Generating Emergent Physics for Action-Adventure Games

Joris Dormans
Amsterdam Univeristy of Applied Sciences
Duivendrachtsekade 36-38
1096 AH Amsterdam, The Netherlands
j.dormans@hva.nl

ABSTRACT

Action-adventure games typically integrate levels, progression with the physical gameplay. In order to generate content for this type of games, this paper explores how procedural techniques can be expanded to beyond the domain of generating levels, and into generating physical interactions. It suggests a formal graph language to represent physics and the network of causal relations between game entities. Leveraging transformational grammars, the principles of model driven architecture, and component-based architecture for the game engine, it is argued that physics diagrams are well suited to generate emergent physical gameplay.

Categories and Subject Descriptors

K.8.0 [**Personal Computing**]: General—*Games*;
F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems; J.6 [**Computer-Aided Engineering**]

1. INTRODUCTION

Current research into procedural content generation for games usually interprets game content as stories, missions and levels. Most published games with procedural content (such as *Elite*, *Rogue*, *Diablo*, or *Civilization*) also focus on generating levels. In this paper I want to push the boundaries of game content beyond level design, and investigate how game mechanics can be generated as well. This paper is the result of my ongoing research towards building a procedurally generated action-adventure game. While previous research also focused on generating missions and spaces for this type of game, with this paper I aim to generate mechanics that govern the game's physics. For action-adventure games, novel mechanics is an important aspect of the game, equally important to level design. Successful games in that genre (such as *The Legend of Zelda*) closely integrate the game physics with items and abilities that unlock new 'moves' for the player character and often act as a 'lock and key' mechanism; the players progress through the level is controlled by these items and abilities [2].

In this paper I will take my research one step further by generating physical interactions between different entities in the game. The goal is to be able to generate interesting behaviors for enemies, items, and power-ups that allow the creation of interesting lock and key mechanisms, but also generate interesting gameplay by themselves. The paper is mostly theoretical, it outlines the general approach and suggests ways of implementing these techniques; it does not describe a working prototype. However, it does suggests how a prototype could be implemented; it is based on current efforts to implement the ideas presented within.

2. PREVIOUS AND RELATED WORK

Procedural content generation is a growing research topic within academia and the games industry. Within this field, generation of game mechanics is a relative new and not yet explored as much. A recent survey of search-based techniques for procedural content generation [25] lists a couple of dozen works in five categories that affect the main gameplay and at least as many works focus on optional content. Of these categories, the category that focuses on rules and mechanics, has only five works, two of which focus on board games. Needless to say generation of game mechanics is a huge and complex topic, that is best not tackled at once. Mark Nelson and Michael Mateas suggest to break down the problem of generating game mechanics into interrelated four domains: abstract game mechanics, concrete game representation, thematic content, and control mapping [17]. The research presented here focuses on the first domain: abstract game mechanics. The work in this paper has much in common with the approach taken by Adam Smith and Michael Mateas [22]. However, where Smith and Mateas use a code base representation of game rules, this paper uses graph notation to achieve a similar aim. The advantage of graph based representation of game mechanics is that they are more accessible to non-programmers, while they are still easily generated using graph grammars [19]. Accessibility is important because the research drew inspiration from the notion of mixed-initiative procedural content generation [23, 21]. This approach aims to have the computer and a human designer to collaborate during the design/generation process, instead of generating a complete level or a complete game with automatically with no input from a designer. The research presented in this paper follows the same line: the aim is not just to generate game

mechanics, rather to create a automated method that supports designers to create game mechanics fast and effectively. This approach suggests that procedural content generation is not an aim in itself, but rather a means to formalize and advance the game design lore.

