Game-O-Matic: Generating Videogames that Represent Ideas

Mike Treanor, Bryan Blackford, Michael Mateas and Ian Bogost
University of California Santa Cruz, Expressive Intelligence Studio
Georgia Institute of Technology
{mtreanor, bblackfo, michaelm}@soe.ucsc.edu
ibogost@gatech.edu

ABSTRACT
In this paper, we describe *Game-O-Matic*, a videogame authoring tool and generator that creates games that represent ideas. Through using a simple concept map input system, networks of nouns connected by verbs, *Game-O-Matic* is able to assemble simple arcade style game mechanics into videogames that represent the ideas represented in the concept map. Inspired by a view that videogames convey messages through their mechanics, *Game-O-Matic* makes use of the rhetorical affordances of explicitly defined abstract gameplay patterns, which we call micro-rhetorics. This paper explains how *Game-O-Matic* uses the concept map input to select appropriate abstract patterns of gameplay and then how these mash ups of patterns are shaped into coherent playable games that can be said to represent the user's intent.

Categories and Subject Descriptors
K.8.0 [Personal Computing]: General – Games. I.2.4 [Artificial Intelligence]: Knowledge Representation Formalism and Methods – Representations (procedural and rule-based).

General Terms
Design, Theory.

Keywords
Procedural content generation, game generation, game design, procedural rhetoric.

1. INTRODUCTION
Game-O-Matic, a Knight News Challenge funded collaboration between the Georgia Institute of Technology and the University of California at Santa Cruz [3], is a piece of software that is able to generate simple games based on input that lists objects, actors and their relationships. *Game-O-Matic* was conceived to address a problem facing newsgames: journalism has been hesitant to adopt the form because news organizations don't have the resources to train or hire game designers and integrate game development into their workflow. The difficult processes of game design and programming are automated so that the journalist need only conceive of their stories in a way that fits the input.

As much as possible, *Game-O-Matic* strives to create games that represent ideas through their processes. Bogost argues that the unique meaning-making strategy of games is "procedural rhetoric," the act of making an expression or argument through a game's processes or rules [2]. In this way, *Game-O-Matic* operationalizes a theory of procedural rhetoric for simple arcade-style games and enables the rapid creation of editorial newsgames.

Previous work performing deep analysis of classic arcade games has enabled us to create a framework for "proceduralist readings" that describe how games rules, dynamics and instantial assets interact to represent ideas to a player. Of course, interpretation is inherently subjective, and no single media artifact can be said to represent any one thing, so a proceduralist reading strives to be a well formed, hierarchical argument for why a unit of gameplay might be said to represent an idea. We call this explicit hierarchical structure a *meaning derivation* [12]. For a proceduralist reading, an interpretation is only as strong as the meaning derivation that supports it. Thus we strive to create games with as convincing a meaning derivation as possible.

Users of *Game-O-Matic* input actors (nodes) and the verbs that describe the relationships between them (arrows between nodes). This concept map structure is not an arbitrary interface for entering news stories. It was carefully chosen because it establishes a paradigm for reporting that values systems over stories, and why/how over who/what/where/when. Static details are best suited for traditional media, while the computational nature of videogames excels at depicting dynamics [1].

Game-O-Matic generates games using a feed forward pipeline with several points of generativity (figure 2). To start with, verbs are entered to describe the relationship between entities and are mapped to patterns of game mechanics (authored for the component based PushButton game engine [6]). These mechanics have *rhetorical affordances* that, when thematic elements are applied, produce what we call *micro-rhetorics*. A micro-rhetoric is a representational segment of gameplay within a videogame that could be supported by a convincing meaning derivation. Often, micro-rhetorics are combined to form the complete rhetoric of a game. Previous work analyzing Activision's *Kaboom* was the first detailed analysis of micro-rhetorics and the rhetorical affordances of game mechanics [10].

Next, these bundles of micro-rhetorics are analyzed by *Game-O-Matic*'s recipe system to discover opportunities to make the partially formed game (a bundle of micro-rhetorics) into more coherent experiences. A recipe is a set of weighted precondition predicates and modifications for the partially formed game. *Game-O-Matic* makes use of three types of recipes: win (what win condition would make the most sense given the bundle of micro-rhetorics), lose (what lose condition would make the most sense given the bundle of micro-rhetorics) and structure (how might the entities be placed on the screen to afford interesting gameplay). Appropriate recipes are selected by evaluating the definition predicates to score the applicability of the recipe. Once the recipes are selected and their recommendations are applied, all appropriate "patches," selected by preconditions, are applied. These clean up any loose ends created by conflicts between recipes.

