DATA-DRIVEN TOOLS
FOR SCENE MODELING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Matthew Fisher
May 2013

Abstract

Detailed digital environments are crucial to achieving a sense of immersion in video games, virtual worlds, and cinema. The modeling tools currently used to create these environments rely heavily on single-object modeling: designers must repeatedly search for, place, align, and scale each new object in a scene that may contain thousands of models. This style of scene design is enabled by the large collections of 3D models which are becoming available on the web. While these databases make it possible for designers to incorporate existing content into new scenes, the process can be slow and tedious: the rate at which we can envision new content greatly exceeds the rate at which we can realize these imagined constructs as digital creations.

In this dissertation, we aim to alleviate this bottleneck by developing tools that accelerate the modeling of 3D scenes. We rely upon a data-driven approach, where we learn common scene modeling design patterns from examples of 3D environments. We show which properties of scene databases such as Google 3D Warehouse are most important for data-driven tasks, and how to transform existing scene databases into a form that is amenable to pattern learning. We also describe a custom scene modeling program which serves as a testbed for the modeling tools we develop, and which we use to create a curated corpus of scenes that enable the development of powerful modeling tools.

Our tools require the ability to compare arrangements of objects. We present several techniques to do so, including kernel density estimation and graph kernels, and show how these approaches can be applied to produce practical modeling tools. We use this machinery to support basic modeling operations such as searching for

or orienting single models. We show how to use a corpus of 3D scenes to automatically categorizing and aligning collections of objects by group objects into *contextual categories*. Finally, we combine these contextual categories and our arrangement comparison algorithm to enable example-based 3D scene synthesis, where the artist provides a small number of examples and we generate a diverse and plausible set of similar scenes. All of the methods we develop use a data-driven approach in order to enable the rapid construction of large virtual environments without the need for an artist to try and specify the "rules of design" for each possible domain.

Acknowledgments

I am grateful for financial support from the Fannie and John Hertz Foundation, which provided immense freedom in pursuing novel research directions. Through the Hertz Foundation I have met some of the most innovative and intelligent researchers whose broad areas of expertise continuously push me to find new ways to connect my work to a wider audience. Several of my projects were also generously supported by the Intel Science and Technology Center for Visual Computing.

I would like to thank my amazing advisor Pat Hanrahan for supporting me and my research group through our diverse and sometimes risky research directions. Pat's advising style is unique and his lab cultivates a style of research that continuously makes advances in the most unexpected areas. Pat also connected our group with Tom Funkhouser at Princeton who continues to be an invaluable collaborator.

Scott Klemmer and Barbara Tversky both generously agreed to read drafts of this thesis and their contrasting backgrounds have contributed greatly to its structure and style. Scott has always encouraged me to think very cautiously about who might use the technology we develop and why, and my group is indebted to Barbara for provided us with a strong psychological basis for our work.

In high school, Mary Whitton provided me with my first introduction to the research project at the virtual reality lab at UNC Chapel Hill. My undergraduate advisors Mathieu Desbrun and Peter Schroeder were also vital to my development as a researcher and continue to be important sounding boards for my career and research directions.

All of my work has been presented at the SIGGRAPH and SIGGRAPH Asia conferences, and I am grateful to the sometimes exhausting but always extensive and

rewarding reviewers who make these journals and conferences such a powerful force in the graphics research community.

Finally, my most important thank you goes out to Mom, Dad, Megan, and Max for their love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Environments in the real world are characterized by a richness in object diversity and complexity. Attempting to enumerate all the unique objects in a house could end up with a list containing tens of thousands of objects — especially if one considers every unique book or brand of cosmetic a distinct item. The arrangement of objects in environments like this are typically built up through a combination of intentional effects (such as books placed on a bookshelf) and incidental effects (such as socks left on the floor). Environments that lack this richness in object diversity and arrangement often feel stale and artificial — the entire home staging industry is built around this premise.

To feel immersive, virtual environments require object richness comparable to that of the real world, and yet at present the task of modeling scenes containing many objects is an extremely time consuming task. This content creation bottleneck is an endemic problem in the construction of 3D environments: we can think up new content much faster than we can transfer it to a digital representation. There are many different ways to decompose the task of modeling digital content. We use the term *object modeling* to refer to the modeling of individual objects such as a television or a mug. We use the term *scene modeling* to refer to the modeling of environments composed of many objects such as a laboratory or a bedroom.

The goal of this dissertation is to present novel approaches to overcoming the content creation bottleneck, focusing on the task of modeling 3D scenes. Although each

digital creation is ultimately unique, it is never totally independent of all previous design tasks. Through our experiences with the real world or other imagined content we establish a strong prior over the types of entities we might expect to create. A human is typically characterized by two feet, two hands, two eyes, and a mouth, and a bedroom typically contains a nightstand, a dresser, a blanket, and a bed. These repeating design patterns are pervasive in both real and virtual worlds. Understanding these patterns allows us to anticipate the intent of an artist as they create new content. With sufficient mastery of a design pattern we can perform complex tasks such as automatically instantiating a pattern many times to create novel variations of the same type of artifact.

There are many different ways we might learn patterns that occur in 3D content. One option is to have humans explicitly encode rules about the patterns. This is challenging because it is not straightforward to enumerate the design principles for a domain: what rules describe a mad scientist's laboratory or a messy college dorm room? Furthermore, rules must be quantified and prioritized. General concepts such as "nightstands occur near beds" must be annotated with numeric values that provide a distribution over allowable distances that are not as natural for humans to specify. Another option is to learn design patterns from examples. Making examples is a very common design task and one well suited to humans. The challenge is then to learn from examples which of the many possible relationships are incidental and which are the important design principles for the domain being exemplified. In this work, we focus on learning design patterns from examples of 3D environments.

1.1 Virtual Worlds

3D content creation is a task performed for a wide variety of target applications including video games, animated movies, and special effects for film or advertisement. The types of tools that will best improve the content creation bottleneck will depend on which application is being targeted. A motivating application for the research in this dissertation is the task of designing virtual worlds. Virtual worlds are large digital environments designed to support the simultaneous interaction of many users.

Modern virtual worlds are characterized by expansive 3D scenes populated by complex architecture and contain both interior and exterior designs. Some worlds, such as Blizzard Entertainment's World of Warcraft, are designed by a dedicated team of professional artists. Other worlds, such as Second Life, are designed incrementally by casual users as part of their interaction with the environment.

The larger virtual worlds have millions of users and development times in the millions of person-hours. Since high user interest is necessary to sustain a monthly subscription model, popular virtual worlds also undergo continuous (and expensive) expansion: World of Warcraft has released four large expansions over the course of a decade each containing large, new environments that share many design patterns with the original release of the virtual world. The total revenue from an American user with a continuous subscription to World of Warcraft over its current product lifetime is 1500 USD, which greatly motivates Blizzard Entertainment to maintain continuous interest in the world. The popularity of virtual worlds combined with their lengthy construction times indicates that there is a strong desire for tools that make it faster and easier to design virtual worlds.

Here, we briefly explore some of the properties of virtual worlds that will guide our development of scene modeling tools:

- **Design Pattern Coverage** — To support the interaction of multiple users and to maintain user interest in the world, virtual worlds contain a large number of diverse environments. This is important because it provides us good coverage over the space of possible models and the relationships between them. As with most knowledge extraction tasks in machine learning, the quality of the results improves with increasing amounts of data.

- **Artistic Design** — Environments in virtual worlds are often modeled by a dedicated team of professional artists. These artists design the world with the intention that various zones embody a distinct set of construction and design styles. This dataset provides a rich opportunity to explore how to learn and adapt to the many stylistic variations in the world.

- **Segmentation and Tagging** — To facilitate user interaction, many virtual

worlds provide a good segmentation of the scene into meaningful objects. This allows users to perform actions such as "sit on that chair" or "pick up that plate". A good segmentation of the scenes into meaningful objects is important because we need to learn relationships between objects. Likewise, many objects in these environments are tagged by artists to help with the maintenance and upkeep of the world. Good model tagging is useful because tags can relate objects that are geometrically distinct but functionally very similar, and tagging is a natural way to query a model database.

Unfortunately, one key obstacle preventing virtual worlds from being used as a data source is accessibility. Few virtual worlds were designed with the intention that the decomposition of their environments into meaningful objects be readily accessible by people outside the designers of the world such as graphics researchers. Although the inaccessibility of commercial virtual worlds requires us to use examples drawn from other collections of 3D environments such as Google 3D Warehouse, the tools we develop are designed to take advantage of the properties that might be expected from virtual world datasets.

1.2 The Basic Scene Modeling Pipeline

1.2.1 3D Model Collections

Attempting to model a large environment by specifying the precise geometry, down to the level of individual triangles, for every object in an environment is a considerable undertaking. Instead, most scene design tasks are performed by using models designed by other artists. Typically, these models are drawn from a 3D model collection such as Google 3D Warehouse or the Blender Model Repository. Sometimes these collections are maintained by communities of modeling enthusiasts in order to show off their work and allow others to reuse their creations. Other model collections are maintained by professional companies for a specific use such as a virtual world. In all cases, an artist modeling an environment needs to be able to search the database to find models they want to use in their scene. There has been significant research

in evaluating different approaches to querying 3D model databases such as searching using keywords, searching using 2D sketches, and searching for models similar to an existing 3D model [Min 2004]. All previous search algorithms are designed for searching for models independent of the task these models are being used for. In this work, we will show that it is possible to take advantage of knowledge about the scene being modeled to improve querying 3D model databases for the task of scene modeling.

1.2.2 Modeling Interface

There are a wide range of programs that can be used for 3D modeling tasks, such as Autodesk 3ds Max, Autodesk Maya, Google SketchUp, and the open source Blender. Each 3D modeling program focuses on different use cases and depending on the program different tasks will be significantly easier or more challenging. For example, 3ds Max has many tools that focus on adding fine-scale details to 3D surfaces, while Google SketchUp is designed for the rapid prototyping of buildings, homes, and other large 3D environments. To understand what tools might benefit artists modeling scenes, we need to understand the pipeline that is currently used in existing modeling programs. We center our analysis on Google SketchUp (version 8) since this program focuses on ease-of-use and designing large environments, but the scene modeling process does not vary significantly between the popular modeling programs. Google SketchUp is also tightly integrated into a community-driven 3D model corpus called Google 3D Warehouse which greatly facilitates the construction of richly decorated environments. We do not focus on the modeling of architecture since this varies significantly across programs and requires very specialized design constraints and tools to model correctly. For some domains such as houses, automatic tools have been developed to help with this process [Merrell et al. 2010].

We start by assuming that the user has some very high level idea of what they want to model, such as a house or a shopping mall. They then begin the following iterative process:

- **Think of something to add** — Based on their observation of the current

environment, the artist needs to think of a model that is missing from the scene and needs to be added.

- **Search for the model** — Given a concept of what to add, such as "a large flatscreen television", the user needs to acquire an acceptable model of this type. In Google SketchUp, this is accomplished by bringing up a website view to Google 3D Warehouse and performing keyword searches. Once found the model is added to the current SketchUp environment. If an appropriate model cannot be found, the artist may need to model its geometry themselves or think of a new type of model they wish to insert.

- **Insert the model into the environment** — Once a model is added to the environment, SketchUp does not attempt to match the scale of the imported object to the current scene, or to translate it to a meaningful location. The artist needs to scale the object to an appropriate size, and navigate the active camera to point to the location where they want to insert the object. Once they are at the target location they need to translate the object to its final position, and rotate it into a desirable orientation.

These tasks are not performed in a linear order: artists often think of many objects to insert at the same time, it is common to first navigate to the location where an object is going to be inserted to help visualize what types of objects might be most appropriate to insert, and the location, scale, and rotation of existing objects are continuously tweaked as new objects are added or undesired objects deleted. Nevertheless, the above tasks of thinking, searching, and placing are extremely common operations in the modeling pipeline used in Google SketchUp and will guide our development of scene modeling tools. We refer to operations such as navigating, searching, scaling, rotating, and translating as *low-level modeling operations* because they have a single, well-defined effect and are the simplest modeling operations exposed in modeling programs like SketchUp. These operations are sufficient to complete modeling tasks which have a small number of objects, but they rapidly become cumbersome for scenes with thousands of distinct objects.

1.3 Modeling Tools

We want to use knowledge of design patterns that we learn from examples to enable tools that improve the process of modeling large scenes. We break down the tools we develop into roughly two categories. First, we develop tools that speed up different stages of the standard "think, search, insert" scene modeling process described above. Second, we develop tools that enable entirely new types of modeling operations. Because these operations perform a large number of simple operations at once, we refer to them as *high-level modeling operations*. Here we briefly summarize some of the tools we develop:

- **Scene retrieval and suggestion** — This tool looks at the current scene the user is modeling and finds similar environments in a backing scene database that may contain models relevant to the artist. This can help the artist think of new models to add or give them inspiration in the form of possible variations in style or composition explored by other artists [Lee et al. 2010].

- **Context-aware model search** — As a user adds models to an environment, we can gain a lot of insight into the types of models they might add next. This tool searches for possible new models to insert, conditioned on the scene being modeled and other types of information that might be specified by the artist. For example, the user might place a bounding box in the scene to search for models appropriate to that size and location, or they might ask for objects with a certain relationship to an existing object, such as "give me objects that belong on top of this nightstand".

- **Object categorization and alignment** — To help deal with the sheer number of objects in the world, humans group objects into categories according to various properties such as function, size, and location. This tool automates the process of partitioning a large set of objects into disjoint categories, and aligns the objects within each category into a canonical coordinate frame. Having an aligned set of categories enables rapid exploration of the space of possible models for a scene. For example, a user might insert a television into their room,

then rapidly cycle through a large number of other televisions without having to manually search for, replace, scale, and align each new television they wish to try out.

- **Synthesis from examples** — This tool learns a model for a type of environment from a small set of examples and generates more environments in a similar style. The tool introduces novel variations on the input examples to create a diverse set of results while respecting the important design constraints and preserving the functionality of the environment, without the need for the user to enumerate what relationships are important and which are not. The artist can then rapidly browse a large number of generated results to find desirable scenes. The synthesis can be controlled by the artist by either modifying the examples or constraining the sampling process.

1.4 Comparing Arrangements

Few interesting design tasks occur in exactly the same state. Each instance of patterns such as bedroom layouts or a blacksmith's forge will be related to each other but ultimately unique. To learn these patterns we need the ability to compare one arrangement of objects to another arrangement in order to understand what data is most relevant to a user's request. The task of comparing arrangements is challenging because one must simultaneously compare individual entities such as two 3D models along with the relationships or connections between sets of such entities. This is a commonly recurring problem in many scientific disciplines, and we will look to many other fields for inspiration when designing our modeling tools.

One instance of the problem of comparing arrangements is object recognition in photographs, where researchers have looked at using context information to disambiguate between visually similar objects [Rabinovich et al. 2007]. Another instance is the problem of comparing two images, where researchers have reduced images to segmentation graphs then defined a graph kernel between two such graphs that can capture the structural similarity between images [Harchaoui and Bach 2007]. Finally,

the problem of understanding protein folding and function requires comparing the molecular arrangement at two different binding locations [Bagley and Altman 1995]. In each case, different techniques have been used to solve the problem of comparing arrangements, and we will build upon this research as we adapt their approaches to the scene modeling tools we develop.

One commonality among most solutions to the problem of comparing arrangements of entities is to decompose the problem into two comparison tasks: how to compare two entities in isolation, and how to compare the arrangements given the similarity between all the relevant entities. We will make use of the same decomposition when comparing arrangements. In our case, the entities being compared will be 3D models.

1.5 Dissertation Road Map

The majority of this dissertation is devoted to the technology that supports the tools described above. We organize this work into the following chapters:

Chapter 2 describes different datasets that can be used to learn design patterns from examples. It looks at what dataset properties are most important for the tools we develop, and which of these properties are missing from scene databases such as Google 3D Warehouse. We also detail Scene Studio, a design tool we developed that serves as a practical testbed for our scene modeling tools.

Chapter 3 presents our method of comparing two objects in isolation, a subcomponent required by many of our tools. Objects are compared based on their size, geometry, texture, name, tags, and text description.

Chapter 4 describes context-based search. It presents an algorithm for this type of search based on the Visual Memex Model, an approach used in computer vision to classify unknown objects by using kernel density estimation over a set of training examples [Malisiewicz and Efros 2009].

Chapter 5 presents our work on applying graph kernels to scene modeling tasks. We first transform scenes into a relationship graph whose nodes represent objects and whose edges represent different types of relationships between nodes. Using this

representation we show how to apply established graph comparison techniques to support 3D model context search and scene retrieval and suggestion tasks.

Chapter 6 develops a generative model for 3D environments. An artist provides a small number of examples of a 3D scene, and our algorithm synthesizes new scenes of a similar type. This model uses the machinery developed in Chapter 4 to group models into *contextual categories* that are a set of functionally or semantically interchangeable objects. The scene comparison algorithm developed in Chapter 5 is leveraged to produce plausible, diverse scenes by drawing additional training samples from a larger database of scenes.

Chapter 7 discusses the scene modeling task as a whole. It highlights the areas where our tools are most effective and suggests areas for future development.

Chapter 2

Datasets

We want to learn general design patterns by observing many different examples of 3D scene design tasks. The first step towards accomplishing this is to acquire a set of examples of 3D environments that exemplify the design patterns we are trying to learn. 3D scenes can be described in many different formats ranging from a list of objects to range scans to unstructured triangle soups. We start by going over the properties we want a 3D scene corpus to have for pattern learning tasks. We then contrast this against existing scene corpora and show how to recover some of the properties that are most important for our scene modeling tools. Finally, we describe Scene Studio, our scene modeling tool, and the properties of the dataset generated using this tool.

2.1 Ideal Examples

Some scene representations make it very easy to extract design patterns, while others, such as unstructured point clouds, make it very challenging. Since data-driven scene modeling tools are not yet common, most existing scenes are not designed with data-driven tasks in mind so are missing many useful properties. Below we delineate the properties that ideal examples would possess.

- **Many environments** — The more examples we have, the more design patterns they exemplify and the more information we can learn about each pattern. As

with most data-driven tasks, having sufficient amounts of data often allows simpler approaches to be effective and combats overfitting or noisy data. As a concrete example, in natural language processing a very simple technique known as Stupid Backoff proves surprisingly effective at dealing with certain data processing tasks once sufficient examples are available [Brants et al. 2007].

- **High object detail** — Because adding object detail to 3D content is time consuming, many existing virtual environments are very barren. They contain rooms with only a few pieces of furniture and no decorative or functional objects. Even among scenes with some functional objects, examples are best if they contain detail at the same level that is desired in the environments an artist is designing.

- **Relevant environments** — Any set of examples is going to encode only a small subset of possible design patterns. For a data-driven tool to be useful, it needs to have examples that encode patterns that are desired by artists using the tool. A database of interior, 21st century scenes might share only a few design patterns that are useful for artists modeling an Elven treehouse or a 25^{th} century spaceport.

- **Clear object segmentation** — If we want to learn what possible arrangements of objects correspond to a given design pattern, we need to first understand how to decompose a scene into meaningful objects. Oversegmentation down to the level of polygons makes it very challenging to recover meaningful relationships, and failure to segment below the level of collections of objects such as "bedroom" or "house" prevents us from learning relationships between the objects that composed these scene categories.

- **Semantic annotation** — To support rendering, all objects in our environments must contain certain properties such as geometry, texture, and a material description. Although humans can usually infer the function and type of an object from these properties, modern algorithms to compare geometry are much less effective at this task. To achieve good semantic correspondence between

two functionally similar but geometrically different objects, our algorithms will benefit greatly from per-object text labels that describe their type, function, or style. Text information is also necessary to support many common search techniques such as keyword search.

The best source of examples for a tool are environments made for the tool's target application. A tool meant to help with home rearrangement tasks should draw examples from well-designed homes. This is what makes virtual worlds so appealing as a target application of data-driven modeling tools: they already contain a large number environments that encode relevant design patterns with an acceptable level of object detail and typically have a good decomposition into meaningful objects to enable user interaction with the environment.

2.2 Google 3D Warehouse

A growing demand for massive virtual environments combined with increasingly powerful tools for modeling and visualizing shapes has made a large number of 3D models available. These models have been aggregated into online repositories that other artists can use to build up scenes composed of many models. Some of these databases are publicly accessible on the web. Unfortunately, a 3D model database that contains only isolated objects, such as the Princeton Shape Benchmark, provides no information about the relationships between objects in real scenes [Shilane et al. 2004]. Instead we need to focus on scene databaes that contain collections of objects. The most popular such repository that is publicly accessible is Google 3D Warehouse, which we will use extensively throughout this dissertation. Here we explore some of the important properties of Google 3D Warehouse.

- **Individual components and complete environments** — Google 3D warehouse contains many different types of models that have been uploaded for a variety of purposes. Some are individual objects uploaded by artists or corporations with an interest in making their models available such as furniture manufacturing companies; Google SketchUp refers to these as components. Other

models uploaded to Google 3D Warehouse are complete environments, typically created by combining components uploaded by other artists. Finally some models on Google 3D Warehouse do not fall into either category, such as 3D reconstructions of buildings intended for use on Google Earth or Google Maps. The distinction between individual components versus complete environments is not explicit: users searching for individual components will often find complete scenes and vice-versa.

- **Textual information** — As an artist uploads each model to Google 3D Warehouse, it is annotated with three different types of textual information: a model name which describes the type of model, model tags which further refine the object type or describe its style or function, and a longer model description. The textual information available for a typical Google 3D Warehouse component is shown in Figure 2.1.

- **Types of environments** — Because Google 3D Warehouse is community-driven, there is no specific style of environments common to all scenes. Users model a wide range of environments ranging from table decorations to dorm rooms to spaceship interiors, each with a wide range of artistic styles and object densities. This results in a very large number of design patterns that can potentially be learned.

Unfortunately there is no simple way to acquire all models on Google 3D Warehouse. Instead, we accumulated a set of scenes by searching for keywords that suggested scenes with multiple objects ("room", "office", "garage", "habitación", etc.). We then manually filtered out scenes that represented individual objects, 3D buildings intended for Google Earth, or otherwise did not correspond to complete environments. After filtering, in total we acquired 4876 scenes this way.

2.3 Scene Processing

Before we can learn patterns from these Google 3D Warehouse scenes we need to segment the scenes into meaningful objects and (if possible) acquire tags for the objects.

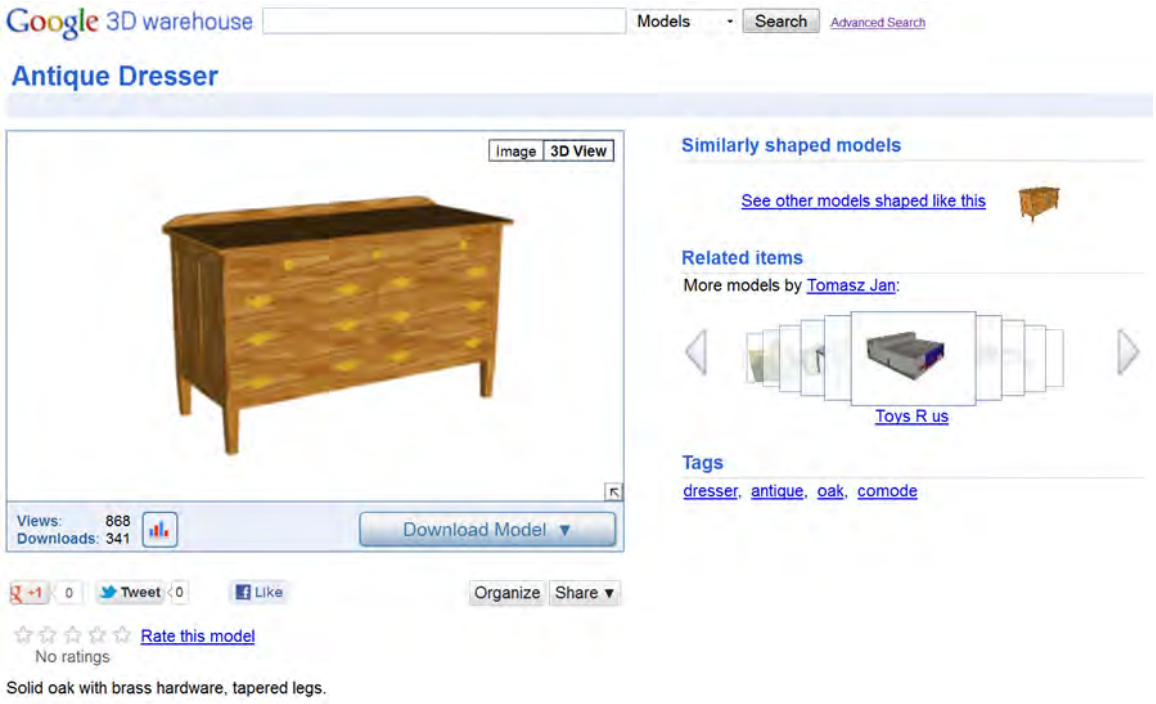

Figure 2.1: A typical Google 3D Warehouse component. Artists annotate uploaded models with a name, set of tags, and text description.

Our first step is to convert all the scenes in the input database into a standardized format (we chose to use the COLLADA file format because many different applications can process this format). The scene graph encoded in a typical COLLADA file from Google 3D Warehouse is shown in Figure 2.2. The text labels attached to each node are optionally added by the artist as they model the scene and are much less detailed than the textual information attached to individual components in the Warehouse which contain a name, tags, and description. Most scene graph nodes have no textual information at all.

