Reality Skins: Creating Immersive and Tactile Virtual Environments

Lior Shapira*
Google Machine Intelligence, Seattle WA, USA

Daniel Freedman†
Microsoft Research, Haifa IL

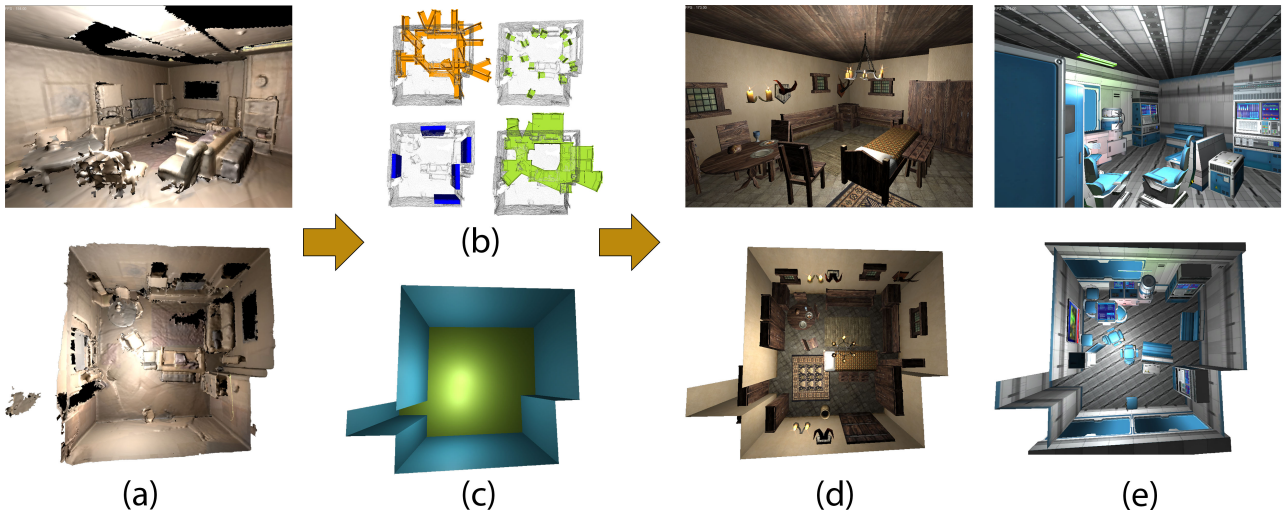

Figure 1: (a) We scan and process the surrounding environment (b) We populate the scene with different objects from a specialized data-set, and optimize to find the best combined layout (c) We detect planar areas and infer a complete floor-plan (d) We combine the floor-plan and object layout, creating a **Reality Skin**, a large scale, dynamic and tactile VR experience (e) Each scene can be transformed into a variety of worlds, such as this space station.

Abstract

Reality Skins enables mobile and large-scale virtual reality experiences, dynamically generated based on the user's environment. A head-mounted display (HMD) coupled with a depth camera is used to scan the user's surroundings: reconstruct geometry, infer floor plans, and detect objects and obstacles. From these elements we generate a **Reality Skin**, a 3D environment which replaces office or apartment walls with the corridors of a spaceship or underground tunnels, replacing chairs and desks, sofas and beds with crates and computer consoles, fungi and crumbling ancient statues. The placement of walls, furniture and objects in the Reality Skin attempts to approximate reality, such that the user can move around, and touch virtual objects with tactile feedback from real objects. Each possible **reality skins world** consists of objects, materials and custom scripts. Taking cues from the user's surroundings, we create a unique environment combining these building blocks, attempting to preserve the geometry and semantics of the real world.

We tackle 3D environment generation as a constraint satisfaction problem, and break it into two parts: First, we use a Markov Chain Monte-Carlo optimization, over a simple 2D polygonal model, to infer the layout of the environment (the structure of the virtual world). Then, we populate the world with various objects and characters, attempting to satisfy geometric (virtual objects should align with objects in the environment), semantic (a virtual chair aligns with a real one), physical (avoid collisions, maintain stability) and other constraints. We find a discrete set of transformations for each object satisfying unary constraints, incorporate pairwise and higher-order constraints, and optimize globally using a very recent technique based on semidefinite relaxation.

Index Terms: H.5.1 [Multimedia Information Systems]: Artificial, Augmented and Virtual Realities—; I.3.7 [Three Dimensional Graphics and Realism]: Virtual Reality— [G.1.6]: Optimization—Constrained Optimization

1 Introduction

When a user dons a head-mounted display (HMD) to immerse herself in a virtual reality (VR) environment, she becomes isolated from her surroundings. In order for the VR experience to be safe and immersive, she must be able to interact with the virtual world without fear of bumping into unseen obstacles. The simplest solution would be to remain seated (immobile), and use a controller to move within the simulation. This solution however, limits the type of experiences, and is often more prone to causing nausea [30].

Being able to physically walk in a virtual environment increases immersion and reduces discomfort, but increases the challenge of providing a safe experience. A common solution is to cordon off a safe area, whose boundaries are known, and is guaranteed to be empty of obstacles [15]. If the user approaches the boundaries, she can be visually or audibly warned. This approach requires a large empty space, and precludes the possibility of large-scale mobile exploration with VR (barring methods such as redirected walking which are still limited).

We propose to use an HMD integrated with a depth sensor. The easy availability of consumer-grade depth sensors such as Kinect [21] or Structure [23] makes such integration easy. These sensors can scan and reconstruct the user's environment, as well as

*e-mail: liorshap@google.com
†e-mail:danifree@microsoft.com

track his position and orientation. With such a device we no longer need to rely on static VR experiences, designed beforehand and forced to fit the user's surroundings. Instead, we create a dynamic large-scale virtual environment, which approximates the geometric and semantic properties of the real environment. In such a virtual world, any position which a user can reach is clear in the real world; any table on which the user may lean, provides support in the real world.

A **Reality Skin** (RS) is a dynamically created 3D environment consisting of an enclosed space (the "level" in game terms), populated and furnished with objects, to approximate the layout of a real environment. The virtual environment can appear to be the stark corridors of a space station, the gloomy exam rooms of an abandoned hospital complete with gurneys, exam beds and zombies, or the fungi-filled caverns of a fantasy world. An RS is created using a blueprint, which we call a **Reality Skins World (RSW)**. Each RSW is created by a designer, containing instructions on how to transform 2D floor-plans into corridors and rooms, and how to populate the virtual world with objects and game elements. The RSW contains attributes and constraints which help guide the optimization process generating the RS.

For a given environment we process the scanned geometry to create a floor-plan. The floor-plan is based on the observed walls and floor area, but is constrained to be closed and complete. We then extract feature-vectors for each voxel in a volume-based representation of the environment. The generation of an RS should be fast (a few seconds at most), as the user is walking with a headset on, ready to engage in the experience; therefore we construct features which are quick to compute and compare.

To generate a Reality Skin for the user's environment we (i) use the inferred floor-plan and the RSW to construct the floors, walls and ceiling, the general structure of the virtual world. (ii) Populate the virtual world with a set of objects from the RSW. The solution space for a set of objects satisfying a large set of unary, pairwise and higher-order constraints is a extremely large. Since many of our constraints are unary constraints – that is, the matching of each object to the environment – we first find, for each object in the RSW, a list of candidate transformations into the scene. We refer to these candidate transformations as **modes**. An object's attributes and constraints drive this process; for example, it might be the case that an object may only be placed on the floor, or with its back to a wall. Most important though, is the geometric match of an object to the real world, such that when the user reaches out to touch a virtual surface, it will have a parallel in the real world. From the list of modes of all objects, we aim to find a subset that minimizes an objective function. The constraints embedded in the function take into account unary constraints (already calculated for each mode), as well as pairwise (e.g. avoid collision) and higher-level constraints (e.g. limit the number of instances for a specific object). The problem can posed as a non-convex integer quadratic program, which is then solved by relaxing the problem to a (convex) semidefinite program, and using an appropriate rounding scheme; this technique is based on the state of the art Concave Quadratic Cuts algorithm described in [27].