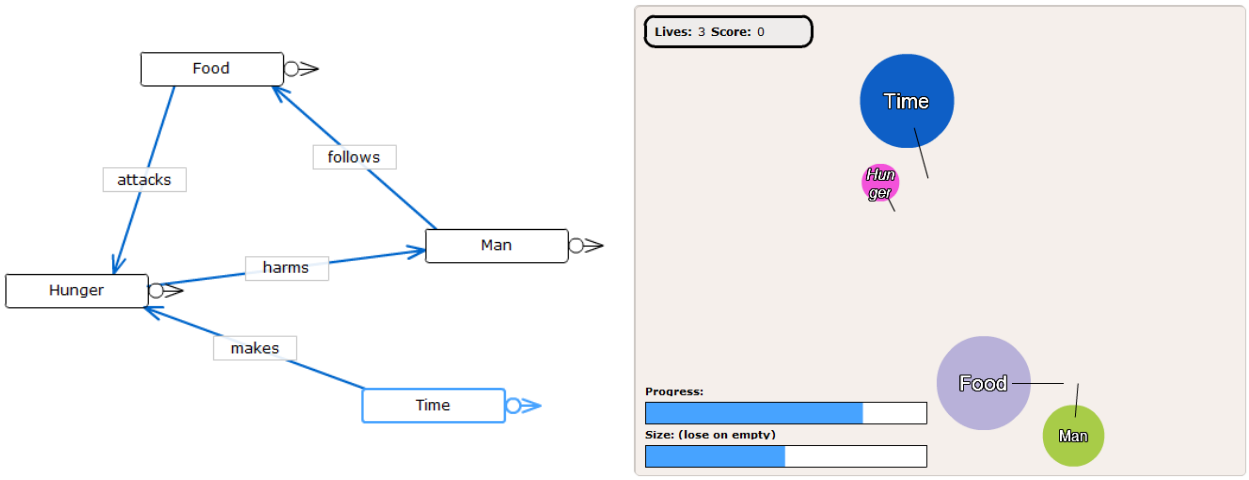

Figure 1. *Game-O-Matic's* concept map input (left) and a game that was generated to represent the ideas in the concept map.

The left side of figure 1 shows concept map input for *Game-O-Matic* that represents a situation where Time makes Hunger, Man needs Food, Food attacks Hunger and Man follows Food. On the right is a game generated from that map where the player controls Man trying to avoid collision with the Hunger that Time is shooting towards it. If Hunger collides with Man, Man shrinks. The player wins if he survives for an amount of time, and loses if he gets too small. To win this game the player maneuvers Man behind the Food which destroys the Hunger.

Game-O-Matic utilizes hand authored libraries of micro-rhetorics and recipes to create games that reasonably represent specified relationships between objects. It is our aim for *Game-O-Matic* that it be able to generate games where an interpreter could create a convincing meaning derivation that matches the concept map input. This paper describes the details of how *Game-O-Matic* preserves the intended representation from the user's input while maintaining a high degree of variability in the games it generates.

2. RELATED WORK

Generating videogames is a relatively new field with few examples of combining basic rules into playable games. All of these works (including *Game-O-Matic*) create single-screen arcade-style games. The first two related works are focused on a formal specification that reliably produces playable abstract games, while the third provides a method for creating games with a sensible representation, the final paper is an earlier work on automatic game design.

Variations Forever is a game generator developed by Adam Smith and Michael Mateas [8] which generates games using a combination of answer set rules that define the game's ruleset, space, controls, etc. Answer set programming offers random selection over a range of values, such as position or who is the player, yielding generative space. The generated games can be constrained to fall within certain mini-genres by adding to the rules. Variations Forever's games are selected from the solutions to the set of rules. *Game-O-Matic* also uses a rule-based approach, but the variations are selected based on a score against the narrative mapping. *Game-O-Matic*'s preconditions are not strict in order to provide greater variability between games or flexibility in accommodating unusual narrative maps. This may occasionally result in unwinnable games, but the genre of newsgames contains several examples of unwinnable games such as Gonzalo Frasca's *Kabul Kaboom*. These games may be interpreted as expressing a rhetoric of failure [11]. This type of game can be well suited for expressing certain ideas, so we allow for a small percentage (less than 5%) of unwinnable games, which will typically only be produced after generating several winnable games.

The ANGELINA system, developed by Michael Cook and Simon Colton [4], like Variations Forever, generates abstract arcade games. The games are generated in 3 parts: the map, the layout (of entities), and the ruleset (collisions, movement types, time limit, and score limit). Each part is evolved through many generations with separate fitness functions, and occasionally testing the fitness of the parts together. The map part is a maze-like arrangement of bricks on the grid that can impede or encase entities. This is a

