Computerized Clutter: How to Make a Virtual Room Look Lived-in

Joshua Taylor and Ian Parberry

Technical Report LARC-2010-01

Laboratory for Recreational Computing
Department of Computer Science & Engineering
University of North Texas
Denton, Texas, USA

April, 2010

UNIVERSITY OF NORTH TEXAS
Discover the power of ideas.

Computerized Clutter: How to Make a Virtual Room Look Lived-in

Joshua Taylor[*]
Ian Parberry[†]
Dept. of Computer Science & Engineering
University of North Texas

April 19, 2010

Abstract

We describe an algorithm for placing room contents including furniture and general clutter in a virtual environment according to design constraints specified in the form of a Petri net. We have implemented a prototype system and present images generated by it. The outputs of our prototype are believable, varied, random, yet controllable and structured room layouts suitable for video games.

Keywords: Procedural content creation, clutter, object placement, furniture, Petri net.

1 Introduction

Game content, including 3D models, bitmapped graphics, levels, textures, maps, and audio, is important to a variety of game genres (Forbus [5]). The demand for content is increasing to the point that its creation has become the most time-consuming and costly part of game development. Traditionally game content has been largely hand crafted by artists and designers. By having the computer take over some of the process of content generation, the time and money required can be reduced without compromising either the amount or quality of the results. This is called *procedural content generation* (see, for example, Roden and Parberry [20] and Nelson and Mateas [15]). In addition to taking some of the fiscal and temporal burden from game developers, procedural content generation can also open up new methods of increasing a game's replayability by incorporating the generators into the game itself. Prior work has shown the feasibility of procedural methods (see, for example, [6, 16, 23]) and there has been much interest in the procedural generation of a number of different types of game content, including clouds and stars [21], terrain [4], plants [3, 25], forests [2], cityscapes [10, 11], animation [17], levels [1, 22], and textures [14].

Procedural generation of room contents has not been well studied. We will use the term *clutter* to refer to any non-architectural elements of a room including furniture and other miscellaneous objects. Howard and Broughton [7] offer an offline method where the major furniture pieces are added by hand, and smaller pieces of clutter are added by a genetic algorithm. This is not particularly fast, and since the major furniture pieces are not part of the process, it is mainly a tool

[*]Author's address: Dept. of Computer Science & Engineering, Univ. of North Texas, 1155 Union Circle #311366, Denton, Texas 76203–5017, U.S.A. Email: `JoshuaTaylor@my.unt.edu`.

[†]Email: `ian@unt.edu`.

for the level designer. Tutenel *et al.* [24] offer a more complete solution that involves defining a set of tagged bounding boxes for each object and then using a constraint solver to place the objects. This approach is attractive since the bounding boxes are intuitive and visual enough for use by a professional designer, and the method appears fast enough to be used in game.

There are a number of desirable properties in any procedural content system. The most important of these are *novelty*, *structure*, *interest*, *speed*, and *controllability*:

- Novelty: The generated content should be recognizably different each time. Differences that are not noticed by the player are not novel.
- Structure: The content should not be merely random. It should have some structure recognizable to the player.
- Interest: The content should achieve a balance of structure and novelty that engages the players.
- Speed: The generator needs to run quickly enough. This includes both the time to generate one instance as well as the number of instances needed to find an interesting one.
- Controllability: The generator needs to be controlable enough that a designer can use it to make interesting content.

Novelty in generated room clutter means that rooms generated from the same constraint set should be sufficiently different from each other to be distinguishable.

While a procedural clutter generator should create unique outputs, which means using enough randomness to avoid duplicating identifiable features, at the same time this randomness should not lead to a loss of *structure*. Structure means that the room content makes sense and are internally consistent, as opposed to looking as if a random set of objects were thrown into a room.

Interesting content is subjective, but partially comes from a balance of novelty and structure. That is to say, novelty and structure are necessary but not sufficient conditions for a room to be interesting. Designers spend numerous hours learning what makes a room interesting, so rather than try to replicate that, this system aims to enable designers to use that knowledge.

The idea of the *speed* of a content generation system is somewhat like the definition of real-time, which is "fast enough that the results are ready when needed." One could imagine the contents of a room being generated while the door is opening to admit the player, say within a small number of animation frames.

Finally, any content generation system needs to be *controllable*. The system needs to allow the designers to express their ideas. Besides being flexible enough for those ideas to be possible, it also needs to be usable enough that the designer can actually implement them. Petri nets can be, and in this case are, Turing complete and can be used to enforce any arbitrary conditions the designer wants. Additionally, Petri nets can be built with a drag-and-drop interface that is often easier on non-programmers.

We present a procedural clutter generator based on Petri nets that we have designed to satisfy these five constraints. The remainder of this paper is divided into five sections. In the first section we introduce the concept of an *anchor*, and describe how anchors can be used to guide the placement of objects. The second section describes our Petri net language for describing room clutter constraints. The third section describes our implementation of this system, and the fourth section presents some resulting images of room clutter generated by it. The fifth and final section is a Conclusion that includes further work.