Our contributions are a declarative framework for designing dynamic VR experiences, and the algorithms required to turn those blueprints into (virtual) reality, adapting them to fit a real world environment which the user can engage. Designing an RSW in our system is a simple and straight-forward process, which we have integrated into a popular game creation engine. Scanning an environment and generating a Reality Skin takes seconds on any device equipped with a modern GPU.

The rest of the paper is organized as follows: In Section 2 we discuss related work. In Section 3 we discuss scanning and processing the user's environment. In Sections 4 and 5 we discuss the design of an **RSW**, and how from an RSW we generate a **Reality Skin**, a complete virtual world. In Section 6 we discuss technical details of our implementation and show results. Finally, in Section 7 we draw conclusions, discuss limitations, and outline possible future work.

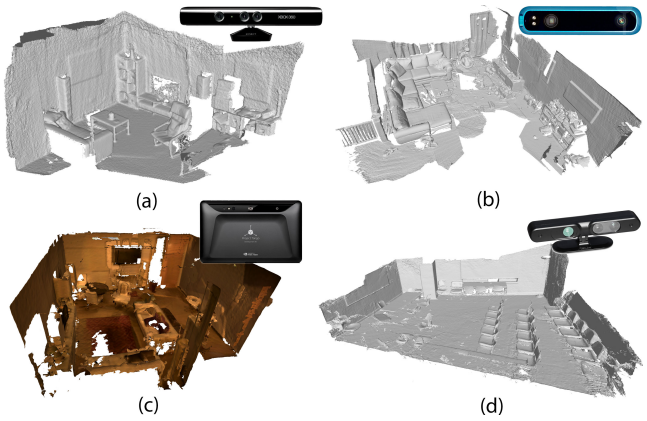

Figure 2: We use different devices to capture home and office environments for Reality Skins: (a) Kinect [21] camera with KinectFusion [17] (b) Structure camera [23] with KinectFusion [17] (c) Google Tango [11] (d) RGBD sequences from Sun3D [38], captured with an Asus Xtion camera and reconstructed with [4]. Overall we have captured over 100 different environments.

2 Related Work

The availability of 3D models has inspired a large amount of work on generating layouts. In [40, 20] a set of rules and spatial relationships for optimal furniture positioning are established from examples and expert-based design guidelines. These rules are enforced to generate furniture layout in a new room. Yu et al [40] employed a simulated annealing method which is effective but takes several minutes, while Merrell et al [20] 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. Fisher et al [7] showed how arrangements of 3D objects can be found using a data-driven example based approach. Yeh et al [39] 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. FLARE is a rule-based framework for generating objects for augmented reality (AR) applications by Shapira et al [9]. In our work the layout of objects is heavily constrained by the captured scene (represented by unary constraints), the dimensionality of the solution space is not fixed (similar to [39]), and there is need for interactive computation times, in which one good solution must be computed.

Most of the papers cited above use Markov Chain Monte-Carlo (MCMC) methods [14] to traverse a large multi-modal solution space. The prohibitive size of the solution space in our case, and the constraints of the environment necessitate a different sort of solution technique. The Concave Quadratic Cuts [27] method is very recent, but is in the spirit of various other semidefinite relaxation techniques for integer programming problems, for example [10, 26]. Such convex programs can be solved using fast modern techniques, such as the alternating direction method of multipliers (ADMM) [1, 25].

When acquiring a geometric representation of the user's environment, we infer a **floor-plan**, which guides the generation and the layout of the Reality Skin. Mura et al [22] take as input multiple point clouds acquired by a mobile scanner with known positions,

and build a 2.5D layout of the scanned area complete with room segmentation. They do so by detecting candidate walls (vertical planar patches), and creating a 2D cell complex, from which they infer the layout by diffusion. With similar requirements, Turner et al [34] partition space into interior and exterior domains, where the interior represents all open space in the environment. The input samples define the volumetric representation via a Delauney triangulation. Each triangle is labeled interior/exterior based on line-of-sight of each wall sample. A final model is constructed by using the boundaries between interior and exterior, and simplifying. In this paper we process a single point cloud, often noisy and with many occlusions. Oesau et al [24] support multi-story reconstructions by detecting vertical peaks in the scanned points distribution, used to find horizontal structure. They employ multi-scale line fitting to find vertical wall candidates. Finally, a graph-cut formulation is used to extract the final structure. Furukawa et al [8] reconstruct architectural scenes using structure-from-motion, multi-view stereo and a Manhattan-world assumption. They produce a simplified 3D model used for exploration of the reconstructed environments using an image based 3D viewer. Ikehata et al [16] reconstruct a grammar-based structured model of an indoor scene from panorama RGBD images. Our parametric floor-plan model side-steps the discretization and alignment issues of graph-cut and is more flexible as a template for a new floor-plan (which might differ in geometry and not only texture). Still, our RS algorithm may use any system for floor-plan inference.

Detecting and classifying objects in the scanned scene adds a layer of semantic understanding, which can help guide the generation of the Reality Skin, and enable semantic constraints. Our system can work with different approaches, such as Salas-Moreno et al [28] which integrate 3D object matching into their dense reconstruction (SLAM) pipeline. An interactive approach is used by [37, 36], with a user in-the-loop to annotate parts of the scene. Finally, unsupervised methods of object detection and classification have proliferated in images, RGBD frames, and 3D environments. We integrate the work of Litany et al [19] which detects 10 common object categories, and finds a closest matching exemplar from a 3D object dataset, into the scene.

Several works in the past have incorporated real world objects into virtual and augmented experiences. RoomAlive [18] is a projection-mapped experience which takes into account room geometry to create interactive games. The user can interact with virtual objects and creatures physically e.g. stepping on a virtual bug. However, they are constrained to a single prepared room and perform specific augmentations on that room. A recent demo [2] shows a crafted VR experience in which users explore an ancient tomb, whose layout matches that of a prepared area, and holding a torch (a physical object). Our work offers a framework in which a designer can create anything he imagines, and a 3D environment is generated to match any layout. Simeone et al [33] present a study on modeling virtual environments based on real ones. They focus on the mismatch between virtual and physical objects, and how it affects user experience. The environments presented in that paper were manually constructed to match specific rooms, whereas our work offers a complete pipeline for generating a virtual environment.

3 Scene Understanding

In order to create a VR experience in which a user can walk around any room, home or office, in a safe and engaging manner, we scan and process the surrounding environment. We've used different devices (Kinect [21], Structure [23], Tango [11]), all capable of capturing depth frames and integrating them together into a volumetric representation of the scene (figure 2). We are able to use any device that is able to export a triangular mesh or point cloud of the scene, with a generalized up direction. Given a scanned environment we perform the following steps:

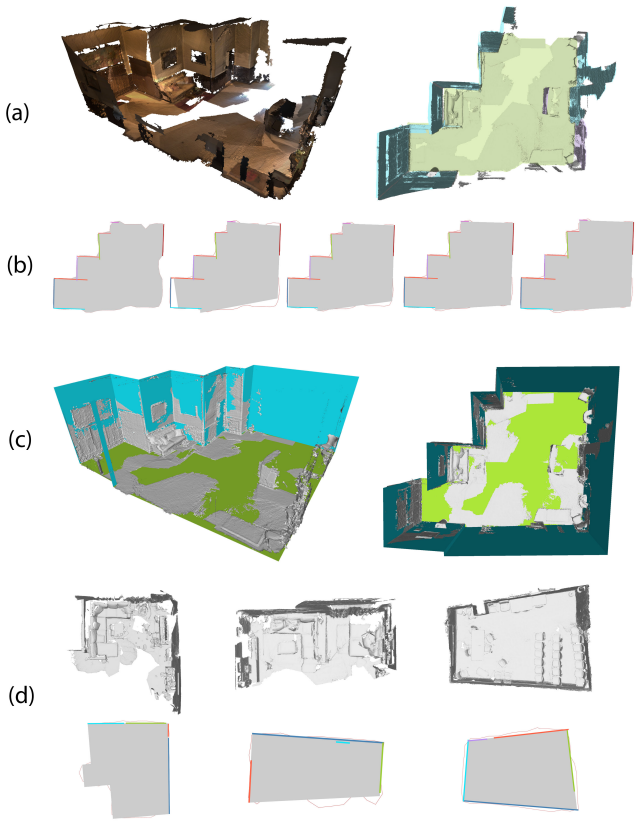

Figure 3: Inferring floor-plan for a scene: (a) We scan a large residential building lobby, and identify major planes (b) An iterative MCMC optimizes a simple floor-plan which adheres to detected walls (visualized as colored lines), covers the observed floor (visualized as brown poly-line) and prefers right angles and parallel walls (c) We extrude the detected floor-plan to create a water-tight 3D environment for Reality Skins (d) Additional examples.

1. Identify and classify the planar areas.
2. Infer a complete floor plan.
3. In a volume-based representation of the scene, compute a feature-vector for each voxel.

3.1 Plane Identification and Classification

We identify the dominant planes from the reconstruction of the environment using a greedy strategy, similar to [32]. First, using the Hough transform [5] we select a finite set of candidate planes. Each 3D point and its surface normal vote for a plane equation parametrized by its azimuth, elevation and distance from the origin. Each of these votes is accrued in an accumulator matrix of size $A \times E \times D$ where A is the number of azimuth bins, E is the number of elevation bins and D is the number of distance bins [1]. After each point has voted, we run non-maximal suppression to avoid accepting multiple planes that are too similar.

Once we have a set of candidate planes we sort them in descending order by the number of votes they have received and iteratively associate points to each plane. A point can be associated to a plane

[1] We use A=128, E=64 and D is found dynamically by spacing bin edges of size 5cm apart between the max and minimum points

if it has not been previously associated to any other plane and if its planar disparity and local surface normal difference are small enough [2]. As an additional heuristic, each new plane and its associated points are broken into a set of connected components ensuring that planes are locally connected (e.g. two identical coffee tables in a room will produce a single plane equation, but two separate components).

Once we have a set of planes, we classify each one independently into one of four semantic classes: Floor, Wall, Ceiling and Internal. To do so, we train a Random Forest Classifier to predict each plane's class using the ground truth labels and 3D features from [31], which capture attributes of each plane including its height in the room, size and surface normal distribution. Planes classified as one of Floor, Wall and Ceiling will be used for inferring the floor plan.

3.2 Floor Plan Inference

The set of planes extracted and classified in the previous section often form an incomplete floor-plan due to limited view-points, occlusions and errors in reconstruction. We aim to find a plausible and complete floor-plan, which fits the observed data and from which we can construct the layout of the **Reality Skin**.

We model a floor-plan as a pair (φ, ω), where $\varphi = (p_1, ...p_n), p_i \in \mathbb{R}^2$, a set of points which form a closed polygon (the "exterior walls") and $\omega = \{w_1^a, w_1^b), ...(w_m^a, w_m^b)\}, w_i^j \in \mathbb{R}^2$, a set of internal walls represented as points pairs[3].

We project all points from the scanned point cloud onto the floor plane, and find a concave hull [6] encompassing the points and representing the rough shape of the scene $\hat{\varphi} = (r_1, ...r_k)$. We define the set of observed planes classified as walls $\hat{\omega} = \{ow_1^a, ow_1^b), ...(ow_t^a, ow_t^b)\}$.

We define an objective function $C : (\Phi, \Omega) \rightarrow \mathbb{R}$ where $\varphi \in \Phi$ is a closed polygon, $\omega \in \Omega$ is a set of point pairs such that $C = C_{walls} + C_{angles} + C_{parallel} + C_{overlap}$. C evaluates how well the proposed floor plan (φ, ω) matches the scanned environment $(\hat{\varphi}, \hat{\omega})$. The sub-functions are defined as follows[4]:

C_{walls} is an objective function based on the number of walls in the suggested configuration. We compare that to the number of walls observed in the real scene, penalizing too many or too little walls $C_{walls} = abs((|\varphi| + |\omega|) - (|\hat{\varphi}| + |\hat{\omega}|))$.

C_{angles} is an objective function favoring right angles between adjacent walls. It is defined as $C_{angles} = \sum(\overrightarrow{p_{i+1}} \bullet \overrightarrow{p_i})$ where $\overrightarrow{p_i}$ is the normal of the wall between p_i and p_{i+1} (where $p_{n+1} = p_1$).

$C_{parallel}$ is an objective function where for each wall $w \in (\varphi, \omega)$ (external or internal wall) we search a parallel wall (opposing normal) up to a deviation of 10 degrees. It is defined as $C_{parallel} = \sum parallel(\overrightarrow{w})$ where $parallel \in [0,1]$ is equal 0 if a parallel wall exists and 1 if the closest to parallel wall deviates by more than 10 degrees.

$C_{overlap}$ is an objective functions which compares the floor area of the proposed configuration vs. the observed environment. It favors containing the observed environment, but without too much expansion. It is defined as $C_{overlap} = 1 - \cap(\varphi, \hat{\varphi}) / \cup(\varphi, \hat{\varphi})$.

We minimize C using the Metropolis-Hastings algorithm [14, 3]. From a configuration $\boldsymbol{F} = (\varphi, \omega)$ we sample a modified configuration $\boldsymbol{F}^*$ using one of the following moves (selected with equal probability):

1. Translate a random point in (φ, ω) (i.e. a point on the closed polygon or one of the ends of an interior wall).
2. Delete a point in φ.
3. Split a wall (in either φ or ω).
4. Add a new random interior wall.

Acceptance of the proposed move and new configuration $\boldsymbol{F}^*$ is derived from the Metropolis-Hastings algorithm's acceptance probability

$$\alpha(\boldsymbol{F} \rightarrow \boldsymbol{F}^*) = \min\left(1, \left\{\frac{p(F^*)q(F|F^*)}{p(F)q(F^*|F)}\right\}\right)$$

where $q(F|F^*)$ is the proposal distribution from which a new configuration F^* is sampled given current configuration F. $p(F)$ is a probability function defined as

$$p(F) = \frac{1}{Z}\exp(-\beta C(F))$$

where Z is a normalizing factor and β is a temperature constant which is 'cooled' over the iterations, favoring large moves in the beginning and smaller moves as the optimization converges. We run multiple MCMC threads in parallel, retaining the configuration with the lowest score. A visualization of the process and some results can be seen in figure 3.

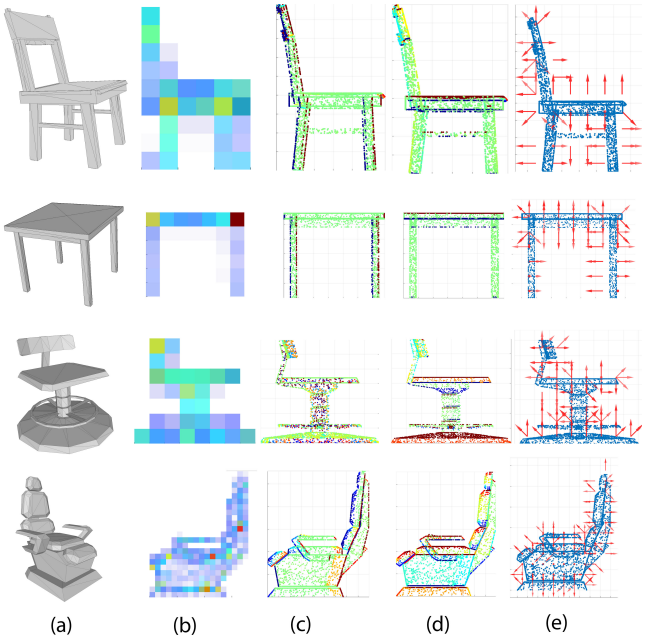

Figure 4: Extracting features: (a) We calculate feature vectors for each voxel (cell) in an object or scene. (b) Occupancy is a normalized count of points within each cell, visualized here from blue to red (c,d) We extract a normal response histogram for each cell by binning the response of each point's normal to a set of directions (Unit-X and Unit-Y shown here) (e) We visualize the maximum direction for each cell.