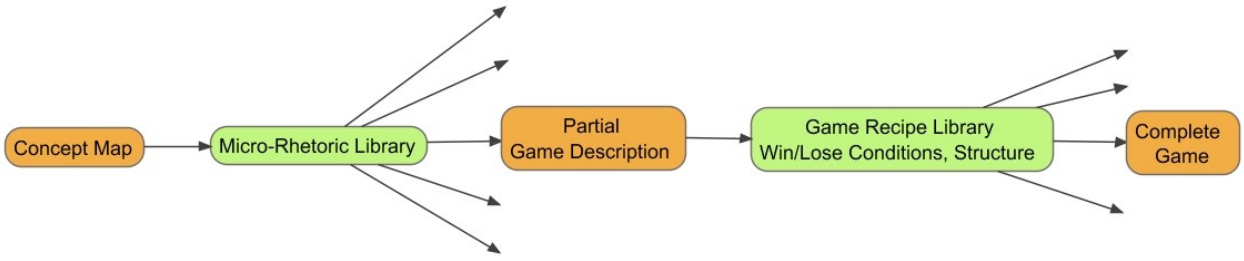

Figure 2. An overall architecture diagram that shows *Game-O-Matic's* main points of generativity.

component which *Game-O-Matic* currently lacks. As of now, we rely on constraints to the movement of entities, and the only environmental factor is the border of the screen. Although sometimes entities will have components that cause them to behave like walls. Still, the genetic algorithms used in ANGELINA set it well apart from *Game-O-Matic*; the independent fitness function produce variations in the games, and the virtual playouts of the game's combined parts ensure playable games.

The work presented in this paper is most similar to Nelson and Mateas' work on generating skins for games with very simple mechanics [7]. Given a verb and a noun, like "shoot a duck," Nelson and Mateas used a common sense knowledge base to find an appropriate skin to apply to A and an appropriate game mechanic from their library of game mechanics. For example, the system would select a game where the player controls a set of cross hairs and tries and click on a frantically moving around duck. The system chose this mechanic and skin as it was more appropriate to select than something like shooting a piano would be, as a bird is something that can be shot. As will be described below, *Game-O-Matic* relies on the user to supply sensible relationships and does not prevent strange pairings, but, it is able to combine multiple game patterns. However, using a similar approach to and putting restrains on what valid verbs are could be an interesting future direction.

Julian Togelius and Jürgen Schmidhuber's foundational work on generating videogame rules evolves games using a fitness function built on theories of fun and learning [9]. The generator needs to play the games to evaluate the fitness function, so controllers are evolved as well. By changing a few parameters regarding the consequences of collision, setup and behavior of entities, and win/lose conditions, their system can generate *Pac-Man*-like games. *Game-O-Matic* avoids the need for evolving games by starting from a user supplied concept map and mapping those concepts to representative game mechanics, the permutations of which are scored to fit various videogame tropes, and the result is typically playable.

3. FROM CONCEPT MAP TO MICRO-RHETORICS

3.1 Concept Map Input

Users of *Game-O-Matic* input their desired stories in the form of a concept map: networks of nodes and arrows where the nodes contain actors in the story (nouns) and the arrows are labeled with their relationships (verbs). This approach was arrived at after analyzing classic arcade games with the intention of describing the rhetorical relationships between game entities and finding that the best way to describe what was happening on the screen was in terms of these sort of simple relationships: "Space ship attacks Invaders" (*Space Invaders*) or "Buckets protect World" (*Kaboom*) [10].

It is important to note that accepting input in this form implies no chronology; any that does appear in the generated games arises from the dynamics of the micro-rhetorics simulated. Also, all relationships are transitive and only involve two nouns. This limitation was introduced to maximize accessibility for non-technical users and is not a limitation of the approach.

The generator creates games treating the nouns as game entities and uses the verbs to determine the mechanics that should be applied to the entities. Currently, *Game-O-Matic* supports the following verbs: arrests, attacks, avoids, carries, collects, deflects, follows, gets, grows, harms, influences, makes, needs, obstructs, prevents, wastes and watches.

3.2 Micro-Rhetorics

Every valid verb that the user enters into a concept map has a corresponding set of micro-rhetorics that can be selected to represent two entities with that relationship. A micro-rhetoric is a representational segment of gameplay within a videogame. For example, a possible micro-rhetoric for "A needs B" could be A and B being represented as sprites, and the game mechanic that unless A is colliding with B, A will constantly shrink. *Game-O-Matic* primarily makes use of metaphorical representation, such as entity A colliding with entity B causing B to be removed representing A eating B, but micro-rhetorics can also take the form of more accurate simulations, such as the mechanics that deal with pollution from industrial zones in *SimCity 4*.

As previous work has shown, purely abstract game mechanics cannot be said to represent concretely. How a set of abstract mechanics are understood is determined by the interpreter's beliefs about the depicted objects participating in the mechanics. For example, if A was a picture of a shoe, and B was a picture of an ant, it is likely that an interpreter would understand a collision between the shoe and the ant, followed by the removal of the ant as the shoe killing the ant. Where if A was a bunny and B was a carrot, the interpreter would understand a collision between the bunny and the carrot followed by the removal of the carrot as the bunny eating the carrot.

