Rule-based layout solving and its application to procedural interior generation[*]

Tim Tutenel and Rafael Bidarra
Delft University of Technology
Delft, The Netherlands
timt@graphics.tudelft.nl, r.bidarra@ewi.tudelft.nl

Ruben M. Smelik and Klaas Jan de Kraker
TNO Defence, Security and Safety
The Hague, The Netherlands
{ruben.smelik|klaas_jan.dekraker}@tno.nl

Abstract

Due to the recent advancement in procedural generation techniques, games are presenting players with ever growing cities and terrains to explore. However most sandbox-style games situated in cities, do not allow players to wander into buildings. In past research, space planning techniques have already been utilized to generate suitable layouts for both building floor plans and room layouts. We introduce a novel rule-based layout solving approach, especially suited for use in conjunction with procedural generation methods. We show how this solving approach can be used for procedural generation by providing the solver with a user-defined plan. In this plan, users can specify objects to be placed as instances of classes, which in turn contain rules about how instances should be placed. This approach gives us the opportunity to use our generic solver in different procedural generation scenarios. In this paper, we will illustrate mainly with interior generation examples.

Keywords: Procedural generation; constraint solving

1 Introduction

Many recent games that play in an urban setting feature huge cities, e.g. "Grand Theft Auto IV" (2008) or "Assassin's Creed" (2007). The player is however limited to entering only a handful of the many hundreds of buildings. It would obviously take too long for designers to model by hand the interiors of all these buildings with rooms and furniture, but with current procedural generation techniques, the interiors of these buildings could have been generated automatically.

This would not only increase the perceived realism of the game, more importantly, it would allow for new gameplay. The player could break into buildings to hide from enemies or to find food or other items like first-aid kits. It could also make chase sequences more interesting, and give players even more to explore in sandbox-style games.

Clearly, the difference in quality between hand-designed and procedurally generated content in the game world should not be too noticeable. In comparison to manually designed content, current procedural content can look dull and repetitive. However, there are several characteristics of building interiors that make them a suitable candidate for automated techniques. For example, often the interiors in a common house roughly follow the same structure. In a kitchen, we often see cabinets and counter being placed against the walls, with a table or perhaps an extension to the counter as an island in the middle of the room. Similarly, in a living room the couches are often placed around a small table and oriented towards the television set. Many such observations could be translated into rules and procedures to automatically generate these interior spaces.

Moreover, research in solving of space layout problems aimed at room interiors has already generated promising results. Several methods of solving different kinds of layout constraints among objects inside a room have been proposed, as we will show in the next section. We developed a rule-based layout solver, which is especially suited for procedural methods: based on a plan or a procedure, objects (e.g. furniture) are fed to the solver, which tries to fit them based on a set of rules defined for those objects and the ones that are already placed. We also define object features to steer the layout (e.g. areas around an object that should remain empty) or to link them to other objects. Furthermore, our approach was designed to be integrated with a comprehensive semantic class library which is explained in Tutenel, et al. (2009). In this paper, however, we will only briefly describe how these semantic classes

[*] This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie).

are used in our method. A more detailed discussion about how semantics can play an important role in the design of game worlds can be found in Tutenel, et al. (2008).

We first present the general idea of our rule-based layout solver in Section 3, after which we go into more detail about the layout planner which feeds the solver in Section 4. Finally, in Section 5, we show several examples where the solver is used to generate different layout problems.

2 Related work

Research on procedural generation of content suitable for game worlds has focused on many different aspects, including a variety of techniques that generate height maps for terrains or models for vegetation. A recent survey of procedural methods can be found in Smelik, et al. (2009).

Because of the characteristics of room interiors, we focus here on buildings and floor plan generation examples. We mentioned before that, in almost every room that has a specific function, patterns are visible. Many of these patterns are also hierarchical in nature: chairs are placed around the table, plates on top of the table and the cutlery, in turn, is placed next to the plates. A similar observation is made with respect to the decomposition of buildings and their facades. A wall contains a door and windows and those windows consist of a windowsill, the frame of the window and the glass inside that frame. Many algorithms based on this kind of decomposition supply shape grammars to generate the buildings and facades. See for example, the work of Wonka, et al. (2003), Yong, et al. (2004), Müller, Vereenooghe, et al. (2006), Müller, Wonka, et al. (2006) and Larive and Gaildrat (2006).

In floor plan generation methods we see the notion of shape grammars come up as well. In Rau-Chaplin, et al. (1996) and Rau-Chaplin and Smedley (1997), a shape grammar is presented to layout the different areas in a house. When this process is finished, these areas are assigned a function. A different method to automatically creating floor plans is proposed in Martin (2006). First a graph is generated in which every node represents a room and every edge corresponds to a connection between rooms. Next, these nodes are given a center location and the rooms are formed using growth rules.