2 Anchors, Objects, and Collisions

A cluttered room is, except in extreme and perhaps pathological cases, not a completely random collection of objects. There are almost always patterns that emerge. In particular there tends to be a spatial organization with certain objects or sets of objects appearing in specific areas of a room. These objects tend to 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 relation to other objects.

We capture this intuition with the concept of an *anchor point*, which marks a place at which an object can be (approximately) located. An anchor point is represented as a position and orientation. Throughout the rest of this paper, although our examples will be in 2D for simplicity, the descriptions will explain where 2D and 3D would differ. In 2D the anchors would be in the corners, spaced along the edges, and spread out in a grid over the center of the room. The spacing along the edges and the size of the grid depend on what type of room is being generated, for example, the desks in a classroom and the tables in a dining room would be placed using different spacings. Additionally, in 3D there can be anchors on the wall and ceiling.

Since just one level of objects is not enough to clutter up a room, each object also has a set of anchor points for further clutter. For example, a table may have anchor points for the chairs to be placed around it, as well as 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, which almost never happens in the real world but can often be seen in virtual worlds, 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. Objects in the corner would normally have a standard deviation in both position and orientation of 0, while a pile of paper in the center of the table might have a standard deviation of position of a foot or so and any orientation. This is specified as late as possible so that it can be changed for a piece of paper on a book versus a piece of paper in the middle of a table.

In addition to individual objects, each object can be classified into one or more categories. The system can then choose one of a list of objects to represent each category needed by the current room. This would give the room a more cohesive feel. If having just one object per category is too restrictive, it is possible to have it pick multiple objects based on some measure of how well they go together or to mark certain categories as open, allowing it to pick anything from that category.

If the objects are placed randomly, it is very likely that some of them will end up colliding. In 3D, collision detection would be done in the normal way, but in 2D some work has to be done to allow a chair to be under a table, or a plate to be on it. One refinement, taken from [24], is to change the bounding volume of objects to include space the object needs to function, such as the room in front of a drawer, or behind a chair.

When an object collides with something that has already been placed in the room, it needs to be moved. The simplest way to do this is to generate new displacements for the position and orientation and try again. There has to be a limit to how many times this can be tried, as the object might not be able to be placed at all. In most cases, it is safe to throw the object away as many can appear in multiple places. Two cases in specific make this approach somewhat lacking though. First, things like chairs around a table look odd when one is missing. Second, some items may be considered important and could fail to generate. In both cases, there are partial workarounds that can lessen the chances of this happening and both cases can be detected by a post-processing step as uninteresting.

3 Petri Nets

Petri nets date from 1939 and were originally used to describe chemical processes [18], but they have found a large number of applications since then, including distributed computing [19] and manufacturing [13].

A *Petri net* (Petri [18]) consists of a directed bipartite graph and a set of *tokens*. The nodes in one partition of the graph are called *places* and the nodes in the other partition are called *transitions*. Tokens can be put on the places, and the transitions describe how the tokens can move around the graph. A transition is called *live* if each of the places pointing to it contain at least one token. Each step, one live transition is chosen to fire, which removes one token from every place pointing to it, and adds one to each place it points to.

In standard Petri nets, tokens are completely indistinguishable from each other. The only information they carry is where they are and how many of them are there. In a colored Petri net (see, for example, Jensen [9]), the tokens carry additional information. We will use tokens to carry the information needed to place objects in a room. Although it seems natural for the tokens to represent objects, we will perhaps counterintuitively use the tokens in the Petri net to represent anchor points. In addition, some generic tokens with no anchor point will be added manually to aid in bookkeeping.

The transitions in our colored Petri nets differ slightly from normal Petri net transitions in three significant ways. Firstly, since the tokens are carrying important information, the relation between incoming edges and outgoing edges 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. This is a simplified version of the standard approach to colored Petri nets, but it makes more sense in this context.

Secondly, for each incoming edge the user can specify which types of tokens to allow on that edge. This is handled by attaching a nonunique name to each anchor point and then checking those names when deciding if the transition is ready to fire. This allows objects to be placed in corners, but not along edges, while all of the tokens were together in one place. This is also a simplification of the guard expressions used in standard colored Petri nets.

Thirdly and finally, each incoming edge can create an object at the anchor point represented by the token it is working with. Since that anchor point would then be occupied, it is no longer valid to keep a token for it. Instead, all of the anchor points on the object being created become new tokens and are passed along to the appropriate outgoing edge.

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. This probability is how likely it is to actually fire when picked and defaults to 1 (which means it will fire if picked). This allows the user to make very low probability events without making hundreds of duplicated transitions for everything else, or doing some other gymnastics with the shape of the net.

One way of simplifying certain common tasks is with *prioritized Petri nets*, in which a priority is attached to each transition. This would allow the user to say that this object must be placed as soon as possible without worrying about other objects taking up the available anchors. It is possible to do this without priorities, using extra places and transitions, but in many case, it would be a significant complication to do so.

