Unlike in traditional adventure games, items do not have to play a single role. In a traditional adventure game, a piece of paper with a code on it might be used to convey to the player what code to enter when opening a vault. Once the vault door is open, the paper may have no other meaning or use in the game, if none was programmed for it. This narrow model can often be a source of frustration for the player when she believes that such an item could be used in other tasks which it wasn't programmed to do. For example the game may enforce a rule that she has to find some newspaper to light a fire, despite the fact she already has the code written on paper, which should also be burnable. To procedurally generate puzzles, a more flexible model is required that allows items to affect other items that they may not be aware of, or have not even been created at that point. This model is less concerned about particular object interaction, and more concerned about how a type of object affects the environment and how environmental forces affect that type.

Compared to the classic "hard coded" approach, where newspaper is programmed to burn when a lighter is used on it, smart terrain paper doesn't need to know anything about what item will make it burn; rather it simply knows that it should combust if its temperature reaches a certain point. In other words, smart terrain objects only need to know about their own behavior and reactions to other physical forces, rarely reacting to particular instances of other objects.

The interactions between smart terrain objects are implemented at two levels: a physics simulation in the game engine itself that implements the actual interactions, and a symbolic description of the *possible* interactions, which is used to procedurally generate causality chains. In both cases, objects are arranged into object hierarchies, so for example a newspaper is a kind of paper, and inherits the behaviors that all paper has (such as burning above a certain temperature).

B. Physics simulation

The physics simulation provides the "glue" that causes interaction between objects without each knowing about the other: a fire knows to emit heat into the world, and a paper knows to burn when heat passes a threshold. The physics engine is the medium in between the two that propagates the heat from the fire to the paper.

The physics engine for *Space Dust* is implemented in the Unity game engine. The root smart terrain objects are the three classical states of matter: solid, liquid, and gas. Objects can be modified by undergoing the familiar phase transitions—melting, freezing, evaporation, and condensation—as well as by undergoing custom transitions, such as crumbling a piece of paper or burning it.[1]

Objects affect each other by direct collision or indirect energy. Object movement depends on the state: solids can fall via gravity, liquids not in a container disperse along flat and downwardly sloped surfaces, and gases not in a container diffuse through the air. If these movements cause two objects to collide, they may affect each other as a result. Alternatively, objects can emit several kinds of energy, which will affect objects in range that respond to that kind of energy. Energy can come in many forms, of which we've implemented four: thermal radiation, electrical current, radio transmission, and sound emission. Each of these travels in different ways;

[1]The fourth phase of matter, plasma; and the direct solid/gas transitions, sublimation and deposition, are a bit too esoteric to be commonly used in adventure-game puzzle physics.

for example, thermal radiation expands in a gas-like cloud, whereas radio transmission instantly reaches everywhere in range.

C. Generating causality chains

To generate causality chains capturing a sequence of smart object interactions that can solve a puzzle, we need a qualitative version of this physics simulation rather than the full numerical one, capturing which interactions are *possible* when the simulation is actually run; for example, the facts that paper can burn, and water can short-circuit electronics. To keep it lightweight, we don't implement a full qualitative physics system [3], but instead maintain a discrete list of possible actions that can act on each smart-terrain object, together with the cause and the effect of each action. Doing so allows us to implement a simple, greedy backwards-chaining planner directly in Unity, with actions retrieved from an embedded SQLite database.

Each object is assigned an ID, and can appear one or more times in a table of actions, which has four fields:

- **object_id**: The object this action operates on
- **action**: A verb describing the action
- **cause**: A symbol specifying what triggers this action
- **effect**: A symbol specifying the outcome of the action

For example, the following two entries specify that a sprinkler can be activated by smoke, producing water; and a paper can be burned by fire, producing smoke.

object	action	cause	effect
sprinkler	activate	smoke	water
paper	burn	fire	smoke

From this example pair, the causal chaining possibilities should be clear: one possible way to produce water is to use fire (from another object, perhaps a lighter) to burn a paper, which produces smoke, which can activate a sprinkler, which produces water. If a cause is of the form *object(object_name)*, it means that the action can be directly caused by a specific object. If an effect is of that form, it means that the action spawns the named object.