3.3 Extracting Scene Volumetric Features

We populate a Reality Skin with instances of objects from the RSW. Each object instance consists of the original 3D object paired with a transformation which places it in the scene, in a way which minimizes an objective function (discussed in section 5). We perform a uniform and discrete sampling of the transformation space for each

[2] Planar disparity threshold=.1, angular disparity threshold = .1

[3] Internal walls occur when there are thin dividers in the room which are not scanned as two separate walls, in practice these rarely occur.

[4] C functions are weighted as follows: $w_{walls} = 2.0, w_{angles} = 1.0, w_{parallel} = 2.0, w_{overlap} = 2.0$

object by creating a volumetric representation of the scene (and for each object) and running a sliding-window algorithm.

Given the inferred floor-plan of the scene, we create an aligning transformation which rotates the floor normal to the Y-axis, translates the scene such that the floor lies at $y = 0$, and further rotates such that the longest wall lies along the positive X-axis. We voxelize the aligned scene with a default voxel (cell) size of 10 cm on each side. Each point in the internal point-cloud (all points not associated with a wall, the floor or ceiling) is assigned a voxel index. For each cell we calculate **Occupancy**, the number of points falling within this cell (figure 4b) and **Normal Response**, A 26-bin normalized histogram summing the response (dot product) of each point's normal with a set of major and minor direction vectors (figure 4c–e).

We experimented with extracting more advanced features such as described in Litany et al [19]. However, unlike object detection algorithms, which search for near-exact matches, we are looking for best possible matches, often with a very disparate set of objects. Therefore a set of coarse features, which are easy to calculate and compare, worked best for our work.

Note that automated [19] or "user in the loop" [37, 36] methods for object classification and detection may be used at this stage to add semantic properties to the feature vector of each cell. Such semantic information is matched to tags defined on each object in the RSW, and used in the objective function.

4 Designing a Reality Skin World

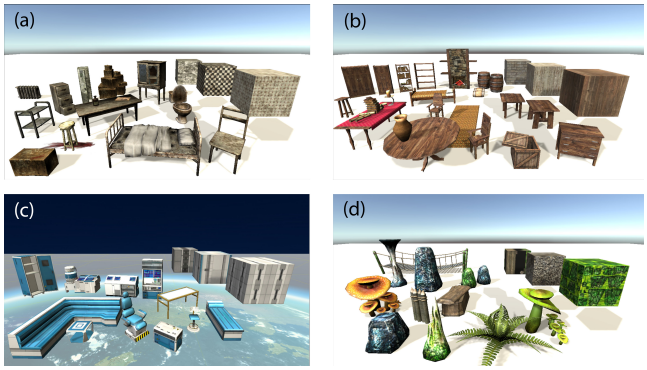

Figure 5: Each Reality Skins World (RSW) is a blue-print for dynamically creating 3D worlds. Each RSW contains materials and code needed to build the world's layout. Sample materials for each RSW are shown on the three floating cubes, on the right of each sub-figure. At the heart of each RSW is a set of objects with attributes and tags attached. The RSW's shown are (a) Abandoned Hospital (b) Medieval Village (c) Space Station (d) Cave.

An **RSW** is a blueprint containing all the rules, materials and objects required to construct a **Reality Skin**. In practice it is a combination of data and code by a designer/developer, and used by the RS algorithm to create the final 3D virtual world. The RSW contains two main parts. The first is a set of instructions detailing how to create the layout of a 3D world, given a 2D floor-plan of the scene. The second is a set of objects, complete with attributes and constraints, which are used by the RS algorithm to populate the virtual world.

In section 3.2 we described how we infer a complete 2D floor-plan for the scanned environment. The most straight-forward way to extrude a 2D plan into 3D is construct a polygon for the floor, ceiling and each of the walls in the floor-plan. Wall height, as well as positioning the ceiling is set to the height of the original ceiling plane. Each polygon is triangulated and given consistent texture coordinates. The designer may specify multiple rendering materials in the RSW, tagged appropriately (e.g. floor, walls) and assign them to each polygon. However, the designer may supply code which builds more elaborate geometry based on the simple 2D floor-plan. For example slanted walls of a space station, some of which overlook a distant nebula, or natural cave walls, closing close overhead. Ambient lighting may also be introduced in this stage. See section 6 for examples.

The RSW contains a data-set of objects, instances of which will be used to populate the Reality Skin. Each object is typically a 3D mesh complete with rendering properties (such as material). In the data-set, each object is also assigned different attributes, used later in the RS optimization. The attributes are detailed in table 1[5].

Attribute	Type	Description
ID	uint	Unique identifier for the **RS** algorithm
Mesh	mesh	A reference to the mesh of the object (for analysis and rendering)
Tags	string list	Textual tags which can be used for semantic constraints
Enabled	bool	Allows an object to be temporarily left out of the optimization
InitT	M^{4x4}	An initial transform meant to rotate the object such that it's Y-axis is up, X-axis is forward, and scale it properly
SitOnFloor	bool	Constrain transformations to place the object directly on the floor
NearFloor	bool	Constrain transformations to place the object within 20cm of the floor
BackToWall	bool	Constrain transformations to place object adjacent to scene walls
StayUpright	bool	Constrain transformations to rotate only around up axis
MinHeight	float	Constrain object minimum height in scene
MaxHeight	float	Constrain object maximum height in scene

Table 1: **RS** Object Attributes

Once the RS optimization is complete, a set of object instances with transformations is retrieved. Code in the RSW is used to instantiate each object in the scene with the appropriate transformations. Custom code can at this point add additional props, lighting, particle effects and other game elements.

5 Reality Skins

The input to the **Reality Skin** algorithm is an RSW, and a scanned (and processed) environment. For each object in the RSW, we wish to find a discrete set of transformations, which optimally place it in the scene. The success metric for each object and transformation pair (as discussed in section 3.3) compares the feature vectors of the matching voxels in both object and scene.

The space of all possible transformations of an object is large, and can be any combination of translation, rotation, scaling (and possibly even deformations). We compose the transformation of each object O from two parts: a set of *base transformations* $B_1, ...B_K$, consisting of rotations (mostly around Y-axis), scaling and sub-cell translations, and a discretized translation T. This separation allows us to extract the feature vectors of an object for each B_i in advance (since it is translation invariant). We then apply a sliding-window algorithm comparing the cells of B_k to a specific location in the

[5]Note that the attributes *SitOnFloor* and *NearFloor* are mapped into *MinHeight* and *MaxHeight* for convenience.

scene volume. The location compared determines T, a translation of $cellSize * [w,l,h]$ where w,l,h are the indices of the compared location in the scene volume.

The number and composition of base transformations (BT) for each object are determined by its attributes. For example, objects which are flagged *StayUpright* are rotated only around their Y-axis. Objects which are flagged *NearFloor* or *OnFloor* have sub-cell translations (along the Y-axis) applied to them, in order to find fine alignment to the floor plane. We apply each BT to the object point cloud, and align the transformed point clouds such that they share a same-size volume, from $[0,0,0]$ to the maximum coordinates of the combined transformed BT point clouds. We extract feature vectors for each BT of the object (see section 3.3) and store the feature-vector grids.