The micro-rhetorics that are mapped to *Game-O-Matic*'s verbs are all said to have the verb as a rhetorical affordance. Rhetorical affordances are the opportunities for representation made available by the rules that govern the relationship between objects and processes in a system. In order for *Game-O-Matic* to reliably generate games that represent the input verbs, the user must input actors that can be reasonably understood to be related by the verb on the arrow between them.

Micro-rhetorics are represented by abstract entities and the game mechanics that should involve them. As a high level example, a micro-rhetoric for "harms" could be that A spawns a shape that moves toward B, and when that shape collides with B, B shrinks.

*Game-O-Matic*s's system of mechanics is based on the highly modular component-based framework of the PushButton Engine (PBE), a Flash game engine [6]. PushButton is able to add a behavior to an entity by simply declaring that the entity should use a component with various parameters. Example components include RemoveOnCollideComponent, DestroyIfOffScreen, FollowBehind, and MouseController. This modularity matches the conceptual theory of micro-rhetorics very well and, as will be explained below, enables us to query the state of a game in the generation process.

Micro-rhetorics also make use of a simple grammar structure to support specifying sets of possible components than can be added to component. These are useful for gameplay patterns where the particulars of a component are not important, and several component assignments could represent the verb. We simply call these assignments non-terminals and they are denoted by an underscore as their first character. For example, a non-terminal of "_isVulnerable to target B" can be added to an entity A and any component that could be understood as making A vulnerable to B could be selected. Every PBE component is given a set of tags that are used when the non-terminals are resolved (described

Figure 3. The structure of a Micro-Rhetoric.

below). The current set of PBE components that are tagged with _isVulnerable are RemoveOnCollideComponent, ShrinkOnCollideComponent and StopOnCollideComponent.

Figure 3 shows the structure of a micro-rhetoric. Micro-rhetorics are defined by a verb that it can represent, a specific id (to distinguish between the multiple micro-rhetorics that represent the same verb in the micro-rhetoric library), and a set of component assignments. Component assignments specify which entity should be assigned the component (the owner), any other entity involved (the target) and the specific parameters that the particular component should be assigned in order behave as desired by the author of the micro-rhetoric. The owner and target values are assigned either the subject or predicate from the "subject-verb-predicate" concept map structure.

For example, consider a micro-rhetoric for the input "A avoids B." In this case A is the subject, B is the predicate and avoids is the verb. This particular micro-rhetoric will describe a set of game rules that will have A striving to avoid collision with B at risk of being harmed in some way. The first component assignments are to make sure that both the subject and predicate have the non-terminal _moveInAnyWay. This non-terminal makes sure that entities have some sort of movement behavior. Next, ChaseDownComponent is assigned to the predicate with a parameter of evaderName being set to the subject. This demonstrates how micro-rhetoric can set variables that are specific to particular PBE components – evaderName in this case. Finally, a non-terminal component of _isVulnerable is assigned to the subject with a target of the predicate. Because this micro-rhetoric is defined with the non-terminals of _movesInAnyWay and _isVulnerable, it can be realized in many different ways. For example, the A could be moving erratically while B moves directly toward it, and would shrink it upon collision, or the player could control A with the mouse while being chased by B which would remove A upon collision. How non-terminals are resolved will be explained below.

With the above understanding of micro-rhetorics, we can explain the first phase of *Game-O-Matic*'s generation process. For each node in the concept map, an Entity data structure is created. These structures mirror closely to the structures PBE uses to run games, but we wait until the game is completely generated before we "render" our internal data structures into a form that PBE will accept. This is done to separate the generation code from the workarounds we had to introduce as a result of PBE's implementation.

Next, one micro-rhetoric is selected for each verb and its parametrized components are added to Entity structures that map the nouns connected by the verb.

4. RECIPES: PARTIAL GAME DESCRIPTIONS MADE COMPLETE

At this point in the generation process, we have generated a partial game description made up of a list of entities and parameterized, or non-terminal, behavior components that each entity should have to represent the verbs in the concept map. However, because micro-rhetorics are authored to be as abstract as possible (to maximize the system's generativity and component compatibility) and there has not been any consideration given to the overall shape of the game, there is no promise that the partially formed game description will even have such necessary features like win or lose conditions, an avatar to control, or logically placed entities. The next phase of generation looks at the partial game description, and selectively applies modifications to add structure to the abstract rules generated in the concept map to micro-rhetoric process.

Each set of modifications to the partial game description we call a *recipe*. We call them recipes because the set of modifications can be understood as instructions for how to make the game become more like the gameplay pattern that the recipe was modeled after. For example, there could be a recipe that specified that the game could be won once all of one entity type has been removed from the screen. Note how this recipe would not make sense to apply if there was no behavior component that removed that entity. To avoid this sort of situation, each recipe has a set of preconditions which query the current partial game, and add or subtract from that recipe's salience score for the current game. The highest scoring recipe is chosen and the selected recipe's modifications are applied.

