Generating graphical content using grammars and graphs

Jess Martin

Abstract

Procedural techniques can automate and allow run-time execution of tedious tasks and generate arbitrarily complex content that would require large amounts of human effort. Also, due to the algorithmic nature of the methods, procedures can be constructed to create new content at run-time and respond during a simulation or video game with content adapted to unique situations that may arise at runtime. Procedural techniques have been in use for many years for content such as textures and trees. We present a generic framework consisting of a generative grammar and a graph encoding, allowing new types of content such as residential floor plans and video game maps to be generated automatically as well.

1 Introduction

As the capability to render complex, realistic scenes in real time increases, many applications of computer graphics are encountering problems related to the content of the scenes. Content refers to any characteristic of the scene which assists with its presentation, including models, textures, and animations. First is the problem of manual content creation. In virtual reality, researchers have found that creating and updating static models is a difficult and time-consuming task [Brooks 1999]. New games require large amounts of content, the creation of which is such an expensive process that it may drive smaller companies out of business [Wright 2005]. Content creators cannot keep up with the increasing demand for high-quality models. Algorithms will have to be called upon to assist. Additionally, as complexity and realism of scenes increases, repetition in textures and models become more apparent. Traditional modeling requires a static model to be created and stored for each variation in size, shape, texture, and even level-of-detail. Due to the increasing need for content, static methods should not alone serve the needs of complex scenes. Dynamically generated content is required.

Second, content generated statically cannot adapt dynamically to accommodate changes in a video game or simulation. Static models must be created for each possible configuration or interaction, limiting the appearance of natural variation and interactivity found in the real world. Static scenes in video games limit the replayability by only allowing the player to play through the game in pre-scripted ways. Procedural methods can adaptively generate new models and scenes to increase the possibility space.

Procedural methods, algorithms that specify some characteristic of a scene and generate a model [Ebert et al. 2002], are a possible solution to the problems of manual model creation and dynamically changing content. With such methods, the algorithm, not a human artist, determines the detailed attributes of the scene.

This paper reports work on two procedural methods for generating content useful for games and simulations. We also observe the similarities between the two methods presented and suggest a general pattern for methods to generate new types of content procedurally.

2 Previous work

Over the last twenty years, procedural methods have been successfully employed in several problem domains and are still an active area of research. Early techniques focused on generating content found in nature. Ken Perlin describes a complete language based on mathematically-generated noise for procedurally generating natural textures such as grass and clouds [Perlin 1985]. Algorithms employing fractals have been shown suitable for generating complex and realistic terrain [Musgrave et al. 1989]. Also, Prusinkiewicz described an implementation of L-systems that is able to procedurally generate complex and realistic plants [Prusinkiewicz and Lindenmayer 1991].

Recently, procedural methods have been applied to generating man-made artifacts including buildings. Parish and Muller used L-systems to generate a city map and geometry and textures for buildings [Parish and Muller 2001]. Focusing on city modeling, Greuter et al. describe a procedure to generate buildings based on vertical extrusion of randomly generated polygons [Greuter et al. 2003]. Wonka et al. focus on the appearance of the buildings using a "split grammar" to control the external appearance so that it conforms to architectural rules [Wonkta et al. 2003]. Muller et al. created a shape grammar called CGA (CG architecture) based on grammar production rules that models a building's exterior shape and facade [Muller et al. 2006]. Merrell introduced a method for procedurally replicating models with logical variations using a technique similar to texture synthesis in three dimensions; the technique is particularly useful in modeling architectural structures [Merrell 2007].

Procedural techniques have also been developed to generate abstract artifacts such as maps, stories and plots. Lechner et al. use an agent-based simulation to generate a city map with realistic zoning [Lechner et al. 2003]. Reidl and Young propose a new procedural technique for generating branching story graphs called narrative mediation [Riedl and Young 2006]. Video games such as Diablo and those like Rogue (so-called "roguelikes") have been producing randomly generated dungeons using maze-generation algorithms for over two decades [Roden 2005].

It should also be noted that languages for encoding procedural representations of textures and models have been researched in some detail over the past twenty years [Snyder and Kajiya 1992]; [Berndt et al. 2005]; [Cutler et al. 2002].

3 A Procedural Framework

Procedural techniques use a variety of methods to generate new content. However, the characteristics shared between techniques have not been widely explored. Consequently, there are no widely accepted patterns or best practices for the development of new procedural techniques. Our research builds on the idea of pipeline stages or steps presented by Roden and Parberry [2004] as a process for procedurally generating spatial content. We then apply that process to two specific problem domains.

We propose two specific steps that lend themselves well to the procedural generation of certain types of content. The purpose of the first step is to capture the topological relationships between elements without worrying about absolute spatial positioning or potential difficulties arising therefrom. The first step employs a set of rules in the form of a context-free grammar. These rules are then iterated to produce a graph representation of the content. In the second step, the nodes and edges of the graph are expanded to a corresponding spatial representation. We apply this process to the creation of house floorplans and video game maps and suggest other problem domains that have similar characteristics.