2.3.1 Segmentation

The first step is to segment each scene graph into semantically meaningful objects. In COLLADA scene graphs, each node may point to any number of child graph nodes and to any number of child geometry objects. As can be seen in Figure 2.2, some nodes

Name	Processed Tags
LuminAiria	None
rouage	Cog
SimpleDome	Simple, Dome
11 by 8 paper	Paper
FineLiner	Fine, Liner
Box Deckel	Box, Cover, Lid
Box halter	Box, Halter
Stift	Pin, Pencil
Stiel	Handle, Stem
Eames Softpad Mgmt. Chair	Soft, Pad, Management, Chair
HSeat	Seat
HFrame	None (frame is a stop word)
iPhone	iPhone
Macbook Pro Open	Macbook, Pro, Open

Figure 2.2: Top: Typical scene and its scene graph decomposition (some nodes omitted for brevity). Objects are labeled with their raw node names. Bottom: Scene graph node names and their final processed set of tags.

such as the Eames chair represent complete objects. Other nodes, such as the children of the Eames chair, represent parts of objects. Distinguishing between whole objects and parts of objects is challenging. Although there has been work on segmenting 3D objects, this is often focused on segmenting individual objects into components and not entire scenes into semantically meaningful objects; we were unable to find an existing automatic approach that outperformed the scene graph segmentation for our dataset [Chen et al. 2009].

Considering all the scene graph nodes to be objects would result in considerable oversegmentation of the environments. We start by performing some limited filtering. We ignore nodes with common names that we found to be extremely suggestive of being an object part, such as "plant stem" or "flower petal" (this list was constructed by sorting the list of node names from most to least frequent followed by manual inspection and rejection).

After performing this limited filtering, we have a reasonable decomposition of each environment into objects, but it still contains some oversegmentation as there are several nodes that would not be judged to be individual objects by humans. Nevertheless, the quality is sufficient for algorithms that are robust to errors in segmentation, such as the Visual Memex approach to model search described in Chapter 4. For algorithms which cannot easily tolerate significant errors in segmentation, such as the tools based on graph kernels described in Chapter 5, we use a crowdsourcing approach to manually filter a subset of the scenes by asking a human to classify each potential scene graph node as either meaningful or not.

2.3.2 Tagging

Additional information about the object can be inferred from the names associated with the objects in the scene. In our Google 3D Warehouse dataset we gather naming information from three sources:

1. Scene graph nodes are sometimes named by the artist, as seen in Figure 2.2.

2. The root node of each scene (which corresponds to the entire file) is named by the artist as it is uploaded.

Scenes:	4,876
Scene Graph Nodes:	426,763
Objects:	371,924
Unique Models:	69,860
Tagged Models:	22,645
Shared Models:	10,509

Table 2.1: Current snapshot of our database. A tagged model is a model with at least one tag. A shared model occurs in at least two scenes.

3. The same model can be used in multiple scenes. We union the names from all instances of the model.

Unfortunately, there are still many semantically meaningful objects in our database that are poorly labeled or not labeled at all. This motivates the geometric comparison term we describe in Chapter 3.

To improve the chances of successfully comparing two objects using their tags, we start by cleaning up the names and translating them to English. First, we use Google auto-suggest to perform spelling correction and word separation (e.g., "deskcalender" becomes "desk calendar"). Second, we use Google Translate (which can auto-detect the source language) to convert Unicode sequences to English words. Finally, non-English words and stop words are removed.

The second step in the tag processing pipeline is to add related words to each source word. WordNet is used to find the most common hypernyms and synonyms of each word [Fellbaum et al. 1998]. Recall that hypernyms are enclosing categories of a word; for example, color is a hypernym of red which is a hypernym of crimson. We refer to the final set of words as the tags for that object. Figure 2.2 shows a set of scene graph nodes, their raw names, and their processed tags.

2.3.3 Summary

Table 2.1 gives a summary of the dataset after processing. As we will show, this is a rich dataset and can be used to learn contextual relationships between a wide range of objects. However, it has several flaws which make it unsuitable for some

of our more complex modeling tools. First, some scenes remain poorly segmented even after manually removing object parts as described above. This typically occurs when objects are created in certain ways, such as physically extruding the wall to produce books on a bookshelf, or when geometry is imported from another program and the model editing program does not maintain the decomposition into individual components. Second, some scenes are extremely barren and contain only one or two decorative objects, which is typically below the density our tools are targeting. Third, even though many components in these environments are imported from Google 3D Warehouse, the link back to the original model database entry is lost. This makes it difficult to recover a good textual description for all objects in the corpus, which is important for many of the tools we design.

2.4 Scene Studio

Here we briefly describe Scene Studio, a modeling program developed to produce a dataset of scenes that overcome some of the problems with scenes acquired from Google 3D Warehouse and to test the viability of some of the modeling tools developed in this dissertation. Unlike most popular modeling programs, ours does not allow the modeling of individual objects and instead focuses entirely on modeling scenes populated with objects from an existing 3D model database. To obtain a set of base models to be composed into complete scenes, we crawled Google 3D Warehouse using keywords commonly found in interior scenes such as "monitor", "chair", and "wallet". We then manually removed search results that contain multiple objects, such as desks that already contain a chair or computer. Users modeling with our program first select a desired base architecture and then repeatedly search for objects in this pruned model database to insert into their scene. Figure 2.3 shows a screenshot of this modeling interface.

Our modeling program produces scenes that have several important properties for our algorithm:

- **Segmentation:** Because the scene is composed entirely of models from the underlying database, we can easily maintain a good segmentation of the scene

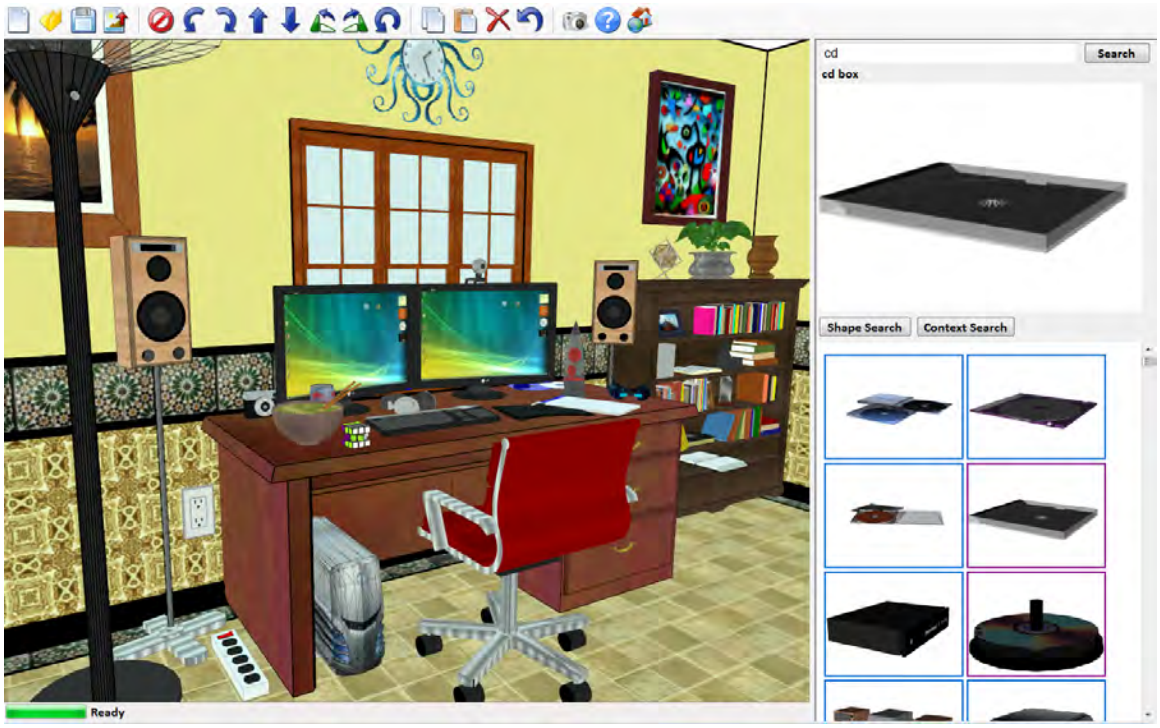

Figure 2.3: Our modeling interface for interior scene design.

into meaningful objects.

- **Tagging:** Models in our database come directly from a single entry in Google 3D Warehouse, which we use to obtain a good name, set of tags, and textual description for each model.

- **Parent contact:** When a user inserts an object, it is always rooted at a specific point on the surface of an existing object in the scene. Although the user is free to then further displace the object, this feature was rarely used when modeling static scenes. Knowing an object's parent avoids the need to guess object contact relationships and makes modeling easier because modeling operations on the parent can be implicitly extended to its children. The set of all parent contact information for all objects in a scene defines a static support hierarchy. We make use of the properties of this hierarchy compared to a traditional scene graph representation in our high-level modeling tools.

The model database used by Scene Studio contains 16847 models, including objects intended for both interior and exterior design. We distributed the program to users on the web to acquire a corpus of 3D environments to use as a dataset for our modeling tools. We asked participants in our corpus-building effort to model any scene they liked using our software. We reminded users to add detail objects such as light switches and power outlets to their scenes. The majority of our participants were students and staff from our university's computer science department, or their friends and family. In total our participants generated 154 scenes, containing 4651 model instances and using 2970 distinct models. We readily acknowledge that this dataset quality is very high and is not yet representative of current modeling programs such as Google SketchUp. Nevertheless, we feel that this dataset reflects the type of data that is available to the designers of large 3D environments such as virtual worlds and that it serves as a viable testbed for our modeling tools.

We provide this dataset as a resource to the research community; it can be found on the project web page: http://graphics.stanford.edu/projects/scenesynth.

Chapter 3

Model Comparison

To learn design patterns from examples, we need the ability to compare arrangements of objects. As discussed in Section 1.4, an important subcomponent of this problem is the task of comparing two objects in isolation. The need to compare objects is a necessary task for both humans and computers to combat data sparsity and allow rapid understanding of new situations. Because we encounter so many distinct objects, neither humans nor computers can afford to learn properties of each object individually. Instead, when we encounter a new object we relate it to other objects we have observed in our experiences in order to understand how to interact with it. This allows humans to transfer knowledge about aspects such as object function between objects, and the data-driven tools we develop will use object similarity to transfer knowledge about the expected surroundings of an object between different observations. Unlike humans, the algorithms we develop will need to precisely quantify the similarity between objects.

What makes two objects similar? Clearly, if two objects are indistinguishable without intense scrutiny, such as two new pencils from the same box, then we can reasonably expect their properties to be similar and we can say that properties we learn about one pencil should transfer to another identical pencil. Likewise, if two objects are extremely different along dimensions such as size, geometry, and material composition, then there is no reason to expect to transfer knowledge between observations: observations of a construction crane tell us nothing about observations of a

grain of sand. Most of the time, however, object comparison is much less clear cut: knowledge of black pens can tell us a lot about red pens, yet there are plenty of subtle differences (red pens are more likely to occur around stacks of homework).

3.1 Attribute Lists

How do humans deal with this problem? One model that has been proposed by psychologists is that we establish a set of attributes across which to compare objects, then groups observations of objects with commonly co-occurring attributes into *natural categories* [Rosch 1973]. In Figure 3.1, we show a subset of what such an attribute list might look like. When designing our object comparison function, some of these attributes can be easily inferred from a geometric representation of an object, such as the object's height. Unfortunately, many properties are lost in the conversion to a geometric representation and are not otherwise directly accessible to a computer, such as the smell of an object. Fortunately, when textual annotations are available for models, we can hope to use this information to compare properties not readily determined from the geometry alone. Although we do not have access to all the attributes humans use when comparing objects, the general idea of comparing objects by comparing sets of attributes remains the same.

Categories are a very powerful way to transfer knowledge between objects — as we will show in Chapter 6 effective modeling tools can be developed using categories as the sole means of object comparison. However using them is not without challenges. The ontology of categories used by humans is very complex. Different languages or even different speakers of the same language do not always agree on the category of an object. Categories are also overlapping and hierarchical — a stool and a chair are different natural categories in English but are both types of single-person seats which is itself a type of furniture. The problems of relying solely on basic categories to relate objects has been observed in many computational fields such as computer vision [Malisiewicz and Efros 2009]. Nevertheless, categorizing objects remains a very effective tool for comparing objects that we will explore in Chapter 6. In this chapter

				
Can sit on it	Yes	Yes	No	No
Has arm rests	Yes	No	No	No
Taller than a table	Yes	Yes	No	No
Is red	No	Yes	Yes	No
Smells nice	No	No	Yes	Yes
Is organic	No	No	Yes	Yes

Figure 3.1: A toy example of an attribute list that might be used to group objects into natural categories.

we will focus on comparing objects using attribute lists, which is an important subroutine for both the modeling tools we develop and the automated object categorization algorithm we describe later.

We want to define a model kernel $K_{\mathbf{model}}(a, b)$ that estimates the similarity between two models. It evaluates to 1 if the models are identical and 0 if they are not similar at all. We compose this function by combining comparison of object attributes derived from text, size, geometry, and texture kernels, which we describe in the following sections.

3.2 Comparing Text

Depending on the data source used, models in scenes may be annotated with textual information. This may come from the original model database used to create the scene, tagged by the artist as they inserted the model into the scene, or be available as part of the object structure of a virtual world. Text information is one of the most effective ways available to a computer to understand aspects of objects that are not easily recovered after the transition to a geometric representation, such as function

or style.

We show three different types of text information available for models that have been uploaded to Google 3D Warehouse in Figure 2.1. Some of this information is more reliable than others: the "name" field likely contains information pertaining to the model's category, the "tags" field often contains information pertaining to the style or function of the model, and the "description" field might contain information such as what modeling program was used. Past work on 3D model search using a similar dataset has unified these fields into a weighted list of text [Min 2004]. This work found the "name" field to be significantly more reliable than the "tag" field and the "description" field to offer essentially no benefit when performing simple precision-recall tests. As such we form a set of words for each Google 3D Warehouse model by taking the union of each word in the name and tag fields, weighting words that occur in the name five times higher than tags. We also apply a standard word stemming algorithm to handle similar words with different stems such as "fishing" and "fish" [van Rijsbergen et al. 1980].

There are many ways to compare two weighted word sets; we found the best results using a variant of the Jaccard index [Levandowsky and Winter 1971]. Let $w \in W$ be the set of all words in the corpus and let $x[w]$ indicate the weight of word w for model x:

$$k_{\mathbf{text}}(a,b) = \frac{\sum_{w \in W} \min(a[w], b[w])}{\min\left(\sum_{w \in W} a[w], \sum_{w \in W} b[w]\right)} \tag{3.1}$$

3.3 Comparing Geometry

People are very good at using only geometry to relate two objects — we can first segment the object, classify the components, determine their function, then relate these properties. While this approach is much harder for a computer, comparing model geometry remains useful. Many categories of objects are well defined by their geometric features and geometry can discriminate between subcategories within a single

natural category, such as separating chairs into those with and without armrests.

Geometric comparison is a common problem in computer graphics and many different approaches have been developed. Most approaches fall into one of two categories: graph-based representations and vector-based representations. Graph-based representations attempt to cluster the model into segments and define a graph connecting the segments. Models are then related by comparing their resulting graphs. Two such approaches are Reeb graphs [Tung and Schmitt 2005] and skeletal graphs [Sundar et al. 2003]. Graph-based approaches can be time consuming to produce and rapid retrieval is challenging. Vector-based representations typically reduce each model into a feature vector in $\mathbb{R}^n$. Two popular approaches are spherical harmonics [Kazhdan et al. 2003] and extended Gaussian images [Horn 1984]. Studies have been done that compare many different shape searching methods [Iyer et al. 2005].

In this work we use 3D Zernike descriptors to compare shapes [Novotni and Klein 2003]. These descriptors have the desirable property that they are invariant under scaling, rotation, and translation. Our Zernike descriptor computation closely follows work on autotagging models from Google 3D Warehouse [Goldfeder and Allen 2008]. We first scale the geometry into a unit cube and then voxelize it on a binary grid V that is 128 voxels on each side. We then thicken this grid by 4 voxels; a voxel in V' is set if there is a voxel set in V inside a sphere with a radius of 4 voxels. V' is used to compute a 121 dimensional Zernike descriptor using 20 levels of moments.

We use the Euclidean metric between two Zernike descriptor vectors as the distance between two shapes. The absolute distance between two model descriptors varies significantly depending on what categories of models are being compared. To mitigate this problem, we will use the distance to the n^{th} closest model as an estimate of the local density of models in the descriptor space. This density estimate is used to normalize the distance between two models. Let d_{st} be the Zernike descriptor distance between models s and t, and let $g_i(n)$ be the distance to the n^{th} closest model to model i. Our symmetric kernel between two models is given as:

$$K_{\mathbf{geo}}(a,b) = e^{-\left(\frac{d_{ab}}{\min(g_a(n),g_b(n))}\right)^2} \tag{3.2}$$

We chose the minimum value of $g_i(n)$ (corresponding to the region of greatest density) as our normalization term. This prevents an object that lies outside any cluster from forming a geometric association with an object inside a good cluster. The results in this work use $n = 100$.

Note that our use of Zernike descriptors is fundamentally a global comparison, which we found to be sufficient for most objects in Google 3D Warehouse. Datasets with a large number of articulated models may benefit from using a partial shape matching algorithm [Gal and Cohen-Or 2006].

3.4 Comparing Materials

Some categories of objects, such as chairs or airplanes, exhibit very distinct geometric features and humans can easily distinguish them using only geometry. Other categories, such as basketballs and soccer balls, are hard to tell apart from geometry alone and humans need to observe the texture to correctly classify and compare objects in these categories. This is especially true in digital environments, where 2D texture mapping is often used to convey the effect of fine-scale geometric features such as the grooves in a basketball.

Determining the similarity between 2D images is a very challenging problem and we refer the interested reader to comparison literature [Kokare et al. 2003]. In the case of 3D models, this problem is significantly more complicated. A single model may be composed of many different parts each using a different texture and material. Although many 2D image approaches can be extended to 3D surfaces, such approaches are complicated by the fact that the geometry itself needs to be aligned. Rather than formulating a 3D model comparison technique we have adopted a simpler approach. Two models are said to use the same texture if, after scaling their 2D diffuse texture maps to be the same dimension, they are equivalent within a small epsilon. We denote such near-exact matches using Kronecker Delta notation, $K_{\mathbf{material}}(a, b) = \delta_{ab}$. This term is most useful for disambiguating objects that are both categorically and geometrically similar, such as highly decorative plates vs. plates designed for eating.

3.5 Model Kernel

We want to combine the individual kernels over properties such as geometry and text described above into a single real valued function that says how similar two models are. We use the simple approach of using a weighted linear combination of the attribute kernels:

$$K_{\mathbf{model}}(a,b) = 0.6K_{\mathbf{text}}(a,b) + 0.3K_{\mathbf{geo}}(a,b) + 0.1K_{\mathbf{material}}(a,b) \tag{3.3}$$

To make inference in databases faster, we zero out this model term if it is less than a small epsilon ($\epsilon = 10^{-6}$). The empirically chosen weights given in this equation reflect our observations about the relative importance of the three features: text information was found to be the most effective way to compare two models. Geometry was found to be most useful for providing addition discrimination on top of text information or linking together two models with very similar geometry but that were uploaded under different sets of tags (such as a model uploaded multiple times using different languages).

This model kernel will be an important component necessary for all the scene modeling algorithms we present in future chapters.

Chapter 4

A Visual Memex for Model Search

The first scene modeling tool we develop focuses on the task of searching for models that belong at a location in the scene the artist is modeling. This chapter describes our approach to responding to context search queries of the form shown in Figure 4.1 [Fisher and Hanrahan 2010]. The inspiration for this type of context-based search for 3D models came from the development of similar search engines for tasks such as source code retrieval [Henrich and Morgenroth 2003]. This search engine uses the user's history and the code the user is currently writing to return relevant results.

Model search and retrieval is the first tool we develop because it is one of the most forgiving in terms of the quality of the results that are produced: the purpose of the task is to suggest ideas for a creative process, and even "wrong" results, such as suggesting a wall sconce on the surface of a desk, are not devastating as long as a few reasonable results are returned. This allows us to work with real 3D scene datasets such as Google 3D Warehouse without any human filtering. Overall, this is the only scene modeling tool we will present in this dissertation that can work well on databases that have extremely sparse tagging and poor segmentation. Although the application to 3D model retrieval is novel, the underlying ranking algorithm we use for model suggestion is based heavily on an algorithm for the Context Challenge on 2D images called the Visual Memex Model [Malisiewicz and Efros 2009].

Like all tools we develop in this dissertation, our context search algorithm uses a data-driven approach, attempting to learn spatial relationships from existing scenes.

Figure 4.1: Scene modeling using a context search. Left: A user modeling a scene places the blue box in the scene and asks for models that belong at this location. Middle: Our algorithm selects models from the database that match the provided neighborhood. Right: The user selects a model from the list and it is inserted into the scene.

First, we extract a large number of complete scenes from Google 3D Warehouse and segment these scenes into their constituent components as described in Chapter 2. We then preprocess this dataset to determine for each model a set of similar models based on properties such as the model geometry and tags. We make the assumption that similar objects occur in similar contexts. Given a user-provided context, our algorithm finds clusters of related models in the database that have appeared in a similar context.

Work by Funkhouser et al. [2004] is perhaps the most related work in 3D modeling to the algorithm presented in this chapter. They present a data-driven object modeling system. In their work, a user starts with a base model and then issues queries for related models that have desirable parts. To determine whether a candidate model is a good match to the query, they approximate the sum of the distances from every point on one surface to the closest point on the other, and vice-versa, weighting selected regions on the surface higher. The main difference between this work and our approach is that our focus is on scene composition containing a large number of disjoint models and not on finding similar parts for a single model. When comparing two scenes, we will not use a surface deformation approach and instead leverage the semantic segmentation and tagging of the scenes.

4.1 Spatial Context in Computer Vision

Computer vision research has made significant progress on using spatial context information in classifying objects in photographs [Rabinovich et al. 2007]. One way of evaluating the success of this work is to measure how accurately it labels a set of test images. Labeling objects in images has a clear analog to the case of 3D scenes, where we are given a 3D scene and are asked to decompose it into a set of semantically meaningful 3D objects that we then label. Although this is one of the most commonly used methods for evaluating contextual understanding in 2D scenes, we do not focus on this problem in this dissertation because we feel it is much less useful in scene modeling applications.

For a model search engine, a more relevant evaluation method is the Context Challenge [Torralba 2010]. In this problem, the goal is to determine the identity of a hidden object given only its surrounding context. Recent work has looked at this problem in both category-based and category-free frameworks [Malisiewicz and Efros 2009]. In the category-based framework, the goal is to directly identify the unknown object by returning a weighted set of possible categories the object belongs to. In a category-free framework, the goal is to provide a set of 2D objects (represented as bitmaps seen in other images) that belong in the unknown region. A category-free framework is sometimes advantageous, since many problems can arise when attempting to categorize the set of meaningful objects. A related problem to the Context Challenge is scene completion, which attempts to fill or replace undesired regions in an input image [Hays and Efros 2007].

The category-free Context Challenge problem, extended to 3D, is precisely the context-based query we present in this chapter: given a query box in a 3D scene, return an ordered set of models that belong at this location. Although there are differences that arise in the 3D version of the problem, many of the techniques used to solve the problem in 2D are highly applicable and can be extended to the 3D case. In 2D, solving problems like the Context Challenge is often seen as a stepping stone towards 2D scene understanding. In the 3D case solutions to the Context Challenge find a direct application in geometric search engines.

4.2 Context Search Algorithm

We begin by defining some basic terminology for the context query. We are given a scene consisting of Q supporting objects, already placed by the user, and a query box with known coordinates where the user wants to insert a new object. Our goal is to rank each object in our database according to how well it fits into the query box.

Although many of the algorithms used for learning 2D spatial context could be used as the basis for our context query, we chose to model our algorithm closely after The Visual Memex Model, because of its focus on the Context Challenge and category-free learning [Malisiewicz and Efros 2009]. Intuitively, whenever two objects f and g are observed in scene A, we take this as a suggestion that if we observe an object f' similar to f in scene A', then an object g' that is similar to g is a good candidate model in scene A', provided the spatial relationship between f' and g' echoes that of f and g.

To show a more concrete example, in Figure 4.2, the user has placed a query box in front of a desk. Suppose our database consisted of only the four scenes shown on the right. The desk in the top-left scene is an excellent match for the query desk. In the top-left scene, because the relationship between the desk and the chair is very similar to the relationship between the desk and the query box, the chair in the top-left scene is an excellent response to the query. On the other hand, the laptop and lamp in the top-left scene are not good models because their relationships to the desk are very different. To rank the models in the other three scenes, we need to answer a lot of highly subjective questions: how similar are the top-right and bottom-left desks to the query desk? Is the table at all like the desk? It is clear that answering these questions is a necessary step to responding to context queries, and we will rely upon the model kernel described in Chapter 3 in our search algorithm.

