Scenario Generation For Emergency Rescue Training Games

Kenneth Hullett Michael Mateas
Expressive Intelligence Studio
University of California, Santa Cruz
Santa Cruz, CA 95064
{khullett, michaelm}@soe.ucsc.edu

ABSTRACT
Procedural methods have long been used for generation of art assets, but procedural generation of scenarios has lagged behind. In particular, training games for emergency rescue workers would benefit from procedural scenario generation guided by pedagogical goals. In such a game, users could select what skills they wish to train for, and the system would generate a unique level containing the elements necessary to train those skills. In this paper, we present a system that uses HTN planning to generate collapsed structure training scenarios that are both internally consistent and allow the user to train for the desired goals.

Categories and Subject Descriptors
I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems - *Games* I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods – *Representations (procedural and rule-based)* K.8 [**Personal Computing**]: Games

General Terms
Algorithms, Design

Keywords
Procedural Level Generation, Serious Games, Qualitative Reasoning

1. INTRODUCTION
In the US, one rescue worker is lost for every two victims rescued in collapsed structure situations; following Sept. 11, 2001, this ratio fell to 1:1[1]. Limited physical facilities are available for training rescue workers in collapsed structure rescue techniques. Furthermore, emergency response teams are composed of volunteers who respond only when their team is activated. They may be geographically dispersed and have day jobs in other fields. Training games could improve emergency rescue worker preparedness by increasing training opportunities.

The Disaster Assistance and Rescue Team (DART) at NASA Ames Research Center has expressed interest in developing a training game for collapse structure rescue training. They host a unique annual training class on collapsed structure rescue techniques that is attended by rescue workers from around the world. The workers train on essential skills such as: evaluating hazards by observing the cracks and deformations in a structure, techniques for breaching collapsed structures, determining the appropriate type of shore to use and how to construct it, and identifying areas where victims could be trapped and how to extract them safely.

This training class is only held once a year and has a limited enrollment of 40 students, so a training game that rescue workers could play on their own PC in their leisure time would allow greater dissemination of collapsed structure rescue techniques. Students planning to attend the class could get a preview of what they will be learning, allowing trainers to focus more on hands-on physical training. Former students could review what they've learned, and students who are unable to attend the class could get training they otherwise would not.

While hand-authored scenarios are useful for training, they lack replayability. Once a user has played a scenario it is no longer useful for training. It follow that in order to provide the most effective training possible, a training game should include dynamically[2] generated scenarios. Such a system should generate interesting, playable scenarios with a high degree of internal consistency. Furthermore, the system should generate scenarios that provide training to satisfy given pedagogical goals. The trainees should be able to specify which skills they want to train, and the system should generate a unique scenario that contains the appropriate challenges.

In this paper we describe a generation system that applies the established technique of hierarchical task network (HTN) planning to the domain. Input to the system is an abstract representation of an intact structure and a set of training goals. The planner's domain model is simple naïve physics knowledge; deformation operators transform the intact structure to a collapsed structure containing the elements necessary to train for the selected pedagogical goals while maintaining a believable consistency between the various damaged sections of the scenario.

[1] Private communication, Robert Dolci, Director of NASA Ames DART

[2] In this paper, we follow the game industry convention of using the term "procedural" to refer to content that is dynamically constructed through computational methods, rather than hand-authored by human beings

ICFDG 2009, April 26–30, 2009, Orlando, FL, USA.

2. RELATED WORK

2.1 Procedural Methods

Procedural methods have been used in computer graphics for decades. The earliest examples involve the use of L-Systems to model plants [11]. Later approaches built on this, developing context-sensitive grammars for the modeling of buildings [7] and cities [4]. These techniques have been incorporated into commercial products used by game developers [10, 16].

Some efforts have been made to develop procedural methods for generating game levels, particularly for 2D game worlds. Most of the existing examples have relied heavily on randomness. The earliest example is Rogue and its various clones (the so-called Roguelikes). Each time the player moves to a new section of the dungeon, the game randomly generates a unique level. Rooms of random size are generated, populated randomly with treasures and monsters, and connected by hallways [13]. While unique, the unconstrained randomness limits the ability to guarantee that generated levels contain specific properties.