5.1 Generating a Reality Skin

Given an object O with K feature-vector volumes (one per BT), we apply a sliding window algorithm over the scene feature-vector 3D volume (of size $W \times L \times H$), where the window size is determined by the volume of O (identical for all BT). For each position in the scene p and for each feature-vector volume k we calculate $score(p,k) = \sum_i \lambda(S(p+i), O_i^k)$ where i iterates over the feature volume of O^k, $S(p+i)$ is the feature-vector in the scene centered on position p (shifted by local iterator i). λ is the scoring function, comparing the feature vectors of a single voxel (cell) of an object with the matching scene voxel. The function is defined as

$$\lambda(s,o) = \begin{cases} 1.2 - \sum_i \frac{(s^n(i)-o_i^n)^2}{s_i^n} & \text{if } s^{occ} > 0 \text{ and } o^{occ} > 0 \\ -0.4 & \text{if } s^{occ} = 0 \text{ and } o^{occ} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\{s,o\}^n$ are the normal response normalized histograms (compared with χ^2) and $\{s,o\}^{occ}$ is the occupancy field. This definition of λ can be combined with semantic features, increasing the score per cell, if it is a semantic match for the scene cell. Note also that cell scores are summed per object (and BT) and not normalized, to reflect a global objective measured by voxel. Therefore a larger object can achieve a higher score, and a larger scene will elicit a larger overall score.

The result of the sliding window algorithm (for object O) is a $W \times L \times H \times K$ grid of real values, scores for a discrete set of transformations. We apply a non-max suppression algorithm to filter out near-identical scores (with a spatial radius of 5 and BT radius of 3 by default, which can be overridden per object), and retrieve the list of modes, i.e. transformations, for each object. Each final transformation for a mode is a composition of the BT and the grid-aligned translation. For visualization purposes in figure 6 we flatten the grid to its width-length dimensions by taking maximal value over height and base position. As can be seen, the non-max suppression distributes the modes over the scene volume (and reduces their number), which significantly speeds up the optimization process.

The selected modes (transformations) for each object are selected to satisfy the **unary** constraints applied i.e. its geometric and semantic fit to the scene, as well as designer specified constraints (e.g. should be two feet from a wall).

Minimizing collisions between the selected modes is of high importance when generating an RS. We check for collisions in two stage: First we compare axis-aligned bounding boxes between each pair of modes, which rules out a large percentage of the possible collisions. For all the collisions detected in this manner, we compare the *occupancy* for each cell between two modes and sum the results. Thus, the collision factor between two objects represents the physical collision volume. A collision between two modes is modeled as a pairwise constraint, and incorporated into the optimization.

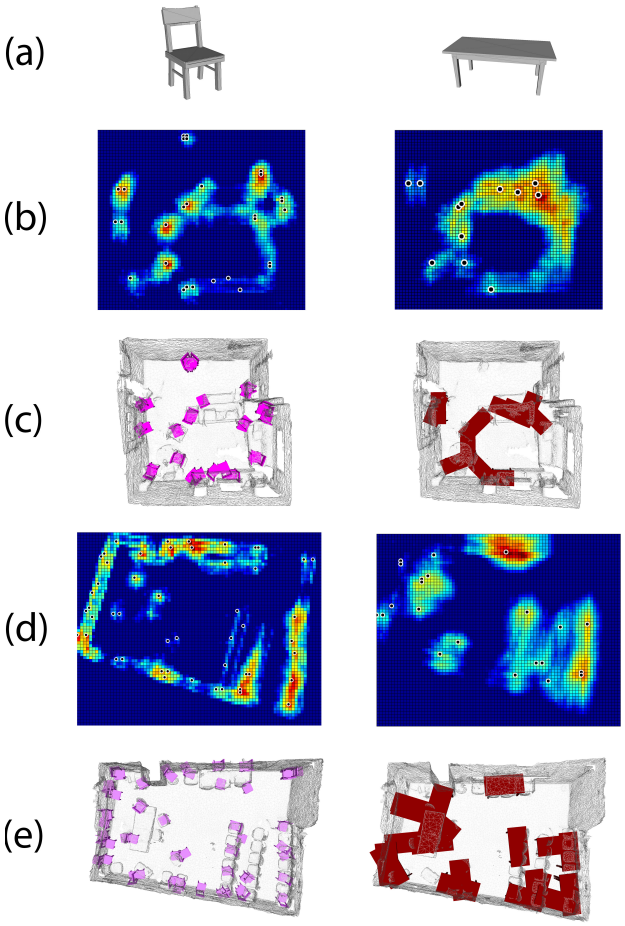

Figure 6: Finding the best transformations (modes) for each object in the RSW: (a) Two sample models from the Hospital RSW (b) Visualization of the score map for each model over the **xilab** scene (captured in Tango). The 4D score map (over scene volume and base positions) is flattened to the X-Z plane via *max*. (c) Non-max suppression is used to extract modes (transformations) from the score map. (d,e) Visualization and modes for the **mit32** model [38].

We formalize the problem as follows: Suppose there are n modes, corresponding to n potential objects, along with their transformation in the scene. For each object i, let the binary variable u_i indicate whether the object is to be included within the RS: $u_i = 1$ indicates the object will be included, and $u_i = 0$ means it will be left out. For object i, s_i is the score which reflects how well that objects matches the underlying scene – including geometric and semantic fit, as well as designer specified constraints, as described above. A higher value of s_i is better. For a pair of objects (i,j), the scalar c_{ij} gives the collision score between the two objects; the lower c_{ij}, the better.

We wish to solve the following optimization problem:

$$\min_{u \in \{0,1\}^n} -\sum_i s_i u_i + \frac{1}{2} \sum_{i \neq j} c_{ij} u_i u_j$$

If we define the matrix P by

$$P_{ij} = \begin{cases} \frac{1}{2} c_{ij} & \text{if } i \neq j \\ 0 & \text{otherwise} \end{cases}$$

then we may rewrite our problem in matrix-vector notation as

$$\min_{u\in\{0,1\}^n} -s^T u + u^T P u \tag{1}$$

This is a non-convex integer quadratic program, which in general will be NP-hard to solve. Progress has been made recently within the optimization community on how to attack such problems. In what follows, we rely on the very recent Concave Quadratic Cuts (CQC) technique of Park and Boyd, whose essential details we now present. The interested reader is referred to [27] for a fuller treatment.

5.2 RS Algorithm Based on CQC

The goal of CQC is to relax the non-convex integer quadratic program to a convex continuous problem, which we can then be solved using standard techniques from convex optimization. The relaxation begins by using the following lifting: let $U = uu^T$. Then the optimization in (1) becomes

$$\begin{array}{ll} \underset{u\in\{0,1\}^n, U\in\mathbb{R}^{n\times n}}{\text{minimize}} & \mathrm{Tr}(PU) - s^T u \\ \text{subject to} & U = uu^T \end{array}$$

Noting that the matrix U is positive semidefinite, a natural relaxation for the non-convex constraint $U = uu^T$ is $U \succeq uu^T$, where the notation $A \succeq B$ means $A - B \succeq 0$, that is, $A - B$ is positive semidefinite. The simplest relaxation of the integrality constraints is to ignore them, leading to the following optimization problem:

$$\begin{array}{ll} \underset{u\in\mathbb{R}^n, U\in\mathbb{R}^{n\times n}}{\text{minimize}} & \mathrm{Tr}(PU) - s^T u \\ \text{subject to} & \begin{bmatrix} U & u \\ u^T & 1 \end{bmatrix} \succeq 0 \end{array} \tag{2}$$

where the semi-definiteness constraint $U \succeq uu^T$ has been rewritten using the Schur complement. The optimization problem (2) is a convex problem, indeed a semi-definite program, which can be solved using standard packages. Let us denote the solution of this problem to be (u_o, U_o).

