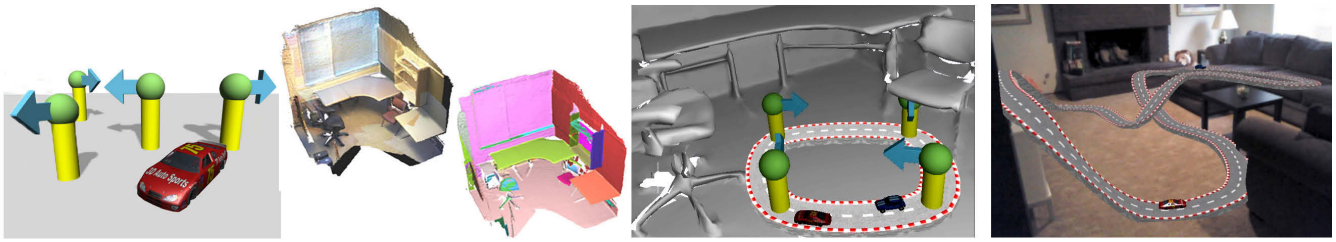
FLARE: Fast Layout for Augmented Reality Applications

Ran Gal* Microsoft Research
Lior Shapira† Microsoft Research
Eyal Ofek‡ Microsoft Research
Pushmeet Kohli§ Microsoft Research



(a) (b) (c) (d)

Figure 1: Designing an immersive augmented reality (AR) application such as a dynamic racing game is difficult. In our framework (a) declarative rules are used to define application elements and the rules governing them (b) in real-time we analyze an environment to extract scene geometry and horizontal and vertical planes (c) our move-making algorithm targets the application to the room (d) an additional result of our system in a different room with a longer track.

Abstract

Creating a layout for an augmented reality (AR) application which embeds virtual objects in a physical environment is difficult as it must adapt to any physical space. We propose a rule-based framework for generating object layouts for AR applications. Under our framework, the developer of an AR application specifies a set of rules (constraints) which enforce self-consistency (rules regarding the inter-relationships of application components) and scene-consistency (application components are consistent with the physical environment they are placed in). When a user enters a new environment, we create, in real-time, a layout for the application, which is consistent with the defined constraints (as much as possible). We find the optimal configurations for each object by solving a constraint-satisfaction problem. Our stochastic move making algorithm is domain-aware, and allows us to efficiently converge to a solution for most rule-sets. In the paper we demonstrate several augmented reality applications that automatically adapt to different rooms and changing circumstances in each room.

Index Terms: F.4.1 [Mathematical Logic]: Logic and Constraint Programming—; G.3 [Probability and Statistics]: Markov Processes—;

1 Introduction

Augmented reality is a growing trend on mobile platforms, as well as on emerging wearable computing platforms. Yet, AR systems have struggled to make the transition from laboratory to the real world. A particular hindrance to the successful deployment of AR systems is the complex and variant nature of reality. AR applications must work in any environment the user finds herself in. Therefore, the layout of the different elements comprising the AR application must be consistent with the environment. Simple applications might consist of planar information overlaid on reality, or virtual objects hanging in free space in front of a user. However, creating an application which truly integrates with the environment, embedding virtual objects among real physical objects is much more complex.

Several issues make this task challenging: First, the layout of virtual objects must be consistent with the placement of other virtual objects, as well as with the geometry of the physical environment they are placed in. For example, an application might require that two elements be placed within two feet of each other, but also that both be placed on an elevated horizontal surface. Second, a user might deploy several applications in the same environment, all of which must be laid out successfully without interfering with each other. Finally, several users might be collaborating using an AR application in a shared environment, further complicating the layout of the application elements.

In this paper we describe FLARE (Fast Layout for Augmented Reality), an application development system that enables targeting of AR applications to a variety of environments. In this system the layout of an AR application is designed using declarative rules, describing the desired mapping of the application elements to an environment. Each element has a state defined by a set of properties (e.g. position, scale, color). The declarative rules refer to these properties and to environment properties, and have a cost function associated with them. Mapping an application to an environment consists of finding an optimal (or close to optimal) state for all elements, such that the overall cost of the rules is minimized. Targeting several applications to a single environment, or sharing an application between multiple users is translated in our system to additional rules constraining the system.