In the late 1990s Blizzard used a similar approach to create levels in Diablo and Diablo II. Similar to Rogue, everything from the layout of the levels to the placement of monsters and treasures was random. There was some scaling for difficulty as the game progressed and different art assets were used to give different game areas unique visual styles. While commercially successful, these games did not represent a great leap forward in procedural level generation.

The approaches used in the Roguelikes and Diablo series are not adaptable to the emergency rescue training domain. The demand for believable physical consistency means we cannot generate scenarios by randomly placing features. For wexample, in a believable collapsed structure, heavily damaged sections should not be adjacent to undamaged sections. Elements like fire or flooding should spread from sources in a realistic manner.

Generation of 3D levels adds greater challenges to those already present in the 2D examples. Roden et all proposed techniques for generating 3D game levels [12]. Their system consisted of placing pre-designed set-pieces randomly and connecting them with passages. This is an extension of the established 2D techniques into 3D, with additional constraints of vertical spacing being satisfied. However, no commercial game has employed generated 3D levels.

Even though the proposed emergency rescue training game would use a 3D environment, this technique is not adaptable to our needs. Again there is a lack of internal consistency in the generated levels. The constraints we are considering go beyond the vertical spacing of elements. While a set-piece approach could potentially be used, it would be necessary to make sure that incompatible set pieces were not placed near each other, and connections were consistent. With a reasonably large library of set-pieces, the combinatorial number of possible interactions could be intractable.

Nitsche et all developed Charbitat, which featured a game world that adapted to the player [9]. Like previous work, their system employed a high degree of randomness and placement of pre-designed segments. Their game world is tile based, and a new area is created when the player first moves to a previously unvisited tile. The game generates both the landscape and places objects in it. The terrain generation is random but constrained at the border with existing tiles so that features match. Objects to be placed on the tile are selected randomly from a set of pre-made objects. Objects are categorized by size, and the selection is constrained to match the desired distribution of one or no large objects, some medium objects, and several small objects.

This approach works well for the outdoor environment of Charbitat, but would not work for the indoor environments of collapsed structure scenarios. Also, while generated tiles could be internally consistent, there is no constraint for consistency across multiple tiles.

All previous approaches to procedural level generation have depended on randomness, though they differ in the degree to which they are constrained. None of the existing approaches could produce the level of internal physical consistency and believability we would desire for an emergency rescue training game.

2.2 Serious Games

The field of serious games has been described as “the use of electronic games technologies and methodologies for primary purposes other than entertainment” [15]. This includes educational games, advertising games, and military simulations, among others.

Emergency rescue training would seem like a natural application for serious games, but only a handful of examples exist. Schurr et all developed DEFACTO as a training tool for incident commanders for large scale disasters [14]. DEFACTO focuses only on the high-level management of rescue operations, rather than the details of rescue operations. The training game we envision would involve rescue operation management, but would be more focused on individual rescue workers and the hazards they face in doing their jobs.

Hazmat: Hotzone is a project developed at the Entertainment Technology Center at Carnegie-Mellon University. This has more of an individual focus than DEFACTO, allowing firefighters to train on hazardous materials scenarios in a multiplayer, 3D game environment. This is closer to what we envision, but is specific to the domain of hazardous material handling. The system we propose is general enough that it could be extended to a wide variety of emergency rescue domains.

A commercial application in this area is Flame-Sim, a training game for firefighters [1]. Like Hazmat: Hotzone, it allows multiple trainees to work together in a networked 3D game environment. The only available scenario type at this time is single-family residences.

These examples all rely on hand-designed scenarios. Hazmat: Hotzone and Flame-Sim both come with scenario editing tools that allow users to create custom scenarios. While the scenario editors streamline the process, the scenario generation system we propose would generate a unique scenario every time with minimal user input.

3. SCENARIO GENERATION METHOD

3.1 Qualitative Modeling

Our approach is essentially an application of qualitative physical reasoning. In qualitative modeling, systems engage in symbolic, qualitative reasoning about physical systems [2, 5]. Instead of

using a system of differential equations to model the relationships between variables in a physical system, a qualitative modeling approach employs a common sense understanding of the relationships between different, discretized quantities in different regions of the system. For example, if modeling a system of pipes, a qualitative approach would model relationships such as "if pressure increases in pipe 7, pressure decreases in pipe 3."