Another important addition is the inclusion of inhibitor edges. These are edges from a place to a transition, but they do not move tokens. Instead, if the place holds an appropriate token, the

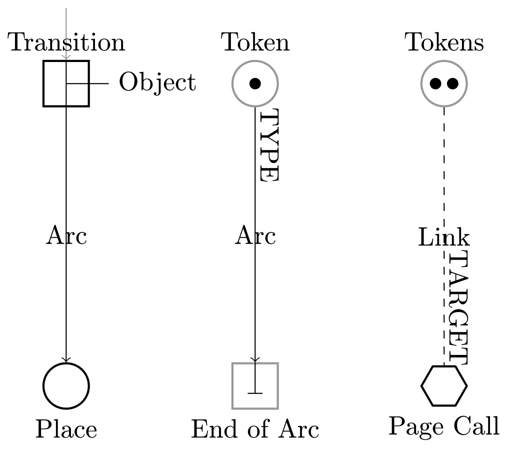

Figure 1: Legend for Petri nets. Edges are often called arcs in other literature on Petri nets and graphs.

inhibitor edge prevents the transition from firing. Petri nets with inhibitor edges are well known to be Turing complete, that is, they compute exactly the set of computable functions.

To enable the designer to organize and modularize the design, the Petri net is broken into pages. This is often called a *hierarchical Petri net* (see, for example, Huber, Jensen, and Shapiro [8]). The version used here allows the pages to be easily unrolled into one large page at run time for ease of execution.

To provide the widest range of possibilities, each place and transition is assigned a scope. In the simplest case, this would be either local or global, but other scopes such as room or page are useful options. Nearly all places and transitions are local, but having a global scope allows the user to, for example, enforce the uniqueness of an item even if it could be created in multiple pages.

Figure 1 shows the different parts of a Petri net. Circles are places and rectangles are transitions. Dots within circles are generic tokens. Labels on edges from places to transitions represent restrictions on the types of tokens that can move along that edge. Labels on the dotted linkages from places to page calls designate which place in the other page to link with. The labels on the parts of the edges within the transitions show which object, if any, to create when that transition fires. Figure 2 shows a small example of a Petri nets for a table.

The anchor tokens for a room are initially put in the START place. In this case, only the CENTER anchors can move on, leaving the EDGE and CORNER tokens in START. The first transition also requires a token from the TableControl place. Since there is only one such token, only one table can be created.

In the process of firing, the CENTER token is used to place a table. After the table is created, all of the anchor points that were part of it will be made into tokens and placed in the TableStuff place. From there, the ChairSpot anchors are used to place chairs around the tables.

4 Implementation

We implemented a prototype of this system in 2D in Java. The room and the initial anchor points are prepared by hand, but the system is designed to be usable along side a room generator (for example, [12]).

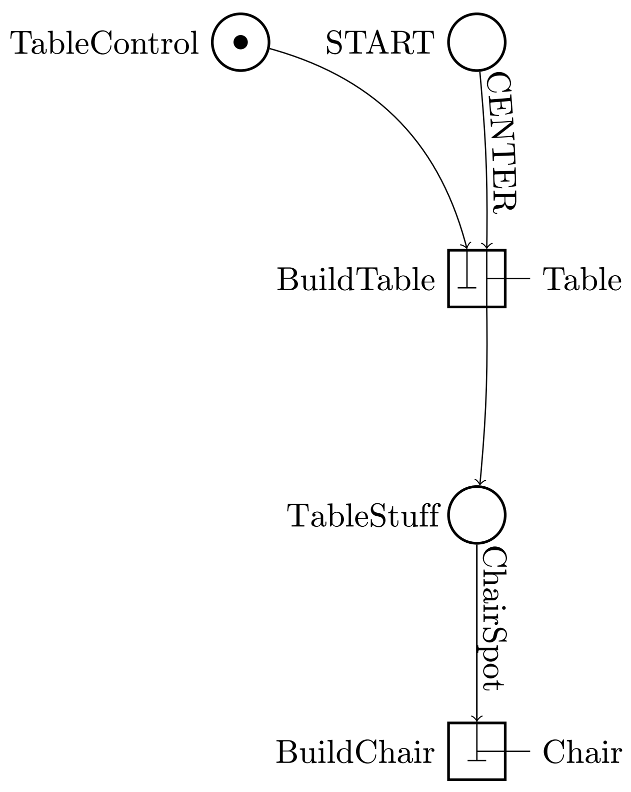

Figure 2: A small petri net for a table and chairs.

The list of objects and the Petri net are stored in two XML files. Table 1 summarizes the tags for the Petri net itself and Table 2 summarizes the tags for the list of objects. Table 3 gives sample XML for the first page of the Petri net and for one of the table objects.

The Petri net is broken into pages that contain places along with any extra tokens, transitions with details on what types are allowed and what objects to create, and calls to other pages and the links for those calls. The deviation in orientation and position are only specified inside the transitions so that a cup fallen on the ground can scatter further than one still sitting on the table or a rug sitting on the floor.

The object XML is simply a list of objects where each object tag provides such details as the image or images, its origin, its bounding volume (only bounding boxes are used here), a tag or list of tags for use with the category system and a list of anchor points. If multiple images are provided, they are used together as an animation.