We capture the user's environment using a Kinect camera (RGB and depth streams), and process it using Kinect Fusion [25] to extract dense scene geometry. We detect planar surfaces in the room and label them as *vertical* (e.g. walls, cabinets), *horizontal* (e.g. floor, table) or *other*. Planar features are common in indoor scenes and are useful to many applications. Adding additional detectors (e.g. object detection, recognizing previously visited rooms) could enable more complex rules and applications. FLARE performs a real-time mapping of the application to the user's current environ-

*e-mail: rgal@microsoft.com
†e-mail:liors@microsoft.com
‡e-mail:eyalofek@microsoft.com
§e-mail:pkohli@microsoft.com

ment, by applying the rules to the application elements, and the geometric information extracted from the scene. Finding the optimal layout under a complex set of rules is difficult, and requires the solution of a weighted constraint-satisfaction problem. A commonly used approach is to explore the solution space by local search, *i.e.*, start from an initial solution and proceed by making a series of changes (moves) which lead to solutions having lower cost. The algorithm is said to converge when no lower energy solution can be found. Our algorithm proposes a new method for generating moves with higher acceptance probability. We demonstrate its efficacy in reducing the number of moves required.

To summarize, our contributions are (1) FLARE, a general framework for designing the layout of an AR application (2) a quick-converging algorithm for finding an optimal layout. The rest of the paper is organized as follows, in section 2 we discuss related work. In section 3 we provide a formalization for the layout design problem and describe our declarative framework. In section 4 we describe our method for generating compact mathematical descriptions of design rules and our algorithm for computing the optimal layout. In section 5 we demonstrate our framework in action, detailing several synthetic rule-sets and some AR applications. We conclude in section 6 by listing some observations regarding our framework and discussing directions for future work.

2 Related Work

Mapping AR to the real world Augmented reality [2, 6], in general, should work in a large range of environments. Mobile AR application such as [19, 31] use the location of the user and the orientation of the mobile device to add a 2D overlay over the user's view. For location-specific apps, the geometry of a site can be computed in advance, for example archaeological sites [1], Museums [23], manufacturing floor [26], projection mapping [12]. In recent years many augmented reality apps and games, were designed for a simple planar world on which the augmented content resides. The world plane is attached to a known pattern, found on a magazine ad or a packaging of a product [19, 15]. Recent works [25, 14] used the recovered 3D geometry of the scene to demonstrate some physical simulation examples.

Layout Synthesis The availability of 3D models of physical spaces has inspired a large amount of work on generating layouts. In [34, 24] a set of rules and spatial relationships for optimal furniture positioning are established from examples and expert-based design guidelines. These rules are then enforced as constraints to generate furniture layout in a new room. [34] employed a simulated annealing method which is effective but takes several minutes, while [24] sample a density function using the Metropolis-Hastings algorithm implemented on a GPU. They evaluate a large number of assignments and achieve interactive rates (requiring a strong GPU). Both papers work with a small number of objects in relatively small rooms and in static scenarios. [10] showed how arrangements of 3D objects can be found using a data-driven example based approach. [33] populate a scene with a variable number of objects (open universe). They present a probabilistic inference algorithm extending simulated annealing with local steps, however the computation cost is high and the procedure takes upwards of 30 minutes.

Constraint Satisfaction for Design The problem of rule based design generation has a long history. Design and layout synthesis consist of rules referencing a set of objects. An assignment to each object can be measured by how well the rules are met, whether they are satisfied or violated. As an example of an early work in this space, the Ultraviolet [4] system used a constraint satisfaction algorithm framework for interactive graphics. The constraints for user interface layout usually form a non-cyclic graph, are hierarchical in nature and container based and are therefore less complex.

Constraint satisfaction problems (CSP) [22] are fundamental in Artificial Intelligence and Operations Research. A variant of the problem, weighted CSP defines a cost function assigned to each constraint, and the objective is to minimize the overall cost. A large majority of CSP algorithms [18] use a search paradigm over a limited set of possible object assignments. More recently, [21] performed Pattern Colorizations by solving a CSP. These approaches are relatively rigid, and do not offer interactive performance. In contrast, our method for computing consistent layouts can adapt to the problem at hand and is inspired from move making algorithms that have been used for image labeling problems.