This paper builds on my previous work in level generation and game mechanic formalisms [5, 6]. In particular the work on Ludoscope, a level generation tool that uses transformational graph and space grammars designed to generate levels for action-adventures [8]. This work builds on the ideas of model driven engineering in software engineering, where the process of creating software is broken down into a series of transformations applied on formal models representing the domain or the software [4]. By breaking down the process of designing levels for an action-adventure game into a series of transformations, and by designing different transformational grammars to specify each transformation, I was able to create a flexible tool that can generate a wide variety of levels. In previous papers, I focused on two steps in the process of designing action-adventure games: the generation of missions (a series of tasks the player needs to perform to complete a level) and the generation of a space to accommodate the mission. The formal models used for each step are graphs. This means that the transformations are described by graph grammars [13].

The approach to the generation of physical interactions in action-adventure games taken in this paper is similar to the approach taken in my previous work: rather than taking algorithms as a starting point, the techniques build on an abstracted, formal understanding of the game mechanics first and foremost. In this case, I will look at a formal description of the structure and qualities of the physical mechanisms and how they contribute to interesting, emergent gameplay. The procedural techniques will leverage these structures in order to generate interesting mechanics.

3. EMERGENCE AND PROGRESSION

An interesting challenge that stems from trying to generate content for action-adventure games is that these games typically combine elements of games of progression and games of emergence. These categories, originally proposed by Jesper Juul [14], are two alternative ways of creating gameplay. In games of progression, gameplay typically stems from carefully designed levels that present the player with a series of increasingly more difficult challenges. Creating this type of games requires a lot of specialized content to be created. Procedural content generation that aims at level design has been successfully applied to structures of progression in games (for example in *Diablo* and *Torchlight*). Action-adventure games use level design to structure the gameplay experience in this way. At the same time, action-adventure games also rely on emergent gameplay. Combat mechanics, the frequent use and combination of special abilities and power-ups are much closer to games of emergence. In action-adventure games, players create personal strategies and tactics to overcome particular obstacles. Action-adventure games are often designed to facilitate this type of exploration of the its mechanics.

Games of emergence rely on a fewer mechanics that generate a large possibility space. The term emergence originates from the science of complexity that studies complex, dynamic systems including, but not restricted to, games. Results from that field suggest that a number of structural features of complex systems play an important role in the creation of emergence. Stephen Wolfram [26] suggest that multiple state machines, with simple rules to govern their interaction can display surprisingly complex behavior. The number of connections between the state machines, and the activity of the state machines (how often they change their state) are good indicators for emergent behavior. In another study, Jochen Fromm [11] relates emergent behavior to the existence of feedback loops within the system. Multiple feedback loops create more emergent behavior.

Feedback loops and their relation to game design has been acknowledged by experienced game designers [15, 20, 1]. In previous works that focus on emergent behavior in economy driven games (such as simulation, real-time strategy, and board games) I also concluded that feedback loops in the flow of resources through the game are vital to create emergent gameplay [5, 7].

In order to generate mechanics for action-adventure games successfully, the mechanics fit the structural qualities dictated by emergence and progression in games. In this respect lock and key mechanisms play an important role, as lock and key mechanisms are vital for the games structure of progression, and at the same time interact with emergent physical properties of the game.

4. LOCK AND KEY MECHANISMS

Locks and keys mechanisms are a central feature of any action-adventure game. Locks and keys are used to structure the players progress through a game. A typical adventure game has actual keys that unlock doors, but also includes many other items that function similarly. For example, a special magic sword can act as a key when it is required to defeat a particular enemy in order to proceed. Designing interesting mechanisms for locks and keys is an important aspect of designing adventure games. Successful design strategies, found across many published games, often resolve around integrating the items or power-ups that act as the key into the physical simulation, or combat system. These keys rarely function for the sole purpose of unlocking a particular doorway; they double as weapons or allow the player to move around the environment in new ways. For example a bomb can be used against enemies but also to clear blocked passages. In order to create a novel experience for the player, designers are always on the lookout for new and interesting ways to create this type of mechanism.