An important advantage of procedurally generating building interiors is shown in Hahn, et al. (2006). Their research focused on generating only the rooms that are visible from the current viewpoint. This is obviously an efficient way of handling large buildings with many different rooms (e.g. office sky scrapers). To maintain changes made in the world, all changes are tracked and stored. When a room is removed from memory at one point, and is regenerated later on, the stored changes are again applied to the regenerated room.

Layout solving based on object rules is also applied in manual scene editing systems. In Xu, et al. (2002), objects contain rules describing which type of objects their surface supports. For example, food, plates or cups can be supported by a table or a counter. Smith, et al. (2001) use similar links, but they apply them to areas. They define offer and binding areas between objects, e.g. the area underneath a table can be an offer area that is linked to the binding area of a chair. In the WordsEye system, described in Coyne and Sproat (2001), natural language sentences are transformed to a list of objects with a set of constraints, based on which a scene is generated.

Declarative modeling of virtual environments (see Gaildrat (2007), Le Roux, et al. (2001) and Le Roux, et al. (2004)) combines constraints and semantic knowledge in the form of implicit constraints, to help the user generate a scene. In the description phase, the designer can express how a scene should look like. These descriptions are translated into constraints that are then fed to some constraint solver. Our layout solver uses a similar workflow: it uses rules defined for objects to come up with a set of constraints for to the solver.

A number of constraint solving techniques have already been researched to create room layouts in the form of space planning problems. Charman (1993) gives an overview of how existing constraint solving techniques that are not specifically focused on space planning can be applied to these problems. He discusses the efficiency of the solving techniques and compares several space planners. Many improvements for the discussed constraint solving techniques have been researched in the years following this study, so the results concerning the efficiency are no longer relevant. The discussed techniques, with their recent improvements, are however still applicable to layout solving. The planner he proposed, steered by the conclusions from his study, works with axis-aligned 2D rectangles with variable position, orientation and dimensions. Users can express geometric constraints on these parameters, which can be combined with logical and numerical operators. While our approach has similarities in the actual solving method, i.e. they both express the possible positions of an object as a union of areas; we propose different ways of expressing rules between the objects. While direct geometric constraints can still be expressed in our system, we allow more freedom to what these constraints can relate to and allow more indirect approaches through the use of features which will be explained in Subsection 3.1.

In the past, several space planning methods were developed using constraint logic programming (CLP). Even some decades ago, this approach was researched (see: Pfefferkorn (1975) and Honda and Mizoguchi (1995)). A more recent system that used CLP was created by Calderon, et al. (2003). It is a framework that generates a number of different layout solutions for a number of objects, through which the user of the framework can interactively find desirable solutions. The rules for the objects are all expressed in predicate logic statements. This gives the opportunity to provide users with more or less natural language-like rules. In our approach we tend to work to a more visual solution, which we think might be more suitable for designers of game worlds.

3 A RULE-BASED LAYOUT SOLVING APPROACH

The main idea of our approach can be summarized as follows: given a starting layout, find the possible locations of a new object, based on a set of rules for that new object and objects in the layout. The relationships between objects can be defined in two ways: the explicit way, for example defining that the sofa needs to face the TV and that it should be no more than five meters away from it; and the implicit way, through the use of features, which we will explain next. An important aspect of our approach is the use of hierarchical blocks in the solving process. When placing a table with some chairs around it and a couple of plates on top, these objects are combined and treated as one block. This way, the solving process is made more efficient.

3.1 FEATURE-BASED CLASS REPRESENTATIONS

In our solving approach, each class can define a geometric representation valid for all instances of that class. This geometric representation consists of a number of so-called object features, which are 3D shapes containing a tag. These tags can refer to specific feature types. For a feature type, rules are defined about which features can and cannot overlap with them. For example, the *OffLimit* features cannot overlap any other features, e.g. the solid parts of objects (usually the entire bounding box). The *Clearance* feature denotes that this area of the object needs to be kept free e.g. for a person to walk or to use the object like the area in front of a closet. These features cannot overlap with other features, except other *Clearance* features.

As an example, a geometric representation for a table consists of five *OffLimit* features, corresponding to the four legs and the table top. The top feature can be assigned a *TableTop* tag. This way we can define that, for example, objects like a cup or a plate should be placed on a feature with the *TableTop* tag. The geometric representation is defined for a specific class, in this example the *Table* class. The shapes of the features are defined relative to the object, e.g. on top, or to the left of the object. This allows designers to link 3D models of different sized tables to the same *Table* class, and the five features will automatically be added to the models. It is however necessary that the models are uniformly oriented to enable the system to handle the relative descriptions of the feature shapes; e.g. when defined that we need a *Clearance* feature *in front of* every closet, the system will not generate the features correctly if a model with the wrong orientation is used.