Smart terrain causality chains are procedurally generated by starting from a set of objectives for a scene, and backwards chaining until they ground out in causes that are all primitive smart-terrain objects that can be placed into the scene. A scene is simply the level of granularity at which we generate puzzles; the outcomes of a scene are used by the author to connect the generated puzzles into the larger story progression. Each scene can have one or more possible objectives, and achieving any of them is considered a puzzle solution. Each objective is a specific effect on a specific smart terrain object. For example, in *Space Dust*, the player escapes the prison cell whenever the effect *turn_off* happens on the smart terrain object *electro_bars* (see Section III).

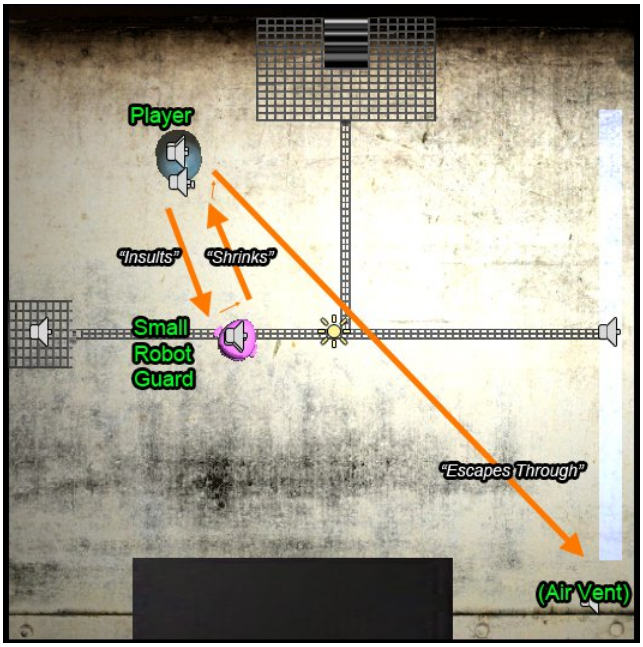
A simple backwards-chaining algorithm suffices to solve this problem. We first choose a random objective, and add the *effect* needed to satisfy that objective to our list of effects that we must cause. Now we look for an action that has that effect, and add it to the causality chain. This action in turn may need some *causes*, and we repeat, finding another action whose *effect* matches up with the needed *cause*. This process continues until it grounds out in the specially treated cause *object(Player)*, indicating something the player can do directly. Figure 2 shows example causal chains generated in our game *Space Dust*.

This formulation can be seen as a subset of STRIPS planning [4], itself a simplification of logical planning in which the objective and all preconditions and postconditions of planning operators must be conjunctions of positive literals. A STRIPS operator corresponds to our actions, a precondition to our causes, and a postcondition to our effects. Since we have no logical operators, we trivially meet the STRIPS conditions, and as a result, simple backwards chaining rather than more complex theorem proving suffices.

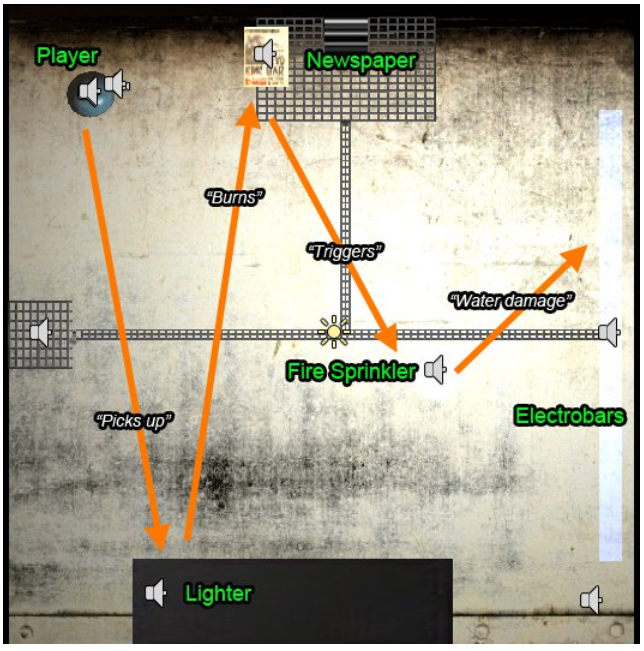
To produce the puzzle, all smart-terrain objects referenced in the STCC are added to the scenario. By construction there will be one or more solutions to the puzzle as a result, taking advantage of the duality between puzzle solving and puzzle generation. In addition, all items placed in the puzzle will have some possible use for at least one of the solution paths, which is consistent with traditional adventure-game design, in which the player finding objects is typically used as a hint towards the solution, so having many objects with no purpose wouldn't be desired. However, if desired, the algorithm can be run multiple times and the runs combined, to produce puzzles with multiple solutions.