At a high level, our algorithm quantifies the concept of object similarity and the similarity between the spatial relationships of two objects, and then uses kernel density estimation over the set of observed object co-occurrences to determine the final ranking over all models. We use the Google 3D Warehouse scene database described in Chapter 2 and summarized in Table 2.1 for all our results. As we will

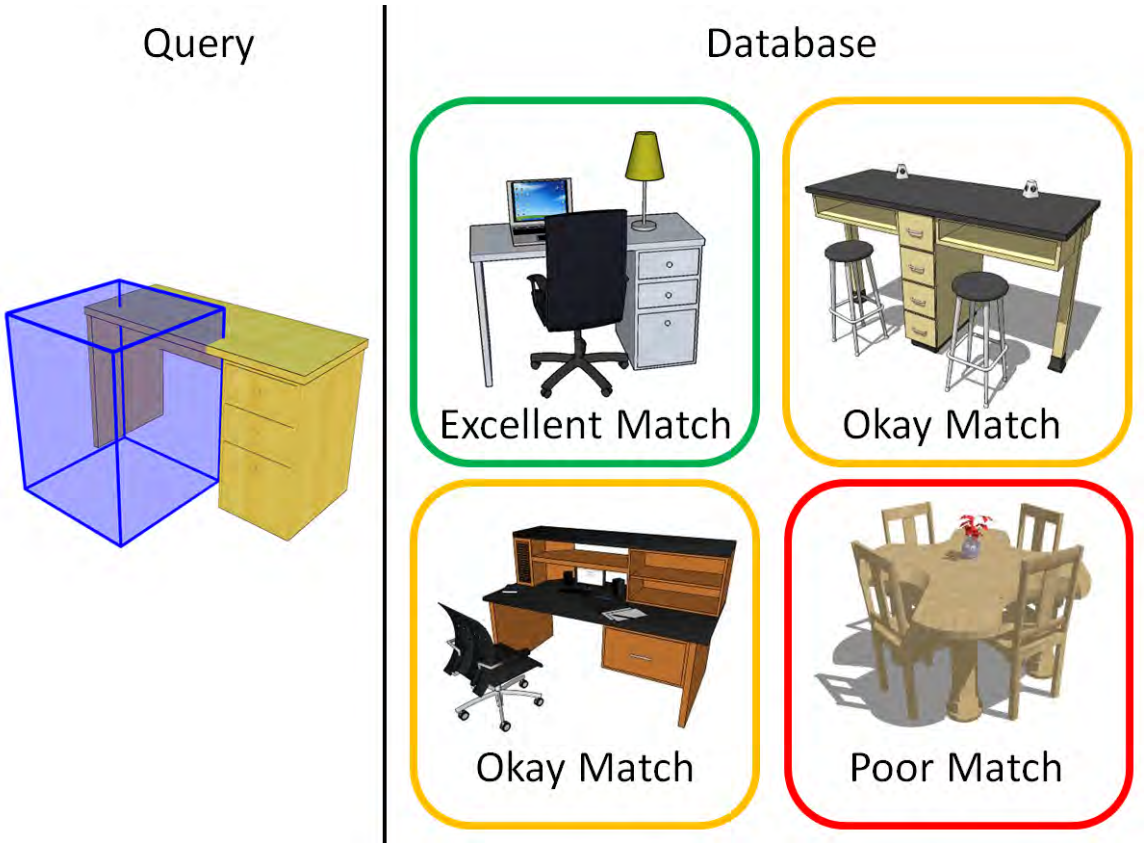

Figure 4.2: Context query using a simple database. To determine if a candidate model is a good response to a given query, we look through our database for similar pairs of models.

show, this is a rich dataset and can be used to learn contextual relationships between a wide range of objects.

4.2.1 Observations

We begin by considering all pairs of object co-occurrence across all scenes, each of which we will call an observation; we refer to the set of all such observations as O. Each observation has the following parameters: the 3D spatial relationship between the objects, and the size, geometry, texture and tags of each of the objects. Two observations are said to be similar if all of these properties are also similar. We use f_{st} as an abstract representation of the spatial relationship between two arbitrary

objects s and t.

In the next two sections we will define the following similarity functions:

- $K_{\mathbf{spatial}}(f_{st}, f_{uv})$: determines the similarity between two different spatial relationships. Evaluates to 0 if the spatial relationships are unrelated, and 1 if they are the same relationship.
- $S_{st}(\sigma_{\mathbf{size}})$: determines the similarity between objects s and t by comparing their size, geometry, texture, and tags. $\sigma_{\mathbf{size}}$ is the bandwidth of the size kernel which we will vary based on the objects being compared. Evaluates to 0 if the models are unrelated, and 1 if they are the same model.

Following the discussion of these two functions, we will describe our model ranking algorithm.

4.2.2 Spatial Relationships

We use a simple model to capture the similarity of the spatial relationship between arbitrary objects s and t. The model depends on two distances: the absolute height displacement z_{st} (along the Z-axis), and the absolute radial separation r_{st} (in the XY-plane). Both are measured between the centers of the bounding boxes of the objects. These distances have units of length, and we benefit from the fact that the input scenes use the same units.

We ran a simple experiment, and found that r_{st} and z_{st} are largely uncorrelated. We model the similarity of each distance with a Gaussian kernel $G(x, y, \sigma) = e^{-\|x-y\|^2/\sigma^2}$, and assume the two kernels are separable. Our final metric is:

$$K_{\mathbf{spatial}}(f_{st}, f_{uv}) = G(z_{st}, z_{uv}, \sigma_z)G(r_{st}, r_{uv}, \sigma_r) \tag{4.1}$$

The two bandwidths σ_z and σ_r have units of length. We choose both to be proportional to the length of the longest dimension of the objects being compared (l). Thus, the rate of fall-off in similarity is proportional to the size of the objects. In this study we used $\sigma_z = 0.05l$ and $\sigma_r = 0.5l$, which encourages objects to be aligned more precisely in z than r.

It would be nice to capture richer spatial relationships between objects. For example, the chair is in front of the desk, the couch faces the TV, or the plate is supported by the table. Measuring such relationships requires more sophisticated geometric analysis and a dataset which very clean segmentations between objects. We explore many of these ideas in the algorithm described in Chapter 5.

4.2.3 Object Similarity

Two objects are similar if both their sizes and their underlying models are similar. We assume that the size and model comparisons are separable kernels, and define the similarity between arbitrary objects s and t as:

$$S_{st}(\sigma_{\mathbf{size}}) = G(\text{Size}(s), \text{Size}(t), \sigma_{\mathbf{size}}) K_{\mathbf{model}}(s, t) \tag{4.2}$$

To compare the size of two objects, we first sort the x-y dimensions of the bounding box of each object based on length. We consider the dimensions of the bounding box of each object to be a vector in $\mathbb{R}^3$. We use the Euclidean distance between these two vectors to compare the size of the objects. $K_{\mathbf{model}}(s, t)$ estimates the similarity between two size-normalized models by comparing their textual annotations, geometry, and texture, and is given in Equation 3.3.

4.2.4 Model Ranking

We can use the two similarity metrics given in Equation 4.1 and 4.2 to estimate the probability that an object would appear in the context of other objects. We first define this probability for the case where the support scene contains only a single object b. Specifically, we need to compute the probability of placing an object a in the query box. Objects are ranked by their probabilities of being in the box.

We model this probability using the following conditional distribution:

$$p(a|b, f_{ab}) \propto \sum_{c \in M} S_{cb}(\sigma_0) \Psi(a, c, f_{ab}) \tag{4.3}$$

$$\Psi(a, c, f_{ab}) = \frac{\sum_{(u,v)\in O} S_{au}(\sigma_1) S_{cv}(\sigma_2) K_{\mathbf{spatial}}(f_{ab}, f_{uv})}{\sum_{(u,v)\in O} S_{au}(\sigma_1) S_{cv}(\sigma_2)} \tag{4.4}$$

Here, $\Psi(a, c, f_{ab})$ is the pairwise compatibility between objects a and c. M is the set of objects in the database and O the set of observed object pairs.

This inference algorithm closely follows that used in the Visual Memex [Malisiewicz and Efros 2009]. The summation in Equation 4.4 can be seen as a weighted sum of kernels, one for each observation. The weight of each kernel, $S_{au}S_{cv}$, measures the similarity between the object pair under consideration (a, c) and the arbitrary object pair (u, v).

The parameters σ_0, σ_1, and σ_2 control how contextual information propagates between objects of different sizes. These have units of length and are chosen proportional to the length of the longest dimension of the objects being compared (l). We use $\sigma_0 = 2l$, $\sigma_1 = 0.2l$, and $\sigma_2 = 2l$. The small value of σ_1 places increased emphasis on finding objects that are close to the size of the query box, while the larger values of σ_0 and σ_2 permits objects of large size variation to contribute context neighborhood information. Increasing the value of σ_1 denotes increased uncertainty in the size of the user-specified query box; as this value approaches infinity the size of the query box becomes irrelevant.

Figure 4.3 provides a visualization of Equation 4.3. In this example, we plot the probability that a keyboard, chair, couch and bathtub are near a monitor as a function of the radial separation between the objects. Note that a keyboard has a high probability of being near a monitor, but is not likely to be far from the monitor. Examining the plot for a chair, we see there is interesting structure. First, chairs are further from monitors than keyboards, and can also be found at larger radii. Couches and bathtubs are much more common at large radii than close to monitors. This shows that our algorithm is able to learn some of the structure in the scenes.

We presented Equation 4.3 for the case of only one object in the scene. One natural assumption to make in the case of multiple support objects is to assume $p(a|b, f_{ab})$ is independent between all support objects b, in which case we would simply compute the product of this term for all supports (this is the assumption made by the Visual

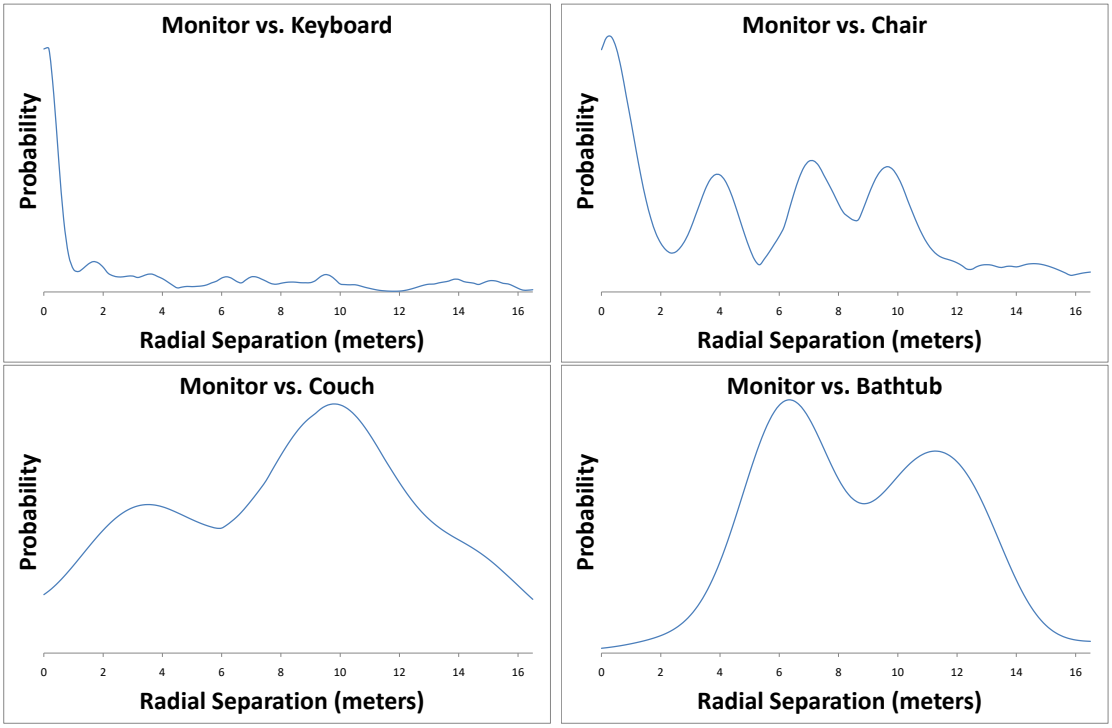

Figure 4.3: Density estimation as a function of radial separation between objects. A representative object of each class was chosen, and Equation 4.3 was computed as a function of the radial separation between the objects. The shape of these curves approximates the expected relationship between these objects.

Memex.) We have found that this tends to unfairly penalize objects that are mostly found in partial scenes, which may have many support objects where this probability score is effectively zero. Instead we compute $p(a|b, f_{ab})$ for all Q support objects in the scene, and store these in a list P for each object a, sorted from most to least probable. Our final model score is then taken as a product over the q strongest supporting objects in the scene (we use $q = 5$:)

$$p(a|b_1, ...b_K, f_{ab_1}, ...f_{ab_K}) \propto \prod_{i=1}^{q} P_i \tag{4.5}$$

Once a weighting over all models is complete, if desired a simple voting process could be used to combine the model weights into a weight over each possible label, in order to produce a set of labels that are likely to belong to objects in the query

box [Medin and Schaffer 1978].

4.3 Results

4.3.1 Context Search vs. Keyword Search

One application of our algorithm is to suggest models to a user who wishes to add another object to a scene. For example, imagine a user who is modeling a dining room table and wishes to add a fork to it. One current approach is to use keyword search. To find a model, the user searches Google 3D warehouse using "fork" as a query term. The results returned are shown in Figure 4.4a. Using our system, the user searches for models using a context query. The query is generated by selecting a 3D bounding box on the dining room table. The context query returns the results in Figure 4.4b.

Figure 4.4a shows that a keyword search can have trouble returning a useful list of models. The keyword search was unable to determine what sense of the keyword "fork" was appropriate. In this case, the result set included only one table fork; the other eleven objects represent other senses of the term "fork" including "fork lift" and "bicycle fork." Although tuning forks are plausible, fork lifts cannot be placed on dining room tables. In contrast, the context search returned four table forks (Figure 4.4b). Since no keyword was given, small objects like pens and brushes were also returned. All the objects could be sensibly placed on a dining room table. Finally, Figure 4.4c show the results of doing a context plus keyword query search. In this example, the result set was all table forks, except for one extraneous model.

4.3.2 Adapting to Context

Figure 4.5 shows the results of a context query when the scene consists of a single object, in this case a desk. The top row shows the results for a query box in front of the desk. For this query, 21 of the top 24 results are chairs. The middle row shows the results for a tall box on the side of the desk. Here the context query returned 17 floor lamps. These two queries demonstrate the algorithm's ability to find different types

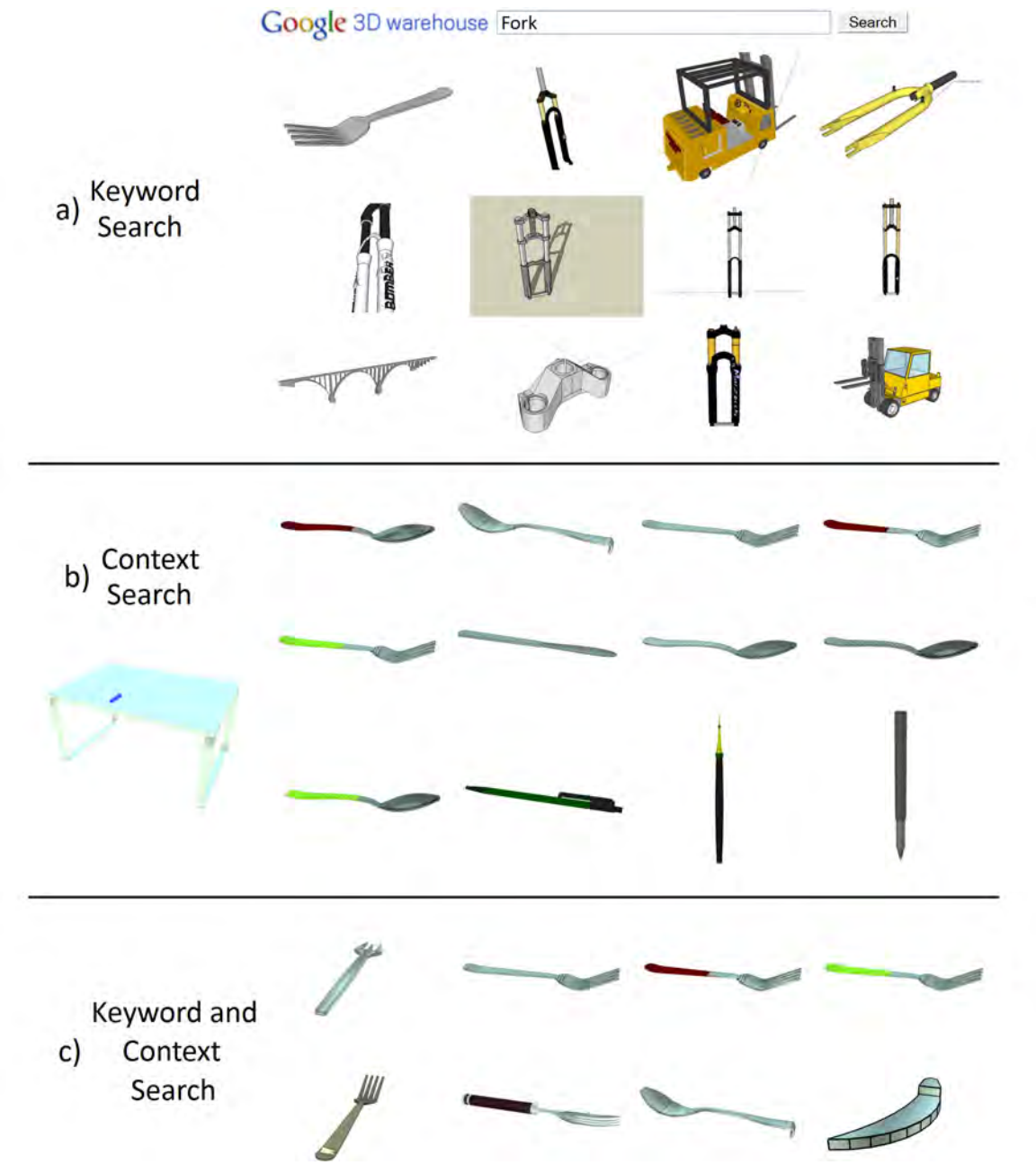

Figure 4.4: Comparing keyword and context search. Top: Front page search results for "fork" in Google 3D Warehouse. Middle: Results of a context search in our database for a query box placed on a table. Bottom: Filtering the results of the context search for models with the "fork" tag. A search engine which can look at the target context and size can better discern the intent of the user's query, especially for keywords with multiple meanings.

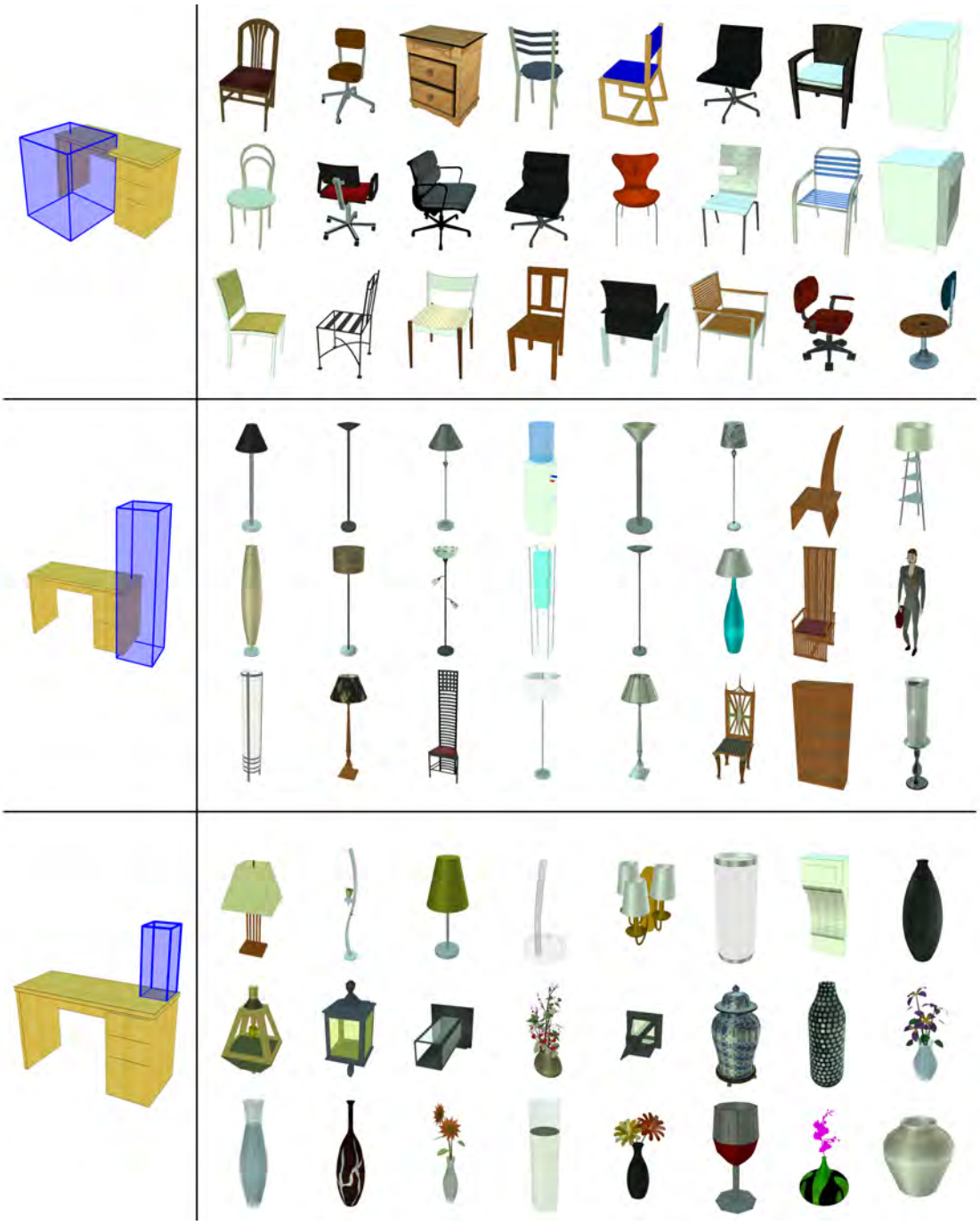

Figure 4.5: Context query results with a desk as the only supporting object. Left: The user places a query box in the vicinity of the desk. Right: The top 24 search results for each query. When models with identical geometry but different textures occur, only the first result is shown.

of models using different contexts. The bottom row shows a query to find an object resting on the desk. This query can be satisfied by many different types of objects. Quantifying the relevance of these results is more challenging; nevertheless, there at least four distinct categories of results that are relevant. These include lamps, plants, goblets, and vases.

It is interesting to note that in all three example queries, the highest rated result (the one in the upper-left corner) would be a very plausible result for the given context. If the user requested a single result be immediately returned (by pressing "I'm feeling lucky"), they would not be disappointed. This is important for the design of more powerful modeling tools such as automatic scene decoration, which we will show in Chapter 6.

4.3.3 Multiple Supporting Objects

We ran an experiment to test the effect of adding additional contextual objects to the scene. In this case, the user wanted to model a kitchen counter. The top row of Figure 4.6 shows the results of the query with only a sink as context. The query returns five sinks, and several other relevant models including a toaster oven, a mixer and a vase, and a few undesirable models such as a printer and at least five objects that are not meaningful. We now add a second object, a blender, to the scene, and reissue the query. The additional context significantly raises the rank of the three microwave models, and slightly increases the rank of the blender, toaster oven, and vase models. It also removes the printers and three meaningless objects. Overall the combination of the blender and sink noticeably improved the quality of the results.

4.3.4 User Evaluation

It is difficult to rigorously evaluate a context search algorithm because so much of it depends on both human perception and user intent. To quantify the quality of our search results, we use human evaluators to perform a standard precision-recall evaluation on our results.

