Randomness + Structure = Clutter: A Procedural Object Placement Generator

Joshua Taylor and Ian Parberry

Department of Computer Science & Engineering, University of North Texas,
Denton, TX 76207, USA
JoshuaTaylor@my.unt.edu, ian@unt.edu

Abstract. *Clutter* is the random yet structured placement of objects in a room. We describe a procedural clutter generator that achieves believable, varied, and controllable object placement using a hierarchical colored Petri net capable of expressing any computable set of object placement constraints.

1 Introduction

The demand for game content has increased to the point that its creation by artists and designers has become one of the more time-consuming and costly parts of game development. *Procedural content generation* is the term used for computer generation of game content (see, for example, Roden and Parberry [1] and Nelson and Mateas [2]). In addition to taking some of the fiscal and temporal burden from game developers, real-time procedural content generation can increase a game's replayability by the incorporation of the generators into the game itself.

We use the term *clutter* to refer to non-architectural room contents. There has been little previous work on clutter generation. Howard and Broughton [3] offer a method in which the major pieces of furniture are added by hand and the miscellaneous objects are added by a genetic algorithm. Tutenel *et al.* [4] offer a more complete solution using a constraint solver that requires a set of tagged bounding boxes for each object.

Doran and Parberry [5] list a set of five criteria important to any procedural content generation system: *novelty*, *structure*, *interest*, *speed* and *controllability*. We present in the paper a procedural clutter generator that we have designed to maximize controllability but not at the cost of the other four criteria. We argue that our generator produces interesting room clutter, and demonstrate this with an implementation of the generator available online for the reader to test for themselves [6] using any standard web browser. We also argue that our generator is flexible enough that a designer can control the output to produce appropriate clutter for different types of rooms.

2 Anchors, Objects and Collisions

A cluttered room does not usually contain a completely random jumble of objects. There are almost always patterns in the way that things are laid out, for example,

J. Anacleto et al. (Eds.): ICEC 2011, LNCS 6972, pp. 424–427, 2011.

certain objects or sets of objects tend to be grouped in specific areas of a room. Objects usually appear in either the corners, spaced regularly but not perfectly along the edges, in a rough grid in the center of the room, or in a specific but logical relation to other objects in the room.

We capture this intuition with the concept of an *anchor point*, which marks a place at which an object can be (approximately) located. Throughout the rest of this paper our examples will be in 2D for ease of discussion, but the principles are the same in 3D. In 2D the anchors, as mentioned above, would be placed in the corners, spaced along the edges, and spread out in a grid in the center of the room. The spacing along the edges and the size of the grid depend on the room type. The placement of these initial anchor points may be done procedurally or by the designer.

Each object placed by our clutter generator also has a set of anchor points for further objects. For example, a table may have anchor points for the chairs around it and points on top for the place settings, center piece, and other clutter.

To avoid having all the objects sit in perfect alignment with each other, objects being placed have a Gaussian displacement in both position and orientation about the normal of the surface the object is on, specified by the standard deviations.

It is very likely that randomly-placed objects will end up colliding. If an object collides with another, we simply generate new random displacements for its position and orientation, repeating if necessary up to some small number of attempts. If this fails, it is often safe to throw the object away. However, some items that are important for gameplay could fail to generate. In this case, we suggest that another random room layout be generated, again up to some small number of attempts. If that fails, it is likely that the constraints should be redesigned.

3 Petri Nets

Petri nets date from 1939 [7] and have since been applied to a wide range of applications including distributed computing [8] and manufacturing [9]. There are many great resources on the basics of Petri nets, so we will avoid repeating that information here. The Petri nets we use are a variation on colored, hierarchical Petri nets with inhibitor edges. The inclusion of inhibitor edges is needed to make the Petri net Turing-complete (see Peterson [10]).

Tokens in standard Petri nets are indistinguishable. Conversely, in *colored* Petri nets the tokens carry extra information. We use this to store the information needed to place objects in the room. While it may seem natural for tokens to represent objects, this approach quickly runs into problems. Instead, each token will carry either zero or one anchor points. Generic tokens, with no anchor point, can be added manually to the initial net or be created at run time. Tokens with anchor points can only be created at run time.

Colored Petri nets also add extra semantics to the transitions to handle this extra information. In our implementation, this leads to three significant differences over standard Petri nets.

First, the relation between the incoming and outgoing edges of transitions must be made explicit. For each incoming edge, the user must specify which outgoing edge

will get the token or that the token is to be discarded. Similarly, for each outgoing edge, there is the possibility of making a new generic token, or taking whatever is coming from one of the incoming edges.

Second, for each incoming edge the user can specify which types of tokens to allow on that edge. This is handled by attaching a non-unique name to each anchor point and then checking those names when deciding if the transition is ready to fire.

Third and finally, each incoming edge can create an object at the anchor point represented by the token it is working with. That token is then replaced by a set of new tokens representing the anchor points on the new object.