When adding smart-terrain objects to a puzzle, there are several additional cases. Objects at the leaves of the smart-terrain object hierarchy are *concrete* objects, and have graphical models associated with them, so they can be instantiated and placed in the world. Objects further up the hierarchy are abstract smart terrain, which can be referenced when building causality chains, but must be specialised to a specific object before the puzzle can actually be produced. By inheritance, however, any object beneath the abstract object in the hierarchy may be selected (we simply select randomly). Secondly, some objects may inherently depend on another object, for physical or conceptual reasons. We keep a second table of object dependencies, and add in any dependencies of the objects contained in an STCC, even if they aren't themselves part of the causal chain.

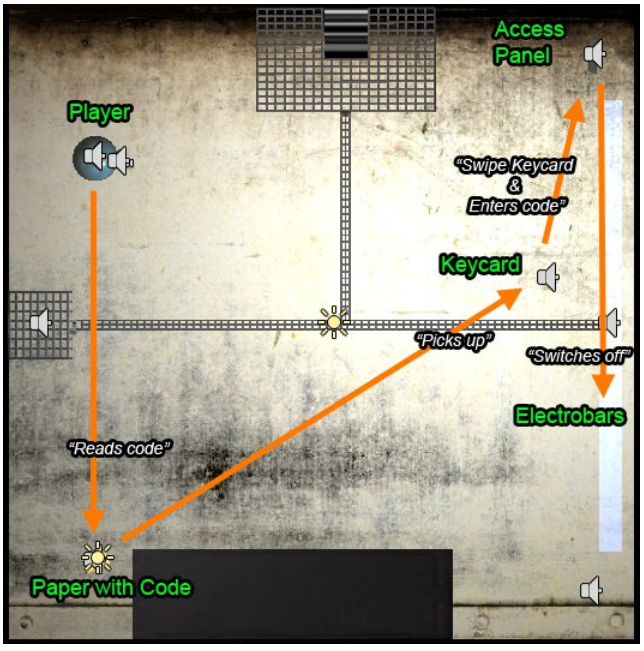
Finally, some objects may need to be included to make the puzzle a challenge at all; each scene includes a list of such mandatory objects. In our *Space Dust* puzzle, for example, there is one solution that involves the player exiting the room without turning off the electric bars blocking the doorway (instead, the player shrinks and exits through a small hole). Therefore, this solution's STCC doesn't include *electro_bars* in it, so they wouldn't need to be added to the scene as a required smart-terrain object for the puzzle solution. But if they aren't added, the player can simply walk out the unobstructed door! The bars are needed here even if the player doesn't use the solution that involves disabling them, because they serve the purpose of enforcing the challenge even for other



(a)



(b)



(c)

Fig. 2: Examples of STCC-generated puzzles in *Space Dust*, with the smart objects and causality chains labeled. See Section III-A for an explanation of each puzzle.

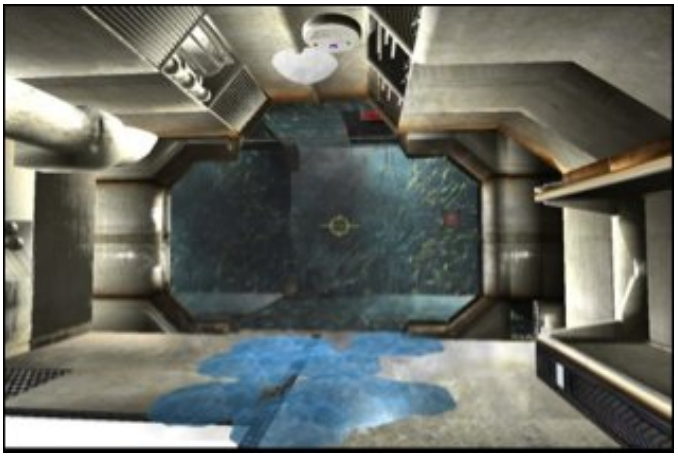


Fig. 3: First-person view of the main *Space Dust* puzzle, in which the player must escape from a prison cell. Here, the fire sprinkler has been activated, spilling water on the floor, which shorts the electric bars blocking the doorway.

solutions. This meshes with Dormans's [5] observation that procedurally generated missions are insufficient for enforcing challenges unless coupled with world boundaries.