Our scenario generator employs a qualitative model of the damage effects of an earthquake (including indirect effects such as fire or trapping of victims), implemented in an HTN planner, to generate plausible collapsed structure rescue scenarios given pedagogical goals. The planner reasons about how the different parts of a structure will interact during a collapse situation. For example, the planner models that ceilings are supported by walls, so when a wall receives a critical amount of damage, the ceiling is likely to collapse.

3.2 HTN Planning

For the implementation of our system we use the HTN planner Simple Hierarchical Ordered Planner (SHOP) developed by the University of Maryland's Automated Planning group [8]. Given a starting state and a goal state, the planner finds a sequence of operators that achieve the goal state. This is achieved by decomposing the tasks into subtasks, which may in turn be decomposed further until primitive tasks are reached. The human author is responsible for the knowledge engineering work required to specify both methods (tasks) and operators.

We apply HTN planning to our qualitative reasoning-based scenario generation problem using the following conventions:

- The initial state is a description of the uncollapsed structure.
- The goal state is a list of structural features implied by the pedagogical goals selected by the user.
- The methods encode knowledge on what subgoals to pursue to achieve structural goals. For example, a method may model that, if you want a ceiling collapsed, a good way to do that is to critically damage the supporting walls. Damaging the supporting walls would now become subgoals.
- Operators implement the assertion and retraction of structural features and subgoals.

Typically HTN planning is fully deterministic, but to increase variation, we have included a random element in the propagation of damage. We took advantage of a SHOP feature that allows users to call external methods. We wrote a function that generates random numbers and use it to vary the amount of damage applied to structural elements as the damage is propagated throughout the structure.

3.3 Domain

This system is developed specifically for collapsed structure rescue scenarios. The major skills users would be interested in training for are breaching walls, placing shores, and rescuing victims. A typical scenario would involve the following steps:

1. The structure would be analyzed to determine the most likely location that surviving victims might be located.
2. External walls may be shored if needed to prevent further collapse while rescue workers are in the structure.
3. Walls may be breached to reach the building interior if doors are blocked or unusable.
4. Once inside the structure, ceilings and internal walls may need to be shored before proceeding.
5. Internal walls may need to be breached to reach trapped victims.
6. Once the victim is reached, they must be extracted safely.

The scenarios generated by the system should vary on the locations and status of victims, the amount and location of damage to the structure, the degree of collapse, and the parts of the structure affected by fire, if any.

The domain model describes buildings along with the structural state of individual building elements. Table 1 lists the predicates used in the domain model, as well as their arguments, while Figure 1 provides a concrete example of a domain representation.

Table 1. Domain Elements

Type	Characteristics
Wall	Strength, Damage, Elements contained
Room	Bordering walls, Elements contained
Ceiling	Strength, Damage, Supporting walls
Victim	Status
Fire Source	Damage
Door	Status

```
(wall w1 0.5 0.0)
(victim v1 injured)
(room r1 v1)
(borders r1 (w1 w17 w18 w19 w20))
(fire-source g2 0.0)
(contains w20 g2)
```

Figure 1: Example domain code

3.4 Methods

Methods are used to decompose tasks into other methods, and eventually primitive operations. Methods have preconditions that that must be satisfied in order to be executed. Since these preconditions may contain variables with many possible bindings, the system can generate multiple plans. Figure 3 shows an example method.

```
(:method (damage-walls ?magnitude (?wall
. ?walls))
       ()
       ((damage-wall ?wall ?magnitude)
(propagate-damage ?wall) (damage-walls
?magnitude ?walls)))


(:method (propagate-damage ?wall)
       ((wall ?wall ?strength ?damage)
(call >= ?damage 0.7))
       ((collapse-ceiling ?wall)
(ignite-fire-source ?wall) (damage-
adjacent-walls ?wall)))
```

Figure 3: Example methods

Figure 3 shows 2 methods. The first, damage-walls, applies a given amount of damage to a list of walls. It has no preconditions. It does three things: First, it calls a primitive operator to directly change the damage level of the wall. It then calls the propagate-damage method on the wall being processed. Finally it calls itself recursively with the remaining portion of the list.

The propagate-damage method spreads the damage done to a wall to the elements that depend on it. If the damage level is greater that the specified amount, it calls 2 methods: one that will determine if the ceiling supported by the wall should collapse, and one that determines if a fire source (i.e., gas line) contained in the wall should start a fire.