Discrete Optimization for Image Labeling In computer vision, many tasks such as segmentation of an image can be formulated as image labeling problems where each variable (pixel) needs to be assigned the label which leads to the most probable (or lowest cost/energy) joint labeling of the image. The models for these problems are usually specified as factor-graphs in which the factor nodes represent the energy potential functions that operate on the variables [17]. In most vision models, the energy function is composed of unary and binary terms and the interactions between objects are generally limited to variables in a 4 or 8 neighborhood grid. The sparse grid-like structure of the object interactions and the limited number of labels allows for fast solution of image labeling problems using techniques such as graph-cuts [5, 11, 20, 28, 32], belief-propagation [27], tree message-passing [30, 16], and dynamic programming [29].

In our case, rules can be defined over multiple variables, and create complex factor graphs which these approaches do not handle well. Further, each object typically has a large space of possible configurations, which increases the complexity in multi-object interactions. Furthermore, in all but the simplest scenarios the factor graph contains cycles that makes the problem NP-hard even if the label space for each object is small. Our method deals with complex factors by generating compact encodings of higher order relations.

3 Layout Design for AR Applications

In the FLARE framework a designer specifies the layout of his application by defining application elements and rules which apply to them. First the elements are defined, each identified by a unique name and belonging to a predefined class. The class of an element defines its properties which are similar in nature to member variables in object-oriented languages. Each property is strongly typed, can be initialized and can be limited to a range of values (discrete or continuous) or even a single fixed value. Most of the elements (objects) in our implementation are of type *Object3D* (unless otherwise noted in the application description). This type is detailed in table 1.

Table 1: Definition of *Object3D* class

Object3D Properties	
Type (enum)	Attach to a certain type of surface (vertical or horizontal), or float anywhere in the room
Position (Vector3)	Position relative to selected surface
Facing (float)	Rotation relative to selected surface
Scale (float)	Uniform scale multiplier

For example, one instance of class *Object3D* might be constrained to be positioned on horizontal surfaces and allowed to scale from $\times 1$ to $\times 2$, while a second instance is constrained to vertical surfaces and its facing is limited to a 30^o arc. Additionally, our system defines the environment itself as an element, the properties of which are detailed in table 2.

A global coordinate system is defined for the current environment, which serves to transform between the different local frames. We detect an *up* vector for the environment using a gyroscope or inference (from the detected floor plane). The *Camera* is positioned in the global coordinate system, and its projection parameters are preset (field of view, and near and far planes). The *Floor* plane is

Table 2: Definition of *Environment* properties

Environment Properties	
Camera	Positioning and projection information for the camera/user.
Floor	Local frame for the floor of the current environment
Ceiling	Local frame for the ceiling plane

detected in the plane detection and classification stage. Its coordinate system is set such that the X axis points towards the camera and Z axis is the up vector. Similarly, local frames for other horizontal surfaces are defined such that the Z axis is the surface normal, while the X axis points towards the camera. Therefore, application elements with a fixed 0 facing, will always face the camera. Vertical surfaces are set such that the X axis point towards the up vector.

Rules are written using algebraic notation, a library of predefined routines and comparison and boolean operators. A rule can reference the properties of any of the elements defined, as well as the environment. For example, a rule might require that the distance between two elements is more than 1 meter. Another rule might stipulate that one element be positioned between 20 and 40 centimeters higher than another. We define a natural cost function on comparison operators, for example

$$cost(a < b) = \begin{cases} 0 & a < b \\ (a-b)^2 & a \geq b \end{cases}$$

Library routines define their own costs. For example *distinct(a,b)* returns 0 if there is no collision between objects a and b, otherwise it returns 1. Finally, each rule has a weight associated with it (default 1). For boolean operators, $cost(OR(a,b)) = min(cost(a), cost(b))$, $cost(AND(a,b)) = cost(a) + cost(b)$.

For practical purposes we've exposed our rule-based framework as both an API, and as a scripting language. For examples and for a complete listing of our supported operators and library routine, please see the supplemental material.

We call the space of all possible assignments to the application elements, the *layout solution space*. We define a cost function

$$cost(s) := \sum_i w_i \cdot r_i(\hat{s}_i) \tag{1}$$

where w_i is an optional weight specified by the application designer (default 1), $r_i : (O_i \subseteq O) \rightarrow \mathbb{R}$ is a rule operating on a subset of the elements (O), $s \in S$ is a specific solution (S is the solution space), and $\hat{s}_i$ is a slice of the solution containing only the objects in O_i. Typically each rule applies only to a small subset of the objects. An optimal layout for an AR application is one which minimizes the overall cost of its rules. In the next section we discuss several approaches to finding an optimal or approximate layout for an AR application.