Another example of how one can use features to position objects is the common case that an object should be placed against the wall. When creating the starting state for laying out a room, *Wall* features are added to the solver on the room's walls. By expressing that an object needs to be located with its back against such a feature, this rule can be enforced.

3.2 CLASS RELATIONSHIPS

In the previous section, we explained how we can implicitly add relationships between objects with the help of features. More detailed relationships can be defined in the rules that are used by the solving mechanism. These rules can be specified in two ways. They can be associated with a class, which will add the rule to every instance of that class, but the rule can also be defined in the layout planner (see Section 4), which gives the opportunity to define rules to objects that are not generally applicable. For example, a chair in a waiting room is generally placed against the wall, so this rule is specified only in the waiting room plan.

When creating the rules, one can express object relationships in a number of ways. First, we can create a direct link with an already placed object in the layout. This is only useful when creating a layout plan (see Section 4), because in a general class rule you cannot be sure which objects are already placed. The second way of expressing a relationship, which can be used in a general definition, is by linking an object to objects belonging to a specific class. We could, for example, define as a rule of the *Sofa* class, that when there is an instance of the *TV* class present in the room, a sofa should be facing that TV.

It is clear that in this approach a hierarchical relationship is created between the objects in a layout. For a designer, most of these hierarchies are clear, so he or she can make use of this knowledge by creating sub-plans. An interesting example of this is an office desk setup. A number of objects like an office chair, a telephone or a computer are all positioned relative to the desk. It is therefore practical to define a custom plan for creating such a desk layout. Not only is it easier to reuse already created plans, it also provides an opportunity to speed up the solving process. Instead of placing the desk and all its related objects immediately, we can first place a *Desk* feature in the layout. This *Desk* feature could contain sub-features to guarantee clearance areas for example; still it will definitely be faster than individually placing the desk and all its related objects. After a suitable layout for an office space is generated using these *Desk* features, the features can then be replaced by all its related objects. This has the advantage that the placement of these related objects has become a sub-problem, for which it is faster to generate a solution.

3.3 SOLVING MECHANISM

In this section we describe how the solving approach works. First, we find all possible locations for a new object, based on the *ground* type of the object, its features, and the features of the already placed objects in the current layout. This *ground* type is the feature type on which the object can be placed. This could be a *TableTop* for a cup, a *Floor* for a table or a *Counter* for a kitchen sink. All these features have a shape which makes up the basic location for the new object. Above, we mentioned that *OffLimit* and *Clearance* features have a special meaning in this phase of the solving procedure. Based on the features of these types, unwanted overlaps are trimmed from the found locations. To allow for different orientations of the object, we perform this procedure for a discrete set of angles. In our system we use Minkowski addition to calculate the unwanted areas: when a feature of the new object should not overlap with an already placed feature, the Minkowski sum of the already placed feature and the new feature is trimmed from the possible locations. Based on the possible locations defined by the rules connected to the object, the list of possible locations is further refined. This mechanism is further illustrated here in pseudo code. Based on the input of a new object and the current layout, this algorithm creates a list of possible locations for the new object:

```
//--- Creating the list of possible locations of the new object based on the object's
//--- features and the features already placed in the current layout
//--- We start off with locations of the ground type features available in the layout
possibleLocList = currentLayout.GetFeatureLocationsOfType(newObject.GroundType)

//--- Now we prune this list of possible locations based on the overlap rules of the
//--- features in the new object and the already placed features in the current layout
for each objFeature in newObject
{
   //--- Each feature in the current layout that cannot overlap with the currently assessed
   //--- feature is subtracted from the possibleLocList
   for each layoutFeature in currentLayout
   {
      if ( ! objectFeature.OverlapAllowedWith(layoutFeature) )
      {
         //--- Using the Minkowski Sum, we create an area that contains all locations for
         //--- which the object feature would overlap with the layout feature
         illegalAreaAroundFeature = (layoutFeature.Shape).MinkowksiSum(objFeature.Shape)

         //--- Now we subtract this area from the list of possible locations
         possibleLocList = possibleLocList.Minus(illegalAreaAroundFeature)
      }
   }
}

//--- The possible location list is intersected with the possible locations based on each
//--- individual rule for the new object
for each rule in newObject
{
   ruleLocationList = rule.CreatePossibleLocations()
   possibleLocList = possibleLocList.Intersection(ruleLocationList)
}
```

We have two kinds of rules that can be handled by the solver. The *area*-based rules define a possible placement area for an object. This could be an area next to a feature or an object, on top of an object, etc. These are handled first by the solver and the intersection of these areas and the already found areas are now the provisional possible locations for the new object. Next, the so-called *grid*-based rules are handled. The current list of possible areas is cut into a grid of smaller areas, of which the size can be set inside the rules (when placing a table in a room, we can use larger grid sizes than when placing a fork on the table). For the center point of these areas a list of geometric constraints can be evaluated, ranging from a required distance to a certain object or feature, whether another object is visible from that area, etc. The areas for which these constraints do not hold are discarded.