As previously mentioned, the first step uses a grammar to generate a representation of the content encoded in a graph. The graph representation of the content must be constructed such that the nodes represent elements (e.g., rooms or territories) and the edges represent a relationship (e.g., adjacency) between the elements. The nodes and edges should represent only relative size or position. The ruleset should consist of human readable rules that control the generation of the graph. However, a ruleset alone suffers from a lack of global state; each rule is applied locally to a single element. Our technique uses a statistics object to maintain a concept of global state and rules can be designed to check the conditions of the global state before they are applied. This is similar to the conditional check in CGA Shape [Muller et al 2006].

The second step focuses on the expansion of the graph representation into an actual physical representation suitable for eventual display. It is during this step that absolute sizes and positions of the elements are determined. There are many different algorithms that could be used at this step due to the fact that the algorithm must be specific to the kind of content to be created. The floorplan and video game map techniques will describe two algorithms for this step in detail and further methods will be explored in the discussion section.

The primary advantage of the two-step process is that it partitions the problem to make it more manageable. The first step solves the problem of the relationship between entities without considering the exact spatial layout. The second step solves the problem of the spatial layout without considering the relationships between entities. We assume that if a model of the relationships can be constructed in the first step then a solution to the spatial layout problem can be discovered during the second step. Additionally, the two steps can be executed independently as long as a well-defined interface exists for the data structure passed between step one and step two. From our own experience, this partitioning of the problem space has shown to be more tractable than tackling both problems together.

4 House Floorplan Generation

To date, procedural methods for creating buildings have produced buildings that are merely textured facades, containing no internal structure. Also, most algorithms do not draw upon insights from architecture. We introduce a method for procedurally generating a single-family residential unit, a house, automatically. The first step in generating the house uses a grammar which constructs a graph of the rooms and connections between rooms. The rules of the grammar are based on insights gleaned from particular architectural styles. Consequently, the algorithm is capable of producing houses in the style from which the rules are drawn. During the second step, rooms reach their appropriate sizes using Monte Carlo semi-deformable growth.

Our method of building generation is fundamentally different from methods previously discussed. While most methods generate only exteriors of buildings with no interior structure or layout, our method focuses on the floorplan with the assumption that the floorplan will eventually dictate most of the exterior appearance. Most other methods focus primarily on commercial buildingsâ€“office buildings, skyscrapers, and other large, fairly regular buildings. In exploring the problem of procedurally generated buildings, we found residential buildings to be highly complex due to their variety and irregularity and chose them as the target for our new method.

4.1 ALGORITHM PRINCIPLES

Two insights from Christopher Alexander's architectural writings provided foundations for our work [Alexander et al. 1977]. First, Alexander points out that architectural patterns vary between cultures and a particular set of rules will not be able to produce houses of all different cultures. Thus, we chose to focus on American-style residences. Second, Alexander's description of the layout of residential units revealed a crucial distinction between two types of rooms: some rooms are designated private, intended to be used by only one individual or couple; other rooms are designated public, intended to be used by a group. This simple distinction formed the basis of our graph generation step. Additional architectural insight came from browsing hundreds of house plans. These plans enabled us to make broad statements about average American houses that guided the development of the algorithm's ruleset which is based on commonalities among houses.

To ensure that the floor-plans generated by our technique are realistic, our method uses statistics and rules from common American homes such as the probability that a room will connect to another room, etc. These statistics and rules were gathered by analyzing many house plans by hand. The rulesets and the statistics are stored in text files in a human-readable format. A user could potentially rewrite the rules and statistics to apply this method to buildings of other cultures.

We also observed the following characteristics of floor plans and utilize them to provide constraints for the algorithm:

Room Size. Although the size of rooms varies depending on the size of the house, we observed that within any particular house the room sizes relative to one another are somewhat fixed according to the type of room. For example, a bedroom is usually larger than a bathroom, and a kitchen is usually larger than a pantry. We used this observation to define relative room sizes based on the type of room.

Footprint Size. The footprint determines the exterior boundaries of the house. The larger the footprint, the more rooms the house can contain. We vary the size of the footprint to allow the algorithm to generate different size houses.

Public to Private Square Footage Ratio. In larger houses, we observed that private and public ratios begin to favor more square footage for public spaces. The increase in square footage is largely accounted for by public space. Conversely, in smaller houses the ratio of public to private space is closer to 1:1. We chose a ratio of public to private space of 3:2 that represents "average-sized" houses.

Room Aspect Ratio. We observed that rooms within houses rarely exceeded an aspect ratio of 2:1. Thus we constrain rooms to not exceed that ratio.