4 Targeting an AR Application

Given an application design in our rule-based framework, we want to be able to target the application to any environment in which the user might find herself. Targeting the application means finding an optimal layout for the application, the global minimum of the *layout solution space*. Finding the optimal layout or even a good one is difficult: Rule cost functions may be non-convex, rules might be unsatisfiable, for example if they conflict with the environment or with themselves, therefore we cannot know the lower bound on the cost and it is difficult to specify a stopping criteria. Finally, the high-dimensional nature of the space and the assumed sparsity of feasible solutions reduce the effectiveness of stochastic sampling.

Similar to [24, 34] we focus on a discretized version of the solution space. An analysis of the environment (detailed in section 5) reveals horizontal and vertical surfaces on which we place most application elements. Each surface is defined by its plane equation and a surface boundary (represented as a poly-line). We generate a finite set of positions for each surface. Other properties such as scale and facing are uniformly sampled at a preset quantization (default 32 bit).

Still, given N objects in the design and k possible assignments per object, the size of the solution space k^N makes performing an exhaustive search prohibitively expensive. Previous methods have attempted to sample from the underlying probability distribution function, using Metropolis-Hastings [13] algorithm coupled with concepts from simulated annealing. These methods require a prohibitively large number of samples (and of course evaluations of the cost function), therefore requiring a long run time or reliance on massively parallel GPU implementations [24]. In many applications performance is an issue, and in some platforms such as mobile devices, computation is costly.

A simple method to find a low-cost solution under the function defined in equation 1 is to explore the solution space by local search *i.e.* start from an initial solution and proceed by making a series of changes which lead to solutions having lower energy. At each step, this *move-making* [5, 20, 29] algorithm explores the neighboring solutions and chooses the move which leads to a solution having the lowest energy. The algorithm is said to converge when no lower energy solution can be found. An example of this approach is the Iterated Conditional Modes (ICM) algorithm [3] that at each iteration optimizes the value of a single variable keeping all other variables fixed. However, this approach is highly inefficient due to the large label space of each variable.

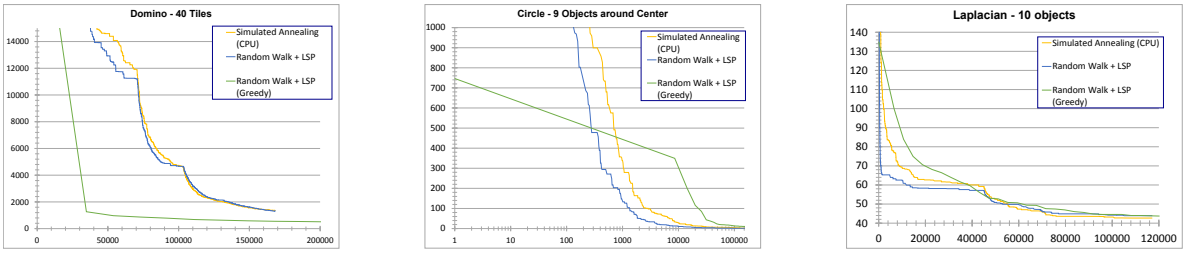
We perform a random walk algorithm (algorithm 1), in each iteration we select a new value for one of the objects and evaluate the cost function. We accept the new configuration with probability α. This probability stems from the Metropolis-Hastings acceptance probability and is dependent upon whether the new configuration improves upon the previous one, and the *temperature* of the system. Over the iterations we cool the system, in effect making smaller moves until the system converges (see figure 2 for an illustration). We refer the reader to [24] for details on the sampling procedure.

Algorithm 1 Random Walk

```
minSolution ← RandomAssignment()
currentSolution ← minSolution
minCost ← Evaluate(minSolution)
for i ← 1, niters do
    for all O ∈ Objects do
        p_O ← nextProposal
        currentSolution ← p_0
        cost ← Evaluate(currentSolution)
        if accept(cost, minCost, temperature) then
            minSolution ← currentSolution
            minCost ← cost
        end if
    end for
    UpdateTemperature()
end for
```

4.1 Locally Satisfiable Proposals

Generating proposals for a move-making algorithm is key to its performance. In [24, 34], a large number of proposals were required to converge to an acceptable solution, requiring a massively-parallel implementation to achieve interactive performance. In targeting an AR application to an environment, we might be required to optimize a large set of constraints over a large set of objects with multiple properties. Furthermore, not all devices have a GPU or can waste computing power. We alleviate this problem by guiding the mechanism through which new proposals are generated.