3.5 Operators

In the SHOP planning system, operators are the primitive-level tasks and the only ones that can make changes to the state. Like STRIPS-style operators, SHOP operators can make new assertions or remove existing ones. An operator consists of 3 parts: a precondition, a list of assertions to remove, and a list of assertions to add.

```
(:operator (!op-damage-wall ?magnitude
?wall)
       ((wall ?wall ?strength ?damage))
       ((wall ?wall ?strength ?damage))
       ((wall ?wall ?strength
?magnitude)))


(:operator (!op-start-fire ?wall)
       ()
       ((forall (?room) ((borders ?room
?walls) (call Member ?wall ?walls) (room
?room nil)) ((room ?room nil))))
       ((forall (?room) ((borders ?room
?walls) (call Member ?wall ?walls))
((room ?room fire)))))
```

Figure 4: Example operators

Figure 4 shows 2 example operators. The first changes the damage level of a wall to a specified level. It has a precondition that a wall exists with the given name. Stating this precondition also has the effect of binding the variables '?strength' and '?damage'. The second part removes the existing assertion, while the third re-asserts the wall with the new damage level.

The second example causes fires to start in the rooms adjoining a wall. This operator would be called by method that has already determined that the given wall contains a fire source that is sufficiently damaged to start a fire, so the operator does not need to check any preconditions. The operator checks all rooms adjacent to the wall and if any currently does not contain fire it asserts a fire condition.

3.6 Goals

The system takes the input set of goals and finds an appropriate propagation of damage to satisfy them in the scenario. The method "add-new-goals" processes the existing goals and asserts new goals to create the desired condition in a specific location. Figure 2 shows a fragment of the add-new-goals method that handles the case of satisfying the goal for a room to contain fire.

The code fragment in figure 2 codifies the following logic: The goal to start a fire in a room is satisfiable if a fire-source (i.e., a gas line) exists in one of the surrounding walls. If that case fails, the planner will select a room with an adjoining wall and assert a goal to start a fire in that room. If this eventually succeeds, the operator to start a fire will propagate the damage through the structure, satisfying the original goal.

```
(:method (add-new-goals)
    ;; precondition for case 1
    ((goal (room ?room fire)) (borders ?room
?walls) (contains ?wall ?fire-source) (fire-
source ?fire-source ?damage) (call Member
?wall ?walls))
    ;; actions to take in case 1
    ((!!remove (goal (room ?room fire)))
(!!assert (goal (fire-source ?fire-source
1.0))) (add-new-goals))

    ;; precondition for case 2
    ((goal (room ?room fire)) (borders ?room
?walls) (call Member ?wall ?walls) (borders
?room2 ?walls2) (call Member ?wall ?walls2))
    ;; actions to take in case 2
    ((!!remove (goal (room ?room fire)))
(!!assert (goal (room ?room2 fire))) (add-
new-goals))
```

Figure 2: Goal processing code

There are multiple ways to satisfy these goals, creating variability in the plans generated. In the first case, the planner could select from any fire source if there are multiple ones in the walls surrounding a room. If the planner falls to the second case, the fire could be started in any adjacent room that does contain a fire source.

There are 9 goals currently implemented in the system, though some are subgoals of others. The high level goals are:

1. Fire, either in a particular room or if no room is specified, anywhere in the structure
2. A victim is inaccessible. This can be specified multiple times to increase the number of inaccessible victims

3. A room is blocked, and therefore only accessible by breaching a wall
4. A ceiling has completely or partially collapsed.
5. An earthquake of a given magnitude has occurred.

4. RESULTS

We tested the scenario generator with three examples: a small, medium, and large structure. We attempted to generate all possible scenarios for each example, but the large number of possible variations caused the system to run out of memory before generating all possible plans. Even with the Java heap size increased to 1GB, we would run out of memory on the medium and large examples. We were still able to generate a large number of variants by limiting the number of plans we asked SHOP to generate. Table 2 shows a breakdown of the size of each example and the results.

We also developed a GUI tool for creating and visualizing scenarios. For this we used the python programming language and the wxWidgets library. The tool allows users to draw walls and rooms and set up other scenario elements. When the planner output is loaded into the viewer, the damage levels are visualized by the amount of stippling on the lines.