We start by making five test scenes, each with a single associated context query,

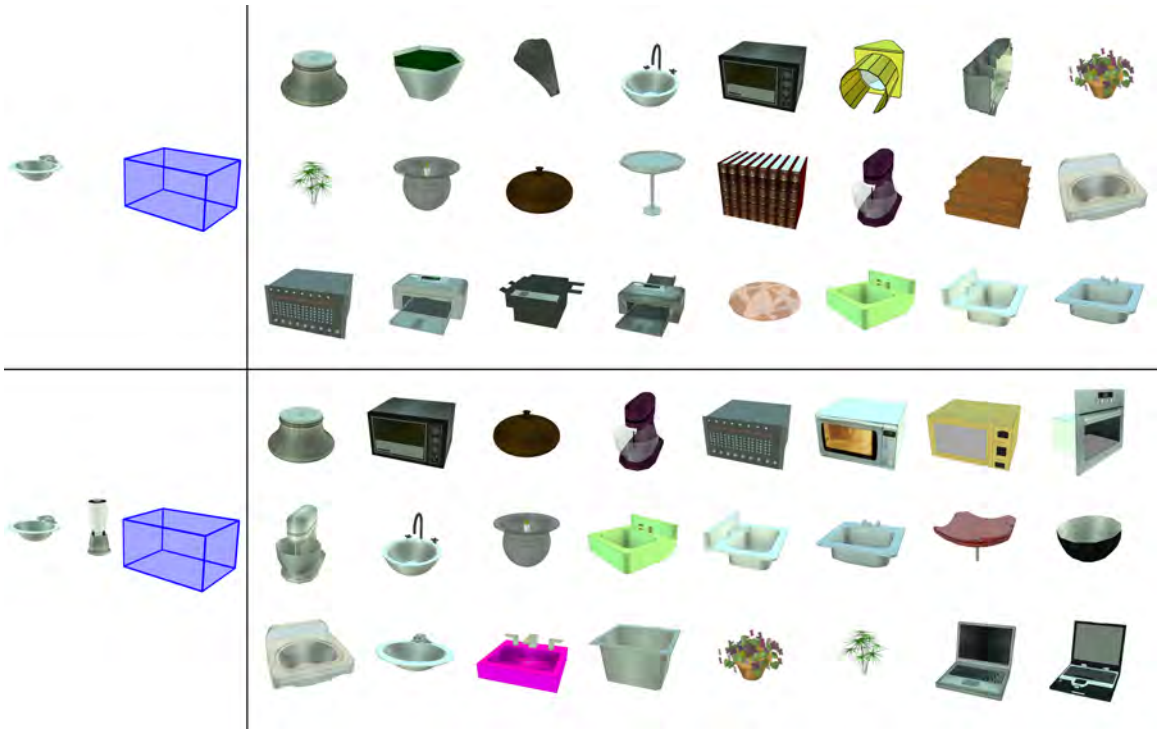

Figure 4.6: Benefit of additional supporting objects. Top: The user places a sink into an empty scene and asks for an object four feet away. Bottom: The user places a blender in the scene between the sink and the target object, and repeats the same query.

and ran our algorithm to get 500 search results for each scene. The number of supporting objects in the scenes ranges from one to fifty. To compute the "recall" capability of our results, we need to estimate the total set of viable results. To do this we union out 500 results with the 1000 objects in the database whose size is closest to that of the query box. Users were then presented with each model in the candidate set in a random order and asked to decide whether the model was relevant to the context query. Our user base consisted of five males and four females, none of whom had any formal design experience.

The results of this study were used to estimate the set of relevant models for each query by using human oracles: we assume that the models marked as belonging in the query box by at least half of the users are the only relevant models in the database.

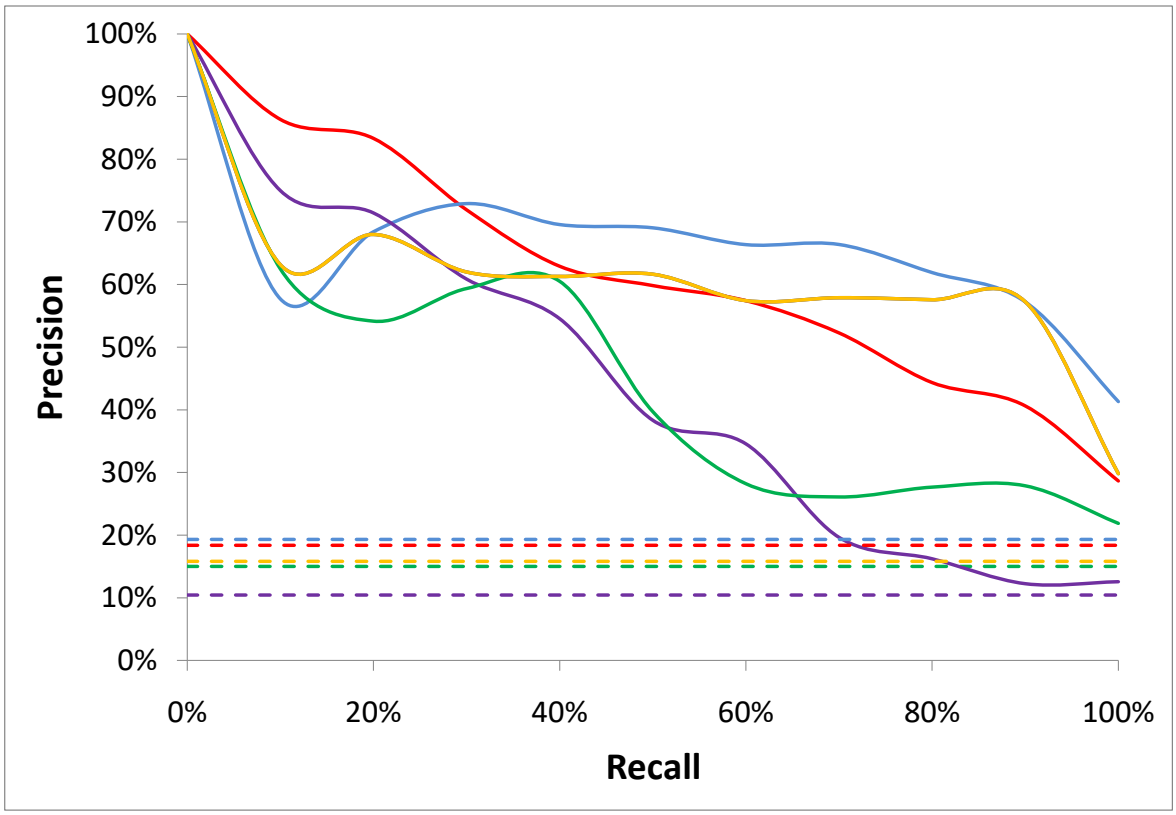

Figure 4.7: The solid curves show the precision-recall curve for our algorithm on five test scenes. The relevance of each model was determined by surveying human subjects. The dashed lines show the expected precision-recall curve if the candidate model set had been presented to the user in a random order.

Given our search results and a relevance set for each scene, we plot a precision-recall curve to evaluate the quality of our results. The solid curves in Figure 4.7 show this curve for each scene. These curves demonstrate that even without keywords, context-based search can successfully favor relevant models. In all five scenes the algorithm returned 50% of the relevant models with a precision of at least 35%, which implies that at least one in every three models was desirable.

To provide a comparison point for our search results, we computed the expected precision-recall curves after randomizing the models in each candidate set and using this as our final ranking. These are the dashed curves in Figure 4.7. All of these curves have a precision between 10% and 20%. Because the candidate sets select

highly for the size of the objects, these results can be taken as an approximation of the performance of a size-only search engine.

The fact that our algorithm provides a noticeable improvement over a random ordering of the candidate set indicates that context provides some useful information. The actual information gained from context information compared against just the bounding box size will depend heavily on the scene and the density of objects in the database around the size of the query box. As a concrete example of the advantage of context information, a direct size-based ranking for the top query in Figure 4.5 would contain 8 toilets in the top 24 results.

4.3.5 Failure Cases

Although our algorithm does a reasonable job of returning relevant models, there are several failure modes that will cause an irrelevant model to be ranked highly. First, an object may be geometrically very similar to a relevant object but semantically very different. Likewise, an irrelevant object may have the same tag as a relevant object. For example, the model in Figure 4.6, 1st column, 3rd row, is tagged as "channel mixer". It is not a very relevant model to the query, but is both geometrically similar to a microwave model and shares a tag with the "mixer" model, both of which are relevant and appear in the top 24 results. Another failure mode can occur because our spatial relationships are overly simplistic. For example, in the top query in Figure 4.5, the filing cabinet object is not very relevant at the selected location, but was ranked highly because it was found on the side of desks in the database and our spatial relationships do not consider the relative orientation. The pairwise independence assumption made by this algorithm can also result in spurrious suggestions, as we will show in Chapter 5 (see Figure 5.9).

4.4 Chapter Summary

In this chapter, we presented an algorithm to suggest models at a given location in a scene that is being modeled. Our algorithm is based on a kernel density estimation method used in computer vision to estimate the probability of an unknown object given its surroundings and a database of images. Our algorithm returns many useful results despite a comparatively low database quality and strong independence assumptions about the occurrence of object pairs. In our experience context search is a very easy tool to use — artists need only find an area they find "barren", then quickly browse through a list of results until something acceptable is found. Perceptually evaluating results is a very fast operation and undesirable results are easily filtered. Keywords can also be used to filter the results of a context search, and our search tool often presents surprising or creative results that help an artist create diverse or interesting environments by borrowing from the creativity of other artists.

Chapter 5

Graph Kernels in Scene Modeling

In this chapter, we present our work on representing 3D scenes as relationship graphs, and using graph kernels defined over these relationship graphs to enable scene modeling tools [Fisher et al. 2011]. The tools we enable include relevant scene suggestion, and a context search tool that overcomes the need to make the the pairwise independence assumption used by the algorithm described in the previous chapter. On the downside, the relationship graphs we define require a good segmentation of the input scene database into meaningful components: the tools cannot tolerate operating directly on the Google 3D Warehouse scenes unless they are very well segmented, and typically some limited human filtering is necessary.

Scene comparison is a challenging problem because scenes contain important structure at many different resolutions. The challenge of comparing highly structured data occurs in a variety of fields, such as web search [Habegger and Debarbieux 2006], protein function prediction [Borgwardt et al. 2005], and image classification [Lazebnik et al. 2006]. In all of these problems, attempting to directly compare the finest-level data is rarely successful. Instead, the data is often transformed into a representation that enables the comparison of important features. In this chapter, we will show how to transform scenes into a relationship graph whose nodes represent semantically meaningful objects, and whose edges represent different types of relationships between nodes. This graph representation greatly facilitates comparing scenes and parts of scenes.

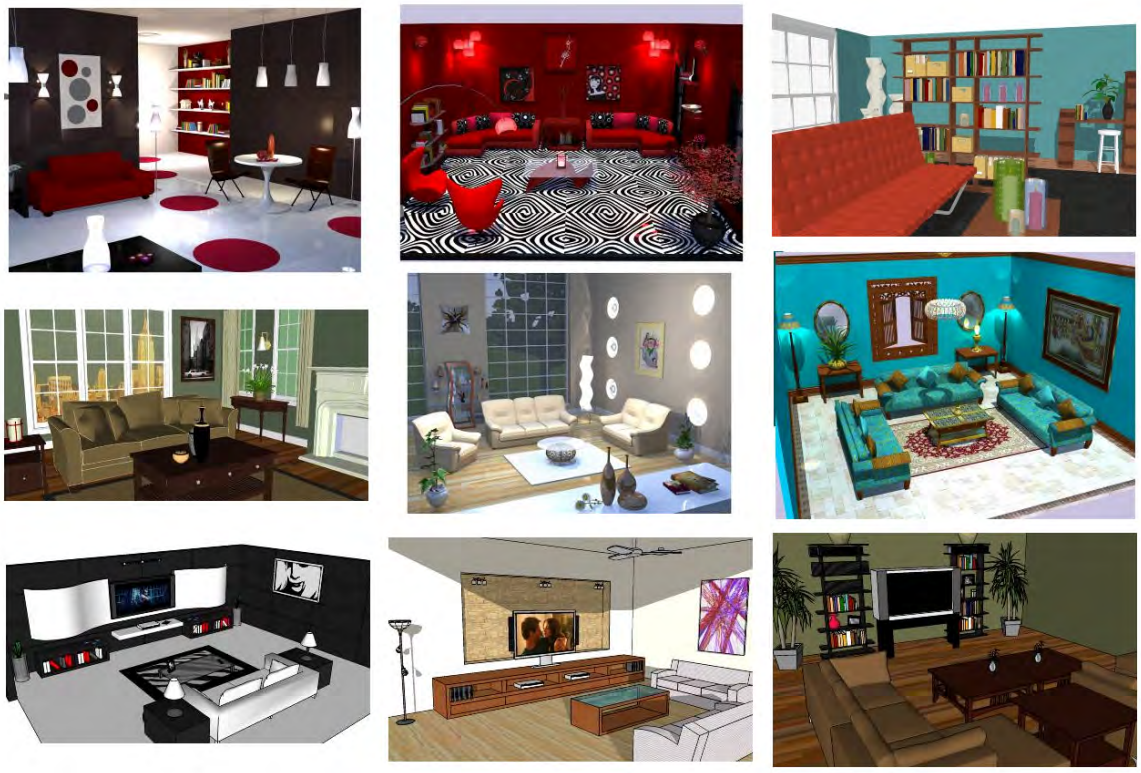

Figure 5.1: A set of scenes in the Google 3D Warehouse with "living room" in their scene name. Many properties of a scene are not reflected well in the scene name. For example, a user looking for models to add to an entertainment center would only be pleased with the three scenes on the bottom.

One simple approach to scene comparison is to directly compare the tags artists have provided for a scene or the name attached to the scene. Unfortunately, while a scene name can provide useful information about the scene's category, it cannot easily express the stylistic variation within these categories. Likewise, it is challenging for the scene tags to encompass all the interesting substructures within the scene. In Figure 5.1, we show nine scenes retrieved from Google 3D Warehouse using a keyword search for "living room". Understanding the relationships between these scenes requires a method to compare different aspects of the scene's internal structure.

In this work we will describe how we can take a 3D scene and extract a set of spatial relationships between objects in the scene. We show how we can use this set of spatial relationships to define a positive-definite kernel between any two 3D scenes.

We use this kernel to execute several different types of queries for complete scenes that incorporate the structural relationships between objects. We show how our scene kernel can also be used to search for models that belong in a particular context and have a specified spatial relationship to other objects. For example, a user could issue a search for models that can be hung on a wall in the bedroom they are modeling.

5.1 Previous Work

5.1.1 Scene Comparison

A small number of attempts have been made at comparing 3D scenes. One approach works by partitioning the mesh of each object into a set of regions, and forming a graph by connecting adjacent regions [Paraboschi et al. 2007]. A feature vector for each graph is constructed using the eigenvectors of the spectral decomposition of the graph topology. This method is sensitive to the mesh segmentation and was not tested on scenes with a large number of objects. Also, because their focus was on manifold meshes it does not map well to the datasets we explore. Nevertheless, it is similar to our approach in that it first reduces scenes to a graph and then compares two scenes using properties of their graphs.

A problem that has many parallels to scene comparison is image comparison, where the goal is to relate two images based on their semantic content. One approach uses local self-similarities of image regions to construct descriptors that can robustly compare visual entities [Shechtman and Irani 2007]. Another approach to image comparison is to first segment the image into regions and then construct a graph by connecting regions that are touching [Harchaoui and Bach 2007]. Two regions are compared by using their color histograms. The images can then be compared by looking at their respective segmentation graphs. Our approach can be seen as the natural extension of this idea to 3D scenes: we first segment our scene into meaningful objects, then insert edges that represent relationships between pairs of objects.

5.1.2 Graph Kernels

Kernel-based methods for machine learning have proven highly effective because of their wide generality. Once a kernel is defined between the entities under consideration, a wide range of learning algorithms such as support vector machines can be applied [Cristianini and Shawe-Taylor 2000]. In addition, techniques such as multiple kernel learning can be used to intelligently combine results from a large number of kernels [Bach et al. 2004].

There is considerable work on defining kernels between data types that are highly structured [Shawe-Taylor and Cristianini 2004]. In particular, several different kernels between two graphs have been proposed [Kashima et al. 2004]. These have been successfully applied to a variety of problems such as molecule classification [Mahé et al. 2004] and image classification [Harchaoui and Bach 2007].

In its most general form, a graph kernel takes as input two graphs with labeled nodes and edges, a kernel $k_{\mathbf{node}}(n_a, n_b)$ defined between node labels and a kernel $k_{\mathbf{edge}}(e_a, e_b)$ defined between edge labels, and returns a non-negative real number reflecting the similarity between the two graphs. The node and edge kernels used depend on the types of labeling used and are application specific. In our work, nodes represent individual objects or collections of objects, and we can use any of the model comparison techniques used in 3D model database search. Our edges represent different types of relationships between objects, and the kernel used depends on the types of relationships.

Given node and edge kernels, constructing an efficient kernel over graphs is a challenging problem with a diverse set of existing approaches [Gartner et al. 2003]. One method that has proven very successful is to first reduce the graph to a finite set of strings, and then use kernels defined over these strings. In particular, a graph walk kernel can be defined by considering all walks of a fixed length as the set of strings. As we will see, this admits a simple and efficient dynamic programming solution. Another mapping from graphs to sets of strings is to consider all possible α-ary tree walks in the graph of a fixed depth [Harchaoui and Bach 2007]. Unfortunately, we found tree walks to be intractable for our problem because there is not a natural way of ordering the edges around a node; successful applications for this type of graph

kernel have relied on properties like graph planarity to obtain such an ordering. It has been shown that as long as the underlying node and edge kernels are positive semi-definite, the resulting walk and tree walk graph kernels will also be positive semi-definite [Shawe-Taylor and Cristianini 2004].

5.1.3 Spatial Relationships

In order to represent a scene as a graph, we need a way to take the geometric representation of the scene and produce a set of relationships between pairs of objects. These relationships might be largely geometric in nature ("object A is horizontally displaced by two meters relative to object B") or largely semantic ("object A is in front of object B"). Capturing semantic relationships is desirable because they are are more stable in the presence of object and scene variation.

Computer vision has used the spatial relationships between two objects in a photograph to assist with problems such as scene content understanding and object categorization. For example, many objects and materials are difficult to tell apart (sky vs. water) but can be disambiguated using spatial relationships (sky is rarely found below grass). One approach uses a conditional random field to maximize the affinity between object labels using semantic relationships [Galleguillos et al. 2008]. The relationships they consider are {inside, below, around, above}. Although these are useful examples of relationships between objects or materials in 2D images, they are not representative of semantic relationships between 3D shapes.

containment	encircled	circlement
contact	**surface contact**	support
attachment	adhesion	hanging
piercing	impaled	proximity
above	below	vertical equality
horizontal support	front	behind
viewport equality		

Table 5.1: A list of spatial primitives used to study how humans reason about spatial relationships [Feist 2000]. The relationships used in this chapter are shown in bold.

Psychologists have tried to understand what set of spatial primitives humans use to reason about spatial concepts [Xu and Kemp 2010]. Although the nuances of human spatial understanding are too complicated to construct a comprehensive list of all possible primitives, in Table 5.1 we show one list of spatial primitives that has been used with some success. Our goal is to test for these relationships given only the geometry of two objects. Many of these relationships are highly geometric in nature (circlement, above, containment), but some are very difficult to infer from geometry alone (attachment vs. adhesion). Work in computer vision has looked at using qualitative relationships between objects in images such as attachment, support and occlusion to infer implicit geometric information from object labels [Russell and Torralba 2009].

5.2 Representing Scenes as Graphs

Our algorithm takes as input a set of well-segmented scenes represented as scene graphs. We start by constructing a corresponding *relationship graph* for each scene. The nodes of a relationship graph represent all objects in the scene and the edges represent the relationships between these objects. Our relationship graph representation is similar to the *3D parse graphs* used in image understanding [Gupta et al. 2010]. Each non-transform node in the processed scene graph corresponds directly to a node in our relationship graph. Given the nodes of the relationship graph, we then determine the set of relationships between the nodes, thus creating a corresponding edge set.

Good relationships should capture features of the scene that correspond with spatial primitives used by humans. We have used a subset of the relationships in Table 5.1 as our set of possible relationships. We chose relationships that were highly discriminative and could also be determined using only geometric tests.

We define a polygon in mesh A to be a contact polygon with respect to mesh B if there is a polygon in mesh B such that the distance between the two polygons is less than a small epsilon, and the angle between the unoriented face normals of the polygons is less than one degree. The results in this chapter use a contact epsilon

of 2mm, using the unit scaling provided with the scene. For the databases we use, we also have a well defined gravity vector that describes the global orientation of the scene.

Below is a list of the relationships we chose and the process used to test for them:

- **Enclosure:** Mesh A is enclosed inside mesh B if 95% of the volume of mesh A's bounding box is inside the bounding box of mesh B.
- **Horizontal Support:** Mesh A is horizontally supporting mesh B if there exists a contact polygon in mesh A whose face normal is within one degree of the gravity vector.
- **Vertical Contact:** Mesh A is in vertical contact with mesh B if there exists a contact polygon in mesh A whose face normal is within one degree of being perpendicular to the gravity vector.
- **Oblique Contact:** Mesh A is in oblique contact with mesh B if there exists a contact polygon that does not satisfy any other test.

The tests are performed in the order given and an edge of the corresponding type is created between two objects for the first test that is satisfied. In addition to the above relationships we also retain the original parent-child edges from the scene graph as a separate type of relationship. Figure 5.2 is a simple example illustrating the resulting relationship graph. Object nodes can be connected by both contact and scene graph inheritance relationships. Note how the monitor and keyboard nodes are both scene graph children of the computer node.

5.3 Graph Comparison

By constructing a node and edge set for each input scene we obtain its representation as a relationship graph. Our goal is to compare two relationship graphs or subparts of relationship graphs. To accomplish this comparison we first need a way to compare individual nodes and edges in these graphs. These will be key subcomponents in our graph kernel.

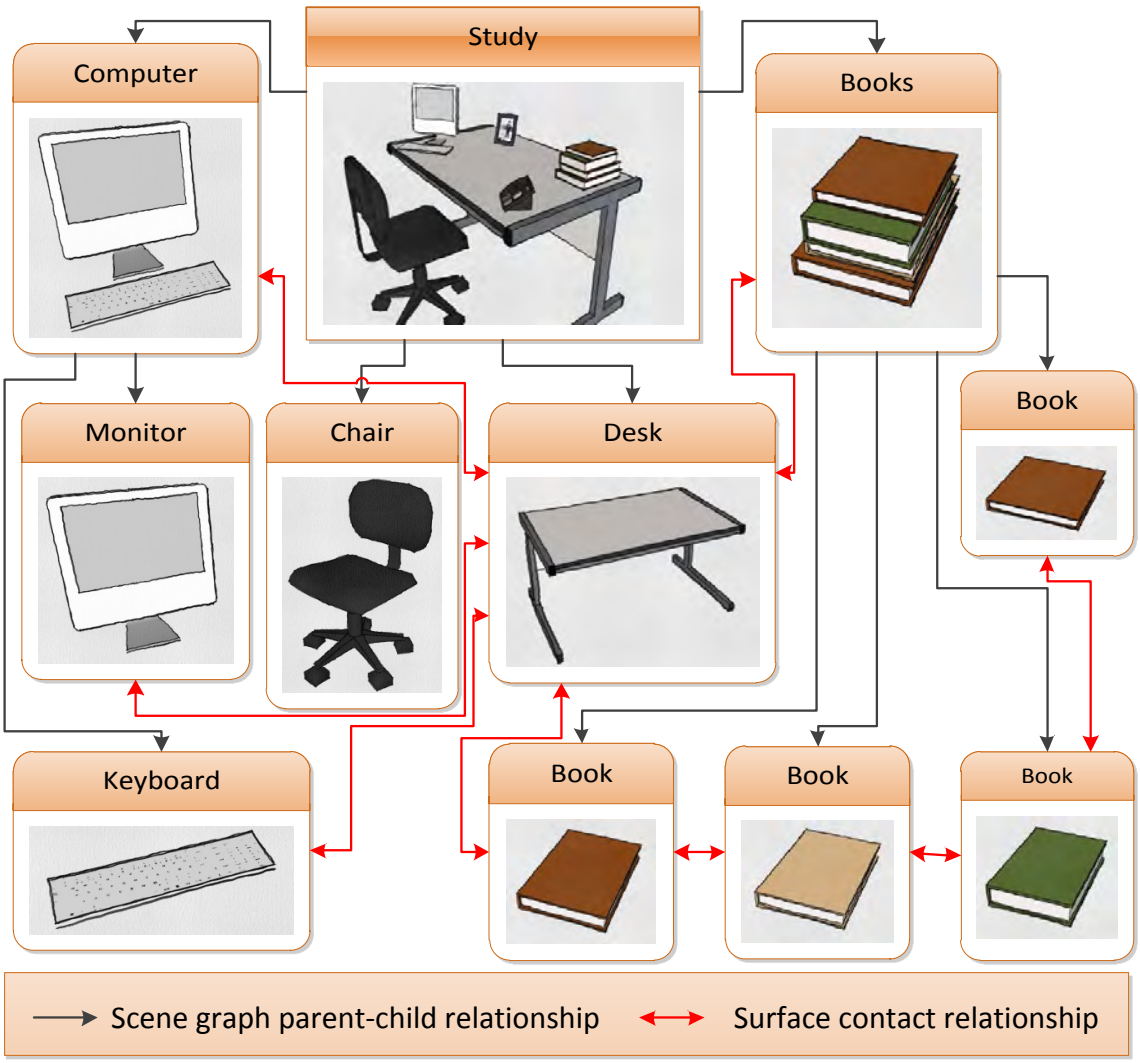

Figure 5.2: A scene and its representation as a relationship graph. Two types of relationships are indicated by arrows between the object nodes.

5.3.1 Node Kernel

The nodes of a relationship graph represent objects within the scene. Each node contains a number of features that relate to the identity and semantic classification of a particular object. These properties include the size of the object and the geometry, tags, and texture of the underlying model. We compare two nodes using the model kernel described in Chapter 3:

$$k_{\mathbf{node}}(r, s) = \sigma(r)\sigma(s)k_{\mathbf{model}}(r, s) \tag{5.1}$$

The $\sigma(r)$ and $\sigma(s)$ terms are node frequency normalization scalars whose computation is described in Section 5.3.4. The result of this kernel evaluation is clamped to 0 if it is less than a small epsilon ($\epsilon = 10^{-9}$). This model kernel can be precomputed between all possible models in the database.

5.3.2 Edge Kernel

We now define an edge kernel to provide a similarity metric between edges representing relationships. In our implementation we choose to represent each relationship as a different edge with a simple string label indicating the type. The kernel between two edges e and f with types indicated by labels t_e and t_f respectively, is then simply $k_{\mathbf{edge}}(e, f) = \delta_{t_e t_f}$. Although we considered performing a more discriminative comparison between relationships, assigning partial matches (values less than 1) runs the risk of devaluing functionally similar relationships.