III. *Space Dust*

We built a prototype game, *Space Dust*, in which a single adventure-game level is replayed multiple times, as a core part of the gameplay (multiple playthroughs are intended to be necessary). The goal was both to test our puzzle-generation system, and to investigate the hypothesis that adventure games can have significant replayability added if the puzzles are varied, even when the general story progression isn't.

The player plays a character imprisoned on board a space ship adrift in deep space, attempting to escape from a prison cell. Each time the player gets caught, or passes out, they wake to find themselves back in the cell with a new puzzle to solve. Figure 3 shows a first-person view, with the electric bars blocking the cell exit being shorted out by water from the fire sprinklers.

Once the player solves the main objective of escaping from the cell, they have a choice to go to the left or right corridor. To the right is certain death by way of a Killer Bots stun gun, and to the left is an airlock. The airlock will eject the player into deep space where they have a short amount of time to find the airlock entrance to the bridge before passing out and waking, once again in prison with another procedurally generated puzzle. Since this involves a series of immediate replays, rather than generating puzzles purely randomly, we store generated puzzles and make sure that subsequent puzzles don't repeat any previous puzzle (until the game is completely restarted).

As intended, no playtester has won the game in their first ten attempts. The reason for this borderline abusive win condition was to guide the gameplay into presenting multiple scenarios to the player in order for them to experience, solve and hopefully enjoy the replay value of the game. One playtester enthusiastically described the experience as being similar to the movie *Groundhog Day*.

The player interface in *Space Dust* is a typical 3d adventure-game interface. The player can walk around using WASD controls and mouselook; players can also pick up, use and even throw Smart Terrain items, as well as interact with characters by selecting them and being presented with a context sensitive smart-terrain actions menu.

A. *Example puzzles*

Three example generated puzzles, schematically illustrated in Figure 2 and glossed in English below, illustrate some of the range of variation:

(a) If the player upsets the small robot guard, he will shrink the player. The game designer specified being shrunk as a winning condition for the prison cell level, because there is a small air vent in the cell which the player can escape through once miniaturised. Even though electro bars are not a part of this plan, they have been automatically included, as they are specified as a required scene object.

(b) The electro bars will short-circuit on contact with water (these aliens were clearly not very bright). A fire sprinkler is randomly selected to create this water; and paper is randomly selected to create smoke. Since paper requires fire to be present to produce smoke, a lighter is selected to instigate the fire. The player is selected to light the lighter, thus ending the puzzle-generation backwards chaining.

(c) An access panel is made available to turn off the electro bars. This panel requires a key card and a code to authorise a switch-off. The player must search the cell to find these two items. Additionally, the access panel is made aware of the paper-with-code, and registers the code on the paper as the only valid code.

B. *Controlling difficulty*

The difficulty level of a puzzle can be altered by changing the number of solutions which are procedurally generated in parallel. Players can have between one and three parallel causality chains generated at a time.

It is not immediately obvious if more solutions in a scene make the main objective easier or harder to complete. More solutions allow multiple routes to complete the objective, but also cause more objects to be placed in the scene. Because players often rely on the objects in a puzzle to guide their line of thinking, having many objects in the scene may confuse the player as a significant portion of objects will play no role in their final solution.

In order to determine whether more solutions increases or decreases the difficulty of solving a main objective, a playtest was performed, in which ten testers were given generated puzzles with between one and three parallel solutions, which they played twice each, before answering a survey which included questions on the difficulty in escaping from the prison cell.

The survey revealed that 70 percent of people found that more solutions were easier. A number of participants commented that it was easier because they could switch strategies and try another solution if they got stuck with a particular item. Several players did find that more objects caused more confusion, however.

Looking at variation within a single puzzle, players generally said that the longer the causality chain was, the more difficult it was to solve. One player commented that a puzzle made of two objects was "insultingly easy". On the other hand, only one person was able to solve the puzzle with the longest chain. Puzzle length is therefore another viable alternative for mapping against a difficulty setting.

IV. Related Work

Symon [6] is another experimental adventure game with procedurally generated puzzles based on item placement. It generates "dream logic" puzzles, in which events are linked in a way that's intended to somewhat make sense, yet not quite. Puzzles are formulated as template-like grammar structures, which are then filled in with items from a database, possibly subject to constraints (*e.g.*, that one template slot can only be filled with a cold item). The inspiration behind *Symon*'s use of grammatical structure in representing puzzles chains is based on Fox Harrell's work in developing a system to procedurally generate poetry, called GRIOT [7]. Somewhat simplified, GRIOT relies on inputs consisting of phrase templates which contain wildcard tokens later populated with domain aware words which form poetry instances. Such inputs are provided by a poetic system designer.