The procedure for generating residential units has two distinct steps:

- Graph generation
- Room expansion

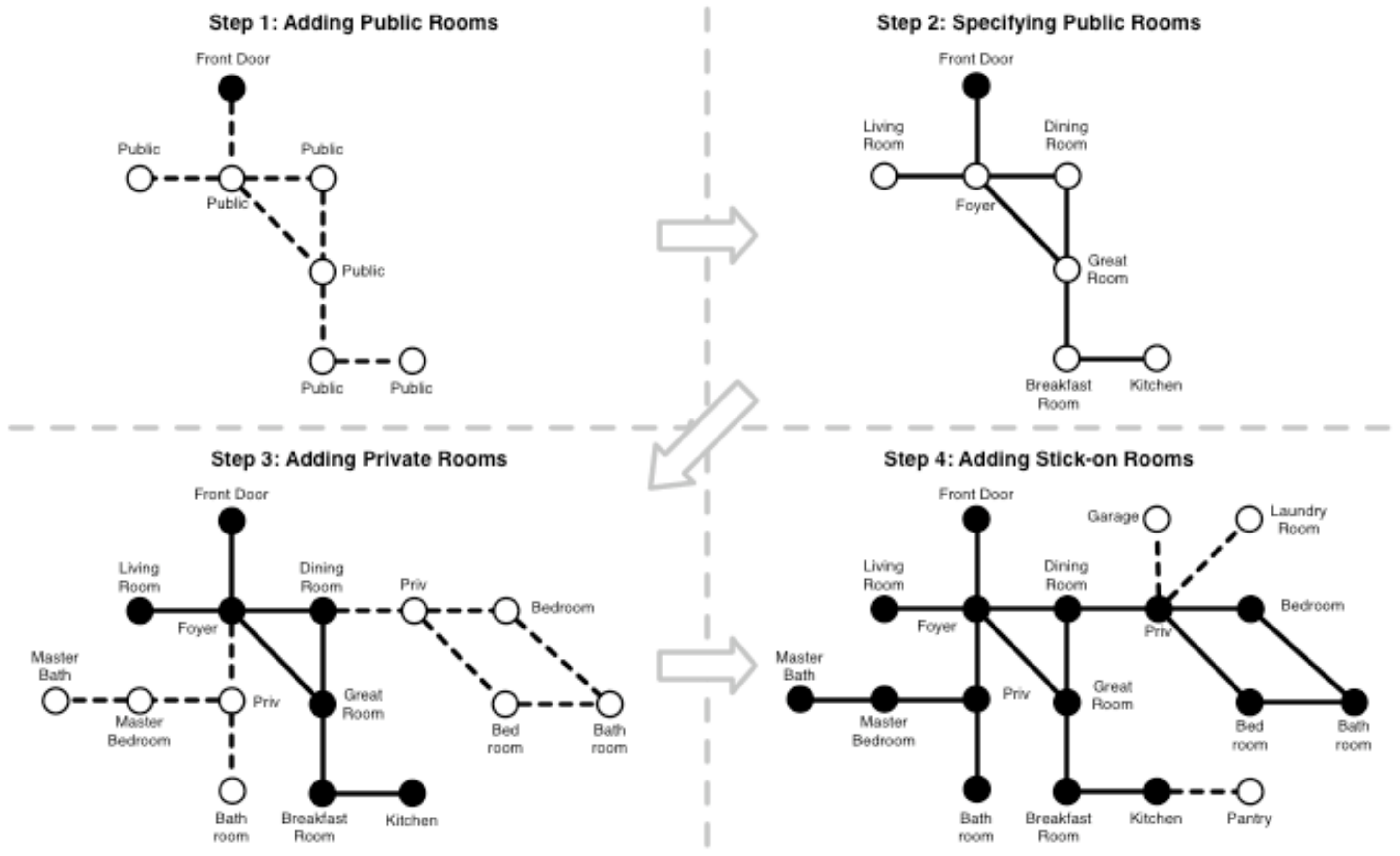

Figure 1: Graph generation overview.

4.2 GRAPH GENERATION

We represent the basic structure of a house as a graph with each node corresponding to a room and each edge corresponding to a connection between rooms. In the first step of the procedure, a graph encoding of rooms is generated. The graph generation phase itself occurs in four steps, as seen in figure 1.

Two techniques are used in the graph generation phase: a context-free grammar and a user-defined ruleset. We found grammars to be appropriate for "growing" the structure of the graphs, but they are ill-suited to capture the semantics of the graph. To account for the semantics, we used a ruleset to maintain both local and global information about rooms in the graph.

4.2.1 Adding Public Rooms

The graph generation phase begins by determining the structure of the public rooms. The front door of the house is added first and production progresses using the context-free grammar according to the ruleset in figure 2. This step is solely responsible for structure, not semantics. Thus, the rooms added are not yet a specific type of room.

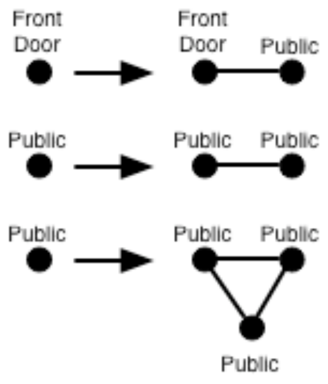

Figure 2: All public rules.