4.1 Precondition Predicates

A recipe's precondition is comprised of a set of *predicates*. Predicates are queries about the current game description that can be evaluated for truth. Each predicate can be a strict precondition (if it doesn't evaluate to true the recipe cannot be applied), or can have independent true or false weights that are added to recipe's overall score.

Predicates can make queries about an arbitrary number of entities in the working game description and are authored using logical variables. For example, one predicate could query whether entity X is controlled with the mouse, and another could ask if X is spawned by Y. When evaluated, all possible combinations of entity bindings to predicate roles are considered. If all strict precondition predicates evaluate to true, and that recipe has gotten the highest score, its score and entity/variable bindings (the assignments to the variables that produced the score) are stored and later its modifications applied.

Predicates can check if an entity has a component (explicit or non-terminal) or if the entity is being controlled by the player. They can also query the original concept map to see if the entity was the subject or predicate connected by a verb in the concept map input

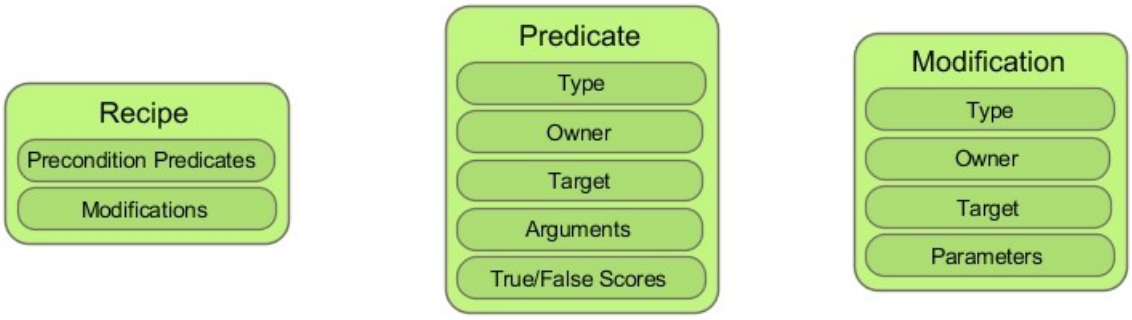

Figure 4. Recipes are selected based on precondition predicates, which query the working game description, change the working game description to make the games more sensible.

4.2 Modifications

Every recipe has a set of modifications that are applied if the recipe is selected. A modification changes the working game description to give it sensible gameplay and structure. Modifications can be made that add or remove components to an entity, give the entity player control and set the scale, rotation and placement of an entity. In a modification, entities are denoted by logical variables that are resolved by the same variable bindings that created the highest score for that recipe during precondition evaluation. For example, one modification would be to add a component to entity X that makes it follow Y closely.

Recipes also have access to a shared blackboard that recipe modifications can write to and predicates can query. This is used to allow recipes to communicate with one another about things that aren't easily represented in entities and components. For example, one recipe can note on the blackboard that an entity is intended to be the primary antagonist to the player and a later recipe can use this when setting entity positions.

4.3 Types of Recipes

Three types of recipes are scored and then applied to a game in sequence: win, lose and structure.

Win recipes determine the player's goal. Examples include remove all of one type of entity, having the player move to the right side of the screen, and surviving for a specified amount of time. Of course, not all partial game descriptions support all win conditions. For example, if the player has no way to remove an entity, it doesn't make sense to have the goal be to remove all of them from the screen. Preconditions and modifications enable us understand the current state of the generated game and modify it to be a more coherent game that players will understand how to interact with.

Lose recipes determine what causes the player to lose the game. Examples include running out of lives, failing to protect one entity from another, and not getting a high score in a specified amount of time.

Both win and lose recipes contain templates that are used on the final game's title screen to tell the player what he or she should try to accomplish and try to avoid. For example, "Make %X% huge to win" and "Lose if %Y% removes %Z%." Also, in the final game, upon triggering a win or lose condition a screen will pop up that can hold custom text that the user can author in *Game-O-Matic*'s interface.

At this point in the generation process, entities are specified, they each have mechanics that represent the micro-rhetoric and there are appropriate ways to win or lose the game. However, the game lacks sensible structure. Where are the entities placed on the screen? How big are they? Should specific entities be limited to specific regions of the screen? This sort of information is what structure recipes are meant to provide.

We roughly model structure recipes after classic arcade games. For example, the structure of *Frogger* could be appropriate to impose on a game where an entity A strives to collide with entity B, but something negative happens to A when it collides with C. In this case, the structure recipe would put A and B on opposite sides of the screen and put C between them moving erratically in order to create a challenge for the player. In terms of *Frogger*, A is roughly the frog, B is the goal area (lily pad) and C acts like the cars. In the current version of *Game-O-Matic*, we have defined structure recipes based on *Frogger*, *Space Invaders*, *Kaboom* and Asteroids.