The result of this procedure will be a real vector $u_o \in \mathbb{R}^n$; but what we desire is a Boolean vector $u_* \in \{0,1\}^n$, which can indicate whether an object is to be included or excluded from the RSW. Thus, we require a rounding procedure. We follow the recently presented technique described in another paper by Park and Boyd [26]; although that paper is concerned with integer programming on convex quadratic functions, the rounding technique is independent of the convexity assumption.

The rounding procedure itself is quite intuitive: it is based on the fact that ideally, one would have $U_o = u_o u_o^T$, and that it was our relaxation that introduced a discrepancy between U_o and $u_o u_o^T$. One would therefore like to "sample the discrepancy" between these two quantities. To make this more formal, one generates a number of random samples $\tilde{u}^k$ from the Gaussian of mean u_o and covariance matrix $U_o - u_o u_o^T$. The samples will be still be real vectors, so one may round them element-wise to Boolean vectors, that is, each element is rounded to the nearest value in the set $\{0,1\}$, which we denote by $\hat{u}^k = \mathrm{Round}(\tilde{u}^k)$ Finally, $\hat{u}^k$ is adjusted by a series of local moves, in which each bit is allowed to flip; this adjustment is performed in a greedy fashion, where at each iteration the bit-flip corresponding to the best improvement to the objective function $-s^T u^k + (u^k)^T P u^k$ is performed. This procedure is terminated once there are no more local improvements to be made, leading to u^k. Finally, one selects the sample u^{k_*} with the optimal function value on the original objective function, i.e. $k_* = \arg\min_k \left(-s^T u^k + (u^k)^T P u^k\right)$. To summarize, the rounding procedure is as follows:

For $k = 1, \ldots, K$:

1. Sample $\tilde{u}^k$ from a Gaussian of mean u_o and covariance $U_o - u_o u_o^T$.
2. $\hat{u}^k = \mathrm{Round}(\tilde{u}^k)$.
3. Greedily perform bit-flips on $\hat{u}^k$ until no more improvements are possible, giving u^k and objective function value $v^k = -s^T u^k + (u^k)^T P u^k$.

Take $u_* = u^{k_*}$ where $k_* = \arg\min_k v^k$.

One detail remains, namely the number of samples K to use. According to the complexity analysis in [26], one should use $K = O(n)$; in practice, we take $K = 3n$.

Note that [27] introduces an interesting way to preserve integrality conditions on u, via the addition of a particular set of concave constraints. We have tried solving the Reality Skins problem with problem (2) as well as with these additional integrality-encouraging constraints, and have found that there is effectively no difference in the quality of solutions we have computed. The program with the integrality-encouraging constraints is, however, somewhat slower. Thus, for the remainder of this paper we solve problem (2) as is, without the extra constraints.

5.3 RS Algorithm Based on CQC With Bounds

A natural extension to the Reality Skins problem is to add in bounds on the number of objects. Here, we will focus on upper bounds, but it is straightforward to use an analogous technique to deal with lower bounds.

Suppose that each object i is endowed with a type $t_i \in \{1, \ldots, T\}$; as an example, $t_i = 1$ could indicate that object i is a chair, while $t_i = 4$ could indicate that it's a table. Our goal in specifying bounds is to ensure that there are not too many of any given object selected. Such constraints looks like

$$\sum_{i:t_i=t} u_i \leq b_t \quad t = 1, \ldots, T$$

Let us define the $T \times n$ matrix

$$A_{ti} = \begin{cases} 1 & \text{if } t_i = t \\ 0 & \text{otherwise} \end{cases}$$

The bounds constraints may then be written in matrix-vector notation as

$$Au \leq b$$

With these bounds, it is easy to amend the original semidefinite program as follows:

$$\begin{array}{ll} \underset{u\in\mathbb{R}^n, U\in\mathbb{R}^{n\times n}}{\text{minimize}} & \mathrm{Tr}(PU) - s^T u \\ \text{subject to} & \begin{bmatrix} U & u \\ u^T & 1 \end{bmatrix} \succeq 0 \\ & Au \leq b \end{array} \tag{3}$$

This problem is still a (convex) semi-definite program, and can handled by the usual solvers. The rounding procedure is also modified slightly, as we now wish to ensure that the new constraints hold on the rounded solution:

For $k = 1, \ldots, K$:

1. Sample $\tilde{u}^k$ from a Gaussian of mean u_o and covariance $U_o - u_o u_o^T$.
2. $\hat{u}^k = \mathrm{Round}(\tilde{u}^k)$.

3. If $A\hat{u}^k \not\leq b$, go to Step 1.

4. Greedily perform bit-flips on $\hat{u}^k$ until no more improvements are possible, where allowed bit-flips preserve $A\hat{u}^k \leq b$. The end of this procedure gives u^k and objective function value $v^k = -s^T u^k + (u^k)^T P u^k$.

Take $u_* = u^{k_*}$ where $k_* = \arg\min_k v^k$. In the above, the changes are the addition of Step 3, and the modification to Step 4, both of which enforce the constraints on the bounds.

6 Results

6.1 Data Collection and Implementation

We captured environments for Reality Skins (see also figure 2) employing: Google Tango [11], Structure [23] and Kinect [21]. We also used rooms from the Sun3D [38] dataset which have been reconstructed by [4]. Each device has different properties, e.g. the Tango is capable of scanning large environments and loop closures, but produces coarse and noisy meshes. The Structure and the Kinect provided high-quality meshes, but the localization was not stable, and could not scan large environments. Overall we collected 157 environments (65 Tango, 59 Structure, 19 Sun3D, 14 Kinect) ranging from offices to homes, small rooms to large spaces.

The RealitySkins framework and algorithm was developed on a desktop machine in a client/server architecture. Scene plane inference and classification was implemented as multi-threaded CPU code. The feature-vector extraction for the scene, and preprocessing of each object in an RSW was implemented on an Nvidia GPU using CUDA.

The sliding-window algorithm comparing the feature-vector grid of each object (with multiple base positions) with the scene was implemented on the GPU as well. In our implementation, each execution of the kernel computes a dense score map for one object. The grid size of the CUDA kernel is equal to the dimensions of the scene grid. The threads in each block correspond to the different base positions. By default each block contains 64 threads. If number of BT exceeds the default, each thread iterates over several. Each block first copies a sub-volume of the scene (corresponding to the block index) into shared memory, after which each thread executes the sliding-window comparison. Once all blocks finish execution, we activate a non-max suppression kernel, and copy back the modes of the object.

We then calculate collisions between the modes as previously described, and solve the optimization problem in (3) using CQC with CVX, a package for specifying and solving convex problems [13, 12]. The result of the CQC optimization is a list of selected modes. We save each result as a list of pairs: originating objects and associated transformations.

The design of an RSW as well as the experience of Reality Skins in virtual reality were implemented in Unity [35], a popular and accessible game engine. The designer develops an RSW in *edit* mode, by dragging and dropping objects into an RSW interface, and setting attributes and tags. In *play* mode, the application can process an environment from a pre-scanned mesh (or pointcloud) file, or access any depth camera with a Unity interface (which exists for all cameras used). Once an environment is scanned, the game engine requests service from the RealitySkins executable, passing a mesh and an RSW configuration. The returned result is a 2D layout for the environment, and instances of transformed RSW object instances.

Given a scene of dimensions $80 \times 30 \times 60$ (with a voxel size of 10 cm on a side), plane detection and classification takes $0.1ms$. Floorplan inference takes $3-5$ seconds. Processing a 10-object RSW (base transformations and feature-vector grids) takes 10 seconds. Detecting modes for all objects over the scene takes $5s$. Solving (3) using CQC takes $5s-15s$ (depending on number of modes and collisions). Note that each RSW in processed in advance. Therefore the total average time to produce an RS is $15s-25s$. We found that in single room environments this time was about $5s$ in practice.