5.3.3 Graph Kernel

Given node and edge kernels we now define a graph kernel to perform the comparison between two scenes. Our approach is heavily based on a graph kernel algorithm used for image classification [Harchaoui and Bach 2007].

A walk of length p on a graph is an ordered set of p nodes on the graph along with a set of $p-1$ edges that connect this node set together. Unlike a path, nodes in a walk may be repeated.

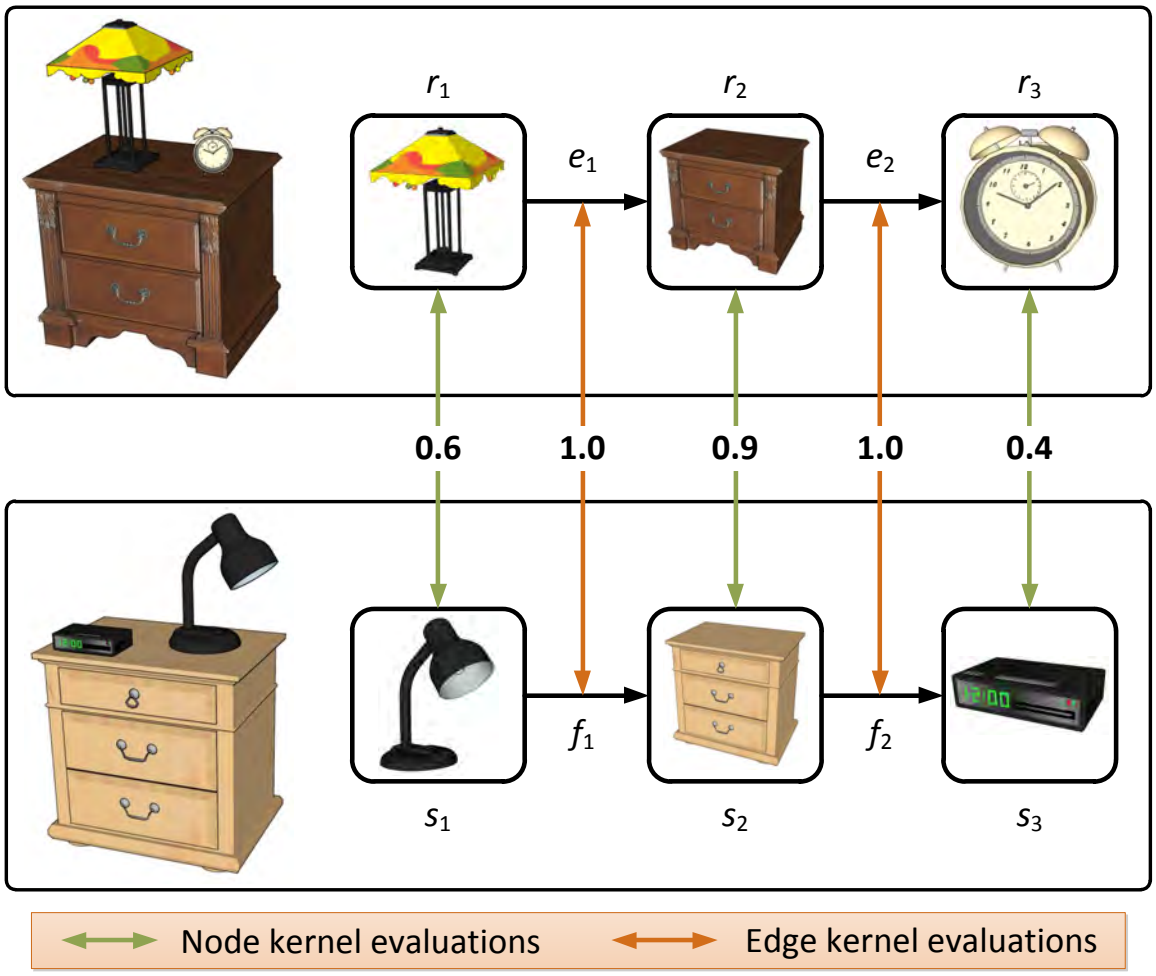

Figure 5.3: Comparison of two walks. Left: The two scenes being compared. Right: Two walks in each scene, both rooted at the lamp node. The two walks are compared by taking the product of kernel evaluations for their constituent nodes and edges. The similarity between these two walks is $0.6 * 1 * 0.9 * 1 * 0.4 = 0.22$.

Let $W_G^p(r)$ be the set of all walks of length p starting at node r in a graph G. As defined earlier, $k_{\mathbf{node}}(r,s)$ and $k_{\mathbf{edge}}(e,f)$ represent the node and edge kernels. Considering nodes r and s in relationship graphs G_a and G_b respectively we now define the *p-th order rooted-walk graph kernel* k_R^p:

$$k_R^p(G_a, G_b, r, s) = \sum_{\substack{(r_1,e_1,\ldots,e_{p-1},r_p)\in W_{G_a}^p(r) \\ (s_1,f_1,\ldots,f_{p-1},s_p)\in W_{G_b}^p(s)}} k_{\mathbf{node}}(r_p, s_p) \prod_{i=1}^{p-1} k_{\mathbf{node}}(r_i, s_i) k_{\mathbf{edge}}(e_i, f_i) \quad (5.2)$$

This kernel is comparing nodes r and s by comparing all walks of length p whose first node is r against all walks of length p whose first node is s. The similarity between two walks is evaluated by directly comparing the nodes and edges that compose each walk using the provided kernels for these object types. In Figure 5.3, we visualize one step of the computation of k_R^2 for two nightstand scenes.

If we also define $N_G(x)$ to be the set of all neighboring nodes of x in the graph G we can formulate a recursive computation for $k_R^p(G_a, G_b, r, s)$:

$$k_R^p(G_a, G_b, r, s) = k_{\mathbf{node}}(r, s) \times \sum_{\substack{r'\in N_{G_a}(r) \\ s'\in N_{G_b}(s)}} k_{\mathbf{edge}}(e, f) k_R^{p-1}(G_a, G_b, r', s') \quad (5.3)$$

where $e = (r, r')$ and $f = (s, s')$ are the edges to neighboring nodes of r and s. The above computation can be initialized with the base case $k_R^0(G_a, G_b, r, s) = k_{\mathbf{node}}(r, s)$. We can use this recursive expression to construct a dynamic programming table for each pair of relationship graphs. We store values for all node pairs between the two graphs and for all walk lengths up to p. The kernel we have thus defined can be used to compare the local structure of two relationships graphs rooted at particular nodes within those graphs.

We can use k_R^p to define a *p-th order walk graph kernel* k_G^p which compares the global structure of two relationship graphs. Here we use V_G to mean the set of all nodes of graph G. This kernel is computed by summing k_R^p over all node pairs across

the two graphs:

$$k_G^p(G_a, G_b) = \sum_{\substack{r \in V_{G_a} \\ s \in V_{G_b}}} k_R^p(G_a, G_b, r, s) \tag{5.4}$$

The running time complexity for computing k_G^p between two graphs is $O(pd_G d_H n_G n_H)$ where d_G is the maximum node degree and n_G is the total number of nodes in the graphs [Harchaoui and Bach 2007]. As we show in Section 5.5.4, in practice these evaluations are very fast.

The graph kernel we have presented can be interpreted as embedding the graphs in a very high dimensional feature space and computing an inner product $\langle G_a, G_b \rangle$. While inner products are widely useful for many applications, some applications such as the Gaussian kernel $K(G_a, G_b) = e^{-\|G_a - G_b\|^2/\sigma}$ operate on a distance metric instead of an inner product. Given a positive-definite kernel, there are many possible distance functions that can be defined over the feature space spanned by the kernel [Ramon and Gärtner 2003]. For a p-th order walk graph kernel k_G^p, a simple corresponding distance function is:

$$d(G_a, G_b) = \sqrt{k_G^p(G_a, G_a) - 2k_G^p(G_a, G_b) + k_G^p(G_b, G_b)}$$

5.3.4 Algorithm Details

Graph Kernel Normalization. Normalization of the walk graph kernel is used to account for the fact that scenes containing many objects will tend to match better against all other scenes by virtue of their broader coverage of the dataset of models and relationships. For example, the relationship graph formed by a union of all the scenes in the database would match well to every scene. To combat this problem, we implement a normalization term by dividing the result of a graph kernel by the maximum of the evaluation between each graph and itself. For each graph kernel k_G we have a normalized graph kernel $\hat{k_G}$:

$$\hat{k_G}(G_a, G_b) = \frac{k_G(G_a, G_b)}{\max\left(k_G(G_a, G_a), k_G(G_b, G_b)\right)}$$

This normalization ensures that a graph will always match itself with the highest value of 1 and other graphs with values between 0 and 1.

Node Frequency Normalization. The importance of an object within a scene is intrinsically affected by the number of occurrences of that object within the scene. For instance, the existence of a book model in a scene is an important cue from which we can infer the type of room the scene is likely representing. The existence of hundreds more book models will naturally influence our understanding of the scene. However, the relative importance of each additional book diminishes as it is in essence an instance of an agglomeration. In order to represent this we introduce an occurrence frequency normalization factor for the node kernel evaluation following a term weighting approach used for document retrieval normalization [Salton and Buckley 1988]. Concretely, for a node n_a in the set of nodes V_G of a graph G:

$$\sigma(n_a) = \frac{1}{\sum_{n_b \in V_G} k_{\mathbf{node}}(n_a, n_b)}$$

This normalization factor scales the node kernel evaluation defined in Equation 5.1. The computed value for n_a is equal to 1 if the node is unique and decreases to 0 with more similar or identical nodes. Using this approach we avoid the problem of large agglomerations of object instances, such as books in a library, drowning out other interesting structure in a scene.

Parameter Selection. Our graph kernel, k_G^p, is parameterized by p, the length of the walks taken. Different choices for p will capture scene features at different resolutions, and it is unlikely that a single value of p will be the best kernel for any given task. This is a very common problem in machine learning and several multiple kernel learning techniques have been developed to allow learning tasks to benefit from information provided by multiple kernels [Bach et al. 2004]. Although it is possible for machine learning classifiers to directly make use of multiple kernels, it is very convenient to define a single kernel that is a linear combination of k_G^p for different values of p. We use the term *basis kernels* to refer to the individual kernels that are

summed to form the final kernel. Given any machine learning task, we can use cross-validation to decide on good weights for each of the basis kernels. Here we formulate a machine learning task that we will use to automatically learn the parameters.

Relevance feedback is a technique used by search engines to improve results. The user selects relevant scenes from a candidate list of results, and the search engine attempts to transform the feature space to favor results similar to ones the user marks as relevant [Papadakis et al. 2008]. The input to our relevance feedback implementation is a set of scenes selected by the user, each marked as either a good or bad response to the query. Given a specific scene kernel, we train a soft margin support vector machine classifier using the selected scenes as the training examples. We use the sequential minimal optimization algorithm to train our SVM [Platt 1999]. Because the SVM is only trained on the selected scenes this training is extremely fast; even if the user selects 50 scenes the SVM optimization always took us less than 50ms. We then use this SVM to rank all scenes in the database according to their signed distance to the separating hyperplane.

To use relevance feedback to learn our kernel parameters we first need to develop a training set. We suppose that we have N different user-designed queries (an example query might be "scenes that contain interesting sconce arrangements"). For each task, we assume users have gone through every scene in the corpus and evaluated whether it is a good or bad response to the query.

We can use this dataset to evaluate the quality of our relevance feedback results for a given set of parameters. For each modeling task we randomly pick samples from the database, half from the positive class and half from the negative class. Given a fixed set of kernel weights, we can compute the expected classification error of our SVM and average it over the N modeling tasks. We compute this average classification error for a large set of possible parameters, and choose the parameters with the lowest average classification error. In addition to varying this over a large set of possible weights for each basis kernel, we also search over different possible values of the soft margin parameter C. Although we found this cross-validation approach to multiple kernel learning to be sufficient for our purposes, it is also possible to directly optimize the weights of the basis kernels as part of a modified SMO algorithm [Bach et al. 2004].

5.4 Dataset

In this chapter we use a subset of the Google 3D Warehouse dataset described in Chapter 2 after limited human filtering has been applied. Specifically, we chose a subset of scenes and had humans manually select all the scene graph nodes that correspond to meaningful objects. For those nodes marked as meaningful, they were also tagged with a small description such as the object's category or subcategory (similar to the "name" field on Google 3D Warehouse models). This filtering approach mimics the methods used in computer vision to construct 2D image datasets such as PASCAL, MSRC, and LabelMe [Russell et al. 2008]. All root nodes are considered meaningful, as they correspond to complete scenes that have been uploaded to the database. We used this to acquire a set of scenes with approximately uniform segmentation and tagging quality.

Using this per-model information we can convert our scenes into relationship graphs. All nodes corresponding to models marked as meaningful by users become nodes in the relationship graph. Scene graph parent-child relationship edges are added between a node and its parent. If a node's parent node does not correspond to a meaningful model, we recursively move to the next parent until a node corresponding to a meaningful model is found. Geometry-based relationship edges are then added as described in Section 5.2, and the resulting relationship graphs are used as the input to our algorithm.

Because we are using humans to process scenes, rather than using all our 3D Warehouse scenes we focus on a subset that are relevant to a specific category of queries. We consider all scenes that contain more than one object and have any of the following tags: "kitchen", "study", "office", "desk", or "computer". In total we have chosen to use 277 such scenes with approximately 19000 models (including different instances of the same geometry). Indoor room scenes are an interesting area to study because their interior design contains structural patterns for our algorithm to capture.

Although we have observed most Google 3D Warehouse scenes to be over-segmented,

the architecture is usually not well segmented. For scenes such as a house with multiple rooms, the architecture itself contains interesting substructure that is often not captured in our scene graphs. While one might imagine several ways to automatically perform this segmentation, the subset of Google 3D Warehouse scenes we are considering are at most as complicated as a single room and usually do not have complex architectural substructures. Nevertheless, our algorithm can easily take advantage of more detailed information about the architecture that may be provided by some databases, such as computer-generated building layouts [Merrell et al. 2010].

5.5 Tools

Here we present scene modeling tools based on the graph kernel approach described above.

5.5.1 Relevance Feedback

Recall that k_G^p is parameterized by p and different choices for p capture scene features at different resolutions. As proposed in Section 5.3.4, we can use relevance feedback to perform parameter selection and determine a good aggregate kernel that captures features at different scales.

To build a training set we presented four different users with our scene database and asked them to think of a scene modeling task of their choice. For example, they might want to find scenes with interesting computer peripherals such as webcams or fax machines, or find scenes with wall sconces that fit well in a study they are modeling. They then classified each scene in the database as either being a good or bad response to their task. Unlike the experiment describe in Section 4.3.4, we have a comparatively small number of scenes so it is possible to enumerate the relevancy of every possible result.

We use this training set for parameter selection. Our basis kernels are walk graph kernels of length 0 to 4. We consider all possible linear combinations of these five kernels with weights ranging from 0 to 1 at 0.1 increments. We also consider the

following values for the soft margin parameter C: $\{0.001, 0.01, 0.1, 1, 10, 100\}$. For this test we used 6 positive and 6 negative examples, and averaged the classification error over 10,000 randomly chosen training sets and over each of the four modeling problems. The best set of parameters is shown in Table 5.2 and had an average classification error of 20.9%. The varied nature of these coefficients suggests that different settings of our graph kernel are capturing different features of the scenes.

C	k_G^0	k_G^1	k_G^2	k_G^3	k_G^4
10	0.5	0	0.1	0.1	0.3

Table 5.2: Weighting used for combining graph kernels with different path lengths.

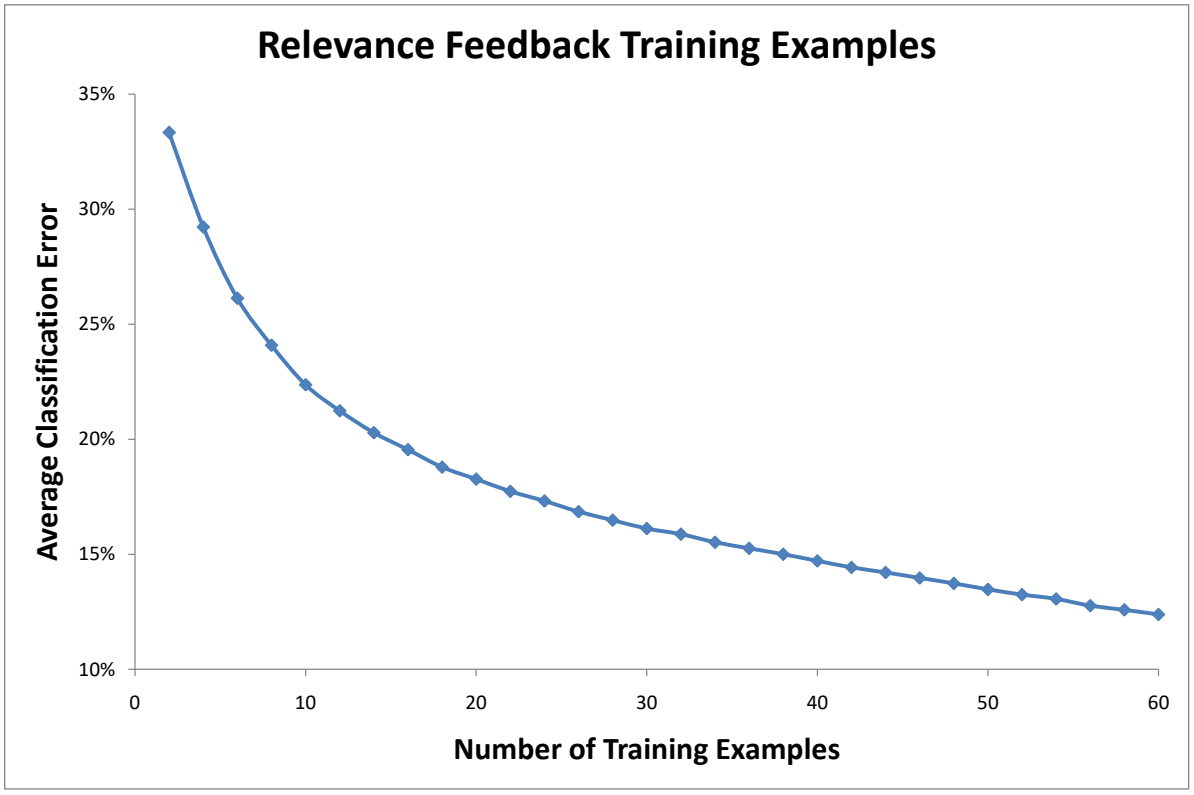

Figure 5.4: Classification error using a support vector machine to distinguish between relevant and irrelevant scenes as a function of the number of user-selected training examples.

In Figure 5.4 we compare the classification error using the weighted kernel from

Table 5.2 as a function of the number of training examples used. Although the classification error decreases steadily, it remains close to 12% even when 30 positive and 30 negative training examples are used. This suggests that many of the queries designed by our users contain challenging subtleties that are not easily captured by our graph kernel.

Because the SVM ranks scenes based on confidence, the user is first presented with suggestions that the algorithm determines are very likely to be relevant. In Figure 5.5 we show relevance feedback results using 4 training scenes and the coefficients given in Table 5.2. Even with a small number of selected scenes the algorithm is able to infer the user's preferences. All of the top 18 results are relevant to the query.

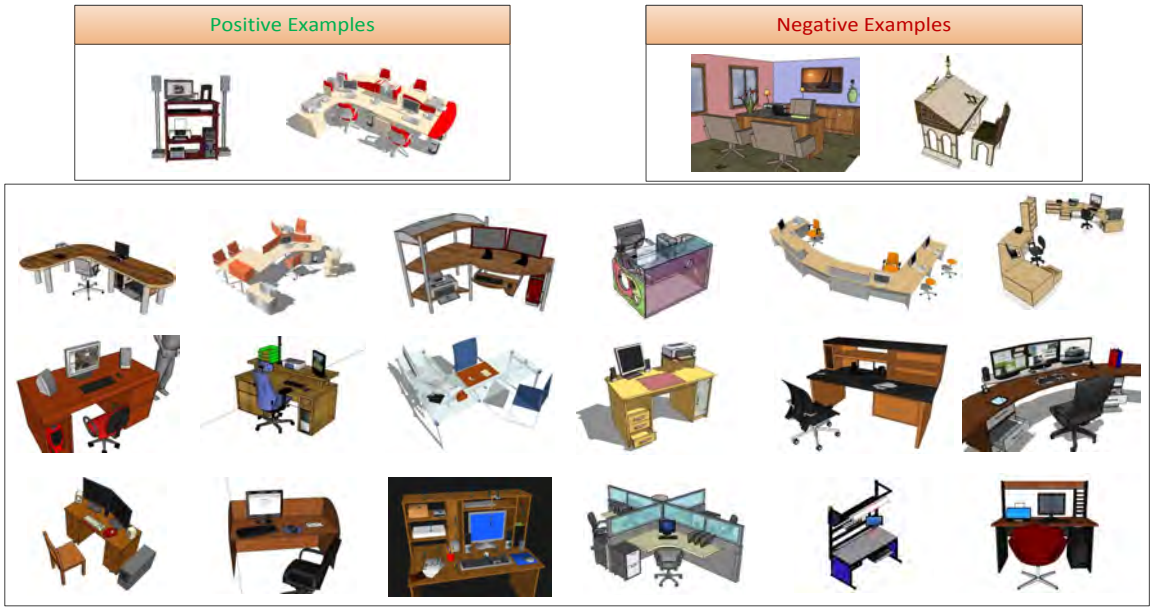

Figure 5.5: Search results using relevance feedback. Top: A user looking for scenes with interesting computer peripherals selects two scenes they like and two scenes they do not like from the database. Bottom: The top 18 results using a scene search guided by the user's selections.

At first glance relevance feedback may seem cumbersome — users must first issue a query, classify multiple results, and finally ask for relevance feedback based on their selections. However it is possible for search engines to passively use relevance feedback methods by aggregating search history information from many previous users. For

example, if a user issues a common query that the engine has seen many times before, click-through rates or model insertion rates could be used as a proxy to predict the relevance or irrelevance of results.

5.5.2 Find Similar Scenes

One application of our scene kernel is to suggest scenes that are related to the one the user is modeling, to allow them to easily find and import related content. We use the aggregate graph kernel described in Table 5.2 to compute the similarity between the chosen scene and all the scenes in the database. Scenes are ranked using this similarity value in decreasing order.

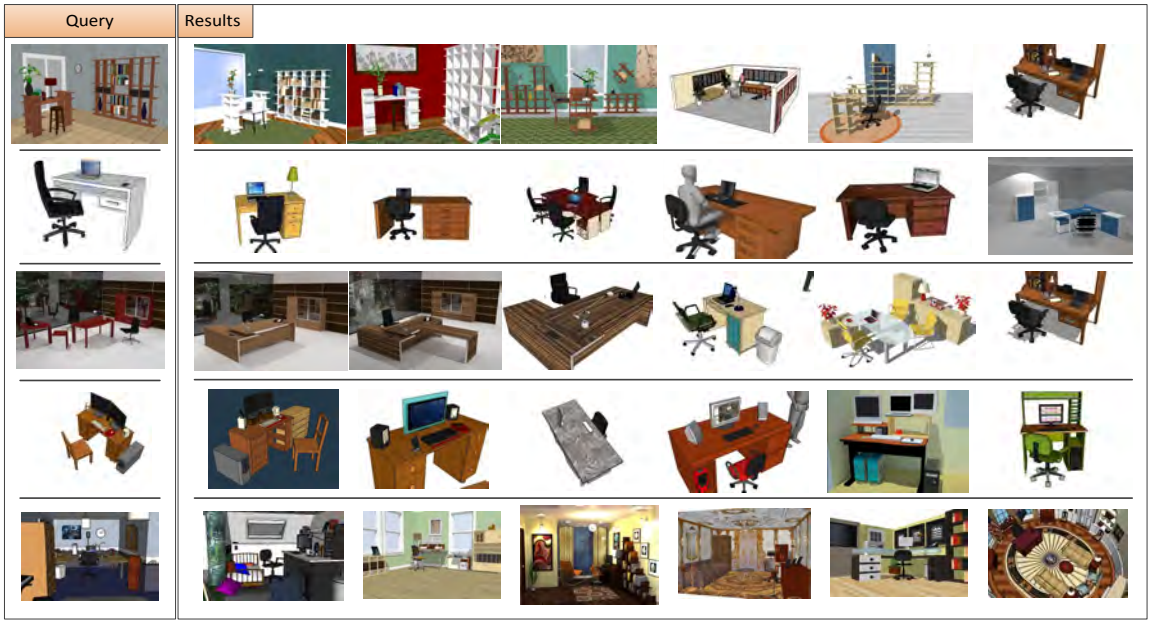

Figure 5.6: "Find Similar Scene" search results. Left: The query scene provided by the user. Right: The top six scenes in the database that match the query. The best match is shown on the left.

In Figure 5.6 we show the top-ranked results for five different queries. Our algorithm returns scenes that share many structural similarities with the query scene. For example, in the first scene many results contain a similar style of shelving. Likewise,

in the second scene the top results are all simple desk scenes with laptop computers. These results also demonstrate the large amount of structure sharing used by artists in Google 3D Warehouse. For example, in the fourth scene query, the top ranked result uses the exact same models on the top of the desk, while changing other aspects of the furniture and layout.