To this end, integration into the physics simulation is a 'key feature'. In general, the integration is best realized by designing a system that rather maximizes the number of possible interactions between elements, not by introducing many different (types of) entities. This allows designers to design puzzles that require the player to combine different items and/or moves.

Lock and key mechanisms already played an important role in the Mission/Frame framework [9]. In particular locks and keys mechanisms allow a designer to transform a linear set of tasks in a nonlinear, branching structure (see figure 1).

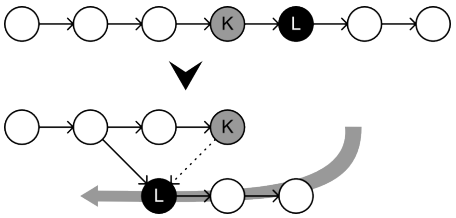

Figure 1: Transformation in a mission structure allowed by lock and key mechanisms.

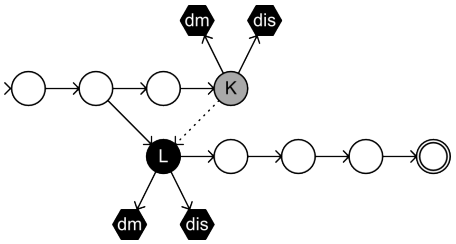

Figure 2: The properties of a bow can be used to unlock a door.

These branching structures lend themselves better to the generation of game spaces. In this case the transformation itself is realized through the execution of a transformation graph grammar. The process of generating an entire level can be broken down into a series of transformation each described by different grammars. However, in this previous work the locks and keys remain abstract constructions, I made no attempt to generate more interesting physical mechanisms for them, or to give them additional purposes in the game.

More detailed mechanics and multiple purposes can be generated by adding properties to the tasks in the mission diagram that describe the lock and the key. Figure 2 illustrates this, the properties are represented as hexagons that are connected to the locks and keys respectively. In this case, the diagram represents how a bow and arrow can function to open a lock requires damage to be dealt to it at a distance. This condition can later be realized by creating a door that is activated by a switch that needs to be struck and place the switch behind a barrier that cannot be traversed but that can be shot over (such as a body of water). Figure 3 illustrates a space graph that fulfill these requirements. In this figure the large circles represent distinct rooms or places in the level while the black circles represent items or features located in those places. Solid arrows indicate how the player can move between places. The dotted arrow indicates that a particular switch unlocks a door while a the solid arrow between the body of water and the switch indicates the switch is 'protected by the water'. Figure 4 represents the transformation rules that could generate that construction.

Similar properties, such as causing fire damage, the ability to grab distant items, or resisting fire, can be used to distinguish key items from one and another, but also to test whether or not one item might unintentionally act as a key for a lock designed for another item. For example, a sword might be modeled as a key that has the damage property,

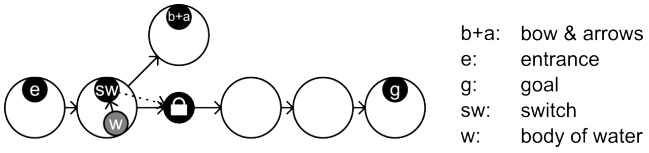

Figure 3: A space graph that realizes a lock and key mechanism for a bow.

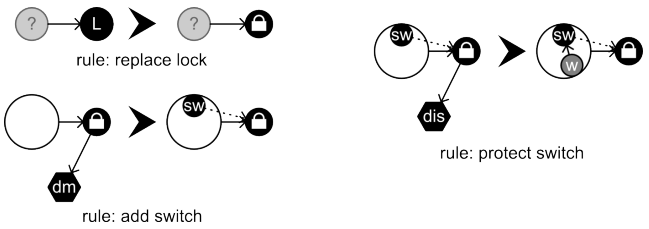

Figure 4: Transformational rules to generate the lock.

while a bow as a key that has the damage and the distance properties. This means that a lock that is opened when it takes damage might be opened using both a sword or a bow. In this case it makes more sense to generate a level where the player needs to find the sword before she finds the bow. Another interesting case, would be a lock designed for bow to have a similar switch behind a field of lava. However if that same level contained an item that would allow the player to cross the lava safely the player can process without needing to acquire the bow. In both cases it becomes important that a transformation does not generate an unwanted situation. The most intuitive way to prevent the system from generating these cases is to specify constraints to identify them and using those constraints to eliminate transformations that would generate them before they are applied.