Dormans [5] approaches adventure-game puzzle generation starting from the higher-level structure of the adventure genre's scenarios. In his system, a graph grammar is used to generate missions, and a shape grammar to generate spaces (the world). Dormans describes how adventure games are composed of spaces and missions and goes on to explain that, although separate concepts, they have certain relationships to each other. For example, some treasure (mission) must be placed behind a door (world) otherwise the player will find the challenge trivial.

Such grammar-based methods, compared to our causal-reasoning approach, make different things explicit. In grammar-based methods, causal relationships are implicit, and enforced only to the extent that the grammar expansion prevents causally impossible things from being produced, but the higher-level structure of a puzzle is explicit and enforced by construction. In causal-reasoning methods (such as ours), on the other hand, causal relationships are explicit and enforced by construction, but higher-level structure is implicit, and enforced only to the extent that the causal constraints, or other explicitly added constraints (such as our required objects) indirectly produce such structure.

The procedural cartoon gag generator of Olsen and Mateas [8] targets a somewhat different problem (comedic gags rather than puzzles), but uses a hierarchical task network (HTN) planning technique that comes closer to our planning approach. The planner in their system places props, and plans action sequences, in a simulated Coyote and Road Runner scenario so that a comedic *failure* in the coyote's plan to catch the road runner will result. Thus it uses similar techniques to achive almost the opposite result—solvable puzzles are the goal in adventure games, but failure of the coyote's schemes are the core of the Road Runner and Coyote cartoons' humor. In future work, if we generate longer puzzles, an HTN planner might allow us to combine the benefits of our backwards-chaining causal reasoning and the higher-level structure given by the grammar-based puzzle generators, since it allows an author to specify hierarchical decomposition within the planning process (*e.g.*, that a particular puzzle should be solved in three specific phases).

V. Conclusions

Smart terrain causality chains (STCCs) are a technique for procedurally generating adventure-game puzzles by generating adventure-game puzzle *solutions* from a set of smart-terrain objects, and then placing those objects into the scene. Smart-terrain objects are annotated with effects they can trigger or have triggered, but are not directly linked to other objects in a classic adventure-game style, so can have multiple uses—the note on which a code is found can later be burned to produce smoke, for example, without this explicitly being programmed. STCC-based generation can produce multiple puzzles fitting into a common place in a larger story progression, by constraining a specific generated *scene* to end with a specific objective; this allows for increasing replayability by making puzzles procedurally generated without having to solve the full problem of story generation for adventure games.

Our experimental game *Space Dust* showcases this replayability in an admittedly somewhat extreme way, by designing a game where the typical player has to replay the same scenario approximately ten times before they're able to successfully beat it. Each replay has a newly generated puzzle with the same objective (escaping from the ship), and the player retries until they successfully escape. Our preliminary experiments showed that players found this engaging, despite it being the "same" puzzle played repeatedly, if viewed from the perspective of story progression. This opens up a new space of possible procedural-puzzle-based design tropes in adventure games, in addition to potentially increasing the replayability of more traditional, story-driven adventure games by varying the puzzles on subsequent playthroughs.

VI. Future Work

There are three main areas of future work: object placement, action/effects inference, and broadening the scope of procedural generation.

A. Object placement

Object placement is currently not automatic: an STCC provides a list of objects to be entered into a scene, but not *where* they should be placed. In some puzzles this can add another dimension of puzzle solving, since placement can

make a significant difference to difficulty or interestingness. We currently use some heuristics to avoid particularly bad solutions; for example, objects that can effect each other at a distance are placed at initial positions out of range, so they don't immediately start interacting without the player having done anything. In addition, other objects' placements are hard-coded, so the toilet for example must be on the floor and against a wall.

Improvements would involve generalizing these heuristics to a set of constraints on placement, which could then be passed off to a separate placement routine, based on optimization or constraint-solving. Doing so would also require semantic information about the environment [9] and/or the game mechanics [10], [11], whereas currently only information about the semantics of the smart-terrain objects themselves is maintained.

B. Action/effects inference

In the current version of *Space Dust*, all cause and effect information pertaining to smart terrain must be manually entered by the author. For instance, the author was required to enter that if a fire sprinkler detects fire, it will emit water.