Table 2. Example descriptions

Type	Size	# of Goals	# of Plans	Average Plan Length
Small	7 Walls, 2 Rooms, 2 Other	2	324	~30
Medium	12 Walls, 4 Rooms, 3 Other	3	>2500	~60
Large	30 Walls, 7 Rooms, 11 Other	5	>600	~100

Figure 5 shows the initial state of the small example, while figure 7 shows a fragment of the same information as a SHOP plan. The domain consists of 7 walls that form 2 rooms. The goals are for a fire in room 1 and for the ceiling to collapse. Figure 8 shows fragments of the final state resulting from one of the plans, fragments of which are shown in figure 9. To satisfy the goal of starting a fire, the system determines that the wall containing the fire source must be damaged, and in order to do so an earthquake of a certain magnitude must be applied to the structure. The damage from this earthquake is then propagated throughout the structure, resulting in a ceiling collapse that satisfies the second goal.

Given the same goals and initial state, there is limited potential for scenario variation given the size of the small example. But even with this small size, SHOP generates 324 plans for this example. Many of these plans result in the same scenario but perform the operations in a different order. Others differ in the amount of damage applied to parts of the structure.

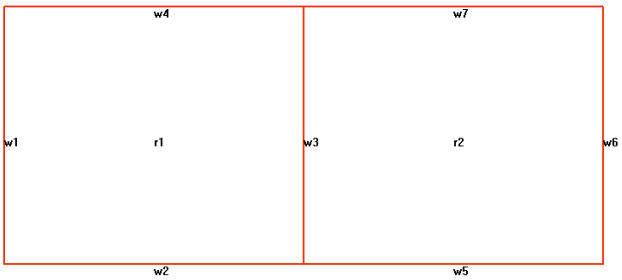

Figure 5: Small example, initial state

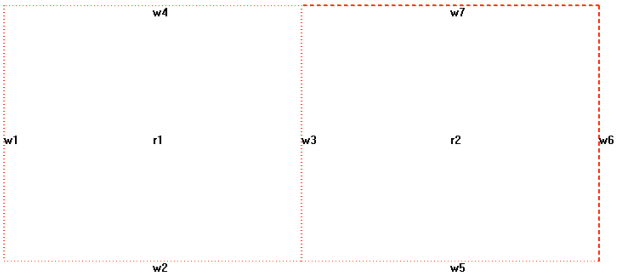

Figure 6: Small example, final state

```
(wall w1 0.5 0.0)
(wallcoords w1 (50 50 50 350))
…
(room r1 v1)
(borders r1 (w1 w2 w3 w4))
(ceiling c1 0.5 0.0)
(supports c1 (w1 w2 w4 w5 w6 w7))
…
(fire-source g1 0.0)
(victim v1 injured)
(contains w1 d1)
(door d1 outside)
(achieve-goals ((room r1 fire) (ceiling
c1 0.5 1.0)))
```

Figure 7: Small example domain fragments

```
(wall w5 0.5 0.7)
(wall w6 0.5 0.5)
…
(ceiling c1 0.5 1.0)
(fire-source g1 1.0)
```

Figure 8: Small example output fragments

```
[ 1 ]   (!!assert (goal (room r1 fire)))
[ 2 ]   (!!assert (goal (ceiling c1 0.5 1.0)))
…
[ 8 ]   (!!assert (goal (earthquake 0.7 (w1 w2 w3 w4))))
…
[ 10 ]  (!op-damage-wall 0.7 w1)
[ 11 ]  (!op-collapse-ceiling c1)
[ 12 ]  (!op-damage-wall 0.7 w2)
…
[ 17 ]  (!op-ignite-fire-source g1)
[ 18 ]  (!op-start-fire w3)
```

Figure 9: Small example plan fragments

For the medium example we show 2 alternate scenarios, Figures 11 and 12. The scenarios differ in terms of which walls are damaged and the amount of damage. The goal of a fire in room 1 is satisfied in both cases by a high degree of damage to wall 3. The system achieves this by initiating an earthquake which damages the main supporting walls 1, 2, 5, 6, 8, 9, 11, and 12. The damage is propagated to the internal walls 3, 4, 7, and 10. In both variants wall 3 is damaged enough to initiate a gas leak, and thus start a fire. The second goal of trapping the victim in room 2 is satisfied by initiating a ceiling collapse, thus both versions also show a high degree of damage to wall 7. The scenarios vary in the amount of damage done to the other interior walls.