Note that applying the structure of a classic arcade does not mean that the generated game will be a simple skinned clone of the arcade title. Not only will the player have different win and lose conditions, but the mechanics of the game will be completely different. For example, a game with the structure recipe inspired by *Space Invaders* could have the player controlling the bullets of the invaders trying to avoid colliding with the ship at the bottom of the screen. Also, as structure recipes don't modify the movement behavior components established by the micro-rhetorics, entities move in ways that can make the original game inspiring the structure recipe unrecognizable. The purpose of structure recipes is to ensure entities are spaced sensibly, in reasonably familiar patterns, such that movement around the screen maximizes the entity interactions specified by the micro-rhetorics, such as the *Frogger* recipe maximizing the negative interactions between entities A and C above.

4.4 Finalizing the Game

At this point, any remaining non-terminal components still remaining in the entities after the recipe modifications have been applied are resolved to specific PBE components.

Finally, a set of patches are applied to fix any unexpected problems that arise from combining all of these independently authored structures. These have the same form as the recipes, except all preconditions are strict and all relevant patches are applied (rather than just one). Patch preconditions often make use

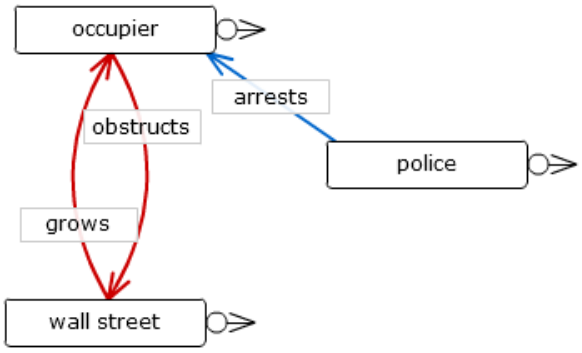

Figure 5. An example concept map created to represent a news paper article

of the blackboard. For example, if the win condition is to make it to the right side of the screen, and a structure recipe has moved the player's starting location away from the left side, a patch would recognize this and move it back. The patch phase allows an easy to author, case specific final check to make sure the generated game is as good as it can be.

After the patches are applied, the complete game structure, made up of entity and components specifications, is written out to Pushbutton Engine's XML level file format and the game can be played.

5. EXAMPLE

The following section explains *Game-O-Matic*'s processes using a specific example.

5.1 Concept Map

On the six month anniversary of the Occupy Wall Street movement, protesters returned to New York's Zuccotti Park and several were arrested [5]. Figure 5 shows a simple concept map meant to capture a high level description of this story. From the diagram we can see that the occupiers are obstructing Wall Street and are being arrested by police, but Wall Street is also growing the occupy movement. The concept map represents three relationships between Wall Street, the occupiers and the police.

5.2 Micro-Rhetorics

For each verb, *Game-O-Matic* randomly selects one micro-rhetoric that is tagged as representing it. For "arrests" in "police arrests occupier," it selects "take custody" micro-rhetoric. This gives the police and occupier entities a _movesInAnyWay component, and the occupier gets a StopOnCollideComponent which targets the police. _movesInAnyWay is a player-agnostic, non-terminal, which will be converted into a specific PBE component later.

For "occupier obstructs wall street," the obstructs micro-rhetoric with the id of "freeze" is selected. This gives the StopOnCollideComponent to Wall Street, thus preventing Wall Street's movement while colliding with an occupier. Other possible micro-rhetorics for obstructs could have been "redirect", which would have given Wall Street a ReflectOnCollideComponent with a target of the occupiers, which would cause Wall Street to bounce off of the occupiers

Next, for "Wall Street grows Occupiers" a grow micro-rhetoric is selected that gives the GrowOnCollideComponent to the Occupiers with a target of Wall Street.

5.3 Choosing Recipes

At this point, the system knows all of the entities, and the micro-rhetorics have given them a small set of components. Next, the win, lose and structure recipes are scored based on the partial game description and one of each type is applied. First, the win recipes are scored. The first win recipe sets the win condition to be for the player to "score 100 points." This recipe has the precondition:

- Y has a component of type _isVulnerable to X. [True: +4/False: -0]

The Y and X in the preconditions are entity bindings, each recipe will be scored for each possible combination. In the case of Y=Wall Street, X=Occupier; the recipe has a score of +4, because StopOnCollideComponent is tagged as being type "_isVulnerable." For Y=Occupier, X=Wall Street, the score is 0, because GrowOnCollideComponent is not tagged in this way. If a precondition such as "_isCollidable" (which both the Stop and Grow components are tagged with) were used instead of "_isVulnerable" the two bindings would have equal scores. The highest scoring recipes are chosen from at random.