5.5.3 Context-based Model Search

Using our framework, there is an intuitive mapping between a context-based search for 3D models and our rooted-walk graph kernel (k_R^p). We implement such a search by placing a virtual query node in the graph. The relationships that the desired object should have with other objects in the scene are defined through a labeled set of connecting query edges. Geometry, tags, and other node properties can optionally be provided by the user to refine the query beyond just considering relationships to other models in the scene. Figure 5.7 illustrates a query for an object that is in contact with a desk.

Once we have placed the virtual node within the query scene's relationship graph, we then evaluate k_R^p between this virtual node and all other relationship graph nodes in our scene corpus. The k_R^p evaluation for each node indicates how similar the environment around that model is to the environment around the query node. We use this evaluation to rank all models in the database.

The walk length parameter p in k_R^p controls the size of the contextual neighborhood that is compared by the model context search. When $p = 0$, all models are ranked by considering only the geometry, tags, or size provided by the user for the query node. When $p = 1$, the algorithm additionally considers the geometry and tags of the models connected to the query node by query edges (in Figure 5.7, this would just be the desk model). Increasing to $p = 2$ and beyond provides an intuitive way of visualizing the region of support considered by the context search.

In Figure 5.8 we show the results of two context-based queries. In both cases the user is searching for models that would belong on top of the desk in their scene. A query node was inserted into these scenes and connected to the desk node through

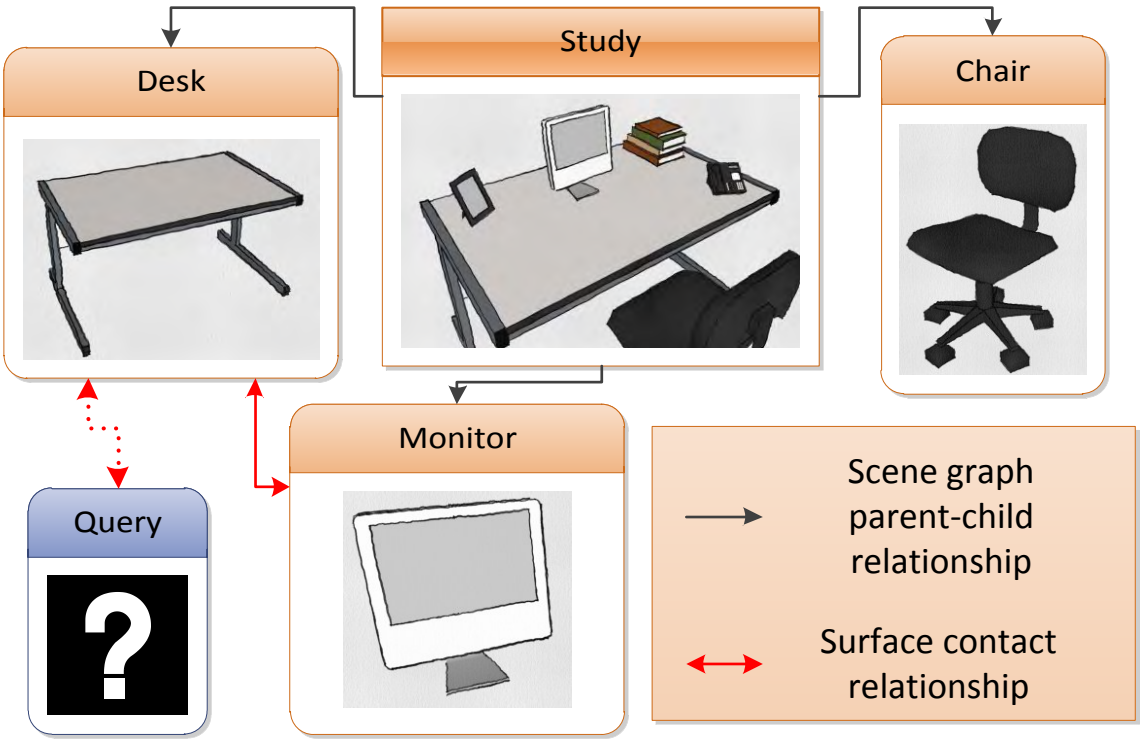

Figure 5.7: A model context search expressed by introducing a query node to a relationship graph. The dotted edge represents the contact relationship that connects the query node to the scene and defines its context. Multiple edges of different relationship types can easily be introduced for a single query node.

a horizontal support relationship edge. We then evaluate k_R^p for walk length $p = 3$ between the query node and all other relationship graph nodes in the database to determine model suggestions.

The results indicate how the existence of a computer and related objects on the desk in the top scene produces computer peripheral and accessory suggestions. In contrast, the bottom scene's context induces results that are more appropriate for a study desk. Also observe that in both cases all of the highly ranked results are at least potentially relevant — only models that have been observed to be horizontally supported by desks (or models geometrically similar to desks) can be highly ranked.

The context-based search algorithm described in Chapter 4 assumes all pairs of

Figure 5.8: Context-based model search results. Left: A user modeling a desk scene issues a query for a model to be placed on the desk. Right: Highest ranked models for each query. Note how the context provided by the models in the query scene influences the categories of the suggestions.

objects are random independent events. In contrast, the graph kernel approach considers the structural relationships between all objects in the scene. To illustrate the difference between these two approaches, we modeled a desk scene with a bowl and two cups, shown on the left side of Figure 5.9. Consider a user who wants to search for other objects to place on this desk. We want to execute this query using both methods.

There are some differences between the formulation of context search queries in these algorithms that need to be resolved before a comparison can be made. The graph kernel approach expresses the desired location as a relationship to existing objects, while the approach from Chapter 4 expresses the location as a 3D point in the scene. We have chosen a point 10cm above the center of the desk to correspond to our algorithm's "horizontal contact" relationship to the desk.

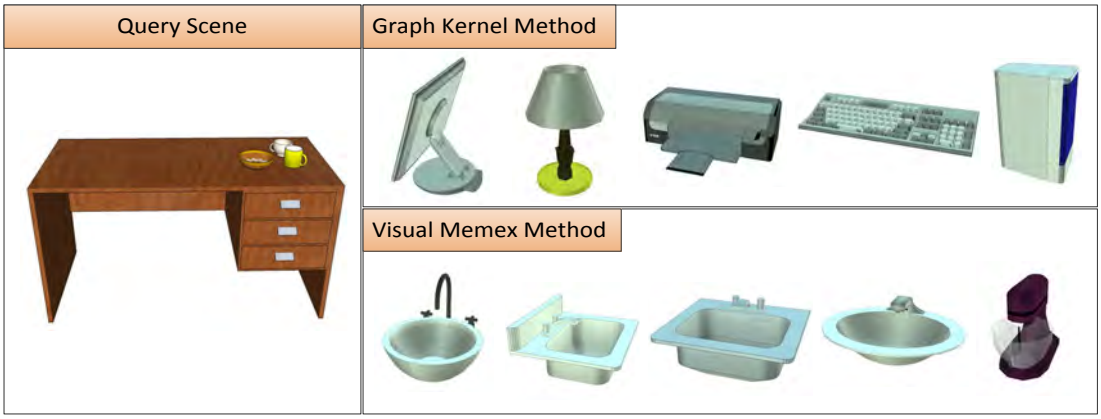

Figure 5.9: Left: The user asks for an object on the desk. Right: Results comparing the results of the graph kernel method and the Memex method (Chapter 4).

In Figure 5.9, we show the results using both approaches on our database. Because the graph kernel algorithm considers the structural relationships in the scene it returns many plausible objects that have been observed on desks in the database, such as lamps and speakers. On the other hand, the Memex algorithm considers all object pairs independently. It returns objects such as sinks and mixers because these objects are often found in the vicinity of bowls, cups and drawers — by making the assumption that object occurrences are independent, it has been "fooled" into thinking that this is a kitchen-like scene. By not considering the semantic relationships between objects, the Memex algorithm is not able to determine that sinks are not commonly found on top of desks — the graph kernel approach will only make such suggestions if there is a scene in the database with a mixer or sink on top of a desk.

5.5.4 Performance

One nice property of graph kernels is that dynamic programming permits extremely efficient evaluation. To test the performance of our algorithm we computed the walk graph kernel k_G^4 between all pairs of scenes in our database. The average graph to graph k_G^p evaluation took 0.150ms. Using this average value it would take approximately 1.5s to exhaustively compare a query scene against a 10,000 scene database.

This experiment was run on a quad-core 2.53GHz Intel Xeon CPU. Note that this is also the approximate cost of executing a 3D model context query — a subcomponent of evaluating k_G^p between two scenes is to compute k_R^p between all possible node pairs. Overall, we feel our algorithm is fast enough for interactive use on most databases.

5.6 Chapter Summary

We have presented a novel framework for characterizing the structural relationships in scenes using a relationship graph representation. Our basis kernels compare the local information contained in individual models and relationships, and the walk graph kernel aggregates this information to compare the topological similarity between two scenes. We have shown that our algorithm can be used for several scene modeling tools, including suggesting similar scenes and context-based model search. A positive-definite kernel is a very powerful tool, and by defining one between two scenes we gain access to a rich body of machine learning techniques that have been developed. In practice we have found scene suggestion to be a convenient tool for exploring design alternatives. However its usefulness is most pronounced on large databases such as Google 3D Warehouse, and the tool suffers on smaller databases such as the Scene Studio database discussed in Chapter 2. Without a critical number of scenes there is not enough variety within the different types of scenes for the discriminative power of our relationship graph kernel to provide significant benefit.

Chapter 6

A Generative Model for 3D Scenes

6.1 Introduction

In this chapter, we will present an algorithm to generate new environments similar to a set of input examples [Fisher et al. 2012]. Our pipeline is exemplified in Figure 6.1. A user provides the system with examples illustrating the desired type of scene, and the system returns a large set of synthesized results. Unlike previous tools, which focus on low-level operations such as model search or scene retrieval, this tool simulates the effects of a very large number of operations at once. Although considerably more powerful, performing so many operations without the artist being involved at each stage is very challenging and we will present several contributions that enable us to generate plausible and diverse environments.

There are several criteria an example-based synthesis algorithm should meet. First, it should generate plausible scenes; they should look believable to a casual observer. Second, it should generate a variety of scenes; they should not be copies or small perturbations of the examples. Third, users should only have to provide a few examples, since they are time-consuming to create.

These goals are challenging to meet, and some stand in conflict with one another. Generating a variety of scenes is difficult when the system can only draw data from a few examples. Scenes can contain a large number of different objects, the full range of which can be difficult to specify in a few examples. Just because the user

Figure 6.1: Example-based scene synthesis. Left: four computer desks modeled by hand and used as input to our algorithm. Right: scenes synthesized using our algorithm. Our algorithm generates plausible results from few examples, incorporating object composition and arrangement information from a database of 3D scenes to increase the variety of its results.

omitted some type of object in the examples, does that mean she does not want it in her scenes? Some objects are connected via precise functional and geometric relationships, while others are more loosely coupled. To generate plausible scenes, an ideal example-based algorithm would infer these relationships from data without additional user guidance.

The example-based scene synthesis method presented in this chapter meets the above challenges through three main contributions. Our first contribution is a *probabilistic model for scenes*. It consists of an *occurrence model*, which specifies what objects should be in the generated scenes, and an *arrangement model*, which specifies where those objects should be placed. The occurrence model uses a Bayesian network, drawing on recent work in example-based shape synthesis [Kalogerakis et al. 2012]. The arrangement model uses a novel mixture of Gaussians formulation.

Our second contribution is a clustering algorithm that automatically discovers interchangeable objects in a database of scenes and forms them into groups. We call these groups *contextual categories*. A contextual category can contain a greater variety of objects than the basic categories (i.e. "table," "chair," "lamp") used by most applications. To find these categories, our algorithm exploits the insight that objects that occur in similar local neighborhoods in scenes (i.e. "on a plate," "beside a keyboard") are likely to be considered interchangeable when building plausible scenes. Using contextual categories, our algorithm can incorporate a wider variety of objects in synthesized scenes.

Our third contribution is a method for learning the probabilistic models from a mix of example scenes and scenes from a large database. In doing so, we treat the database as a "prior" over possible scenes. This provides a wide variety of scene content, and the examples guide that content toward a particular desired target. We allow the user to control the strength of the database prior through simple parameters, trading similarity to examples for increased diversity. Using these *mixed models*, our algorithm can synthesize scenes with a greater variety of both objects and arrangements.

Our results demonstrate the utility of our method for synthesizing a variety of scenes from as few as one sparsely populated example. Through a judgement study with people recruited online, we found that approximately 80% of synthesized scenes

are of adequate quality to replace manually-created ones. This level of quality is more than sufficient for a user to browse a sorted list and find synthesized scenes of interest.

We refer to our learned model as a *schema*. In psychology and learning theory, a schema is a mental structure that represents knowledge about the world [Neisser 1967]. Similarly, our schemata are data structures that represent knowledge about a particular type of environment.

Our learning algorithm relies upon having a static support hierarchy defined for the input scenes. The results we will show all use the Scene Studio database described in Chapter 2 as the backing scene database, and the examples are also generated using our scene modeling program.

6.2 Related Work

Related work has tackled problems similar to example-based scene synthesis. While our algorithm uses some of the same underlying techniques, none of these methods alone are sufficient to meet our goals.

Component-based object modeling Recent work has demonstrated a graphical model for individual objects (such as chairs, planes, and boats) that encodes the cardinality, style, and adjacencies of object sub-components [Kalogerakis et al. 2012; Chaudhuri et al. 2011]. This model can be trained on input objects of a particular class and sampled to generate new objects that are recombinations of input object components. This approach is well-suited for object synthesis, since objects in a given class typically have few functional sub-components whose placement is well-specified by mesh adjacency information. Scenes do not share these properties: they can contain dozens of different, loosely-related objects, and each object is free to move continuously on its supporting surface. Our algorithm uses a separate *arrangement model* to capture the continuous spatial relationships between many possible objects. The *occurrence model* determines which objects should go in a scene and is based on the Bayesian network formalization used in Chaudhuri et al. [2011], with several modifications to better support scene synthesis.

Evolutionary object modeling Another approach to object synthesis evolves a set of diverse objects that is iteratively fit to a user's preferences [Xu et al. 2012]. With this scheme, generated objects are always an interpolation of the initial input set of objects. Thus, the algorithm cannot introduce any new object sub-components that were not present in the input set. This restriction is acceptable for objects, since they typically have a handful of functional subcomponents. Scenes can contain dozens of loosely-related objects, however, so using this method on a few input scenes will generate repetitive content. Since our algorithm extracts *contextual categories* and learns *mixed probabilistic models* from a database of scenes, it can incorporate new types of objects not found in the examples and increase the variety of synthesized results.

Inverse procedural modeling Researchers have also synthesized objects from examples by inferring a procedural model that might have generated those examples [Bokeloh et al. 2010]. This system searches for partial symmetries in the example object geometry, cuts the objects into interlocking parts based on those symmetries, and then generates a grammar that can synthesize new objects by stitching compatible parts together. This method works well with objects that are defined by repeated, regular sub-structures—a property that most scenes do not exhibit. Salient relationships in scenes cut across levels in the scene graph hierarchy, so context-free grammars are unlikely to model them well. In contrast, our probabilistic models for object *occurrence* and *arrangement* can learn salient existence and placement relationships between any pair of objects.

Automatic furniture layout Outside the domain of object synthesis, methods have recently been developed for optimizing the layout of furniture objects in interior environments [Merrell et al. 2011; Yu et al. 2011]. These methods define an energy functional representing the 'goodness' of a layout, and then use stochastic optimization techniques such as simulated annealing to iteratively improve an initial layout. While they can generate plausible and aesthetically pleasing furniture arrangements, these methods do not completely synthesize scenes, since they require a user to specify

the set of furniture objects to be arranged. In contrast, our algorithm chooses what objects exist in the scene using its *occurrence model*. These methods also require non-example input: Yu et al. [2011] uses example layouts but requires important object relationships to be marked, and Merrell et al. [2011] uses domain-specific principles from interior design and requires a user in the loop. Our *arrangement model*, based on Gaussian mixtures and Gaussian kernel density estimation, can be trained entirely from data. It uses a heuristic for relationship salience to automate arrangement of scenes with dozens of objects.

Open-world layout Closely-related recent work seeks to automatically generate 'open-world' layouts, which are layouts with an unspecified number of objects [Yeh et al. 2012]. The underlying generative model is a probabilistic program, which is compiled into a factor graph representation and sampled using a variant of Markov Chain Monte Carlo. The algorithm succeeds at synthesizing interior environments, such as coffee shops, with varying shapes and scales. However, the underlying probabilistic program is written by hand and can only generate patterns that were expressed in the code. In contrast, our algorithm learns its generative model from data and does not require any programming ability on the part of the user.

6.3 Approach

We want to synthesize scenes from a few input examples. The fundamental challenge is that scene synthesis is hard due to the number of possible configurations. Even for a single type of scene with a modest number of objects, such as an office desk scene, the restricted set of plausible configurations that we would like to synthesize is large and very high-dimensional. Furthermore, a user who provides a few examples can only express a small number of desired configurations, but the assumption that they would like synthesized results to conform to only the objects and arrangements in the examples is usually false. We need to address this combination of a large output domain and a restricted set of inputs with an approach that avoids generating repetitive results while retaining plausibility and similarity to the examples.

Our insight is that, much like users have diverse background knowledge from which they draw to construct examples, we can turn to a large database of scenes in order to "fill in the gaps" between the examples. The scene database provides many models instantiated in plausible configurations which we exploit in two ways to improve the diversity of our synthesized scenes. Firstly, we compute contextual categories by using object neighborhoods in scenes to group together objects that are likely to be considered interchangeable. These contextual categories then allow the synthesis algorithm to use more objects in any given role. We describe a simple algorithm to compute these categories using bipartite matching and hierarchical agglomerative clustering. Secondly, we treat the scene database as a prior over scenes and train our probabilistic model on both the examples and relevant scenes from the database. We use a recently-developed similar-scene retrieval technique to choose scenes relevant to the examples and introduce diversity without impacting plausibility. The user can control the degree of mixing between the two data sources via intuitive blending parameters.

For our investigation we use the Scene Studio database described in Chapter 2.4. Each object is tagged with a *basic category label* such as "clock" or "bed." Each object is also rooted to a point on the surface of another object; the set of all such "parent-child" relationships defines a *static support hierarchy* over the scene that our algorithm exploits. Section 6.9 describes the results of our database construction effort in more detail.

Our system begins by retrieving contextual categories from a large database of scenes (Section 6.4). Given these categories, it then learns mixed models from both the user provided examples and the scene database (Section 6.5). It first trains the occurrence model, which describes what objects can be in synthesized scenes (Section 6.6). It then trains the arrangement model, which describes where those objects can be placed (Section 6.7). Finally, it samples from these probabilistic models to synthesize new scenes (Section 6.8).

6.4 Contextual Categories

We would like our synthesis algorithm to be able to draw from a wide variety of objects in order to increase the diversity of our scenes. Previous work addressing furniture layout used predefined sets of basic functional categories, such as "chairs" [Merrell et al. 2011; Yu et al. 2011]. For scene synthesis, such a categorization can be restrictive, leading to repetitive scenes. We note that many objects are interchangeable with other objects not necessarily from the same basic category. For example, a contextual category of "objects that belong on a plate in a kitchen" may contain many different basic categories such as "apples," "bread," "cookies," etc.

Our insight is that such interchangeable objects are frequently surrounded by objects that are similar both in type and arrangement. We use the term *neighborhood* to refer to the arranged collection of models around an object. This insight suggests that neighborhood similarity can predict interchangeability to automatically construct contextual categories.

Neighborhood similarity To group objects using the similarity of their neighborhoods, we must first quantify neighborhood similarity. We reduce each object x to its centroid point plus its basic category label L^x. Comparing two neighborhoods then reduces to matching two labeled point sets, which is a well-studied problem in computer vision and structural bioinformatics. Popular methods include techniques based on geometric hashing [Wolfson and Rigoutsos 1997], bipartite matching [Diez and Sellarès 2007], and finding maximum-weight cliques in association graphs [Torsello et al. 2007].

Our approach is based on bipartite matching. To compare objects A and B, we transform the database scenes in which they occur such that A and B are at the origin and the normals of their supporting surfaces are aligned with $\vec{z} = [0, 0, 1]^T$. We form the complete bipartite graph between the objects in these two scenes, where the weight of each edge in the graph is:

$$k(a, b) = \mathbf{1}\{L^a = L^b\} \cdot G(|\vec{a} - \vec{b}|, \sigma_d) \cdot G(\min(|\vec{a}|, |\vec{b}|), \sigma_n)$$

Here, $\mathbf{1}\{\}$ is the indicator function, $\vec{o}$ is the vector-valued position of object o and $G(x, \sigma) = e^{-x^2/2\sigma^2}$ is the unnormalized Gaussian function. This equation states that objects should only match if they have the same basic category and if the distance between them is small. The third term decreases the significance of matches that occur far away from the objects that we are comparing (A and B). In our implementation, $\sigma_d = 15\,\text{cm}$ and $\sigma_n = 90\,\text{cm}$.

We solve for the maximum-weight matching $\mathbf{M}$ in the graph using the Hungarian algorithm [Kuhn 1955]. Our neighborhood similarity function is then:

$$\begin{aligned} n(A, B) =& \mathbf{1}\{\text{isLeaf}(A) = \text{isLeaf}(B)\} \cdot \\ & G(\frac{|A| - |B|}{\min(|A|, |B|)}, \sigma_s) \cdot \sum_{(a,b)\in\mathbf{M}} k(a, b) \end{aligned}$$

where $\text{isLeaf}(o) = \text{true}$ if object o is a leaf node in its scene's static support hierarchy, and $|o|$ is the diagonal length of the bounding-box of object o. The first term states that two objects are similar if they serve the same support role (i.e. they either statically support other objects or do not). The second term compares the sizes of the objects themselves, and the third term compares the similarity of their neighborhoods. We use $\sigma_s = 1.5$.

We then discretize all possible rotations of B's neighborhood about $\vec{z}$ and find the orientation for which $n(A, B)$ is strongest

$$\bar{n}(A, B) = \max_{\theta} n(A, \text{rot}(B, \vec{z}, \theta))$$

and then normalize the result to be in the range $[0, 1]$

$$\hat{n}(A, B) = \frac{\bar{n}(A, B)}{\max(\bar{n}(A, A), \bar{n}(B, B))}$$

Clustering We cluster all objects in all database scenes using hierarchical agglomerative clustering (HAC) [Steinbach et al. 2000], where the merge score between two

clusters C_1 and C_2 is

$$Merge(C_1, C_2) = \max_{A \in C_1, B \in C_2} (1 - \lambda_{\mathbf{cat}}) \cdot \mathbf{1}\{L^A = L^B\} + \lambda_{\mathbf{cat}} \cdot \hat{n}(A, B)$$

The influence of neighborhood similarity is controlled by $\lambda_{\mathbf{cat}}$. When $\lambda_{\mathbf{cat}} = 0$, clustering only considers basic category labels and will recover these basic categories exactly. As $\lambda_{\mathbf{cat}}$ increases, objects from different basic categories that occur in similar neighborhoods are encouraged to merge together. For our database, $\lambda_{\mathbf{cat}} = 0.7$ yielded many useful categories; see Section 6.9 for examples. The merge score above is the single linkage criterion for HAC [Murtagh 1984], which states that two clusters are as similar as their most similar objects. Since it is susceptible to outliers, in practice we use the 90^{th} percentile rather than the maximum. We stop merging when the best cluster merge score is less than a similarity threshold $\tau = 0.1$.

Alignment Our synthesis algorithm requires all objects in a category to be aligned in a common coordinate frame. We align our objects using both their neighborhood similarity and their geometric features. We first choose an anchor object at random and align the neighborhoods of all other objects to the anchor's neighborhood using the neighborhood similarity function $\hat{n}$. This process alone can successfully align objects with strong neighborhood cues, such as keyboards. To further refine an alignment, we compute the sum of squared distances for all possible rotations, which has been shown to be effective at aligning certain categories of 3D models [Kazhdan 2007]. We snap our neighborhood alignment to the nearest local minima of this inter-object distance function. For some categories that are sensitive to changes in rotation, such as computer speakers and alarm clocks, we manually inspect and correct the inter-object alignments.