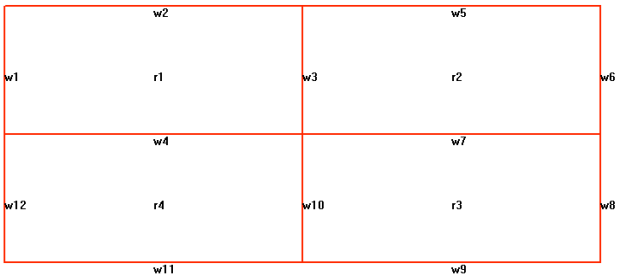

Figure 10: Medium example, initial state

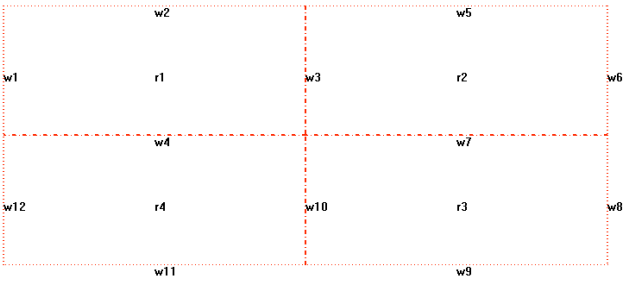

Figure 11: Medium example, final state

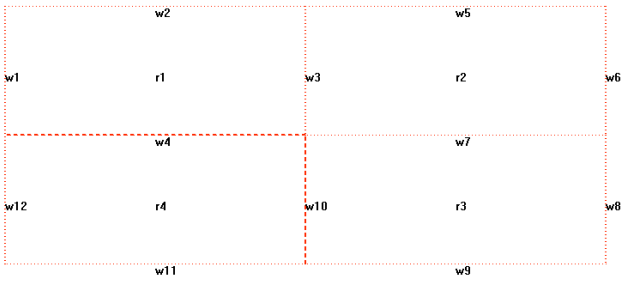

Figure 12: Medium example, alternate final state

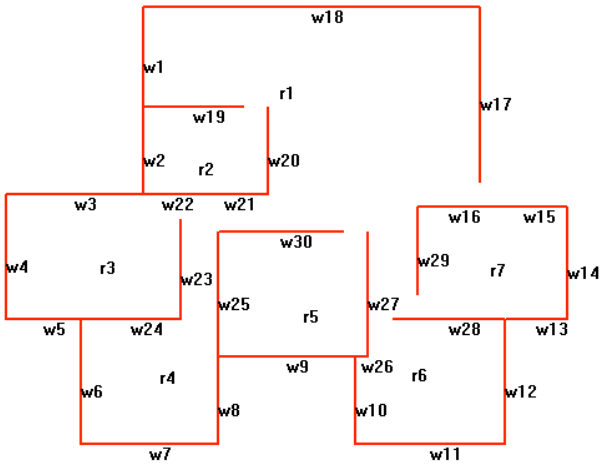

Figure 13: Large example, initial state

In the large example, shown in figures 13 & 14, the system ran out of memory after generating around 600 variants, many fewer than in the medium example. Because the domain in this case is much larger, each plan is significantly larger. On average, the plans generated in the large example contained about 100 steps, compared to the 60 steps in the medium example. Given the difference in plan sizes it is reasonable that the medium example generated about 2500 plans before running out of memory, compared to 600 for the large example.