5. CONSTRAINTS

Constraints are not an integral part of using traditional transformational grammars. The traditional approach would require that the grammar is constructed in such a way that the certain construction cannot be generated. In many cases this is possible, but it is hard and requires a thorough understanding and experience with transformational grammars. Mixed-initiative procedural content generation for games has game designers as an important target group. I aim to create automated tools to aid designers. For designers specifying a set of constraints is far more intuitive that figuring out how grammars can be designed that never generate unwanted constructions. For that reason I propose to extend the implementation of transformational grammars in Ludoscope the with constraints.

In Ludoscope constraints are defined in a similar way as rules or defined. A graph is created that illustrates the situation that should not occur. For example the constraint in figure 5 indicates that a key for a specific door can never be positioned behind the door it unlocks. In this case the star above the edge indicates that there might be any number of connections that between the lock and its key; the key need not follow its lock directly. Constraints like these can

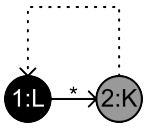

Figure 5: A constraint that would prevent the generation of a key that is placed behind its own lock.

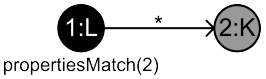

Figure 6: A constraint that would prevent the generation of a key that follows a lock with the same properties.

be easily checked when a grammar is looking for applicable rules. Ludoscope already compiles a list of all applicable rules before selecting which rule to apply. If the application of a rule results in a diagram that matches the constraint, the rule is simply removed from that list. This makes the procedure for selecting and applying rules slower as the process intensity increases. However, fast implementations for matching subgraphs exists and can be used in this case (see Marlon Etheredge's paper presented at the same workshop [10]). In addition, applying rules and looking for constraints while compiling a list of applicable rules has the advantage of offering the opportunity to run more heuristics that might affect the suitability of the rule. For example, it might check the new graph's size and adjust the likeliness of the rules selection based on a specified target size.

Implemented in this way, the definition of constraints can be easily extended to include useful short-cuts. For example, the constraint in figure 6 uses a special command that specifies that the constraint only applies when a match is found where node 1 has at least the same set of properties as node 2 has.

Constraints are specified with each grammar. In Ludoscope, execution of a grammar corresponds with a single transformation in a multi-step generation process. This means that after each step the generated content should be consistent and coherent. It also adds the flexibility of later steps being able to break the constraints that where applicable for earlier steps. Especially because in this model driven architecture for content generation later transformations tend to be more and more specific to the desired game. For example, while it is a good idea not to have keys behind their locks in the earlier generic transformations that create the a game structure, it makes equal sense to define a grammar that allows keys to be placed behind locks, because in this particular game that lock might initially be unlocked and will be locked after the player passed through it.

6. PHYSICS DIAGRAMS

Adding properties to locks and keys and using constraints can be leveraged to create a larger variety of locks and keys for action-adventure style game. However, it does not guarantee that it also generates emergent physics interactions for the game. To that end, we need a better understanding of the structural qualities in game physics that gives rise to emergence and a more direct way to model those qualities.