Figure 6.2 demonstrates our alignment process for two desks and two alarm clocks. For desks, the optimal contextual neighborhood alignment is close to the correct alignment, but is off by a small rotation. On the other hand, the optimal geometric alignment corresponds to a 180-degree rotation of the correct alignment. We achieve the best results by using the nearest local geometric minima to the global contextual

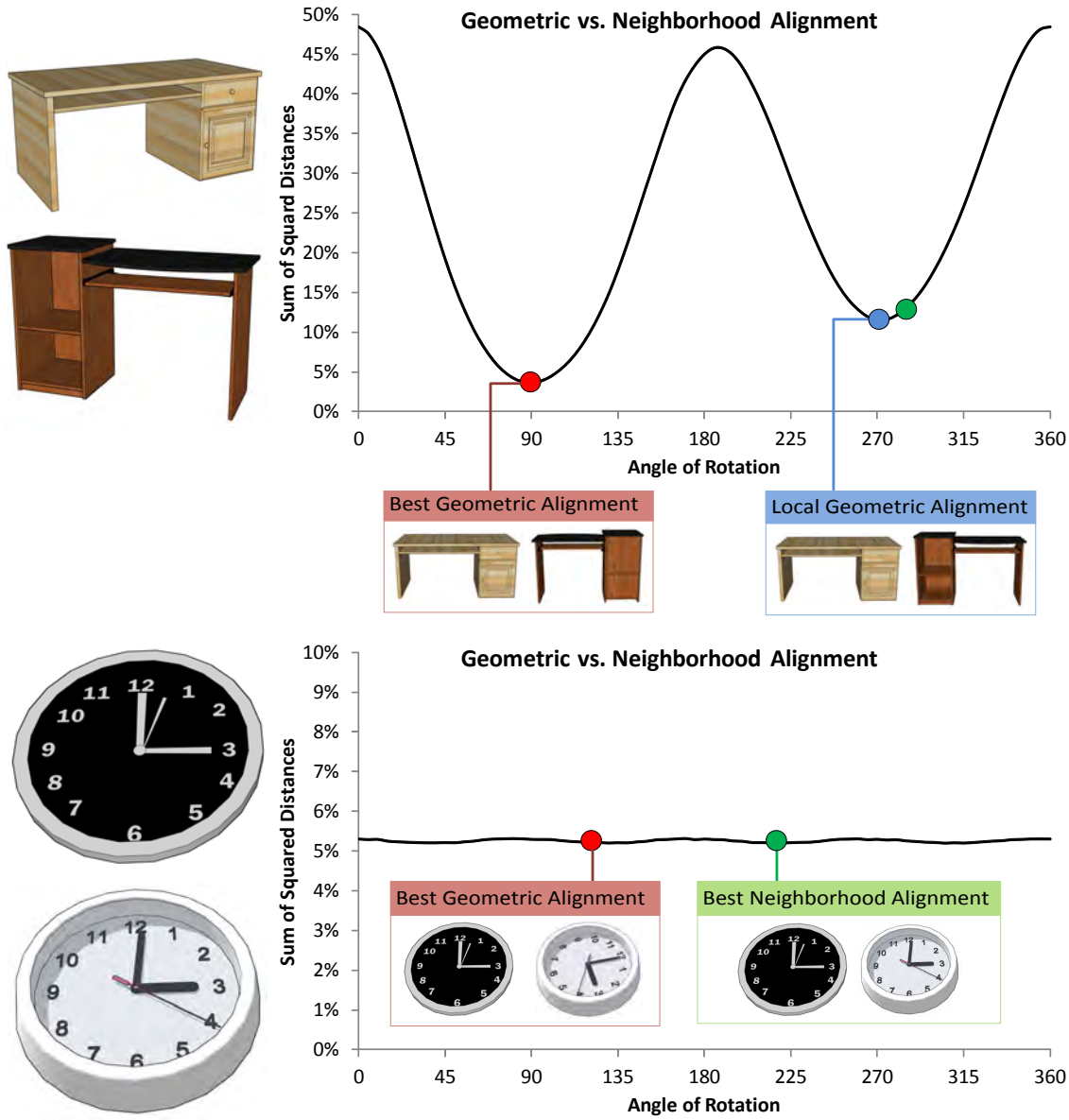

Figure 6.2: We use both contextual and geometric information to align objects within a category. On the left, we show the optimal alignment, using only our context-based neighborhood comparison function. On the right, we show the geometric alignment term as a function of rotation angle; smaller scores correspond to better alignments. The red circle indicates the optimal alignment considering only the object's geometry. The green circle indicates the optimal alignment using only the neighborhood score. We achieve the best performance by combining both contextual and geometric information when aligning categories.

optimum, shown in blue. For alarm clocks, geometric information provides almost no information: to correctly align these two models, textural information must be used. However, by using the local neighborhood information we can achieve very good alignment between the two clocks.

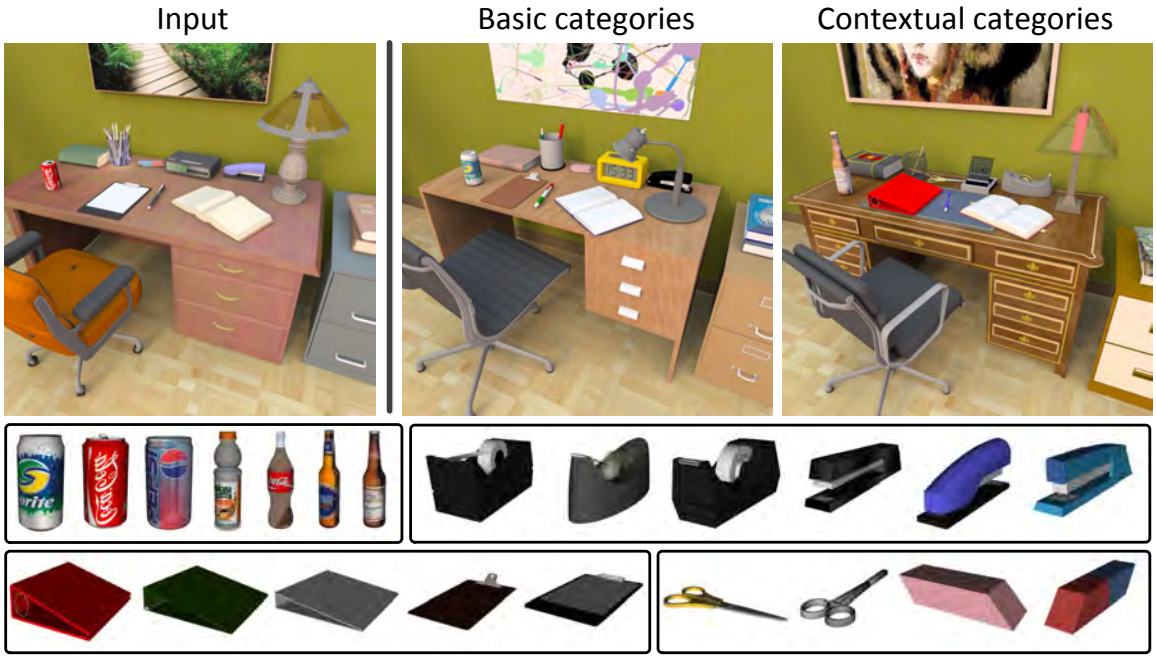

Figure 6.3: Comparing basic and contextual categories. We show an input scene and the synthesized results using basic and contextual categories. On the bottom we show four of the relevant contextual categories generated by our clustering algorithm. Contextual categories allow synthesis with a greater variety of plausible models.

Figure 6.3 shows some of our contextual categories. Later in this chapter, we will show some of the advantages of using contextual categories for scene synthesis.

6.5 Learning Mixed Models

Creating many diverse scenes from a small number of examples is difficult even if we draw upon contextual categories. Frequently, users intend examples they provide to be rough guides of the type of scene they would like and may be missing many details, which the user may not have recalled but has seen in the past. Our insight

is that a database of scenes can act as a prior over scenes and can be used to fill in missing details or enrich the synthesized scenes. The degree to which this is a desired behavior may vary depending on the user and task, so we would also like to control how strong we want the influence of the prior to be by using intuitive mixing parameters.

In Sections 6.6 and 6.7, we learn two models from a mixture of data provided by the few examples and a larger scene database. We define these models here and describe how to mix separate input data sources but defer discussion of the details of each model to the relevant sections. The occurrence model $\mathcal{O}(S)$ is a function which takes a scene S as input and returns a probability for the static support hierarchy of the objects in the scene. The arrangement model $\mathcal{A}(o, S)$ is a function which takes an object o positioned within a scene S and returns an unnormalized probability of its current placement and orientation.

During synthesis, we cannot blindly draw information from the database: if the examples depict bedrooms, we don't want to draw from bathrooms introducing toilets in our synthesized scenes. The need to isolate 'semantically similar' content from a large database has been recognized in prior work on data-driven content generation in computer graphics [Hays and Efros 2007]. During model training, we retrieve scenes similar to the example scenes $\mathfrak{E}$ using the graph kernel-based scene comparison operator described in Chapter 5. To threshold the number of retrieved scenes we sort them by similarity value and use only the results with more than 80% of the maximum value, forming the relevant set of scenes $\mathfrak{R}$.

Our system then learns its probabilistic models from the scenes in both the examples $\mathfrak{E}$ and the relevant set $\mathfrak{R}$. The extent to which each data source is emphasized is determined by two mixing parameters $\lambda_{\mathbf{occur}}$ and $\lambda_{\mathbf{arrange}}$. For these parameters, a value of 0 denotes 100% emphasis on $\mathfrak{E}$ and a value of 1 denotes 100% emphasis on $\mathfrak{R}$.

Mixing the arrangement model is straightforward: we train one model on $\mathfrak{R}$, another model on $\mathfrak{E}$, and the linearly interpolate between them using $\lambda_{\mathbf{arrange}}$. Section 6.7 describes the procedure for training a single model.

Mixing the occurrence model is less trivial since it uses Bayesian networks which

cannot simply be interpolated. Instead, we use an enrichment approach where we replicate each input scene many times to create a larger training set containing N observations. Each scene in $\mathfrak{E}$ is replicated $\lceil \frac{N(1-\lambda_{\mathbf{occur}})}{|\mathfrak{E}|} \rceil$ times and equivalently each scene in $\mathfrak{R}$ is replicated $\lceil \frac{N\lambda_{\mathbf{occur}}}{|\mathfrak{R}|} \rceil$ times. The results in this chapter use $N = 500$. The model is then learned as described in Section 6.6.

6.6 Occurrence Model

We learn from the replicated set of scenes described in Section 6.5 a model $\mathcal{O}(S)$ over the static support hierarchy of a scene S. Our support hierarchy model is broken down into two parts. First, we use a Bayesian network $\mathcal{B}(S)$ to model the distribution over the set of objects that occur in a scene. Second, given a fixed set of objects we use a simple parent probability table to define a function $\mathcal{T}(S)$ that gives the probability of the parent-child connections between objects in a scene.

6.6.1 Object Distribution

Following previous work on learning probabilistic models for part suggestion in 3D object modeling, we use a Bayesian network to represent our learned probability distribution [Chaudhuri et al. 2011]. We can sample from this Bayes net to produce plausible sets of objects that do not occur in the examples. Prior to performing structure learning, we transform the network to reduce the number of free parameters introduced, which is important when learning from a small number of examples. We also constrain our Bayesian network to guarantee that the set of objects generated can be connected together into a valid static support hierarchy.

We start by representing each input scene with a *category frequency vector* that counts the number of times each category occurs in the scene. We consider these vectors to be observations of discrete random variables over which we want to learn a Bayesian network. To guarantee that our network generates plausible scenes, we will define below a set of constraints that specify edges which must be present in the learned network. Given these edge constraints, we use a standard algorithm to learn

Lab table	Test tube rack	Test tube	Notebook	Calculator
1	1	2	0	0
1	0	1	0	1
1	0	0	1	1

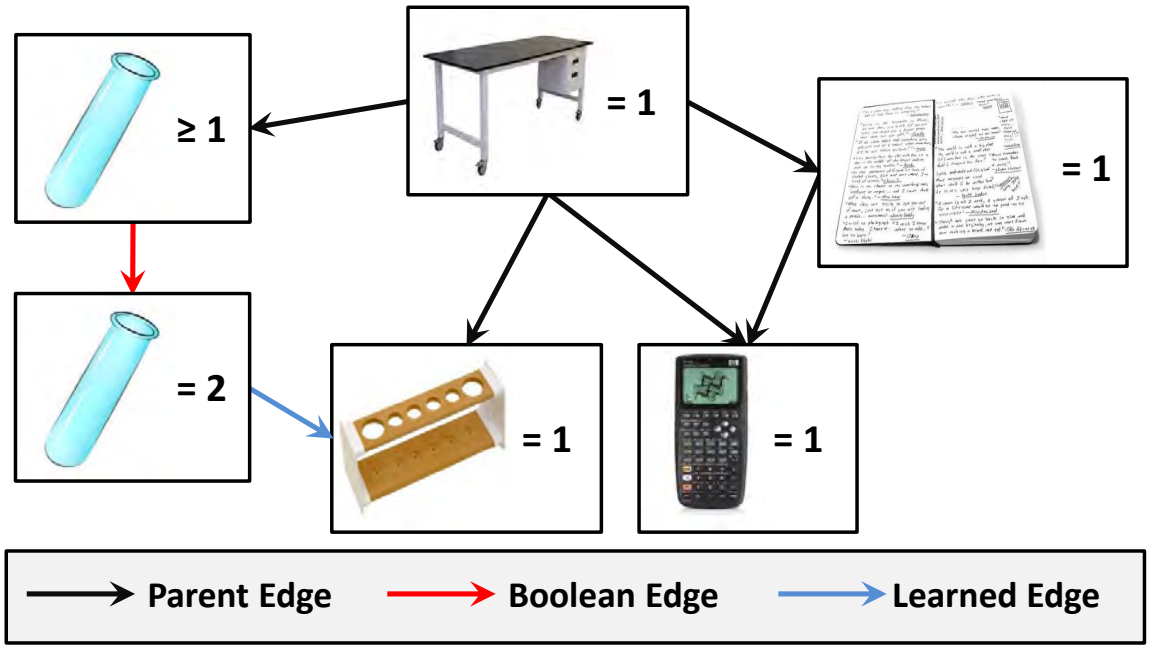

Figure 6.4: Bayesian structure learning example. We start with three input scenes and their corresponding category frequency vectors. At the bottom we show a Bayesian network learned from these vectors. The black edges are enforced by observed static support relationships. Network booleanization splits the test tube variable into two nodes and enforces the edge shown in red. In blue, we show one edge learned by our structure inference algorithm that captures the positive correlation between multiple test tubes and a test tube rack.

the structure of our Bayesian network following prior work on object modeling by Chaudhuri et al. [2011] with the following modifications:

Support constraints Our Bayesian network should guarantee that for every generated object, there is another object that can support it. Thus we require that for every category C, the variable representing that category must be conditional on the variable for each parent category C' it has been observed on.

Booleanization Prior to learning, we transform our problem to better handle the case where we have few input scenes. Each random variable in our network has a maximum value that can be large if that category occurs many times. This makes Bayesian network learning challenging, as it introduces many free parameters. We address this problem by breaking each discrete random variable into a set of boolean variables. If a category C occurred at most M times in an input scene, we introduce M boolean variables $(C \geq 1), (C \geq 2), ...(C = M)$. Higher count boolean variables can only be true if lower counts are true, so we constrain our learning algorithm by requiring that each node $(C \geq a)$ is a parent of $(C \geq a + 1)$. After transforming our problem into a set of boolean variables, we combine the set of booleanization-enforced edges with the support constraints defined above and apply our structure learning algorithm. Figure 6.4 shows an example of this learning process.

Input enrichment Bayesian structure learning algorithms can result in undesired correlations when few unique examples are provided. To help combat this problem, we use a perturbation method to add variety to our replicated scenes. We form new scenes by selectively removing objects from each input scene. We define a *decay coefficient* for each category $e^{-\alpha R(C)}$ where $R(C)$ is the fraction of input scenes that contain C (the results in this chapter use $\alpha = 4$). Ubiquitous categories have small coefficients and are likely to be preserved, while infrequent categories have large coefficients and are more likely to be removed. Previous work has used similar perturbation methods to improve the robustness of learning algorithms for OCR and for speech recognition [Varga and Bunke 2003; Lawson et al. 2009].

6.6.2 Parent Support

To generate a static support hierarchy over a set of objects, we must also define a probabilistic model over possible parent-child static support connections. Let $\mathbf{Parents}(C)$ denote the set of categories that have been observed supporting a given category C in any input scene. We make the assumption that an object's support parent depends only on the existence of each category in this set. We build a *parent probability table* for each of the $2^{|\mathbf{Parents}(C)|}$ different states of existence of the parent categories. For each observation of C in the input scenes, we look at the existence of each possible parent and record the actual parent in the corresponding parent probability table. We use this table to define the probability of any given parent-child support relationship. The probability $\mathcal{T}(S)$ of a given support hierarchy arrangement in a scene S is taken as the product of all its constituent support relationships according to this table.

6.6.3 Final Model

Given the components above, we can define the final probability of a given support hierarchy as the product of its object occurrence model and parent support model:

$$\mathcal{O}(S) = \mathcal{B}(S)\mathcal{T}(S)$$

where $\mathcal{B}(S)$ is the probability our learned Bayesian network assigns to the set of objects in the scene, and $\mathcal{T}(S)$ is the probability of the parent-child support hierarchy given the set of objects in the scene.

6.7 Arrangement Model

We must use the input scenes to learn, for each object category, the kinds of surfaces it can be placed on, spatial locations where it can go, and directions it can face. This is a challenging task because objects can have many valid configurations which are determined by functional relationships with other objects. People can identify which relationships are salient, but an example-based algorithm must infer this from data.

Previous object arrangement models do not meet our goals. Yu et al. [2011] represents each position and orientation relationship with a single Gaussian. However, a single Gaussian does not account for multiple valid configurations. Also, users must specify which relationships are salient. Finally, their system does not address likely supporting surfaces for objects, since it arranged furniture on a flat floor.

We represent position and orientation relationships between all pairs of object categories using Gaussian mixture models. Gaussian mixtures can model multimodal distributions, thus handling objects with multiple valid configurations. We also describe a simple but effective heuristic for determining relationship saliency using object co-occurrence frequencies. Finally, we learn likely placement surfaces for objects using a simple Gaussian kernel density estimation approach.

6.7.1 Spatial Placement

From a set of input scenes, we learn a model of how objects are spatially arranged with respect to one another. Formally, we learn a probability distribution $\mathcal{P}_{C|C'}$ for every pair of categories C and C'. $\mathcal{P}_{C|C'}$ describes where category C objects tend to occur in the coordinate frame of category C' objects. It is a four-dimensional distribution over (x, y, z, θ), where $[x, y, z]$ defines an object's spatial position and θ defines its rotation about the normal of its supporting surface.

To learn these distributions, we first extract training data from the input scenes. To build robust models we need a large number of (x, y, z, θ) tuples, but the set of input scenes may be very small. Consequently, we extract many jittered (x, y, z, θ) samples from each object; 200 is more than enough, in our experience. We jitter object positions by $\alpha \sim \mathcal{N}(0, 25\,\text{cm} \cdot \mathbb{I}_3)$ and orientations by $\omega \sim \mathcal{N}(0, 5^\circ)$.

We represent $\mathcal{P}_{C|C'}$ with a Gaussian mixture model, trained using the expectation-maximization algorithm on the data described above. The number of Gaussians in the mixture is a latent variable; we choose the value that maximizes the Akaike information criterion [Akaike 1973]. This combats overfitting by favoring a low-complexity model unless the added complexity significantly boosts the model's likelihood. Figure 6.5 visualizes some of these learned models.

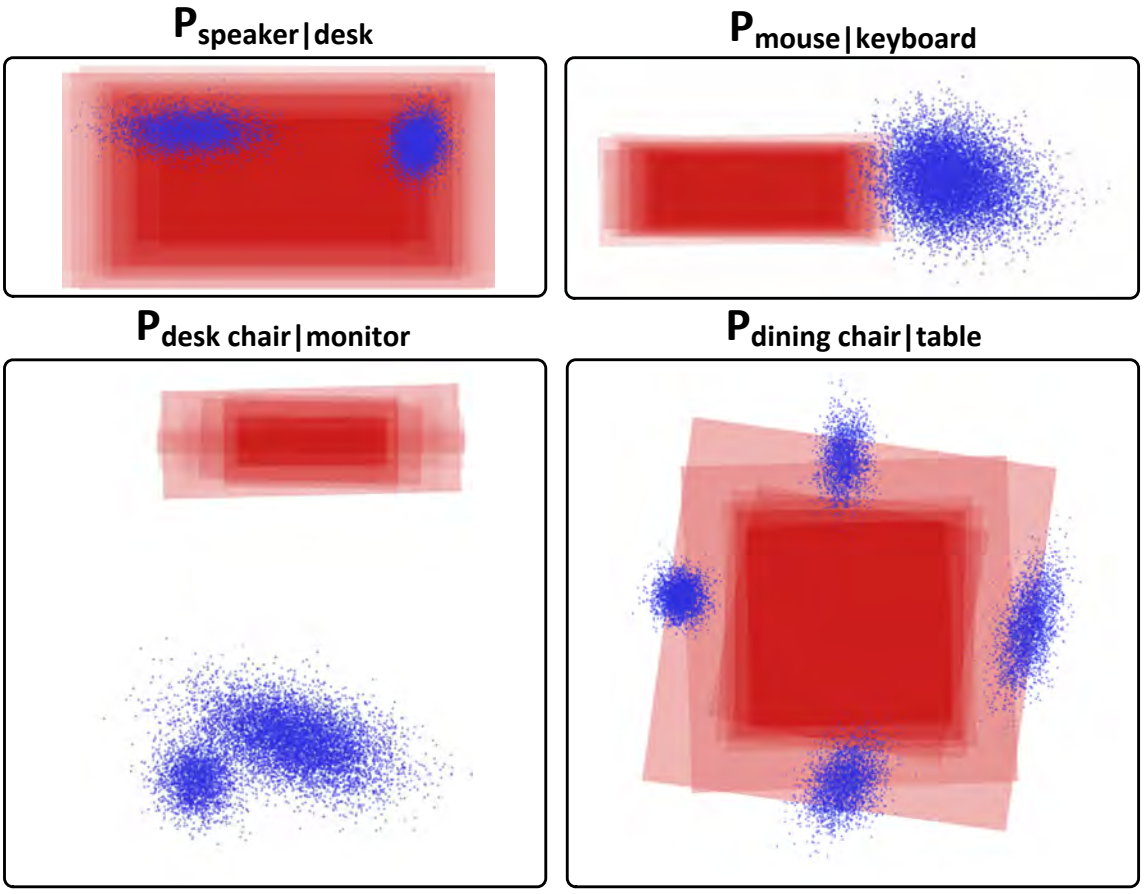

Figure 6.5: Pairwise spatial distributions for object arrangement. Distributions are visualized as points drawn from a learned mixture of Gaussians. The bounding boxes for objects in the category the model is conditioned on are shown in red. Points have been projected into the xy plane; the z and θ dimensions are not shown.

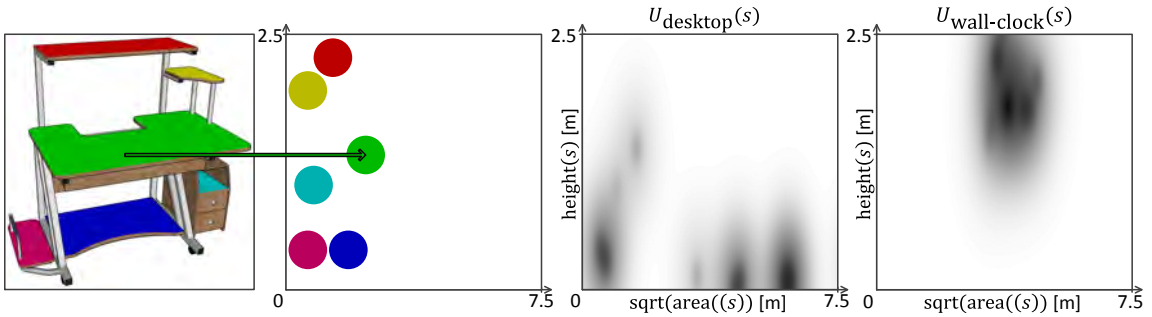

Figure 6.6: Left: Map from supporting surfaces on a computer desk onto the 2D surface placement probability density function. Right: probability density functions for desktop computers and wall clocks.

These distributions should not all be treated equally; we would like to emphasize 'reliable' relationships that occur more frequently in the input scenes. Thus, we also compute a set of pairwise weights $w_{C|C'}$, which we use in synthesis to indicate the relative importance of each distribution. $w_{C|C'} = f(C, C')^{30/n}$, where $f(C, C')$ is the frequency with which categories C and C' co-occur in the input scenes, and n is the number of input scenes. As desired, this weighting scheme emphasizes frequent relationships, where the definition of 'frequent' becomes more stringent with fewer input scenes.

6.7.2 Surface Placement

Different objects are supported by surfaces with different physical properties, often reflecting functional constraints: light switches and mice are found at particular elevations above the ground, and keyboards are unlikely to be placed inside a desk drawer. To capture this knowledge for each category, we observe how objects in that category are supported by other objects in a set of input scenes. For each support observation, we record the height above ground and the area of the supporting surface in a *surface descriptor*. We then treat these descriptors as independently sampled vectors in $\mathbb{R}^2$ and use kernel density estimation to construct a probability distribution over possible supporting surfaces.

First, we perform mesh segmentation to extract planar support surfaces for all

objects in the scene database that statically support other objects. We use a simple greedy region growth algorithm which handles non-manifold input models and robustly extracts planar and near-planar surfaces [Kalvin and Taylor 1996].

After segmentation, we compute surface descriptors for all segments that support any other objects. Each descriptor is a point $(\sqrt{area}, height) \in \mathbb{R}^2$, where the square root of area enforces consistent units between dimensions. We can estimate the underlying probability density function using any kernel density estimation method [Silverman 1986]. For each object category C, we approximate the function by summing Gaussian kernels centered at the $\mathbb{R}^2$ point for each surface descriptor. The Gaussian bandwidth is set using the normal distribution approximation: $h_C = 1.06\hat{\sigma}n_C^{(-1/5)}$, where $\hat{\sigma}$ is the standard deviation of all surface descriptor observations and n_C is the number of observations in the category. This approximates the variability over surface descriptor space using $\hat{\sigma}$ while enlarging the bandwidth to account for less certainty when fewer observations are available.