4.2.2 Specifying Public Rooms

Step two assigns a type (living room, dining room, etc.) to each public room by using four characteristics for each type:

- Must-attach room: A room that must be attached to that particular room. This captures the notion of rooms that always come in pairs, such as kitchens and breakfast rooms.
- Potentially attaching rooms: A list of all the room types which may attach to the particular room, such as a bathroom potentially attaches to a den.
- Minimum number: The minimum number of rooms of a given type that should be in any given house. Some rooms have a minimum of zero because they may not occur in a house, such as an office. However, other rooms such as a kitchen always occur at least once.
- Maximum number: The maximum number of rooms of a given type that should be in any given house. Few houses have more than two dining rooms, no matter how large the house is. This statistic allows the total number of rooms to be limited.

The use of these statistics allows the creation of more realistic spaces than using the grammar alone. Additionally, these statistics can easily be modified by a user who would like finer control over the types and connections of rooms in a house. However, in the absence of user specifications, the statistics system can still behave intelligently to mimic average American houses.

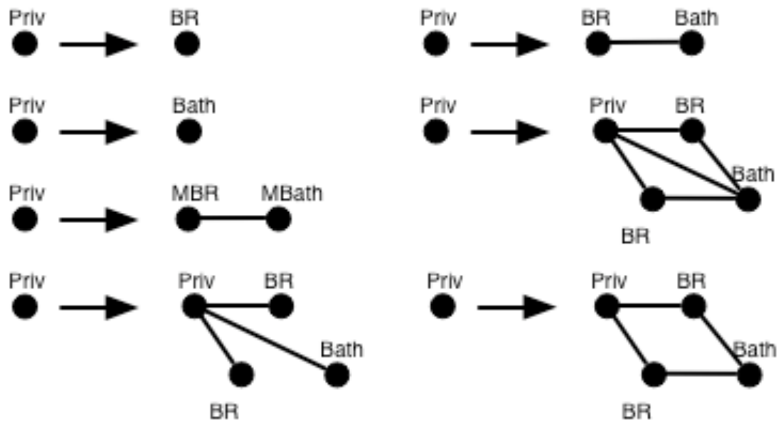

Figure 3: All private rules.

4.2.3 Adding Private Rooms

After steps one and two, the graph has a set of public rooms. Step three randomly places private rooms adjoining public rooms until the private space is filled. The private space is considered filled when the square footage allotted to private rooms has been exhausted. Private rooms should only be accessible by means of a public room. The private ruleset in figure 3 is used to determine what combinations of private rooms are added.

Some of some of the rules in figure 3 feature a private room on the right side of the production denoted by the label 'Priv.'

These nodes function as hallways to private areas and are expanded later in the graph expansion phase.

4.2.4 Adding Stick-on Rooms

The final step in creating the graph consists of adding rooms that are well-suited to being "stuck on" at the last minute, for example closets and pantries. These rooms can be added quickly and without impacting any of the other rooms already in the graph. For example, a pantry can be added to a kitchen without having to change other rooms. The statistics system also aids in this step to ensure logical connections.

4.3 GRAPH EXPANSION

At the beginning of this step, the types of rooms and the connections between them have been completely determined. However, the rooms are not located in absolute space. The placement phase distributes the rooms over the footprint. This placement process treats the graph as a tree with its root at the room that adjoins the front door. From that root, the root's child nodes are then evenly distributed beneath the root with each child spread an equal distance from other children and an equal distance from the root. The process is then repeated for each of the children nodes.

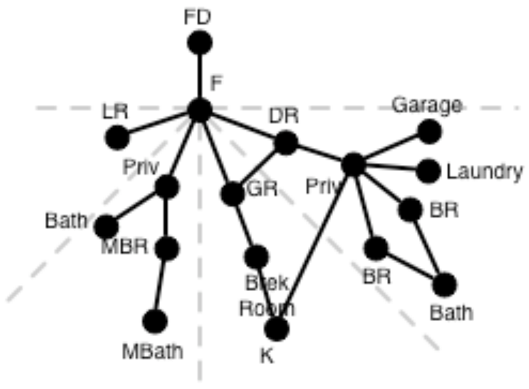

Figure 4: A graph with all the rooms placed.

4.3.1 Complications

The placement algorithm assumes that the graph will be mostly tree-like. Truthfully, it is not always a tree. Occasionally, there are connections between nodes on the same level. This is usually not a problem in the case of private rooms because the two rooms will most likely end up adjacent and a private room will rarely have any child nodes, unless they are a bedroom with a bathroom. However, if a public room has a connection to another room on the same (or higher) level, it can cause problems for the placement algorithm. To circumvent this problem, a cycle-detection algorithm is used and branches with cycles are relocated adjacent to one another. See figure 5 for an example of this.