As was argued above, emergence can be attributed to structural qualities in the game design. Emergence depends on many active and interrelated elements. For games' internal economies that consist of flows of resources produced and consumed by different game elements, feedback structures play an important role in the rise of emergent gameplay. Feedback is generated when state changes in one element generate changes in other elements and over time cause new changes in the original element. As it turns out, a similar perspective can be applied to the physics mechanics of a game, although instead of looking for feedback loops in the production and consumption of resources, we need to look for feedback loops in the network of causal relations that the physics allow. This was argued independently during two recent talks at the game developers conference in 2012. Game designer Randy Smith who talked about his recent work on *Waking Mars*, addressed the importance of having long chains of causal relations in a game [24]. For example, in *Walking Mars* the player might throw seeds at patches of fertile soils to make them sprout. The plants produce new seeds that fall to the ground and might collide with other entities in the game, including crab-like creatures that carry the seeds away to eat them. However, the player might scare the crabs causing them to drop the seeds, etc. Designer and researcher Andy Nealen, who worked on the game *Osmosis*, suggested to structure these chains into loops (thereby creating infinitely long chains) in order to create complex behavior with only a few features and game mechanics [16].

The perspective of relative simple and few mechanisms that allow many interactions that leads to emergent physics in games partly explains the success and elegance of the 2D platform game. In these games a few simple interactions (moving and jumping) create many possible interactions with other elements in the game (jumping on platform, jumping into platforms to reveal hidden items, jumping on enemies, or walking into them). Often one interaction leads to another interaction: landing on unstable platforms will collapse the platform, which in turn might kill an enemy, and so on. The simplest indication that these long chains of events, and loops exists is by looking at the number of possible interactions. Emergent physics stem from many interactions between relatively few elements.

Networks of causal relations can easily be represented by graphs, and therefore easily be generated using graph grammars. All it requires is a graph language build to represent these physical interactions. For Ludoscope I propose a graph language that has three types of nodes to represent entities (circular nodes), conditions (diamond shaped nodes) and actions (triangles). For example figure 7 represents a network for the physics of simple platform game. In this case there is an entity called a player that responds to four different actions: it is accelerated by the gravity, move and jump actions, while it bounces as a result of the support action. There are several conditions in this diagram: the support action is activated when the player collides with a platform entity, the move action is only active when a certain key is pressed, while the jump action is active only when the player is supported and a certain key is pressed.

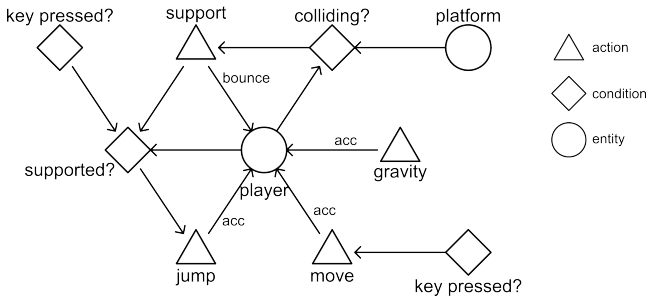

Figure 7: Physical mechanics for a platform game.

The types of conditions required to represent the physics of games is constrained. For most games conditions to check for overlap or collision goes a long way to describe the physical interactions between entities. Many other conditions can be derived from the syntax of the diagram itself: a condition that has an action and an entity as its input is met when that action currently acts on that entity. Conditions that represent player input form another, highly constrained set.

An important aspect this representation of physics mechanics is that one entity might contain another entity; entities can double as properties of other entities. All the actions affecting the container entity automatically also affect its contained entities. This allows for a flexible way of defining physical mechanics for a game. For example, figure 8 defines the mechanics for a sword (top-left) that simply specify that if a sword overlaps any other entity it will activate the 'damage' action on that entity. It also defines a health entity (top-right) that responds to two actions: 'damage' and 'heal', while it might also generate a 'kill' action on its container. Now, if we create a simple 'enemy' entity and have it contain a health entity, that becomes enemy can be damaged and eventually be killed by the sword. At the same time we can define a switch to respond to receiving damage by activating an entity it is associated with (as indicated by the dotted arrow), and a door that opens when activated (also see figure 8).