5.4 Applying Recipe Modifications

After selecting the winning win recipe, its modifications are applied. Assuming it was "score 100 points" with Y=Wall Street, X=Occupier; this recipe would make the following modifications:

1. Write to blackboard: removeToWin Y
2. Remove component: Y _isVulnerable with target X (this will remove any components Y has that are tagged as _isVulnerable)
3. Add component to Y: _isRemovedBy with target X
4. Add component to Y: ScoreRemovalOfComponent with parameters: winScore=100, scoreEachRemoval=10
5. Add component to Y: RespawnOnRemoveComponent
6. Make X the player

As recipes are applied, the variables are substituted for their bound entity. First we'll write on the blackboard "Wall Street is being removed to win," which could be checked in the preconditions of a later recipe.

Modification 2 removes the "_isVulnerable" tagged component StopOnCollideComponent from Wall Street and modification 3 replaces the removed component with a stricter "_isRemovedBy" component, such as RemoveOnCollideComponent in order to guarantee that the player will be able to remove Wall Street and win the game. Replacing a StopOnCollide from "obstructs" with RemoveOnCollide constitutes a change to the micro-rhetorics first built from the concept map. Occupier removing Wall Street as a form of obstruction seems reasonable. This rhetorical leap enables *Game-O-Matic* to give novel interpretations of the system represented in the concept map.

Modification 4 adds a ScoreRemovalOfComponent to Wall Street, so that each time it is removed, 10 points are added to the score, and if the score is 100, the game is won. Modification 5 makes Wall Street respawn each time it is removed, so that the win score can be reached. Finally, modification 6 gives the player control over the Occupier..

Next, the lose and structure recipes are scored and the highest scoring one's modifications are applied. For the example below, the "run out of time" lose recipe is selected, which adds a

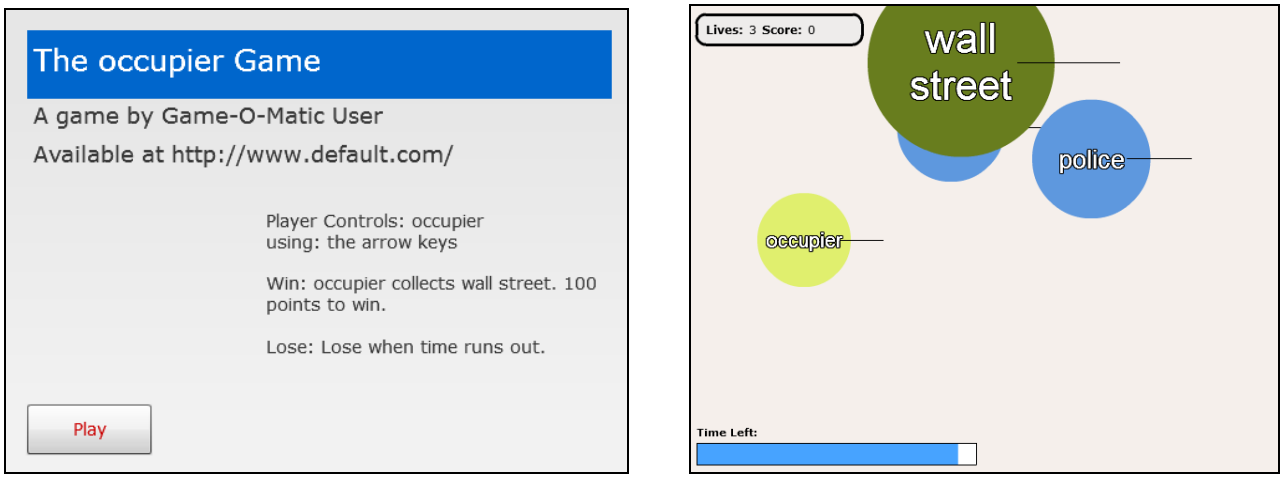

Figure 6. The instruction screen and a screenshot of the game generated from the concept map in figure 5.

MeterComponent to the World. The World is an entity which holds global components, such as UI elements and components which instantiate entities into the game.

Next assume the *Frogger* structure recipe is selected, which places the player on the left, an entity which collides with the player on the right, and several of another entity in the middle, in this case the Police. The middle entities have their movement restricted to vertical bars. The right-side entity, Wall Street, is set to double size.

5.5 Finalizing the Game

If a recipe hasn't already set the player, one is selected randomly from the nouns on the concept map. Non-terminal components are resolved to terminal components, then patch recipes are applied. Patch recipes aren't scored; they are all applied if their preconditions are met. The "everything moves" patch gives a movement component to every entity which does not have one. After all patches are applied, any remaining non-terminal components are resolved, and the instruction text for the start screen is generated.

At this point, all entity components are in place and we can start generating the XML which will be read into the PushButton engine. As we generate, some component parameters will have a single value, but others will have a range of values, which we select over randomly. For example, if a micro-rhetoric or recipe modification has not set a components parameter, such as movement speed, the value is randomly selected from a defined range defined per parameter.