While running, one live transition is picked randomly to fire. To control how often things happen more precisely, a probability is attached to each transition representing how likely it is to fire if picked. This defaults to 1 which means it *will* fire if picked.

The pages of hierarchical Petri nets make running the net somewhat difficult. To avoid that, we instead unroll the net into a single page. To provide the widest range of possibilities, each place and transition is assigned a *scope*. A local scope means that places and transitions with the same name on different pages are considered different, while a global scope would mean they are the same and should be combined when the net is unrolled. To make sure that the unrolling works we also introduce *links*, a subtype of places. Links are treated like any other place but they cannot have generic tokens and can only be locally scoped. All page calls connect places in the current page to links in the called page, which are merged when that page call is unrolled.

4 Implementation

We implemented a prototype of our system in 2D in Java. The room and the initial anchor points were prepared by hand, but the system is designed to be used alongside a room generator. Figure 1 shows some of the generated rooms. (The reader is invited to visit Taylor and Parberry [6] for higher resolution images or to try the generator for themselves.)

The list of objects and the Petri net are stored in XML files. The system takes these inputs plus the room with the initial anchor points. It then unrolls the Petri net, creates a token for each anchor point, and feeds those into the starting place in the net. Execution consists of making a list of live transitions and firing one randomly. This continues until there are no live transitions.

Judging our approach by the criteria mentioned in the Introduction, we claim that our content is novel in that the room contents are unpredictable, yet there is structure in the way the contents are laid out. As evidence, we provide the pictures in Figure 1. Our approach is comparable in speed to that of Tutenel *et al.* [4], and the Turing-completeness of Petri nets offers better controllability. The approach used by Howard and Broughton [3] is designed for a specific subset of clutter generation and so suffers in both categories.

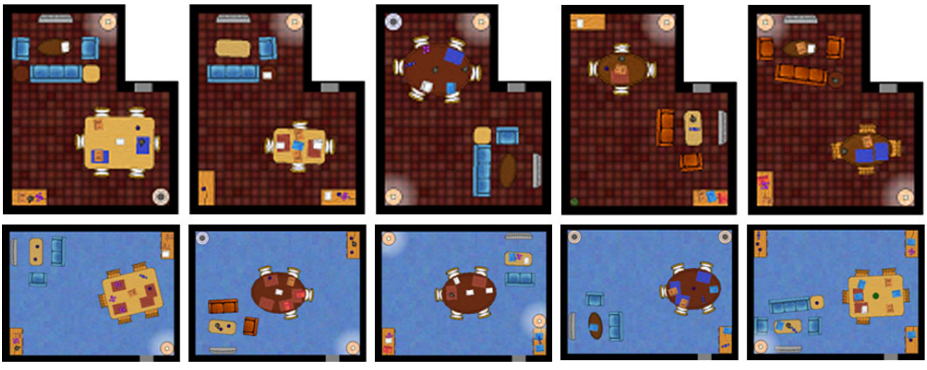

Fig. 1. Ten rooms generated with our system using the same Petri net and object set

5 Conclusions and Further Work

This paper introduces a procedural clutter generator based on hierarchical, colored Petri nets that can express any arbitrary computable set of constraints between objects. It remains to construct a graphical user interface for the designer. This interface should ideally output XML scripts that can be read by our generator.

References

1. Roden, T., Parberry, I.: From artistry to automation: A structured methodology for procedural content creation. In: Proc. 3rd International Conference on Entertainment Computing, pp. 151–156 (2004)
2. Nelson, M., Mateas, M.: Towards automated game design. In: Basili, R., Pazienza, M. (eds.) AI*IA 2007. LNCS (LNAI), vol. 4733, pp. 626–637. Springer, Heidelberg (2007)
3. Howard, T., Broughton, R.: Introducing clutter into virtual environments. Journal of Ubiquitous Computing and Intelligence (3) (2007)
4. Tutenel, T., Smelik, R.M., Bidarra, R., de Kraker, K.J.: Rule-based layout solving and its application to procedural interior generation. In: CASA Workshop on 3D Advanced Media In Gaming And Simulation (2009)
5. Doran, J., Parberry, I.: Controlled procedural terrain generation using software agents. IEEE Trans. Computational Intelligence and AI in Games 2(2), 111–119 (2010)
6. Taylor, J., Parberry, I.: Clutter (2010), `http://www.eng.unt.edu/ian/research/clutter/`
7. Petri, C.A.: Kommunikation mit Automaten. Schriften des IIM Nr. 2. Institut für Instrumentelle Mathematik, Bonn (1962)
8. Reisig, W.: Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets. Springer, Heidelberg (1998)
9. Murata, T.: Petri nets: Properties, analysis and applications. Proc. IEEE 77, 541–580 (1989)
10. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall, Englewood Cliffs (1981)