This black-or-white approach is not always desirable. We want to be able to define that an object of a particular class attracts or detracts objects of another class. For this we use attractors and detractors. These assign weights for the possible placement areas found previously. These weights will deem some locations as unlikely but not completely invalid for a specific object. In the final step of the process, we first pick an area based on the weights and subsequently we pick a random location within that area.

The same approach can be applied when designing a room by hand. When the designer wants to add a new object in a layout, the possible locations can be shown as a guide. Another possibility is to snap to a valid position closest to where the designer dragged the new object. This makes our solving approach suitable for both manual and automatic layout systems.

Performance is always an important issue for any solving approach. However, a generic approach will not be able to take advantage of many optimizations available for more specialized solving methods. In the algorithm above, it becomes clear that the number of features available in the layout could create an important bottleneck for the performance. Every time an object is added to the layout, all its features are added as well and the pruning of possible object locations based on the features in the layout will take longer and longer. The worst-case complexity, i.e. when every feature cannot overlap with every other feature in the layout, is near $O(n^2)$ with *n* the number of features in the new objects that need to be added to a layout. This is the reason why the hierarchic subdivision of the scene is very important, since the new object will only take into account the already placed features in the sub layout that is considered at that moment, so *n* will be the number of new objects added to the sub layout which can be considerably smaller. An optimized scene management system that stores the features of the layout can obviously further improve the performance of our approach. Our current implementation of this approach does not include mechanisms to detect contradicting rules. However, because of the step by step approach, it is quite easy to give the user insight in how a specific rule influences the possible locations for an object in an example. The system could show what areas are deemed valid or invalid by each rule, after which these rules could be adjusted by the user.

4 LAYOUT PLANNER

The job of the layout solver, discussed in the previous section, is to place an object in a layout, making sure the rules defined for that object in the class description hold. The layout planner submits objects to the solver one by one. A planner works based on a procedure: a list of rules that need to be executed in order. Examples for such rules might be: "place X instances of class Y", or "place as many objects of class Z as possible". This planner can also contain elements like if-then-else statements or loops: "keep adding cupboards until the total amount of storage space exceeds 1.3 cubic meters", or "if context is "dinner" place 5 plates on a table in front of a chair and stack 10 plates in storage features, else stack 15 plates in storage features". We can extend the previous example and define that in a context "after dinner" the plates should be placed on the sink. Note that currently, context in our solver is simply a list of tags that describe the current context, e.g. {"weekend", "after dinner", "near Christmas"…}. Based on these rules the planner will feed the layout problem solver. The problem solver, in turn, will make sure the rules defined in the classes are respected and therefore a valid solution will be generated.

An important rule that is available in the planner is the backtracking rule. When a point in a layout process is reached where an important object cannot be placed anymore, this could be solved by backtracking and choosing a different location for previously placed objects. At their old location, these objects might have prevented the placement of the new object.

When adding a new object to the layout, the corresponding features are placed accordingly, but the planner also allows the designer to directly add features to the layout. One of the feature types that is useful in a planner step is the *Area* feature. It is used to create a rough provisional layout, which in later steps can be filled with specific objects. This way, the *Area* features serve as kind of placeholders for the eventual objects. We could, for example, start our living room plan by first placing a sitting area and a dining area. Later, when we add a dining table to the layout, a rule is added that it should be placed inside the previously marked dining area. When placing a new *Area* feature, it behaves like an *OffLimit* feature, i.e. it may not overlap with any other feature and therefore guarantees the new feature area is empty. But when the feature is already placed, overlap with all other features *is* allowed with the exception of other *Area* features, and so the feature areas can be filled with the appropriate objects.

To edit and test the layout plans, we created a tool that allows designers to see how changes he or she makes to the plan affect the result. Because this result is obviously very dependent on the situation, we immediately show the effect of the changes on multiple examples (see Figure 1). Due to the nature of the problem designers will never be able to check all possible solutions, but by checking some solutions under different circumstances, major problems or unwanted behavior will quickly be noticed.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.