(a) (b) (c)

Figure 3: Experimental results comparing the performance of LSP with that of standard sampling parallel tempering (or random walk). In all three graphs, the x axis is number of candidate evaluations (log-scale in (b)) and y axis is the solution cost.

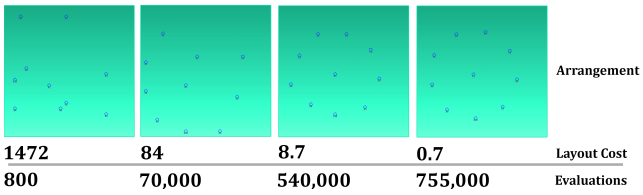


Figure 2: Example of Random Walk: We attempt to arrange 9 objects in a fixed radius circle around a single object, such that the angular distance between each two objects is at least 40 degrees. The overall cost of the layout drops as the number of evaluations grows.

For many types of rules, assignments that satisfy these rules can be found efficiently. In other words, these rules are *locally* satisfiable. In simple terms, given an assignment to some of the objects referenced by r we can generate good proposals for the rest, without resorting to blind sampling in the layout solution space. Our approach could be seen as performing Gibbs sampling [7], taking advantage of a known partial probability function, to sample from the whole solution space.

Within our framework, rules built using a subset of library functions and operators are defined as *locally satisfiable*. For example given a rule $dist(a,b) < 5$, and the position of a, an optimal position for b would be within a sphere of radius 5 around a. Combined with a typical preference of objects for horizontal or vertical surfaces, we are able to sample proposals for the position of b efficiently. A *locally satisfiable proposal* (LSP) is a proposal for an application element o which was proposed by a locally satisfiable rule r. In practice, when generating p_O (see algorithm 1), *nextProposal* is defined as in algorithm 2.

Algorithm 2 Generating a Single Proposal

```
function NEXTPROPOSAL(o)
    p ← rand(0,1)
    if (p < 0.5)&&(∃r : O ∈ r)&&(locallySatisfiable(r)) then
        return LSP(o,r)
    else
        return GenerateProposal(o)
    end if
end function
```

where $LSP(O,r)$ generates a locally-satisfiable proposal for o given the current state of application elements. In some cases a locally-satisfiable proposal might over-constrain our sampling, for example in rule-sets where a least-squares solution is optimal. Therefore we only generate an LSP 50% of the time. If no locally satisfiable rule exists for O, $GenerateProposal(O)$ proposes a standard move.

A variant on the LSP algorithm is a *greedy* LSP generation. In this variant generating an LSP for object o (via rule r) prompts selecting object o' for the next iteration, such that o' is also referenced in r. The reasoning behind this is to traverse the graph of constraints, fixing objects as we go. We have found that in practice this heuristic often accelerates convergence (figure 3(a)). We apply this variant within the *LSP* procedure 20% of the time.

Following are examples of locally satisfiable rules

1. $dist(a,b) == 4$ is locally satisfiable as given a we generate proposals for b on the circle centered around a with radius 4
2. $collinear(x_1,..x_n)$ is locally satisfiable given assignments to two of the objects. As we can sample the rest of the objects on the line defined between them.
3. $withinFrustum(a)$ requires a to be in the camera frustum. This is locally satisfiable as generate proposals only from a slice of the 3D space.
4. A constraint on the material properties of two objects, $complementary(a,b)$, is locally satisfiable as given the color of a, the color of b is easy to calculate.

Using LSP reduces, in most cases, the number of proposals required to achieve a good layout.

4.2 Experimental Evaluation

In order to demonstrate this we created three distinct rule-sets. We targeted each rule-set to different rooms multiple times. We measure the performance and quality of a layout optimization algorithm by counting rule evaluations. Previous papers have counted the number of samples the algorithm performs for all objects in all iterations. However, this measure favors algorithms which perform an exhaustive search over limited combinations of values. For each rule-set we plot the cost of the solution vs. number of evaluations, comparing our approach to a parallel tempering algorithm we simulated on the CPU. In all three designs, the rules are geometric, and each object in the design can be assigned position on a surface, facing (rotation on the surface) and scale. For each rule-set we ran each algorithm 30 times (in different rooms) and plot the median of the results. The three rule-sets are

Domino - Forty tiles arranged in a curve i.e. each tile t_i has the following rules applied (i) $2 < dist(t_i,t_{i+1}) < 5$ (ii) $\langle t_{i+1} - t_i, t_{i+1}.facing \rangle \leq 0.97$ (iii) $\langle t_{i+1}.facing, t_i.facing \rangle \leq 0.9$. A sample layout can be seen in figure 4(a).