Automating this process is possible in a number of ways. The cause/effects information could be more closely linked with the physics system by implementing a qualitative-spatial-reasoning system [3], or even something more ambitious, such as commonsense reasoning about objects that can determine their potential uses [12]. An alternate solution that would allow us to retain the current simple runtime would be to run offline analysis when a new object is created, to automatically extract some of its properties. For example, newly created smart-terrain objects could be placed into a virtual analysis box which applies various physical forces to it and then automatically records into the actions table any effects that are observed.

C. Broader procedural generation

Although STCC-based puzzle generation generates a key portion of adventure games—the puzzles—that is still a long way away from procedurally generating entire adventure games. The primary difference between our approach here and a fully procedural adventure game is that we make no attempt to generate the environments, stories, or items themselves, but limit ourselves to generating puzzles *using* items that are then placed *in* environments, and are situated *within* a story. Our generation process runs once those three parts of the game are already defined.

Combining puzzle generation with Dormans's research on procedural missions and spaces [5] is one possible approach, using each system as a separate layer in the generation process. Another avenue would be to move to a richer model of dependencies than our STRIPS-like model of preconditions and effects; planners that can handle more complex planning formalisms together with smart objects [13] may open up new possibilities. More generally, theories of causality often make finer distinctions among kinds of causal relations, and which ones matter for which purposes [14]. In addition, ours is a simplified account of how causally related game elements structure gameplay. In our model, puzzle games consist of a player traversing/connecting causality chains to solve puzzles; but the relationship between gameplay and causal chains in general can be considerably more complex [15].

Finally, items themselves could be generated using procedural item generation, a form of procedural content generation that would simultaneously generate the graphical manifestation of game-world content (as is currently done with, *e.g.*, rocks, trees, and textures), *plus* that content's uses within a puzzle.

Acknowledgments

Thanks to Julian Togelius for helpful discussions at various stages of the project, and to an anonymous reviewer for a number of suggestions that improved the paper.

References

[1] N. Kirby, "Solving the right problem," in *AI Game Programming Wisdom*, S. Rabin, Ed. Cengage, 2002, pp. 21–28.

[2] M. Kallmann and D. Thalmann, "Modeling objects for interaction tasks," in *Proceedings of the 9th Eurographics Workshop on Animation and Simulation*, 1998, pp. 73–86.

[3] M. Cavazza, S. Hartley, J.-L. Lugrin, and M. Le Bras, "Qualitative physics in virtual environments," in *Proceedings of the 9th International Conference on Intelligent User Interfaces*, 2004, pp. 54–61.

[4] R. Fikes and N. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3–4, pp. 189–208, 1971.

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

[6] C. Fernández-Vara and A. Thomson, "Procedural generation of narrative puzzles in adventure games: The puzzle-dice system," in *Proceedings of the Workshop on Procedural Content Generation at FDG 2012*, 2012, pp. 18–23.

[7] F. Harrell, "Shades of computational evocation and meaning: The GRIOT system and improvisational poetry generation," in *Proceedings of the 2005 Digital Arts and Culture Conference*, 2005, pp. 133–143.

[8] D. Olsen and M. Mateas, "Beep! beep! boom!: Towards a planning model of Coyote and Road Runner cartoons," in *Proceedings of the 4th International Conference on the Foundations of Digital Games*, 2009, pp. 145–152.

[9] T. Tutenel, R. M. Smelik, R. Bidarra, and K. J. de Kraker, "Using semantics to improve the design of game worlds," in *Proceedings of the 5th Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2009, pp. 100–105.

[10] A. M. Smith, M. J. Nelson, and M. Mateas, "Ludocore: A logical game engine for modeling videogames," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 2010, pp. 91–98.

[11] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović, "A case study of expressively constrainable level design automation tools for a puzzle game," in *Proceedings of the 7th International Conference on the Foundations of Digital Games*, 2012, pp. 156–163.

[12] J.-L. Lugrin and M. Cavazza, "Making sense of virtual environments: Action representation, grounding, and common sense," in *Proceedings of the 12th International Conference on Intelligent User Interfaces*, 2007, pp. 225–234.

[13] T. Abaci, J. Ciger, and D. Thalmann, "Planning with smart objects," in *Proceedings of the 13th International Conference in Central Europe on Computer Graphics*, 2005, pp. 25–28.

[14] J. Pearl, *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.

[15] M. Eskelinen, "The gaming situation," *Game Studies*, vol. 1, no. 1, 2001.