With the XML generated, we can load it into the game engine. Components like MeterComponent and ScoreRemovalOfComponent will report to the UI to get their elements put on the screen.

The left side of figure 6 shows the start of the game which tells the player that he controls the Occupier with the arrow keys, and will need to collect 100 points worth of "Wall Streets" before the timer runs out to win. When the game starts (right of figure 6), Wall Street dashes past the police until the player manages to make it run into the occupier. At this point, Wall Street begins to shrink (the system chose ShrinkComponent when it resolved the non-terminal "_isRemovedBy "). Occupier is growing and will soon be stopped by police as the player moves the occupier to collide with Wall Street. Wall Street shrinks until it bleeps out of existence, the player gains 10 points, and a new one is spawned to take its place. As the occupy movement grows to fill the screen, overwhelming the police forces, removing Wall Street happens without any actions from the player. And this is just the first game generated! With just the press of a button user can generate other games that carry different interpretation of the user's input and different gameplay.

6. FUTURE WORK

The first next steps for *Game-O-Matic* will be to improve on and add to the recipe library as well as to increase the verb to micro-rhetoric library. As we expand these libraries we will add new functionality and improve the authoring process.

Another area that will be improved upon is how we choose micro-rhetorics from the concept map verbs. As of now, there is a simple mapping between verbs and the micro-rhetorics, but often additional relationships are implied by the concept map that could be caught by recognizing simple patterns and using those to select micro-rhetorics. For example, if "A protects C" and "B harms C," it would make sense to use a micro-rhetoric that represented that "A protects C from B" as opposed to representing each relationship independently (which would likely not represent the user's intent).

We also will be looking at applying several smaller structure recipes as opposed to just one. As of now, the games of *Game-O-Matic* bear resemblance to the classic arcade games that helped inspire it, and it is our hope that it will soon be able to generate mash-ups.

We will also perform an evaluation in the form of having players play a generated game, and then putting together a concept map representing the game using the concept map interface. The player's map will be compared to the map used to generate the game. While we expect that players will often create a map similar to the map that generated the game, implying that the system does create games which represent the meanings intended by the game designer, we also expect that players will recognize additional representations that were not part of the input which will inform our understanding of videogame interpretation.

7. CONCLUSION

Using the techniques described above, *Game-O-Matic* is able to create simple arcade-style games that represent the ideas put into

the concept map input. While it has yet to be formally evaluated, we believe that this approach enables the nontechnical users to rapidly create editorial newsgames. We also hope that the games of *Game-O-Matic*, which strive to represent using procedural rhetoric, will help the world better understand how videogames mean and can be used for the purposes of expression.

8. ACKNOWLEDGEMENTS

Game-O-Matic is being developed along with Bobby Schweizer, Simon Ferrari, Chris DeLeon and Emmy Zhang.

9. REFERENCES

1. Bogost, I., Ferrari, S., and Schweizer, B. *Newsgames: Journalism at Play*. MIT Press, Cambridge, MA, 2010.

2. Bogost, I. *Persuasive Games*. MIT Press, Cambridge, MA, 2007.

3. Bogost, I. The Cartoonist Aims to Bring Newsgames to the Masses. *PBS MediaShift Idea Lab*. http://www.pbs.org/idealab/2010/09/the-cartoonist-aims-to-bring-newsgames-to-the-masses243.html.

4. Cook, M. and Colton, S. Multi-faceted evolution of simple arcade games. *IEEE Conference on Computational Intelligence and Games*, (2011).

5. Francescani, C. Dozens arrested at Occupy's 6-month anniversary rally. *Reuters*. http://www.reuters.com/article/2012/03/18/us-usa-occupy-wallstreet-idUSBRE82G0FC20120318.

6. Labs, P. PushButton Engine. 2011. http://pushbuttonengine.com/.

7. Nelson, M.J. and Mateas, M. Towards Automated Game Design. *In AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, (2007), 626-637.

8. Smith, A. and Mateas, M. Variations Forever: Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games. *IEEE Conference on Computational Intelligence and Games (CIG)*, (2010).

9. Togelius, J. and Schmidhuber, J. An Experiment in Automatic Game Design. *IEEE Symposium on Computational Intelligence and Games (CIG)*, (2008).

10. Treanor, M., Mateas, M., and Wardrip-Fruin, N. Kaboom! is a Many-Splendored Thing : An interpretation and design methodology for message-driven games using graphical logics. *Foundations of Digital Games*, (2010).

11. Treanor, M. and Mateas, M. Newsgames: Procedural Rhetoric meets Political Cartoons. *Digital Games Research Association - DIGRA. 2009*, (2009).

12. Treanor, M., Schweizer, B., Bogost, I., and Mateas, M. Proceduralist Readings: How to find meaning in games with graphical logics. *Proceedings of Foundations of Digital Games (FDG 2011)*, (2011).