Circle - Nine objects arranged in a circle (with non-fixed radius) around a central object. The minimal angle between any two objects is at least 25^o (example in figure 4(b)). The experiment results are in figure 3(b). All rules in this design are ternary, and the rules enforcing a minimal angle between all objects create a high inter-dependency between object values.

Figure 4: Visualization of the three qualitative evaluation scenarios: (a) A set of domino tiles set on a curve. Each domino tile is within a set distance from the next, faces in the same direction and approximates a straight line (b) A set of objects arranged in a fixed radius circle around a center object (c) Ten objects such that each one attempts to approximate the average position of both its neighbors, and minimize the distance to them.

Laplacian Cycle - We arrange ten objects $t_1..t_{10}$ such that $t_i = (t_{i-1}+t_{i+1})/2$ and $d(t_i,t_{i+1}) > C$. Since the rules wrap around t_{10} the cost can never be 0 and the best possible solution is a least-squares oval structure (example in figure 4(c)).

Results are shown in figure 3, showing the benefits of using LSP. Note that in the Domino scenario (a) we show how effective our greedy LSP generation. In (b) we use a log scale on the number of iterations. In (c), the o'ptimal solution is a least-squares one, and therefore there is little benefit in using LSP, and still we produce comparable results.

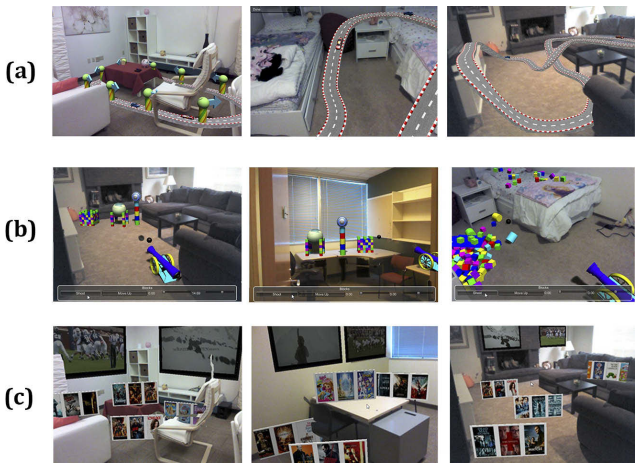


Figure 5: AR applications using FLARE: (a) A race track dynamically generated for a room by placing constrained key points (b) A physics-base game where a cannon shoots at bricks made of castles (c) A media-library app where virtual screens on the walls show clips, and billboards highlight selections from the user's collection.

5 Using FLARE

In order to test FLARE, we captured multiple rooms using Kinect Fusion [25] and produced a triangular mesh for each. Each room was processed to detect planar surfaces using Hough Transform [8]. We classified each plane into horizontal, vertical and other surfaces (using an estimated up vector) and found the concave contour of each [9]. Given an application design rule-set and an environment (including camera position), we target the application to the environment using our move-making algorithm with LSP. The results shown in this section (and in the accompanying video) were generated in real-time (see running times in table 3) and rendered in a Unity3D environment.

5.1 Auxiliary Rules

In some situations, the application rule-set is sufficient for targeting an application to an environment. However, often there is need for auxiliary rules (constraints) in which case our system automatically adds them and optimizes them alongside the designer specified rules.

- *Collision Detection*: For each pair of object (o_i,o_j) we add a constraint $distinct(o_i,o_j)$ whose cost is 0 if there is no collision between the objects bounding boxes, and 1 otherwise. This potentially adds $n(n-1)/2$ rules. However, as we sample positions discretely, we are able to efficiently rule out collisions in most cases.
- *Persistence*: If an application was previously targeted to an environment we would prefer to retain a similar layout. We add constraints for each object to keep its previous values. For example $o == o_{prev}$ with a linearly increasing cost function. Persistence is also useful if a real dynamic object in the room (such as a person walking) forces a retargeting of an application.
- *Multiple apps*: Different application rule-sets by design are not aware of each other. If a user targets two or more application to a room we are able to constrain each app to a different section of the room, or simply avoid collisions between the different application elements.
- *Collaboration*: If two or more users in an environment wish to collaborate on an AR application, we can constrain the application, taking into consideration the multiple camera (user) positions.