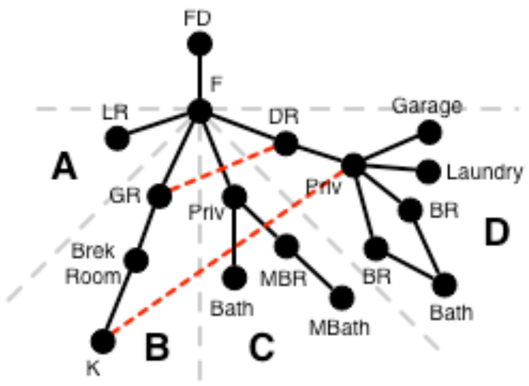

Figure 5: A graph with problematic links between branches of the tree marked by dashed lines.

The solution to the problem in figure 5 is to swap branch B and C. Although this solution is almost always available, sometimes the problem is more complicated. If the graph fails this simple solution, another graph is generated. Due to the speed of the algorithm and the infrequency of the complicated inter-linking, this solution does not significantly impact running time.

4.3.2 Expanding Rooms

Rooms are expanded to their proper size relative to one another using a Monte Carlo method to choose which room to grow or shrink next. Every room in the graph exerts an outward "pressure" proportionate to the size the room should be relative to other rooms. Walls that are shared between rooms have pressure on them from both sides.

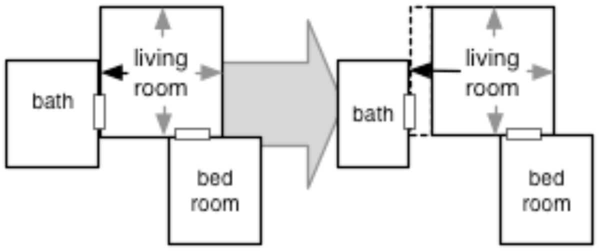

Figure 6: An example of a room expansion step.

The four walls of the room are examined in turn. The decision of whether to grow or shrink a room by moving a wall is based on the difference between the pressure inside the room pushing out and the sum of the pressures pushing in on that wall from adjacent rooms. If the pressure from the inside is greater than the sum of the pressures from the outside, the room will expand by moving the wall under consideration a fraction of the footprint. If the pressure from outside is greater than that from inside, no expansion of that room takes place.

Either of two events can cause a room expansion step to fail:

- The expansion causes a break in connectivity between rooms.
- The expansion causes the room to violate its aspect ratio by more than an acceptable tolerance.

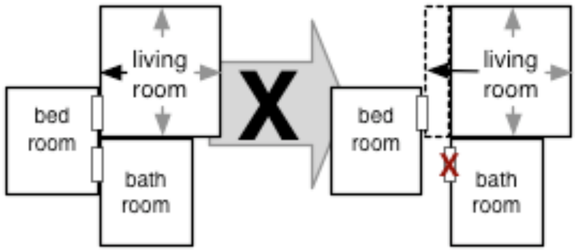

Figure 7: An example of a room expansion step that will cause a break in connectivity.

4.4 RESULTS

We implemented our algorithm in Java 1.5, executing the timing trials on a PowerMac G5 Dual 2.0GHz with 1GB of RAM. The application was single-threaded, and so did not use the second processor. Houses in the test runs had between twelve and eighteen rooms, with an average of fifteen. We used 300 iterations of the room expansion phase, observing that for fifteen rooms the expansion step hit equilibrium at about 250 iterations. The timing trial also generated the geometry for each house through the creation of an object file (OBJ format). The algorithm exhibits run time suitable for real-time applications. See figure 8 for a sample run of between 1,000 and 50,000 houses.

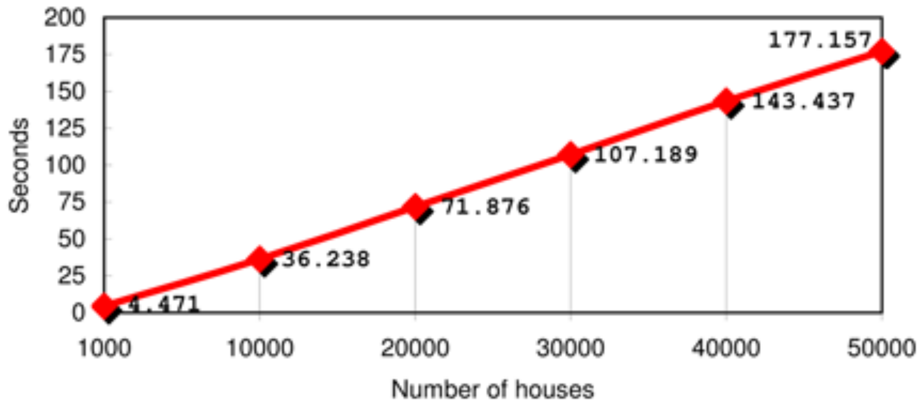

Figure 8: An example run that generates 50,000 houses.

5 Map Generation

Automatic map generation has been present in video games since the earliest games were created several decades ago, specfically beginning with *Rogue* and other so-called *rogue-likes*. Due to space restrictions on early computers and consoles, a level would often be randomly generated to avoid the cost of storing the level layout. As storage constraints relaxed due to advances in technology, the majority of video game creators favored statically generated levels. As an exception, recent games *Diablo* and *Elder Scrolls II: Daggerfall* feature randomly generated dungeons. However, these dungeons represent random areas within a fixed over-arching plot. Procedural methods could also be used to generate a map that enforces a certain "plot".