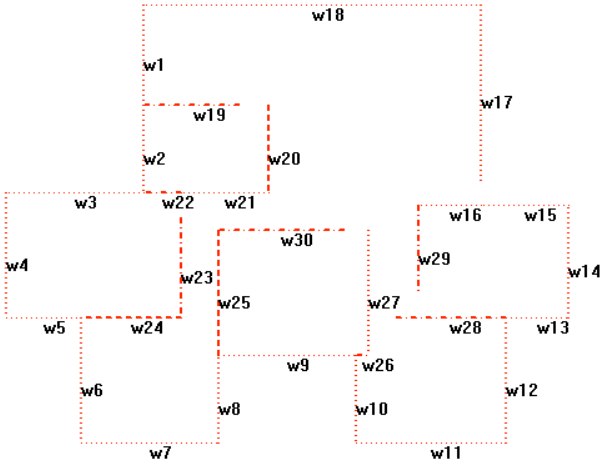

Figure 14: Large example, final state

5. DISCUSSION

The utility of a procedural scenario generation system can be evaluated by 2 measures. First, it should generate a large number of different scenarios, thus showing that the generator is superior to hand made scenarios. Secondly, the variation in the generated scenarios should be superior to the results one would get through random generation. As Rogue and Diablo showed, it is always possible to create a large number of levels by randomly combining elements, but the resulting levels lack internal consistency.

Our results show that our system can generate a large number of different plans. However, some plans result in the same scenario by performing the same steps in a different order. We do know that there is significant variation just because the amount of damage propagated is random. Furthermore, as the size of a scenario grows, the combinatorial number of possible variations grows. The relatively limited medium example contains about 20 elements and generated over 2000 possible variations. From this we can postulate that a scenario with 100 or more elements could potentially generate 10,000 or more different scenarios if given enough time and memory to do so.

We can also create significantly more variation in a scenario by changing the specified goals. We can vary the number of victims or the degree of collapse to create more scenarios from an input structure. With 5 goal types, most of which can be used multiple times in a scenario, there is a large number of possible goals that can be specified for a given input structure. In conjunction with the combinatorially large number of possible scenarios within each set of goals, the number of possible scenarios that can be generated is vast.

To show that this approach is better than the random approaches used in industry, consider the thought experiment of generating collapsed structure scenarios using random generation techniques. A random level generator consists of a library of various types of human designed elements, with a probability distribution for each type of element (e.g. rooms, monsters, and treasures in Roguelikes) that describes the likelihood of drawing a particular element from the library for that type. The level generator proceeds by selecting elements based on its distribution in some specified order. For example, if there were only two element types – rooms and objects – the generator would first select a number of rooms from the room distribution, then select 0 or more objects for each room from the object distribution.

For the collapsed structure domain, we could have a library of set pieces, each of which would be a small playable section of a scenario, e.g., a room with a trapped victim. A random scenario generator would select some of these set pieces and attempt to connect them together to make a scenario. There is no guarantee that the scenario as a whole would be consistent, much less that adjacent set pieces would match. An area containing a fire could be placed next to an area without fire, or highly collapsed sections could be connected by a section with little damage.

The fact that our system models physical constraints such as propagation of damages means that it can produce more believable results than unconstrained selection of elements from a distribution. Thus, our generated scenarios are internally consistent; key elements are changed to satisfy the goals, and the resulting damage is propagated throughout the structure in a believable way.

6. FUTURE WORK

To reach its full potential, the domain needs to be expanded to handle more structural elements. Handling multiple-story buildings would require codifying more chains of causality as collapses propagate from floor to floor. Also, the system could work at a higher level of fidelity, acting on smaller units than walls and rooms. For example, locations within rooms should be considered for victim placement and ceiling collapses, creating scenarios where rescue workers have to remove debris in a room in order to access the victim.

This system could also be expanded to handle a wider variety of scenario types. Besides structures collapsed due to earthquakes, it could consider other types of collapse, such as those resulting from bomb blasts or hurricanes. More types of scenario elements could be added such as hazardous materials or flooding.

6.1 Making a Game

Obvious follow-on work to this project is connecting the planner to a playable game. Such a game could allow the user to play as a rescue worker, breaching walls, shoring up damaged parts of the structure, and locating and rescuing trapped victims. It could also allow the user to play as an operations commander, allocating personnel and resources amongst multiple ongoing rescue operations following a disaster.

Transferring the abstract representation to a 3D game world is a difficult problem. While it's trivial to build a 3D model of a structure in a modeling package like 3D Studio Max, it's difficult to realize the resulting partially collapsed structure. While technologies exist to allow real-time destruction of game objects, they are based on either particle systems or a pre-generated deformation. No technology exists that can perform arbitrary procedural deformations of a model.

One proposed method is to build a 3D model of the structure divided up into modular sections. Each section would have variant models for each of several levels of destruction. Based on the output of the scenario generator, the appropriate variant of each section is added to the model. As long as the sections and their variants are designed to fit together in any combination, the resulting structure will appear consistent. However, to create the potential for an interesting level of variation, the modular sections would have to be very small.