The system reads everything in, then uses that to unroll everything into a single page stored in an easily executable format. Since the pages need to be unrolled, cycles in the page call graph are forbidden. Links, subtype of place, are introduced to make sure that unrolling works out nicely. Links are treated like any other place except that they can only be locally scoped and they cannot have generic tokens to begin with. All page calls connect places in the current page to links in the called page, which are merged when that page call is unrolled.

Finally, the anchors from the room are fed into the starting place and the net is executed. Execution consists of making a list of live transitions and firing one randomly. If specified by the user, some transitions may have a chance of not firing even after being picked. This continues until

XML root	`<net>...</net>`

Pages	`<net>`
	`<page name="START">...</page>`
	`</net>`

Places
tokens: how many generic tokens are here. Defaults to 0.
	`<page ...>`
	`<place name="Somewhere" scope="Global" tokens="2" />`
	`</page>`

Links	`<page ...>`
	`<link name="SomeLink" />`
	`</page>`

Transitions
p: probability that this will fire when picked. Defaults to 1.
	`<page ...>`
	`<trans name="DoSomething" scope="global"`
	`p="0.7">...</trans>`
	`</page>`

Edges/Arcs
in/out: names of places to connect to. Defaults to nothing.
type: name of a type of anchor. Defaults to any type.
object: name of a category of objects. Defaults to no object. *sdp/sda*: standard deviation of the position and angle of the object created by this edge. Defaults to 0.
	`<trans ...>`
	`<edge in="From" out="To" type="CENTER" object="Table"`
	`sdp="3" sda="5" />`
	`</trans>`

Page Calls
target: name of a page.
	`<page ...>`
	`<call target="SomePage">...</call>`
	`</page>`

Call Associations
link: name of a link in the target page.
place: name of a place in the calling page.
	`<call ...>`
	`<assoc link="SomeLink" place="Somewhere" />`
	`</call>`

Table 1: XML Tags and attributes for the Petri nets

there are no live transitions, either due to using up the available tokens or having the tokens end up in a place where they cannot be used.

5 Results

We constructed a Petri net for a sample room design and used it to generate some clutter for differently shaped rooms. Figure 3 shows the first page of the Petri net used to generate the full room pictures below. Figures 4 and 5 show some room layouts generated over multiple runs of the same Petri net for two different room shapes. Figures 6 show some room layouts using a slightly different Petri net from Figures 4 and 5 allowing multiple tables but no TV area. The anchor points for each room are shown in Figure 7.

Judging our approach by the criteria mentioned in the Introduction as being desirable traits for procedural content generation, 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, which makes the room look like it has been used by people instead of laid out by computer or just randomly jumbled together, thus creating interest. As evidence we provide the pictures in Figure 4. Our approach is comparable in speed to that of Tutenel *et al.* [24]. The biggest issue remaining is controllability, which we will address in the next section.

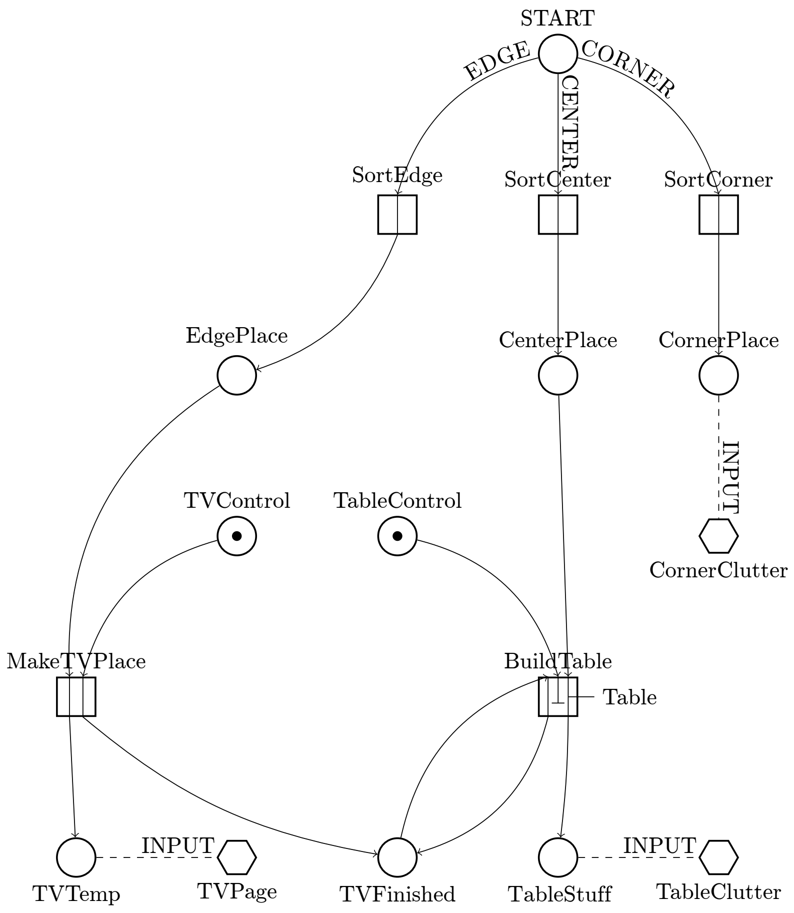

Figure 3: First page of a Petri net for a sample room.