Entities can be defined in isolation of other entities, as was done in figure 8 or in a complete diagram of all physics (as is more or less the case in figure 7). Defining entities in isolation hinges on the implicit relations made possible by actions. A sword will damage anything it overlaps, but only those entities that define a reaction to the damage action, or contain other entities that do so will respond to that action. This allows us to shift between definitions and complete physics networks relatively easily. Definitions can be 'cut' from a complete network by taking an entity and trace its outputs and inputs back to the nearest actions and entities, and all relations between them. Likewise a network can be built from isolated definitions by connecting actions of the same name.

The physics of a game will cause more dynamic emergent behavior as the interactions increase and chains of causal events grow in length. This can be achieved when the actions entities activate and respond to come from a limited set that is shared amongst many entities. For example the more entities respond to the damage action, the more options the player will have for using a sword. At the same time, entities that both respond to and activate actions offer more opportunities for chaining of events. For example, right now the sword will only activate actions, while a door only responds to actions, while switches, timers and health entities both respond and activate. In general it is best to generate or design entities in a way that they do both.

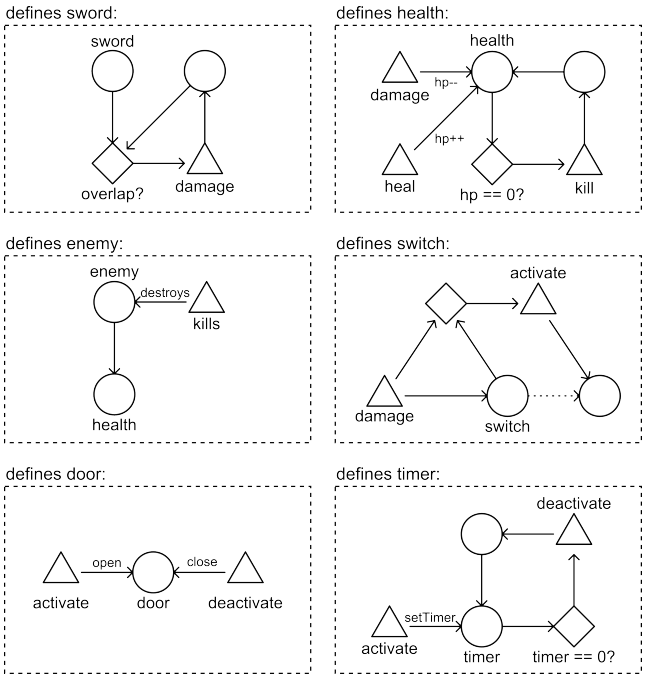

Figure 8: Physical mechanics for an action-adventure (subset).

7. IMPLEMENTING THE PHYSICS

Generating definitions for game entities using physics diagrams using graph grammars is fairly easy. Implementing or generating the associated behavior is more difficult. I suggest that using a game engine following a component based architecture is the best way to approach this challenge. Component based game engines are increasingly popular. It is an effective way of dealing with a wide variety of game object that share different behaviors [18]. Component based architectures are an alternative to inheritance based architectures that are less well suited to deal with sharing behavior among many different types of game entities or objects that follows the composite pattern from general software engineering [12].

At the heart of a component based architecture are two classes: components and composites.[1] Composites contain components. However, as composites inherit from component, they can also be contained by other composites (see figure 9). This leads to a flexible nested architecture that closely matches the physics diagrams presented above where entities can be contained by other entities. In a component based architecture game objects are not as much defined as

[1]Although a stricter implementation of the pattern would have three classes: Component, Composite and Leaf, where Component is an abstract class.

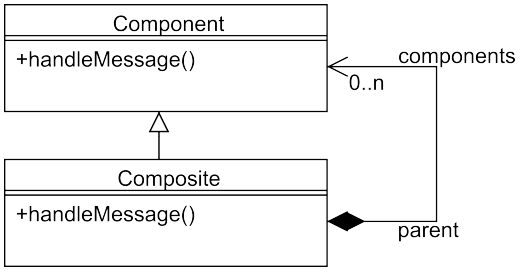

Figure 9: UML for components and composites.