6.2 Procedurally Modeled Buildings

Our system relies on receiving a representation of an intact structure as input. We only show 3 examples, but users who desire even greater variation could create any number of input structures. To further increase variation in the system, procedurally generated structures could be used as input.

Müller et al have developed techniques for generating unique buildings through context sensitive grammars. While their work is primarily concerned with cosmetic elements, it could be adapted to generate usable input structures. The resulting models would have to be converted to our abstract representation and marked up with the necessary domain knowledge.

Such a system would provide a user with a completely unique scenario each time. The user could specify the type of structure desired as well as the goals desired for the scenario. The structure itself would be generated procedurally, and then be transformed

into a partially collapsed structure containing the elements necessary to train on the desired goals.

7. CONCLUSION

In this paper we have described an architecture for the procedural generation of scenarios for emergency rescue training games. We use HTN planning to generate consistent, believable scenarios based on a set of given training goals. The system outputs a large number of variants for each set of inputs, and can generate even more variation by changing the given training goals. The output from this scenario generator could be used as input to a training game to educate rescue workers about collapsed structure rescue techniques.

8. ACKNOWLEDGMENTS

Our thanks to NASA Ames DART, Garage Games, University of Maryland Automated Planning group, UARC, and CITRIS.

9. REFERENCES

[1] FLAME-SIM. http://www.flame-sim.com/features/. Accessed 12/19/2008

[2] Forbus, K., 1996. *Qualitative reasoning*, CRC Handbook of Computer Science.

[3] Gamasutra – Postmortem: Blizzard's Diablo II. http://www.gamasutra.com/view/feature/3124/postmortem_blizzards_diablo_ii.php. Accessed 12/19/2008

[4] Greuter, S., Parker, J., Stewart, N., and Leach, G. 2003. Real-time procedural generation of `pseudo infinite' cities. In Proceedings of the 1st international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia (Melbourne, Australia, February 11 - 14, 2003). GRAPHITE '03. ACM, New York, NY, 87-ff.

[5] Hayes, P. J. 1990. The second naive physics manifesto. In *Readings in Qualitative Reasoning About Physical Systems*, D. S. Weld and J. d. Kleer, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 46-63.

[6] Hazmat: Hotzone. http://www.etc.cmu.edu/projects/hazmat_2005/. Accessed 12/19/2008

[7] Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Van Gool, L. 2006. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers* (Boston, Massachusetts, July 30 - August 03, 2006). SIGGRAPH '06. ACM, New York, NY, 614-623.

[8] Nau, D. S., Cao, Y., Lotem, A., and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the Sixteenth international Joint Conference on Artificial intelligence* (July 31 - August 06, 1999). T. Dean, Ed. Morgan Kaufmann Publishers, San Francisco, CA, 968-975.

[9] Nitsche, Ashmore, Hankinson, Fitzpatrick, Kelly, and Margenau, 'Designing Procedural Game Spaces: A Case Study', in Proceedings of FuturePlay 2006 (London, Ontario October 10-12, 2006)

[10] Procedural Inc. – 3D Modeling Software for Urban Environments. http://www.procedural.com/. Accessed 12/19/2008

[11] Prusinkiewicz, P., Lindenmayer, A., Hanan, J. S., et al. The Algorithmic Beauty of Plants. Springer, New York, 1990

[12] Roden and Parberry, 'Procedural Level Generation', in Game Programming Gems 5, Charles River Media, pp. 579-588, March 2005.

[13] RogueBasin. http://roguebasin.roguelikedevelopment.org/index.php?title=Main_Page. Accessed 12/19/2008

[14] Schurr, N., Patil, P., Pighin, F., and Tambe, M. 2006. Using multiagent teams to improve the training of incident commanders. In *Proceedings of the Fifth international Joint Conference on Autonomous Agents and Multiagent Systems* (Hakodate, Japan, May 08 - 12, 2006). AAMAS '06. ACM, New York, NY, 1490-1497.

[15] Serious Games Institute – About Us. http://www.seriousgamesinstitute.co.uk/about.aspx?item=2. Accessed 12/19/2008

[16] SpeedTree IDV, Inc. http://www.speedtree.com/. Accessed 12/19/2008