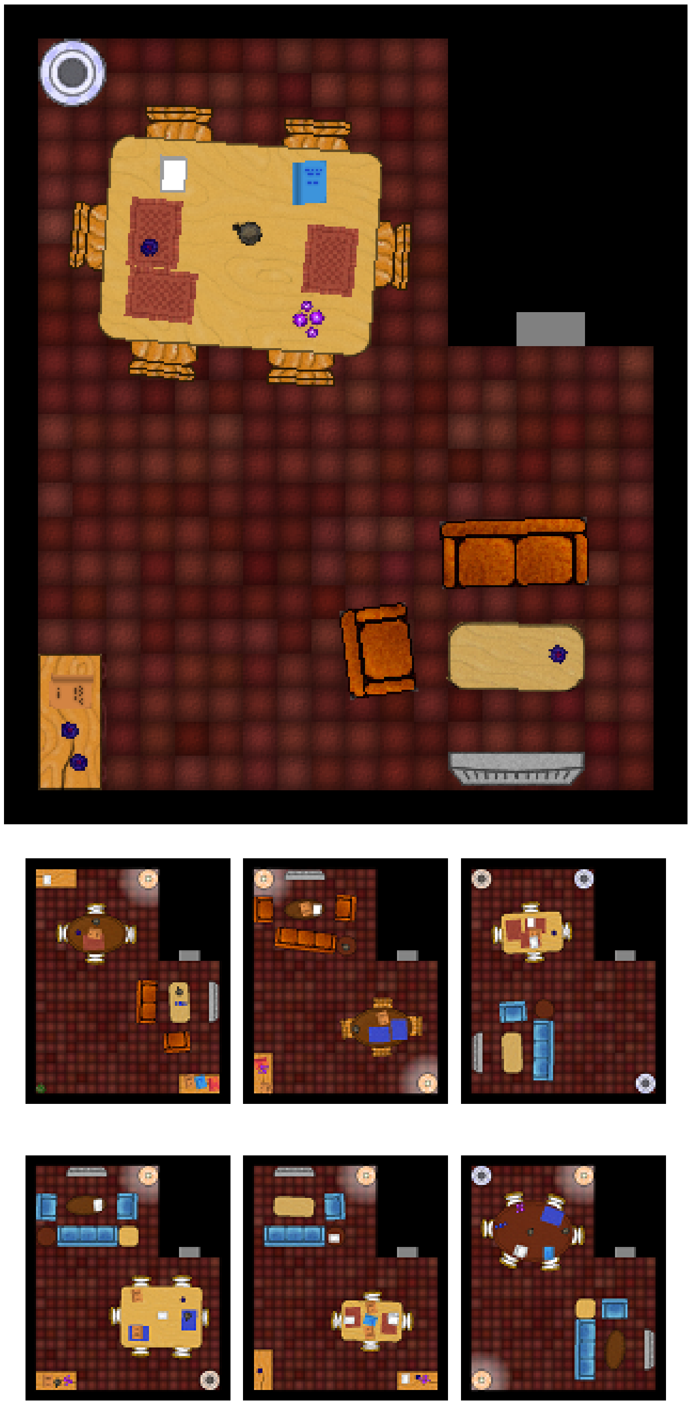

Figure 4: Some clutter generated by our system for an L-shaped room.

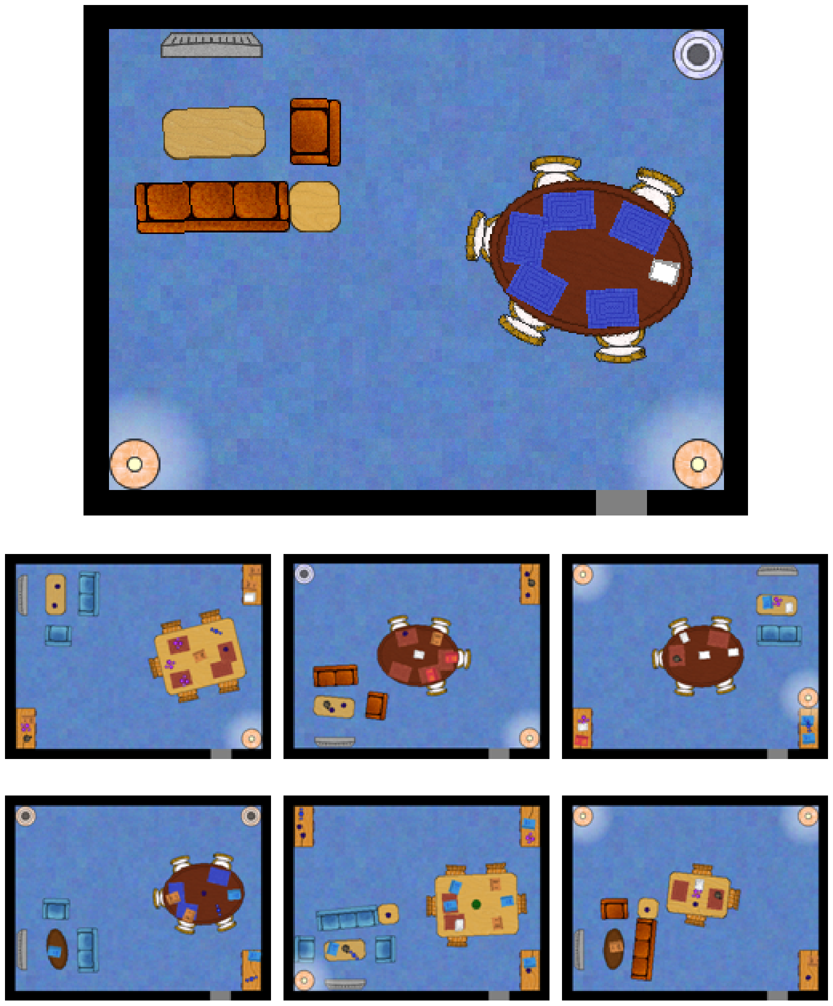
Figure 5: Some clutter generated by our system for a rectangular room using same Petri net as Figure 4.