Table 3: Running times for the various applications.

App Name	# of Objects	Time in Seconds (worst case)
Domino	20	2.74
Circle	10	0.8
Laplacian Cycle	10	1.63
Race Track	16	2.31
Media Library	6	1.6
Angry Cannon	4	0.8

5.2 Sample Applications

We developed several AR applications which contain application logic and graphical assets. These applicatiosn are targeted in real-time into pre-scanned rooms in order to demonstrate the usability of our system:

Angry cannon is a physics-based puzzle game in which a user aims a cannon C at brick castles and bomb pillars $b_1,...b_n$, attempting to knock them down. The castles and pillars are place around the room, within range of the cannon, using existing room features as obstacles. The game objects and their properties are the *cannon* (position, facing), *brick castles* (position, facing, number of bricks) and *bomb pillars* (position, facing, height). The rules are

1. $dist(C,b_i) > 4$
2. $horizontal(C)$
3. $horizontal(b_i)$
4. $collision(C)$
5. $collision(b_i)$

AR Racing is a racing game where the race track is dynamically created for each new room the player visits. Given a desired track length, we create a set of keypoint objects (whose only property is position) $K = \{k_1,...k_n\}$. The rules for each object k_i are

1. $dist(k_i,k_{i+1}) \in [0.5,1]$
2. $lineOfSight(k_i,k_{i+1})$
3. $collision(k_i,K/k_i)$
4. $horizontal(k_i)$
5. $collision(k_i)$

where $k_{n+1} \equiv k_1$ and distances are specified in feet. As the tracks grow longer, the keypoints must select different horizontal surfaces in order to preserve the minimal distance, creating complex tracks, taking advantage of the geometry. In order to render the looped track we pass a spline through the keypoints, and on it we place the racing cars.

The *Media Library* application lets a user browse his collection of videos, in any environment. A selection of movies from a database is divided into categories, and displayed on several tile poster objects $p_1, \ldots p_n$. Each poster has position and facing. Additionally we place two video screens V_1, V_2, meant to hang on the room walls, whose position and scale can change. The rules in this application are

1. $horizontal(p_i)$
2. $vertical(B_i)$
3. $inFOV(p_i)$
4. $inFOV(V_i)$
5. $inner(p_i.facing, eye) \leq -0.8$
6. $collision(p_i)$
7. $collision(B_i)$
8. $scale(V_i) = maximum_scale$

The hanged screens are set to a maximum size by the last rule. The maximal size is set to be bigger than the room, which leads the screens to be of the largest size that can still fit on the wall. The cost of this rule's cost will be minimized but may never reach zero cost. Sample results for all three applications can be seen in figure 5 and the accompanying video.

6 Discussion and Future Work

We presented FLARE, a rule-based design framework for AR applications, that allows a designer to define objects with layout properties as application components, and a set of rules which help target the application to any environment. The environment is represented by a set of features is extracted from recovered geometry and the color video taken at the scene. The richness of the rules is partially dependent on the features extracted from the scene. We used planar features for our examples as they are common in indoor scenes and were sufficient to generate all the examples in the paper. Other environments, such as natural scenes, may require other features. We also introduced the concept of locally satisfiable proposals and demonstrated that their use dramatically reduces the number of evaluations required for finding a rule-consistent layout. In cases where LSP fails, our algorithm degrades to a random sampling approach.

All the examples shown in this paper were generated automatically, from the geometry reconstruction, to the plane extraction and targeting the different apps to the environment. However, the mapping is not without limitations. It is possible to assign a set of rules that will not be satisfied in a given environment. For example, we might wish for an object to be positioned on an elevated horizontal surface above the floor, which may not exist in a given room. In this case the optimal cost function for the design cannot be 0 and the system will approach that minimum (e.g. place the object on the floor). In designs where the optimal solution would be a least-squares solution, our locally-satisfiable proposals do not provide a benefit and our algorithm degrades to random sampling.

As future work, we intend to develop a GPU based implementation of our method that would be similar to parallel tempering, and adapt it to other design problems. We have a strong belief that immersive augmented reality will see a surge in research over the next few years and hope our system can serve as a basis for other mapping algorithms.