In role-playing games (RPGs), the map acts as a series of constraints which restrict the player to a certain path. Along this path, the player encounters people, enemies, objects, and locations. The sequence of encounters the player experiences can be thought of as the plot of the game. Notice that in order to impose a certain plot on the player, what is needed is a properly constructed map. We propose a method that intelligently combines common plot elements to construct an entirely procedural plot for RPGs, going beyond merely randomly generated dungeons.

5.1 ALGORITHM PRINCIPLES

Joseph Campbell observed that many important myths share a similar structure he termed the monomyth [Campbell 1949]. He outlined the basic structure of a hero's journey by dividing the story into three chapters or "acts": Departure, Initiation, and Return. Within each of these acts there are a variety of events which may or may not occur as the hero progresses through them. The genre of console role-playing games share a similar structure, many of which have been documented [Sachs 2004]. We use the act-based structure of Campbell's monomyth and the content of the common genre clichès to construct our algorithm.

The procedure for plot generation has two stages:

- Nested Graph generation
- Graph expansion

5.2 NESTED GRAPH GENERATION

The graph structure used to encode a story is a nested graph. Each node of the graph can itself contain another graph. A node that contains another graph indicates that the elements of the graph occur within the node's context. For example, an act could contain a town and a dungeon, the town could then contain an inn and a weapon shop, and the weapon shop could then contain a blacksmith. Connections between nodes indicate reachability or, in other words, the player can move between connected nodes without impediment. If multiple nodes exist at the same level without connections between them it is implied that they are all reachable to one another. Figure 9 represents an example of a common nested graph.

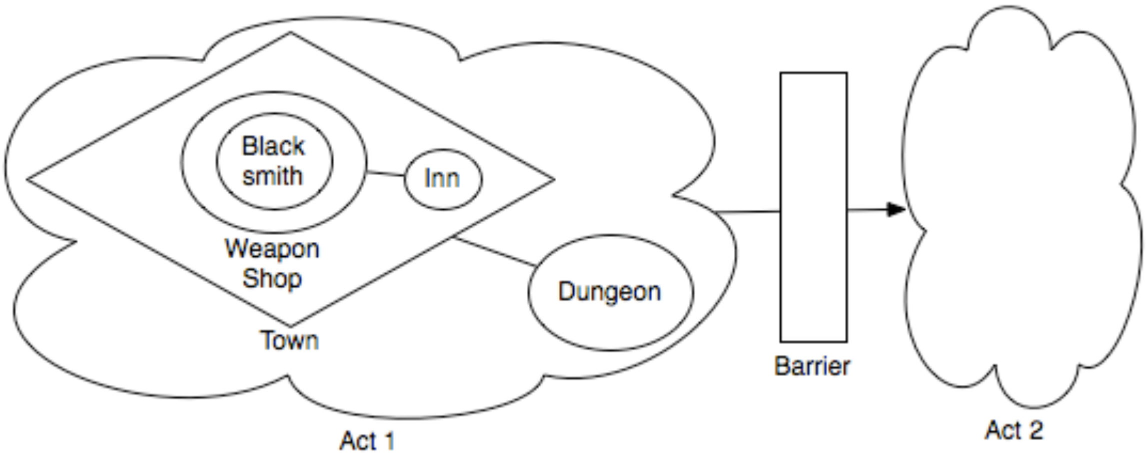

Figure 9: A sample nested graph representing a story plot.

5.2.1 Acts

Acts are the outermost nodes of the nested graph that contain other graphs and are represented by the circular nodes in figure 9. Each act represents a region where some significant amount of the story will take place. Thus, acts also often contain locations such as towns and dungeons to visit, enemies to fight, non-player characters to interact with, items to acquire, and side quests for optional story development. In order to ensure that the player experiences the plot's development in the proper order, acts are typically separated by barriers.

5.2.2 Barriers

Barriers are a special type of node that are represented by the rectangular node in Figure 5. They represent an obstacle to reachability. In order to progress to the next node, the barrier has a constraint that must be satisfied before it can be passed. Common examples of constraints from RPG plots include enemies that have to be defeated, a broken bridge that must be repaired, or a stream that must be crossed. Some barriers may require a specific item in order to be satisfied, i.e., a special hammer to repair the bridge. Others may require a vehicle such as a canoe to cross a stream. The item or vehicle that is required to bypass the barrier must be reachable within the previous area. Barriers generally connect different acts of the plot, but they can also be used within an act to allow more fine-grained control over plot development.