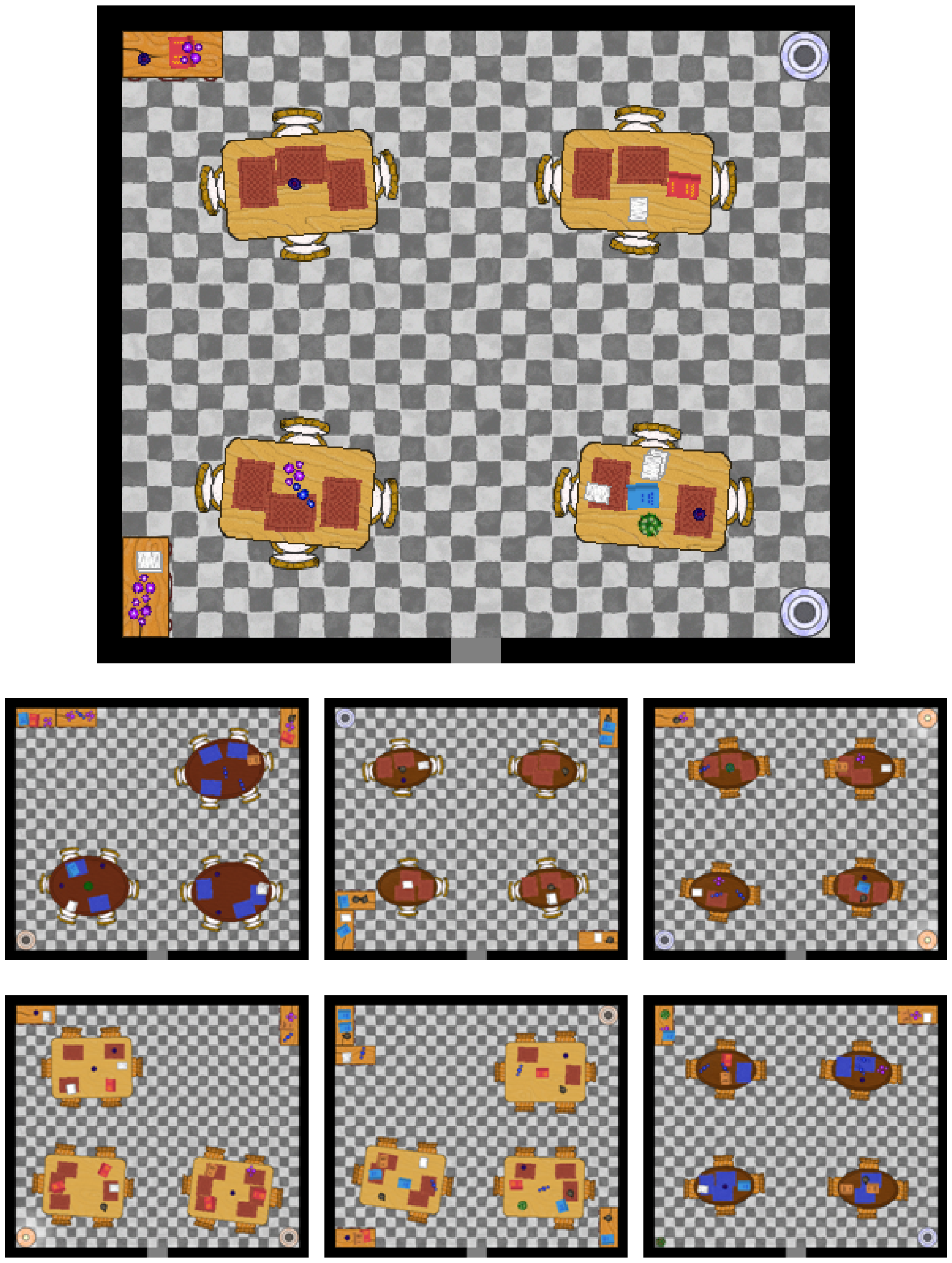

Figure 6: Some clutter generated by our system using slightly different Petri net from Figures 4 and 5 allowing multiple tables but no TV area.