References

[1] Architip. Website, 2013. http://architip.mobi.

[2] R. Azuma, Y. Baillot, R. Behringer, S. Feiner, S. Julier, and B. Macintyre. Recent advances in augmented reality. *IEEE Computer Graphics and Applications*, 21(6):34–47, Nov. 2001.

[3] J. Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 259–302, 1986.

[4] A. Borning and B. Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints*, 3(1):9–32, 1998.

[5] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *PAMI 2001*, 2001.

[6] J. Carmigniani, B. Furht, M. Anisetti, P. Ceravolo, E. Damiani, and M. Ivkovic. Augmented reality technologies, systems and applications. *Multimedia Tools and Applications*, 51(1):341–377, Jan. 2011.

[7] G. Casella and E. I. George. Explaining the gibbs sampler. *The American Statistician*, 46(3):167–174, 1992.

[8] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[9] H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Transactions on Information Theory*, 29:551–558, 1983.

[10] M. Fisher, D. Ritchie, M. Savva, T. Funkhouser, and P. Hanrahan. Example-based synthesis of 3d object arrangements. In *SIGGRAPH Asia*, 2012.

[11] S. Gould, F. Amat, and D. Koller. Alphabet soup: A framework for approximate energy minimization. In *CVPR 2009*, pages 903–910, 2009.

[12] R. Grasset, J.-D. Gascuel, and D. Schmalstieg. Interactive mediated reality. In *ISMAR 2003*, 2003.

[13] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[14] B. R. Jones, H. Benko, E. Ofek, and A. D. Wilson. Illumiroom: Peripheral projected illusions for interactive experiences. In *CHI 2013*, 2013.

[15] Junayo. Website, 2013. http://www.junaio.com.

[16] V. Kolmogorov. Convergent tree-reweighted message passing for energy minimization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(10):1568–1583, Oct. 2006.

[17] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 2001.

[18] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI MAGAZINE*, 13(1):32–44, 1992.

[19] Layar. Website, 2013. http://www.layar.com.

[20] V. S. Lempitsky, C. Rother, S. Roth, and A. Blake. Fusion moves for markov random field optimization. *PAMI*, 2010.

[21] S. Lin, D. Ritchie, M. Fisher, and P. Hanrahan. Probabilistic color-by-numbers: Suggesting pattern colorizations using factor graphs. In *ACM SIGGRAPH*, 2013.

[22] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99 – 118, 1977.

[23] P. S. Margriet Schavemaker, Hein Wils and E. Pondaag. Augmented reality and the museum experience. In *Museums and the Web 2011*, 2011.

[24] P. Merrell, E. Schkufza, Z. Li, M. Agrawala, and V. Koltun. Interactive furniture layout using interior design guidelines. In *SIGGRAPH 2011*, Aug. 2011.

[25] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. W. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *ISMAR*, pages 127–136, 2011.

[26] S. K. Ong and A. Y. C. N. (Eds.), editors. *Virtual and Augmented Reality Applications in Manufacturing*. 2004.

[27] J. Pearl. Reverend bayes on inference engines: A distributed hierarchical approach. In *AAAI*, pages 133–136, 1982.

[28] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. F. Tappen, and C. Rother. A comparative study of energy minimization methods for markov random fields. In *ECCV 2006*, 2006.

[29] V. Vineet, J. Warrell, and P. H. S. Torr. A tiered move-making algorithm for general non-submodular pairwise energies. *CoRR*, abs/1403.6275, 2014.

[30] M. J. Wainwright, T. Jaakkola, and A. S. Willsky. Map estimation via agreement on trees: message-passing and linear programming. *IEEE Transactions on Information Theory*, 51(11):3697–3717, 2005.

[31] Wikitude. Website, 2013. http://www.wikitude.com.

[32] O. J. Woodford, P. H. S. Torr, I. D. Reid, and A. W. Fitzgibbon. Global stereo reconstruction under second order smoothness priors. In *CVPR*, 2008.

[33] Y.-T. Yeh, L. Yang, M. Watson, N. D. Goodman, and P. Hanrahan. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Trans. Graphics*, 31(4):1–11, July 2012.

[34] L.-F. Yu, S.-K. Yeung, C.-K. Tang, D. Terzopoulos, T. F. Chan, and S. J. Osher. Make it home: automatic optimization of furniture arrangement. In *SIGGRAPH 2011*, Aug. 2011.