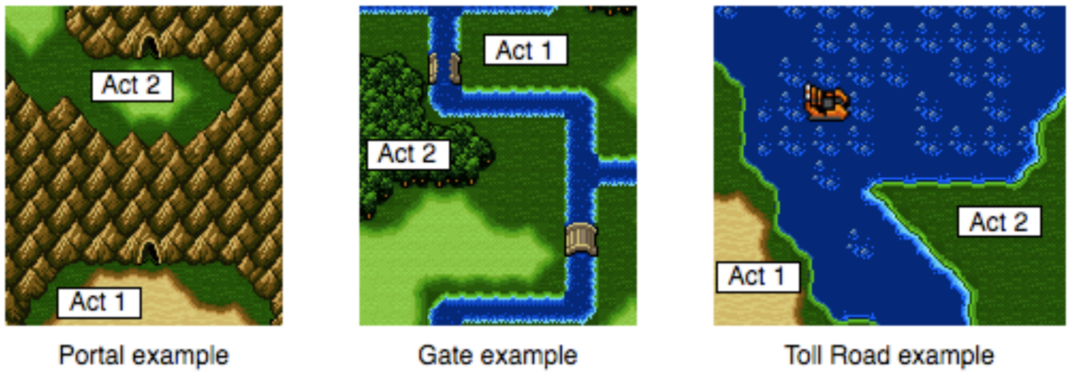

Figure 10: An example of the different kinds of barriers on actual maps.

Barriers were generalized by examining source maps from many different games. Our observation was that barriers fall into one of three types: gates, toll roads, and portals. The type of the barrier influences adjacency restrictions for the two nodes that it connects.

- **Gates** are a single tile that separates two adjacent nodes. The nodes must be located immediately next to one another. Examples include a broken bridge, a locked dungeon door, or a blocked mountain pass. See figure 10 for an example.
- **Portals** are a barrier that actually transports the player from one section of the map to another and therefore do not require the nodes to be located adjacent to one another. Examples include a tunnel through impassable mountains, a

teleportation device, or a ferry across a river. See figure 10 for an example.

- **Toll roads** are a series of tiles that separates nodes. Usually a toll road makes a previously untraversable type of tile traversable. Examples include swamplands traversable by canoe, a body of water only traversable boat, or snow traversable only with snowshoes. See figure 10 for an example.

5.2.3 Rules

The rule system controls the generation of the nested graph via a list of simple grammar-based rules of the form

```
A --> B-C | D
```

where A, B, C, and D are nodes and node A will be filled by either the subgraph of node B connected to node C or the node D. Unlike most grammar-based rule systems, a production rule does not indicate the replacement of the node on the left-hand side, but rather that the right-hand side be nested within the node on the left-hand side. A simple set of default rules are listed in Figure 11.

Ultimately, each node represented in the rule system must have some content associated with it in order for the graph expansion algorithm to know how to represent the node when it is expanded into a two-dimensional map.

```
Arc              =  Start Symbol
Arc             --> Intro-Barrier-Act-Barrier-End
Intro           --> Town | Town-Sidequest
Act             --> Town-Dungeon-Sidequest
End             --> Town-FinalCastleDungeon-Sidequest | Town-FinalCastleDungeon
Barrier         --> GateBarrier | PortalBarrier | TollRoadBarrier
GateBarrier     --> ItemBarrier | NPCBarrier | SkillBarrier | EnemyBarrier
PortalBarrier   --> MountainPass
TollRoadBarrier --> VehicleBarrier
ItemBarrier     --> SkeletonKey | BrokenBridge
NPCBarrier      --> FortuneTeller | TravellingWizard | Princess
EnemyBarrier    --> Dragon | Orc
VehicleBarrier  --> Ship | Chocobo | Snowshoe
SkillBarrier    --> Fire | Ice | Grow
Sidequest       --> Collect10FishSQ | SewerMazeSQ | WaterfallSQ | AlligatorWrestlingSQ
Town            --> DesertTown | LakeTown | MountainTown
Dungeon         --> FireDungeon | IceDungeon | EvilCastleDungeon
```

Figure 11: A sample ruleset for generating a nested graph.

5.3 GRAPH EXPANSION

Once the story graph is constructed during the graph generation phase, it can have a variety of different uses. Our purpose is to provide a tool for generating plots for simple role-playing games. To that end, we will present a method for turning the graph into a simple two-dimensional map.

5.3.1 Act Generation

Each of the acts is represented on the map as an area divided by impassable barriers such as water or mountains. The required area for the act is calculated based on the number and type of things encountered within the act. The shape of the area for each act is generated by a technique we call shape-splatting. The algorithm for shape-splatting with a set of primitive shapes (triangle, rectangle, hexagon) is as follows:

1. Place a random primitive shape that is 70% of the land mass area required.
2. Create another randomly sized primitive and add it to the existing shape with a random offset.
3. Iterate step 3 until within some threshold of the desired land mass area.