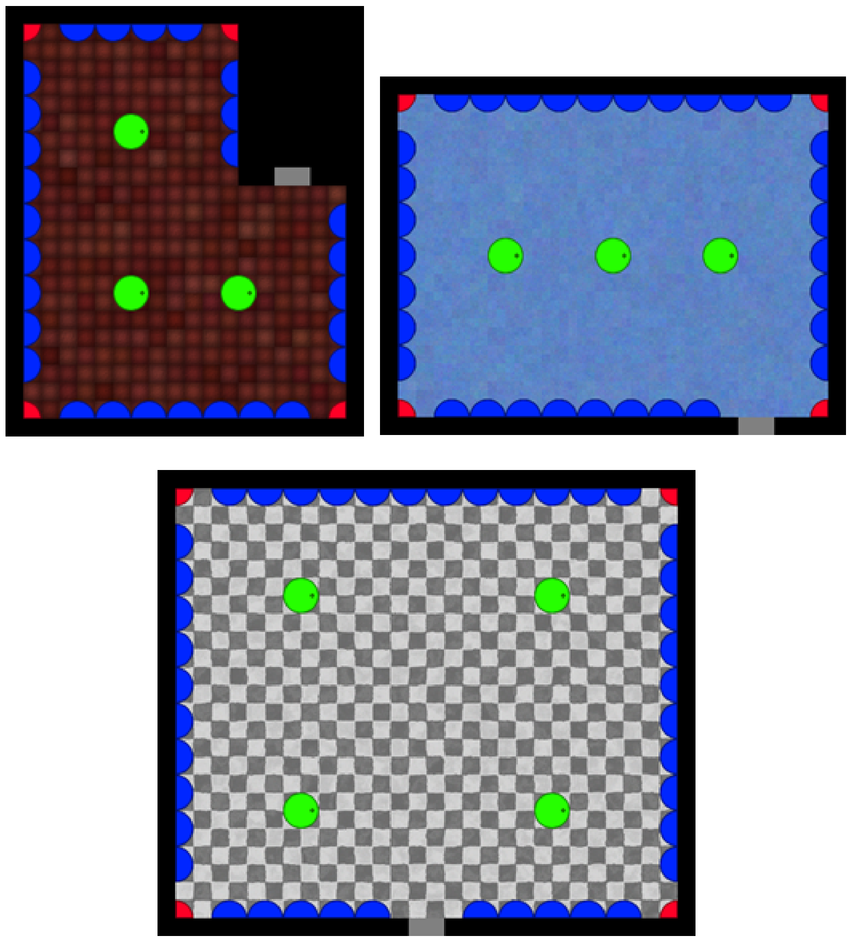

Figure 7: A visual representation of the anchor points in the rooms. Circles in the center of the rooms are center anchors, semicircles along the edges are wall anchors, and quarter-circles at the corners are corner anchors.

XML root	`<objectSet>...</objectSet>`

Objects	`<objectSet>`
	`<object name="Table1">...</object>`
	`</objectSet>`

Animation Frames
x/y: the origin of this frame. Default to 0.
	`<object ...>`
	`<frame filename="Table1.png" x="32" y="24" />`
	`</object>`

Anchors
x/y/a: the location and orientation of the anchor relative to the object's origin. Defaults to 0.
depth: (2D only) shows which objects to draw in front or in back.
	`<object ...>`
	`<anchor name="ChairAnchor" x="16" y="-16" a="90" depth="100" />`
	`</object>`

Bounding Box
x/y: the location of the corner of the bounding box relative to the object's origin. Defaults to 0.
w/h: the width and height of the bounding box.
	`<object ...>`
	`<bounds x="-16" y="-12" w="32" h="24" />`
	`</object>`

Tags	`<object ...>`
	`<tag name="Table" />`
	`</object>`

Table 2: XML Tags and attributes for the object sets

6 Conclusions and Future Work

This paper introduces two new concepts for the design of a procedural clutter generator, anchor points and Petri nets. The use of anchor points seems to allow for a natural placement of objects not restricted to certain positions or angles. The modified Petri net presented above offers a useful intermediate code for placing objects at the anchor points.

As mentioned in the previous section, the biggest issue remaining is controllability. One could argue that a Petri net is not convenient for a designer even if a drag-and-drop interface is provided. We believe that designers could easily be trained to use it however, and in any case the Petri net could serve as a form of intermediate code for some yet-to-be-designed used interface. The construction of such an interface and a usability study remain as further work.

References

[1] Kate Compton and Michael Mateas. Procedural level design for platform games. In *Proceedings of the 2nd Artificial Intelligence and Interactive Digital Entertainment Conference*, 2006.

[2] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 275–286, New York, NY, USA, 1998. ACM.