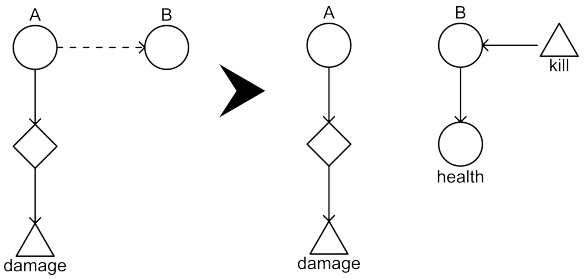

Figure 10: Graph grammar rule to create a physical interaction.

they are composed: they are created by assembling a number of components that define their behavior and rarely define their own behavior.

In a component based architecture individual components do not communicate directly, instead they communicate through a message system. Any component can send and receive arbitrarily defined messages. In this case that functionality is implemented through the handleMessage function. The Composite class overrides the handleMessage function so that it also passes all messages to its components. This ties in nicely with the actions in the physics diagrams.

To generate physics for games a component based game engine can be used in two different ways. First, game objects can be composed from a pre-designed set of components that are specific for the game. If in this case you want an entity A to interact with another entity B, you simple add components to entity B that respond to actions activated by entity A, or vice-versa. Transformational grammars can be used for this. Figure 10 illustrates this by specifying that a relationship between an entity that causes damage and another entity can be created by adding a health component (including all behaviors as defined in figure 8) to the latter. This rule uses a dashed arrow to indicate an unspecified relationship. Note that this rule also forces entity B to implement a response to the kill action.

A second strategy would be to generate definitions for entities or components using a generative grammar and then to generate software code that implements that behavior. For example the health entity in figure 8 could be easily translated to the following code:

```
public class Health implements Component
{
  private hp:int = 3;

  @Override
  public void handleMessage(message:String)
  {
    if (message.equals("damage")) hp--;
    if (message.equals("heal")) hp++;
    super.handleMessage(message);
  }

  @Override
  public void update() //called every frame
  {
    if (hp==0) parent.handleMessage("kill");
    super.update();
  }
}
```

Using this strategy it is possible to generate a physics network first and extract components from it. This way a network can be generated with a desired number of interactions and feedback loops.

8. DISCUSSION

Ludoscope was set up to support a model driven approach to procedural content generation [8]. The physics diagrams and extension of mission graphs with properties for locks and keys must be seen within this light. By designing graph languages specific to these sub-domains of game design, it becomes possible to define steps in the design process as graph transformations within each language. In addition, graph transformations can be used to translate between the different models. For example, generative and transformation grammars can be used to generate a physics diagram, additional grammars can then be used to find mission structures that are suitable for those physics. The design process is not restricted to a particular order. Physics might be generated before a mission is created, or the other way round. The properties for locks and keys are instrumental in this respect, they provide hooks in the generation process for physics or they provide a way of taking into account relevant constraints while generating a mission.

Ludoscope was also set up to support a mixed initiative approach to procedural content generation where a designer and the computer take turns in the generation process. In this case, designers can specify physics diagrams by hand, and use the computer to generate levels to match the physics. Extending the mixed initiative approach to game physics could lead to very fast prototyping tools where the physics diagrams provide designers with an intuitive interface onto that part of the design.

Physics diagrams as presented in this paper can also be used in fully automatic game generation tools such as the Game-O-Matic which was recently presented at the Game Developers Conference by Ian Bogost [3]. In the Game-O-Matic the designer specifies a content map containing entities and relationship between them expressed as verbs. Using graph transformations these could be used to generate physics diagrams where verbs can be translated in various ways. Using constraints and heuristics the tool could quickly search the

possible implementations and gravitate towards those that have the most interrelations and feedback loops.