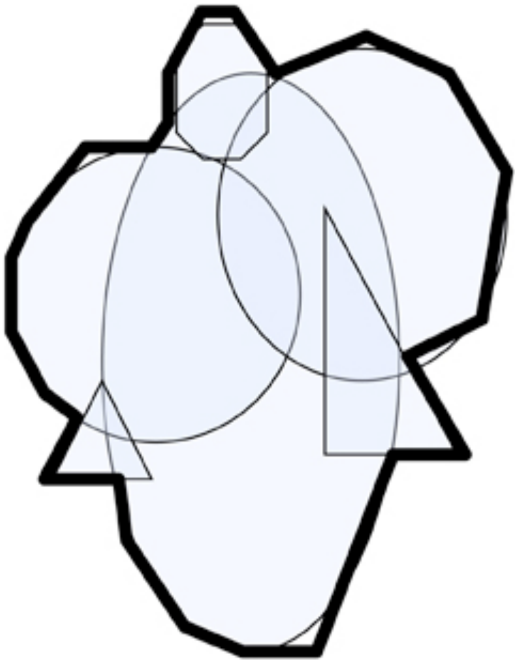

Figure 12: An example of an act region generated via overlapping shapes.

The algorithm produces shapes that resemble the land mass in figure 12. Land masses are connected by finding the outermost extremity of the shapes and connecting them in accordance with the type of barrier that connects them.

5.3.2 Intra-Act Placement

After the land masses for each of the acts have been generated and connected, the act needs to be populated with the locations, enemies, and terrain found in the act. The general strategy for placement of items within the act will be a technique called localized dart-throwing [Dunbar 2006]. This technique will allow enemies, buildings, and terrain elements such as rocks to be placed with a reasonable density without overlapping existing objects. A few rules will need to be observed in the use of the dart-throwing algorithm to ensure a sensible layout.

- Towns need to be contained within a subregion of the act's area. Buildings and non-player characters should be placed within that subregion. Enemies should not be allowed to be placed within the subregion of the town.
- Whatever barrier connects the current act from the next one, the element (item, object, person, enemy, etc.) that satisfies the constraint of the barrier needs to be placed within that act and reachable.
- The terrain (grass, sand, forestry) present in the act should match the type of town specified by the ruleset.

The prior guidelines should be sufficient to prevent any nonsensical configurations. Further implementation details would be dependent on the game or application that is using the map and what specific nodes are present in the rule system.

6 Conclusion

We have shown how the general steps of rule-guided graph construction and graph expansion can generate spatial content automatically while allowing the user to have control through user-editable rulesets. Other procedural techniques could be created based on the two step process described in this paper to generate additional kinds of content. The general problem solving principle of divide and conquer is employed to simplify the creation of an algorithm for each of the steps. The first step merely needs to solve the problem of the relationship between elements. The second step can then focus on the problem of placing the elements in space in such a way that the relationships established in the first step are not broken. We hope to see other algorithms employ this separation of relationship from presentation in order to create additional procedural techniques that model different kinds of content.

Bibliography

Alexander, C., Ishikawa, S., and Silverstein, M. 1977. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, New York.

Brooks, F. P., 1999. What's real about virtual reality? IEEE Computer Graphics and Applications 19, 6, 16-27.

Campbell, J., 1949. The Hero With A Thousand Faces. New World Library, New York.

Dunbar, D. and Humphreys, G. 2006. A spatial data structure for fast Poisson-disk sample generation. ACM Trans. Graph.25, 3 (Jul. 2006), 503-508.

Ebert D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. 2002. Texturing and Modeling: A Procedural Approach, third ed. Morgan Kaufmann.

Greuter, S., Parker, J., Stewart, N., and Leach, G. 2003. Real-time procedural generation of 'pseudo infinite' cities. In Computer graphics and interactive techniques in Australasia and South East Asia.

Lechner, T., Watson, B. A., Wilensky, U., and Felsen, M. 2003. Procedural city modeling. In 1st Midwestern Graphics Conference.

Müller, P., Wonka, P., Haegler, S., Ulmer, A., Van Gool, L. 2006. Procedural Modeling of Buildings. In Proceedings of ACM SIGGRAPH 2006 / ACM Transactions on Graphics (TOG), ACM Press, Vol. 25, No. 3, pages 614-623.

Musgrave, F. K., Kolb, C. E., and Mace, R. S. 1989. The synthesis and rendering of eroded fractal terrains. In SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 41-50.

Parish, Y. I. H., AND Müller, P. 2001. Procedural modeling of cities. In SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques, ACM Press, New York, NY, USA, 301-308.

Perlin, Ken. An image synthesizer. *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques.* New York, NY, USA: ACM Press, July 1985, 287-296.

Prusinkiwiecz, P., and Lindenmayer, A. 1991. *The Algorithmic Beauty of Plants.* Springer-Verlag, October 1991.

Sachs, M., 2004. The Grand List of Console Role Playing Game Cliches. http://project-apollo.net/text/rpg.html. Last accessed 1/23/09.

Wonka, P., Wimmer, M., Sillion, F., and Ribarsky, W. 2003. Instant architecture. In Proceedings of ACM SIGGRAPH 2003 / ACM Transactions on Graphics (TOG), ACM Press, Vol. 22, No. 3, pages 669-677.

Wright, W., 2005. The future of content. Keynote speech, Game Developer's Conference 2005. San Diego, March, 2005.