6.2 Quantitative Evaluation

In order to evaluate the quality of the Reality Skins algorithm, we ran the algorithm on 30 different scenes, ranging from small offices, to living rooms, common areas and lounges. We applied four RSW's to each scene (Hospital, Space Station, Village and Cave) for a total of 120 data points. We normalized the energy by the 'perfect' score for each scene, calculated by summing the coverage score for each occupied voxel in the scene (i.e. matching the scene to itself), disregarding the edges of the scene. Furthermore we visually inspected each resulting RS, finding 87% of the results acceptable. The mean score was 0.57 (standard deviation of 0.14) with a distribution as shown in figure 7. We found that even results with a score of 0.2 produced an acceptable Reality Skin, once the world was textured and lighted properly.

The distribution of the overall results is shown in figure. Overall the mean normalized energy was 0.57 with a standard deviation of 0.14. We found that even a result of 0.2 produced an acceptable RS.

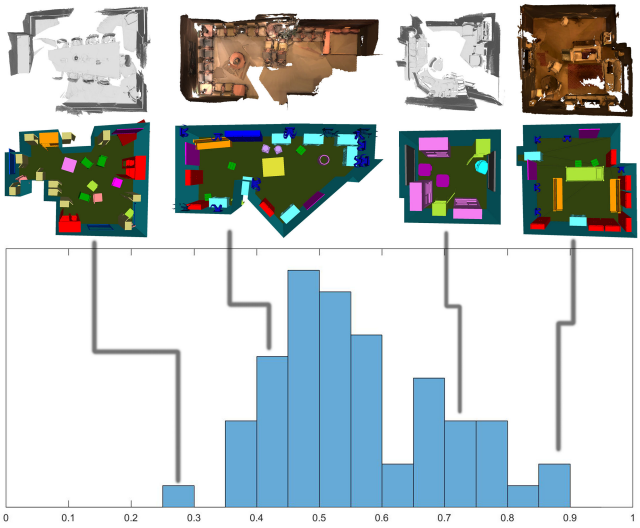

Figure 7: We performed a quantitative evaluation of our algorithm, creating Reality Skins for 30 different scenes over 4 RSW's. We normalize the score per scene and show the distribution. The top half shows four sample results from the evaluation, from left to right ConfRoom:Hospital (0.3), Dentist:Village (0.47), Office:Space Station (0.75), XI Lab:Village (0.88).

6.3 Qualitative Results

The best way to experience Reality Skins, is directly through a VR headset, experiencing the immersion, freedom of movement, and tactile truth. In figure 8 we show different RS's on the MIT-dorm [38]. The plane classification and layout inference is identical in all results. The processing time to generate each RS was $7-9$ seconds. In figure 9 we show additional results: Both (a) and (b) use the Space Station RSW. As discussed in section 4, we customized generation of the 3D floor-plan from the 2D layout, integrating slanted walls and transparent windows overlooking a planet. (c) shows a Cave RS in the XILab model. In the Cave RSW we replace the closed layout with free standing granite slabs and wooden fences. Instead of a closed ceiling, we add floating rough stone ceiling, which gives the illusion of a large space. We refer the reader to the supplemental material for additional results.

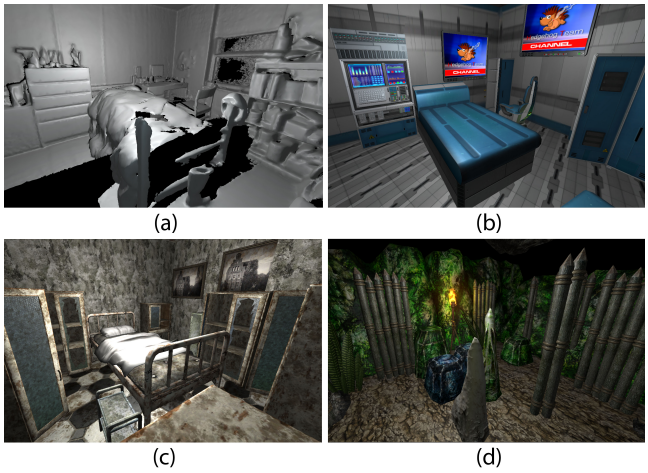

Figure 8: Reality Skin result: (a) We process the MIT Dorm model [38, 4], and produce a Reality Skin for (b) Space Station (c) Abandoned Hospital (d) Cave.

7 Conclusions and Future Work

We presented **Reality Skins**, 3D environments dynamically created to approximate the geometry and semantics of a user's surroundings, a step towards large-scale untethered VR. RS provides a framework for VR developers, to design a world which is adaptive, customizable and unique. We demonstrated how we process RSW data-sets in advance, and interactively build an RS given a large environment, using parallel computation and state of the art optimization techniques.

In this paper we focused on the constraint-based dynamic generation of 3D environments (Some failure cases are in figure 10). The next steps for RS require solving additional problems: Create an avatar for the user in the virtual world so she can touch objects or sit on chairs. A partial solution may be devices such as Leap Motion which can integrate hands into VR experiences. Mismatches which naturally occur between the RS and the environment may hinder the user's safety. When there is a large gap, we render a wireframe of the physical obstacle (similar to Vive's chaperone system). A more drastic example of this is the the static nature of the system, where we scan the environment once, generate a Reality Skin and do not update it, e.g. if a chair falls over. Recently advances have been made in dynamic SLAM pipelines which change over time, and which incorporate object-level detection. We intend to make use of these to refine a RS as the scene evolves. An interesting and crucial aspect of future work would be an extensive user study, exploring how safe and immersive users find RS environments, and their tolerance level for mismatches.

We envision different uses for RS such as adapting pre-designed games and experiences to the user's home, e.g. a game situated in a virtual kitchen which is modified and adapted to his real kitchen. A different use would be exciting single and multi-player games in a large environment such as an office building, re-themed as a space station infested with aliens which the players must vanquish. A third use might be as a gateway into a completely virtual experience: The user puts on his VR headset while on the sofa, intending to launch a racing game. Before starting the game, the sofa and living room might be modeled as a driver's lounge, perhaps integrating a virtual menu system, before letting the user jump into the game.

Mobile VR experiences such as Samsung GearVR [29] and to some extent Google Tango [11] demonstrate the move toward untethered VR, without the need for fixed cameras and sensors. We hope to see Reality Skins incorporated into these and future devices. In future work we would like to integrate more scene understanding and semantic constraints into the system, and enable even larger multi-stored spaces. We would also like to allow the user some authoring control over the generation of the 3D environment, superseding the fixed designer's constraints and objects.

References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[2] C. Charbonnier and V. Trouche. Real Virtuality. Technical report, Artanim Foundation, 07 2015. `http://www.artanim.ch/pdf/Real\%20Virtuality_White\%20Paper`.

[3] S. Chib and E. Greenberg. Understanding the metropolis-hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

[4] S. Choi, Q.-Y. Zhou, and V. Koltun. Robust reconstruction of indoor scenes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[5] 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.

[6] H. Edelsbrunner. Weighted alpha-shapes, 1992.

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

[8] Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski. Reconstructing building interiors from images. In *In Proc. of the International Conference on Computer Vision (ICCV*, 2009.

[9] R. Gal, L. Shapira, E. Ofek, and P. Kohli. Flare: Fast layout for augmented reality applications. In *Mixed and Augmented Reality (ISMAR), 2014 IEEE International Symposium on*, pages 207–212, 2014.

[10] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6):1115–1145, 1995.

[11] Google. Google tango tablet, 2015. `https://www.google.com/atap/project-tango/`.

[12] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. `http://stanford.edu/~boyd/graph_dcp.html`.

[13] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`, Mar. 2014.

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

[15] HTC. Htc vive, 2016. `http://www.htcvive.com/us/`.