Currently, the research is in its early stages. Physics diagrams can be represented by Ludoscope and grammars can be defined to specify transformations. At the moment of writing, there is no prototype that actually implements physics networks into a component architecture. Nevertheless, prototype is under development and its preliminary results, combined with my experience with procedural content generation prototypes, convince me that this approach will lead to interesting results in the near future.

9. REFERENCES

[1] E. Adams and A. Rollings. *Fundamentals of Game Design*. Pearson Education, Inc., Upper Saddle River, NJ, 2007.

[2] C. Ashmore and M. Nietsche. The Quest in a Generated World. In *Situated Play: Proceedings of the 2007 Digital Games Research Association Conference, Tokyo Japan, September 2007*, pages 503–509, 2007.

[3] I. Bogost. Making Games as Fast as You Can Think of Them. Presentation at the Game Developers Conference, San Francisco CA, March 2012, 2012.

[4] A. Brown. An introduction to Model Driven Architecture. 2004.

[5] J. Dormans. Machinations: Elemental Feedback Structures for Game Design. In *Proceedings of the GAMEON-NA Conference, Atlanta GA, August 2009*, pages 33–40, 2009.

[6] J. Dormans. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the Foundations of Digital Games Conference Monterey CA, June 2010*, 2010.

[7] J. Dormans. Integrating Emergence and Progression. In *Think Design Play: Proceedings of the 2011 Digital Games Research Association Conference, Hilversum the Netherlands, September 2011*, 2011.

[8] J. Dormans. Level Design as Model Transformation: A Strategy for Automated Content Generation. In *Proceedings of the Foundations of Digital Games Conference, Bordeaux France, June 2011*, 2011.

[9] J. Dormans. *Engineering Emergence: Applied Theory for Game Design*. PhD thesis, University of Amsterdam, 2012.

[10] M. Etheredge. Fast exact graph matching using adjacency matrices. Paper submitted to the Procedural Content workshop at Foundations of Digital Games Conference, 2012.

[11] J. Fromm. Types and Forms of Emergence. 2005.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston, MA, 1995.

[13] R. Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science 148*, pages 187–198, 2006.

[14] J. Juul. The Open and the Closed: Games of Emergence and Games of Progression. In F. Mäyrä, editor, *Proceedings of Computer Games and Digital Cultures Conference, Tampere Finland, June 2002*, pages 323–329, 2002.

[15] M. LeBlanc. Formal Design Tools: Feedback Systems and the Dramatic Structure of Completion. Presentation at the Game Developers Conference, San Jose CA, March 1999, 1999.

[16] A. Nealen. Minimal vs Elaborate, Simple vs Complex and the Space Between. Presentation at the Game Developers Conference, San Francisco CA, March 2012, 2012.

[17] M. J. Nelson and M. Mateas. Towards Automated Game Design. In *Proceedings of AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, Rome Italy, September 2007*, pages 626–637, 2007.

[18] R. Nystrom. Game programminf patterns: Component. online article, 2009.

[19] J. Rekers and A. Schürr. A Graph Grammar Approach to Graphical Parsing. In *Proceedings of the 11th International IEEE Symposium on Visual Languages, Darmstadt Germany, May 1995*, pages 195–202, 1995.

[20] K. Salen and E. Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, Cambridge, MA, 2004.

[21] R. Smelik, T. Turenel, K. J. de Kraker, and R. Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the Foundations of Digital Games Conference Monterey CA, June 2010*, 2010.

[22] A. M. Smith and M. Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[23] G. Smith, J. Whitehead, and M. Mateas. Tanagra: A Mixed-Initiative Level Design Tool. In *Proceedings of the Foundations of Digital Games Conference, Monterey CA, June 2010*, pages 209–216, 2010.

[24] R. Smith. Landing On Mars: Our Rocky Path to Inventinf New Gameplay. Presentation at the Game Developers Conference, San Francisco CA, March 2012, 2012.

[25] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[26] S. Wolfram. *A New Kind of Science*. Wolfram Media Inc., Champaign, IL, 2002.