[3] Oliver Deussen and Bernd Lintermann. A modelling method and user interface for creating plants. In *Proceedings of the Conference on Graphics Interface '97*, pages 189–197, Toronto, Ont., Canada, Canada, 1997. Canadian Information Processing Society.

```xml
<net>
  <page name="START">
    <place name="START" />
    <place name="EdgePlace" /> <place name="CornerPlace" /> <place name="CenterPlace" />
    <place name="TVControl" tokens="1" scope="Global" />
    <place name="TVFinished" scope="Global" />
    <place name="TableControl" tokens="1" />
    <place name="TableStuff" />
    <place name="TVTemp" />

    <trans name="SortEdge">
      <edge in="START" out="EdgePlace" type="EDGE" />
    </trans>
    <trans name="SortCorner">
      <edge in="START" out="CornerPlace" type="CORNER" />
    </trans>
    <trans name="SortCenter">
      <edge in="START" out="CenterPlace" type="CENTER" />
    </trans>
    <trans name="BuildTable">
      <edge in="TableControl" />
      <edge in="TVFinished" out="TVFinished" />
      <edge in="CenterPlace" out="TableStuff" object="Table" sdp="5" sda="5" />
    </trans>
    <trans name="MakeTVPlace">
      <edge in="TVControl" out="TVFinished" />
      <edge in="EdgePlace" out="TVTemp" />
    </trans>

    <call target="CornerClutter">
      <assoc link="INPUT" place="CornerPlace" />
    </call>
    <call target="TableClutter">
      <assoc link="INPUT" place="TableStuff" />
    </call>
    <call target="TVPage">
      <assoc link="INPUT" place="TVTemp" />
    </call>
  </page>
  ...
</net>
```

```xml
<objectSet>
  <object name="Table1" depth="-100">
    <frame filename="..\Tiles\Objects2\Table1.png" x="48" y="32" />
    <bounds x="-47" y="-31" w="94" h="62" />

    <anchor name="Chair" x="-48" y="0" a="0" />  <anchor name="Chair" x="0" y="-32" a="90" />
    <anchor name="Chair" x="0" y="32" a="270" /> <anchor name="Chair" x="48" y="0" a="180" />
    <anchor name="Top" x="-32" y="0" a="180" />  <anchor name="Top" x="0" y="-16" a="270" />
    <anchor name="TopCenter" x="0" y="0" a="0" />
    <anchor name="Top" x="0" y="16" a="90" />    <anchor name="Top" x="32" y="0" a="0" />

    <tag name="Table" />
  </object>
  ...
</objectSet>
```

Table 3: Sample XML code for the first page of the Petri net used in Figures 4 and 5, and for one of the table objects.

[4] J. Doran and I. Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, To Appear.

[5] KD Forbus, JV Mahoney, and K. Dill. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17(4):25–30, 2002.

[6] S. Greuter, N. Stewart, and G. Leach. Beyond the horizon. *Image Text and Sound Conference*, 2004.

[7] T.L.J. Howard and R. Broughton. Introducing clutter into virtual environments. *Journal of Ubiquitous Computing and Intelligence*, (3), 2007.

[8] P. Huber, K. Jensen, and R. Shapiro. Hierarchies in coloured petri nets. In *Lecture Notes in Computer Science*, volume 483, pages 313–341, 1991.

[9] K. Jensen. Coloured petri nets. In *Lecture Notes in Computer Science*, volume 254, pages 248–299, 1987.

[10] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. In *Proc. Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

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

[12] J. Martin. Procedural house generation: A method for dynamically generating floor plans. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, 2006.

[13] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77:541–580, 1989.

[14] F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and modeling: A procedural approach.* Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[15] M.J. Nelson and M. Mateas. Towards Automated Game Design. *Lecture Notes in Computer Science*, 4733:626, 2007.

[16] M. Nitsche, C. Ashmore, W. Hankinson, R. Fitzpatrick, J. Kelly, and K. Margenau. Designing Procedural Game Spaces: A Case Study. *Proceedings of FuturePlay*, pages 10–12, 2006.

[17] Ken Perlin and Athomas Goldberg. Improv: a system for scripting interactive actors in virtual worlds. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer graphics and Interactive Techniques*, pages 205–216, New York, NY, USA, 1996. ACM.

[18] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[19] W. Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998.

[20] T. Roden and I. Parberry. From artistry to automation: A structured methodology for procedural content creation. In *Proceedings of the 3rd International Conference on Entertainment Computing*, pages 151–156, 2004.

[21] T. Roden and I. Parberry. Clouds and stars: Efficient real-time procedural sky rendering using 3D hardware. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, pages 434–437, 2005.

[22] T. Roden and I. Parberry. Procedural level generation. In *Game Programming Gems 5*, pages 579–588. Charles River Media, 2005.

[23] J. Togelius, R. De Nardi, and S.M. Lucas. Towards automatic personalised content creation for racing games. *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[24] Tim Tutenel, Ruben M. Smelik, Rafael Bidarra, and Klaas Jan de Kraker. Rule-based layout solving and its application to procedural interior generation. *CASA Workshop on 3D Advanced Media In Gaming And Simulation*, 2009.

[25] Jason Weber and Joseph Penn. Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pages 119–128, New York, NY, USA, 1995. ACM.