[16] S. Ikehata, H. Yang, and Y. Furukawa. Structured indoor modeling. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[17] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera. ACM Symposium on User Interface Software and Technology, October 2011.

[18] B. Jones, R. Sodhi, M. Murdock, R. Mehra, H. Benko, A. Wilson, E. Ofek, B. MacIntyre, N. Raghuvanshi, and L. Shapira. Roomalive: Magical experiences enabled by scalable, adaptive projector-camera units. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 637–644, New York, NY, USA, 2014. ACM.

[19] O. Litany, T. Remez, D. Freedman, L. Shapira, A. M. Bronstein, and R. Gal. ASIST: automatic semantically invariant scene transformation. *CoRR*, abs/1512.01515, 2015. `http://arxiv.org/abs/1512.01515`.

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

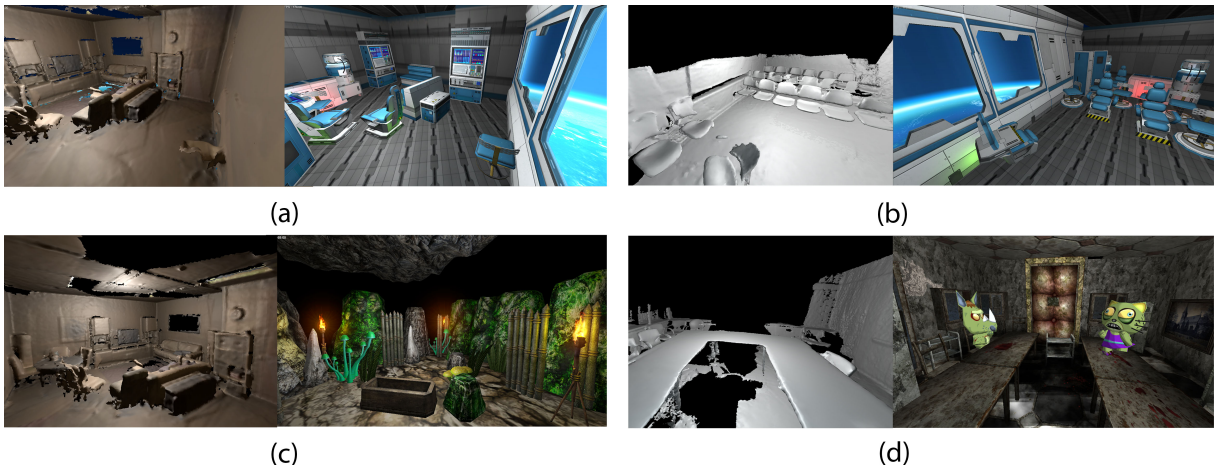

Figure 9: Four additional Reality Skins results: (a) Space Station in the XILab room (b) Space Station in MIT32 [38] (c) An underground cave in XILab (d) An abandoned Hospital complete with cute zombies in HarvardC5 [38].

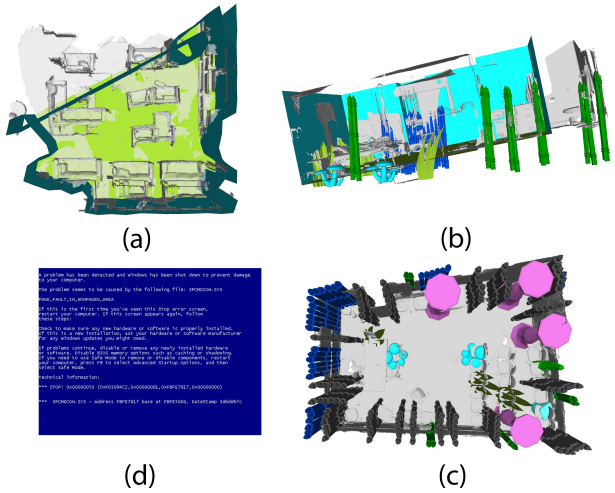

Figure 10: Failure cases: (a) Complex scene structure and missing walls can cause the floor-plan inference to go wrong (b) A noisy scan can cause misclassification of the floor plane and an unaligned volume (c) Large environments, large amount of modes and intensive use of GPU may cause a heart wrenching blue screen of death (d) In a few cases our optimized objective function simply does not produce pleasing results, often fixed by running the optimization again.

[21] Microsoft. Microsoft kinect sensor, 2015. https://dev.windows.com/en-us/kinect.

[22] C. Mura, O. Mattausch, A. J. Villanueva, E. Gobbetti, and R. Pajarola. Automatic room detection and reconstruction in cluttered indoor environments with complex room layouts. *Computers & Graphics*, 44:20 – 32, 2014.

[23] Occipital. Occipital structure sensor, 2015. http://structure.io/.

[24] S. Oesau, F. Lafarge, and P. Alliez. Indoor Scene Reconstruction using Feature Sensitive Primitive Extraction and Graph-cut. *ISPRS Journal of Photogrammetry and Remote Sensing*, 90:68–82, March 2014.

[25] N. Parikh and S. P. Boyd. Proximal algorithms. *Foundations and Trends in optimization*, 1(3):127–239, 2014.

[26] J. Park and S. Boyd. A Semidefinite Programming Method for Integer Convex Quadratic Minimization, Apr. 2015.

[27] J. Park and S. Boyd. Concave Quadratic Cuts for Mixed-Integer Quadratic Problems, Oct. 2015.

[28] R. F. Salas-Moreno, R. A. Newcombe, H. Strasdat, P. H. J. Kelly, and A. J. Davison. SLAM++: simultaneous localisation and mapping at the level of objects. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1352–1359, 2013.

[29] Samsung. Samsung gearvr, 2015. http://www.samsung.com/global/microsite/gearvr/.

[30] S. Sharples, S. Cobb, A. Moody, and J. R. Wilson. Virtual reality induced symptoms and effects (vrise): Comparison of head mounted display (hmd), desktop and projection display systems. *Displays*, 29(2):58–69, 2008. http://dblp.uni-trier.de/db/journals/displays/displays29.html#SharplesCMW08.

[31] N. Silberman, D. Hoiem, P. Kohli, and R. Fergus. Indoor segmentation and support inference from rgbd images. In *Proceedings of the 12th European Conference on Computer Vision - Volume Part V*, ECCV'12, pages 746–760. Springer-Verlag, 2012.

[32] N. Silberman, L. Shapira, R. Gal, and P. Kohli. A contour completion model for augmenting surface reconstructions. In *Proceedings of ECCV 2014*, ECCV'14. Springer International Publishing, September 2014. http://research.microsoft.com/apps/pubs/default.aspx?id=225020.

[33] A. L. Simeone, E. Velloso, and H. Gellersen. Substitutional reality: Using the physical environment to design virtual reality experiences. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3307–3316, New York, NY, USA, 2015. ACM.

[34] E. Turner and A. Zakhor. Floor plan generation and room labeling of indoor environments from laser range data. In *Computer Graphics Theory and Applications (GRAPP), 2014 International Conference on*, pages 1–12. IEEE, 2014.

[35] Unity3D. Unity, 2015. http://www.unity3d.com.

[36] J. Valentin, V. Vineet, M.-M. Cheng, D. Kim, J. Shotton, P. Kohli, M. Niessner, A. Criminisi, S. Izadi, and P. Torr. Semanticpaint: Interactive 3d labeling and learning at your fingertips. *ACM Trans. on Graphics (TOG)*, August 2015.

[37] Y.-S. Wong, H.-K. Chu, and N. J. Mitra. Smartannotator an interactive tool for annotating indoor rgbd images. *Computer Graphics Forum (Special issue of Eurographics 2015)*, 2015.

[38] J. Xiao, A. Owens, and A. Torralba. Sun3d: A database of big spaces reconstructed using sfm and object labels. In *Computer Vision (ICCV), 2013 IEEE International Conference on*, pages 1625–1632, Dec 2013.

[39] 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.

[40] 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.