We call the estimated density functions $\mathcal{U}_C(s)$; for a surface s, this function returns the probability under our model that an object of category C should occur on s. Figure 6.6 visualizes this function for a few object categories.

6.7.3 Final Model

Given the components described above, we can define the final arrangement model as

$$\mathcal{A}(o, S) = \mathcal{U}_C(\text{surf}(o, S)) \cdot \sum_{o' \in S, o' \neq o} w_{C^o|C^{o'}} \cdot \mathcal{P}_{C^o|C^{o'}}(o)$$

where $\text{surf}(o, S)$ is the supporting surface of object o in scene S. Intuitively, this distribution combines the probability of o's surface placement with the probability of its spatial placement according to all other objects in the scene.

6.8 Synthesis

Synthesizing a new scene is straightforward given the models learned in the previous sections. First, we generate a new static support hierarchy that defines the objects in the scene and their parent-child relationships. Then, we determine a plausible spatial layout for the objects.

6.8.1 Static Support Hierarchy

The occurrence model $\mathcal{O}$ admits a very efficient sampling approach. We first use forward sampling to sample from the Bayesian network learned in Section 6.6, which generates a set of objects for the scene. To determine their parent-child support relationships, we independently assign parents to the objects in each category by sampling from the appropriate parent probability table. If the scene contains multiple instances of the sampled parent category, we choose an instance at random. This procedure samples a valid parent for each object, but the overall set of parent-child relationships may contain cycles. We use rejection sampling to generate a valid configuration, and repeatedly sampling parent-child relationships until an acyclic assignment is found.

Next, we assign a specific model to each object in the generated hierarchy. For each category, we compute a *consistency probability*: the frequency with which a category occurs two or more times in an input scene with all objects using the same model. We decide whether all objects from that category in our scene should use the same model according to this probability. If not, we choose models from the category at random.

This procedure can sometimes produce implausible scenes because some objects might be asked to support more objects than can fit comfortably on their supporting surfaces. To avoid this problem, for each object that is supporting children, we compute the total surface area (after projection into the plane of their contact surface) of all the supported children. If the total supported surface area is greater than the total surface area supported by an instance of that object in the scene database, we reject the scene and resample from the Bayesian network.

6.8.2 Object Layout

Given a synthesized support hierarchy, we must determine the exact position and orientation of its constituent objects. This is challenging because we want the generated layout to respect both guidelines implicitly expressed in the examples as well as physical constraints, such as objects not colliding. To arrange our objects, for each object o in the scene we define a density function that describes o's preferred configurations:

$$\mathcal{D}(o) = \mathcal{A}(o) \cdot \mathcal{L}(o) \cdot \mathcal{X}(o) \cdot \mathcal{H}(o)$$

$\mathcal{A}$ is the arrangement model from Section 6.7. The other terms are defined as follows:

Collision Penalty ($\mathcal{L}$) Objects in physically plausible arrangements do not interpenetrate. $\mathcal{L}(o) = 1$ if o does not collide with any other objects in the scene, and 0 otherwise.

Proximity Penalty ($\mathcal{X}$) The arrangement model $\mathcal{A}$ is often multimodal. The modes may represent multiple configurations for a single object (such as a left or right-handed computer mouse) or locations for multiple instances of the same type of object (such as stereo speakers). In the latter case, multiple instances should not concentrate around the same mode. We prevent this behavior with $\mathcal{X}(o) = 1 - G(d, \mu_d)$, where d is the distance from o to the nearest o' such that $C^{o'} = C^o$, and μ_d is the average distance between the instances of C^o observed in the examples.

Overhang Penalty ($\mathcal{H}$) In some cases, o's most likely location according to the terms defined thus far may leave it hanging off the edge of its supporting surface. This is not physically plausible. We address this problem with $\mathcal{H}(o)$, which returns the percentage of o's projected bounding box that is contained by its supporting surface.

The density function $\mathcal{D}$ we have defined typically has a small number of isolated modes and is near zero almost everywhere else. To find a good initial layout, our algorithm places each object one at a time by sampling from $\mathcal{D}$. Objects are placed by order of decreasing size, since large objects often constrain the placement of others.

Finally, the algorithm iteratively improves this initial layout via hill climbing. Each iteration makes a small perturbation to the object configurations, as in prior work on automatic furniture layout [Merrell et al. 2011; Yu et al. 2011]. Proposed perturbations are accepted if they increase the total *layout score*,

$$\sum_{o \in S} \mathcal{D}(o)$$

For all the results in this chapter, the algorithm stabilized within 100 iterations.

Figure 6.7: Effects of varying the $\lambda_{\mathbf{occur}}$ term. Left: a manually created input scene. Right: results generated at three different values of $\lambda_{\mathbf{occur}}$. Even a sparsely populated example can direct the algorithm to retrieve and incorporate relevant content from the database.

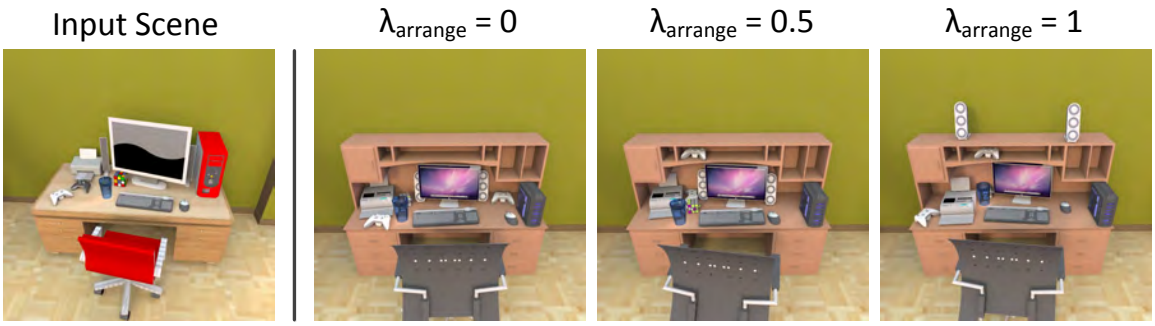

Figure 6.8: Effects of varying the $\lambda_{\mathbf{arrange}}$ term. Left: a manually created input scene. Right: results generated at three different values of $\lambda_{\mathbf{arrange}}$. With $\lambda_{\mathbf{arrange}} = 0$, the objects are only placed on the main desk surface, as in the desk from the input scene.

6.9 Results and Evaluation

To investigate the effectiveness of our method, we synthesize several types of scenes under varying input conditions. We also conducted an experiment in which people provided subjective judgments on the plausibility of synthesized scenes.

6.9.1 Synthesis Results

Figure 6.1 shows scenes synthesized from an input set of four computer desks using $\lambda_{\mathbf{occur}} = \lambda_{\mathbf{arrange}} = 0.5$. The scenes are similar in style to the original four examples without being repetitive. The synthesized scenes use a wide variety of models not found in the examples by drawing from the scene database. By using contextual categories and mixing information from a related set of scenes, our algorithm can insert plausible models not found in the input examples. For instance, the left two generated scenes both contain an alarm clock in a plausible location and orientation on the desk even though an alarm clock was not present in any of the input scenes.

In Figure 6.3, we compare the results of synthesizing using basic and contextual categories. In both cases we set $\lambda_{\mathbf{occur}}$ and $\lambda_{\mathbf{arrange}}$ to zero. When synthesizing using basic categories, our algorithm can only introduce variety by replacing each object with another model from the same basic category. Synthesizing using contextual categories can increase the variety of results by replacing objects in one basic category

with objects from related basic categories. For example, using basic categories the soda can on the desk will only be replaced with other soda cans. In contrast, using contextual categories we draw from a broader set of object types including bottles.

In Figure 6.7 we show how the $\lambda_{\mathbf{occur}}$ term controls mixing in the occurrence model. The input is a single example with only a couple of objects. When $\lambda_{\mathbf{occur}} = 0$ we can only replace objects with other objects of the same category, resulting in equally simple scenes. As we increase $\lambda_{\mathbf{occur}}$ we incorporate more diversity from similar scenes in the database. Relevant new object categories are integrated in consistent arrangements with respect to the existing objects. At $\lambda_{\mathbf{occur}} = 1$, the only contribution of the objects in the input scene is to determine the set of relevant scenes. In the case of the two desk scenes, a difference of only two objects had a significant impact on the type of environment that was synthesized. Note that the scenes in the database are not categorized, nor was a 'desired type of scene' label provided with the input example.

Figure 6.8 demonstrates how the $\lambda_{\mathbf{arrange}}$ term can be used to control mixing of the arrangement model. The single input scene uses a desk with only one supporting surface. Without mixing, the synthesized results do not use other valid supporting surfaces present on the chosen desk models, leading to cluttering of some surfaces and barrenness of others. By increasing the value of the $\lambda_{\mathbf{arrange}}$ term, we leverage observations of model placements from the database to appropriately arrange objects on all available supporting surfaces of the parent model, even if similar models were not used in the user provided examples.

6.9.2 Human Evaluation

To evaluate whether our system consistently generates plausible scenes we ran a online judgment study on three types of scenes: *Gamer Desks*, *Study Desks*, and *Dining Tables*. Our hypothesis is that a human observer will consider a significant fraction of the synthesized scenes to be as plausible as scenes created by hand.

For each of the three scene types, we created scenes under the following experimental conditions:

1. *Manually Created (Manual)*: We manually created four scenes; building each scene took approximately 15-20 minutes for an experienced user of our scene modeling tool.

2. *Synthesized (Synth)*: Scenes generated by our system using a mixed model, trained on the four *Manual* scenes plus relevant scenes retrieved from the database. $\lambda_{\mathbf{occur}} = \lambda_{\mathbf{arrange}} = 0.25$. We generated 50 scenes in this condition.

We then rendered images of all of these scenes. Within a given scene type, scenes were rendered against the same background and using the same camera configuration.

We recruited 30 participants via Amazon Mechanical Turk. Participants were required to be US residents to mitigate cultural influences on familiarity with different scene types. Through a web interface, each participant was shown 51 scene images: four from the *Manual* condition for each scene type, and 13 drawn at random from the *Synth* condition for each scene type. Participants were shown images one at a time in randomized order. Participants were asked to specify, on a 5-point Likert scale, the plausibility of the scene (1 = "Completely random, implausible scene," 3 = "Somewhat plausible scene," 5 = "Very plausible scene"). Rating a large set of more than 50 images helped participants calibrate their responses.

Figure 6.9 shows a summary of the ratings obtained through this experiment. Manual inspection of the data revealed no evidence of 'freeloading' or misbehaving workers, so we did not filter the data prior to anaylsis. Eliciting subjective responses from workers on Mechanical Turk is an inherently noisy process, but as expected, the responses for *Manual* scenes are concentrated toward the high end of the rating scale. In addition, the distribution of responses for *Synth* scenes closely matches the distribution for *Manual* scenes. *Manual* scenes were rated higher, on average, than *Synth* scenes for all scene types, and this difference is statistically significant (Mann-Whitney U test, $p < 0.05$). However, the difference is a small one: across all scene types, ratings for the top 80% of *Synth* scenes are *not* statistically distinguishable from ratings for *Manual* scenes. This result suggests that on average, at least 80% of synthesized scenes are not distinguishable from hand-created scenes by a casual

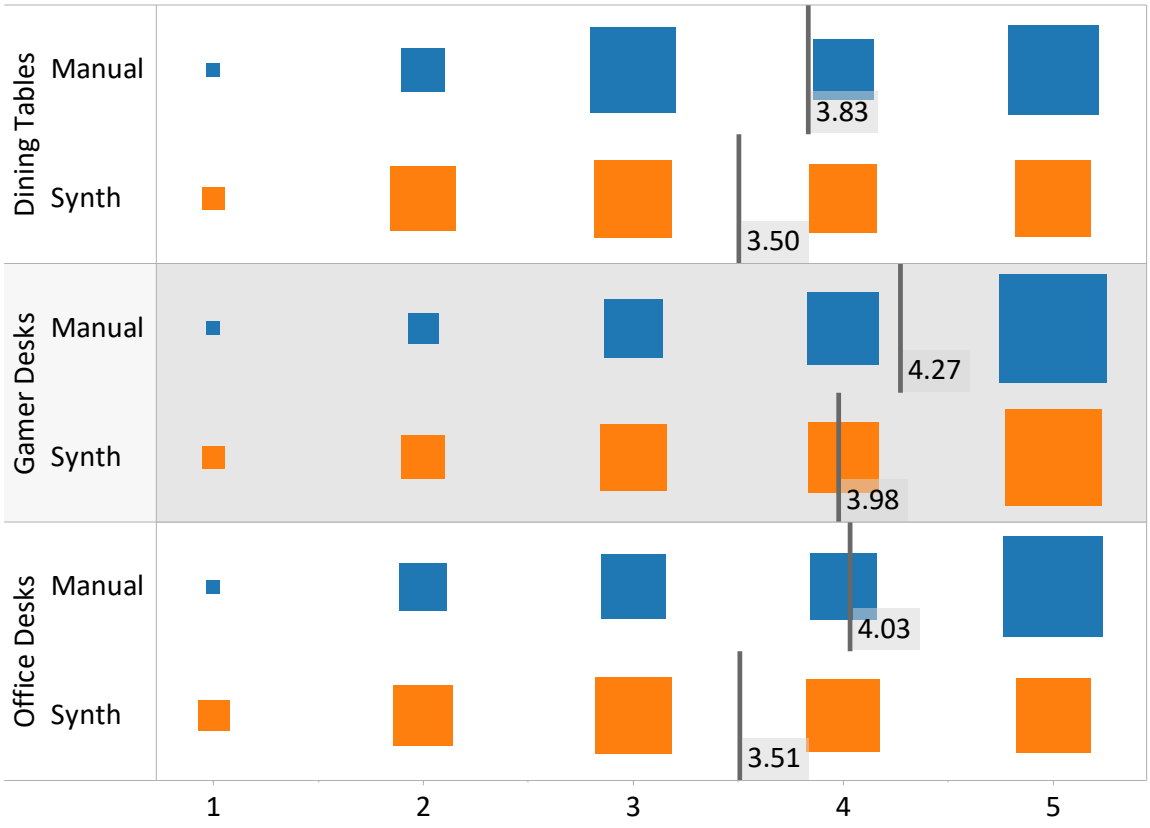

Figure 6.9: Results of an online judgment study in which people evaluated the plausibility of synthesized scenes (*Synth*) and manually created scenes (*Manual*) for three scene types. Each row is a histogram for one scene type/condition pair; the area of each box corresponds to the percentage of responses with that rating. Average ratings are plotted as vertical lines.

observer. It also far exceeded our initial goal of having one third of the synthesis results be of comparable plausibility to manually-created scenes.

6.9.3 Controllable Synthesis

A generative model is a very powerful tool that can be used in many different ways. Up until now, our focus has mostly been on unconstrained sampling. But with graphical models, it's possible to control the sampling process to match different conditions that the artist might want. In Figure 6.10 we show a simple example of controlled synthesis. This type of control makes it easy to perform decoration or clutter tasks that are otherwise really time-consuming, while still giving artists high-level control over the generated scene.

6.10 Chapter Summary

We have presented a data-driven method for learning a probabilistic model for 3D object arrangements from a small number of examples. We introduce the concept of contextual categories and present an algorithm for computing them from a database of scenes. The key to producing innovative variations given a very limited number of training examples is to make use of the design knowledge present in a much large database of scenes. Our approach is entirely example-driven and the artist is not required to specify any design principles beyond what they implicitly express in the examples. Our content generation system should be able to adapt to different environments and styles present in different scene databases, helping artists mitigate the problem of repeated content in these worlds, as well as decreasing the total time it takes to develop rich and detailed environments.

Generative models support many different modeling tools. In this chapter we have focused on synthesis, but many more tools can be developed such as content transfer or content rearrangement. One use that has been explored in related work on object modeling is data-driven part suggestion [Chaudhuri and Koltun 2010]. In this work, the existing object parts are taken as observed evidence and the generative model

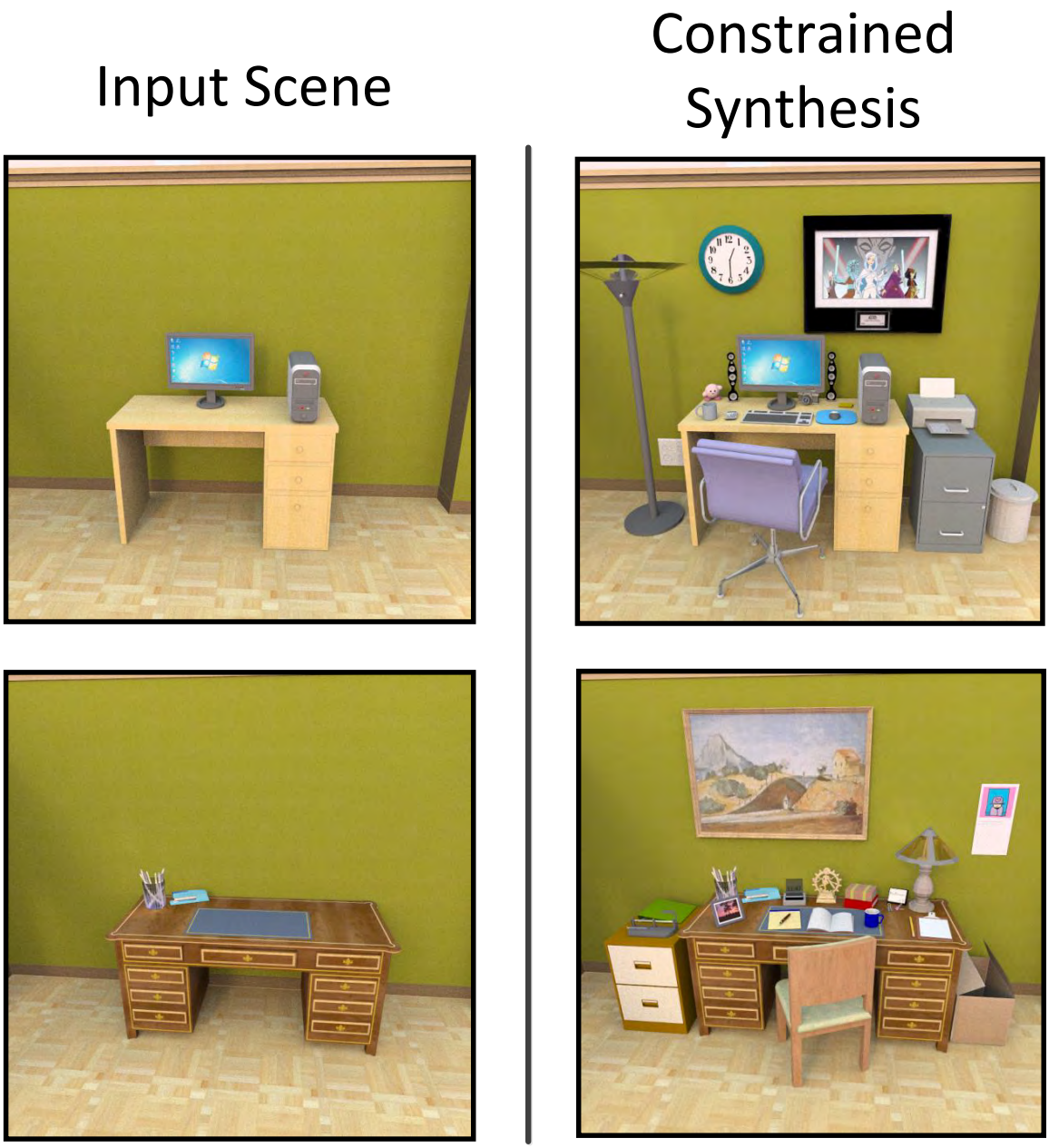

Figure 6.10: Constrained synthesis. The input scenes are shown on the left. On the right, we synthesize novel environments that are constrained to use the same objects at the same position and orientation.

is queried to rank all parts not present in the object, which are then presented to the users as suggested parts they might add. This part-based suggestion approach to object modeling can be easily transferred to the case of scene modeling, and is just one of many other possible design tasks that can be enabled by a generative model defined over the artifact being designed.

Chapter 7

Discussion

Any project that involves the creation of serious amounts of digital content is not accomplished by any one individual or with a single, all-encompassing tool. Virtual worlds, large-scale video games, and digital films are all created through the collaboration of a large number of individuals and with a unique and complex asset pipeline composed of many different tools. Some of these tools have become ubiquitous and shared across many pipelines, such as 3ds Max, and others are designed for a very specific task and are not intended to survive beyond the lifetime of a single project. The many tools we have developed in this dissertation, ranging from context-based model suggestions to unsupervised scene synthesis, are intended to fit into this workflow. Unfortunately, such tool ecosystems are very complex and not easily compared or evaluated, so we have instead evaluated our work in more isolated, task-specific settings. Nevertheless, we feel that there are many real applications such as virtual world design where scene modeling constitutes a sufficiently large portion of the content-creation pipeline to support the development of scene modeling technology. The tools that one day find their way into such pipelines may look quite different than the ones presented in this dissertation, but we believe that this dissertation offers real progress in this direction.

7.1 Data-driven vs. Rule-based Systems

We have placed considerable emphasis on a data-driven approach to tool development. We contrast this against systems that directly encode design principles; for example, one approach to furniture layout works by directly specifying anthropometric constraints such as "A nightstand should be located between 0 and 12 inches to the side of the bed" [Merrell et al. 2011]. Such tools have been shown to be effective at certain tasks, and in practice the most effective tools are likely to use ideas from both rule-based and data-driven approaches. We have chosen to focus on data-driven tools primarily for three reasons:

- **Domain generality** — Each content creation application and its corresponding tool chain is unique, and each application devotes different amounts of energy to different pipeline stages. Because it is not possible to accurately predict which domains are going to require our modeling tools, our tools need to be able to easily adapt to different domains and inputs with wildly different styles and properties. Data-driven approaches excel at this task: all that is required is a collection of data from the corresponding domain. Although a fully data-driven tool may make "childish" mistakes by failing to account for basic domain-specific knowledge, a data-driven tool can serve as an effective and adaptable foundation on top of which such expert knowledge can be encoded.

- **Ease-of-use** — While rule-based systems can be effective at capturing the most important design principles of a domain, they are also challenging to design because in many domains design principles may not exist or may be very difficult to specify. Rule-specification in complex environments often requires considerable trial and error, consulting domain experts, and skill at both rule programming and artistic evaluation. Data-driven systems have the potential to overcome these challenges: the input can be specified purely by example, without the need to codify and program domain rules. Data-driven tools can also easily adapt to changing patterns without the need to understand which rules need to be modified to effect the desired changes.

- **Data-mining and knowledge extraction** — Some patterns in modeling tasks are very clear from only a few examples: it takes only a few observations to learn that keyboards and mice often occur together. Yet many other patterns are seemingly subtle and it is hard to pick out the signal from the noise. This is true across many disciplines and data-mining tasks, yet one thread remains clear: patterns that are quite subtle given only a small amount of data often become easily delineated once the amount of available data grows large enough, even when using seemingly naïve algorithms. For 3D scenes, we are clearly not yet at the "big data" threshold: in comparison to other domains, LabelMe contains over 200,000 images, Google Images serves over a billion images, and Google N-grams contains over one trillion tokens. Despite this, we feel that now is the time to start developing the technology to learn scene design patterns from data; the tools can only improve as progressively more data becomes available.

7.2 Weaknesses of Data-driven Scene Modeling

Throughout the development of our tools, we have encountered several areas that were significant stumbling blocks for our data-centric approach. Perhaps the strongest weakness is cases where design patterns do not repeat, or for which very few examples exist or are needed. Consider an artist who wants to add ambiance to a 30^{th} century nursery room by designing a new board game for the children to play. If only one or two instances of this board game are going to exist in the world, it is not likely data-driven approaches are going to be able to effectively learn this design pattern. Seemingly relevant suggestions, such as pieces from Chess or Checkers, may prove undesired. Of course, if the game has repeating substructures, or there is a need for a large number of instances of the game, the example-based synthesis approach described in Chapter 6 could prove very useful to the artist assigned this task.

Another weakness we have observed in our data-driven modeling tools is architecture. It is very difficult to segment architecture into meaningful, interchangeable components. Analyzing architecture requires sophisticated structural understanding

and it is very challenging to connect different types of architecture together. Furthermore, most digital architectural constructions are designed to be visually pleasing but not structurally sound. It is common for architecture in games, whether of a building or a spaceship, to contain many flaws such as inter-penetrating geometry, as long as these areas are inaccessible or well hidden. For all these reasons, we have focused our tools on assisting with the arrangement of easily manipulated objects. Still, there are several situations, such as bookshelves that are integrated into the walls of a library, where the distinction between object arrangement and architecture becomes blurred.

7.3 Scene Modeling Software

As discussed in Chapter 2, current modeling software such as Google SketchUp creates scenes that are not ideal for data extraction. As scene modeling tasks and tools become more common, scene modeling programs will likely make more of an effort to help artists maintain a functional segmentation of their scene and better track the relationships between these objects. This will make the limited manual segmentation and tagging we performed unnecessary and is extremely useful for enabling user interaction in applications such as virtual worlds.

The benefits of having scenes exemplifying how specific models are used also suggest novel paradigms for scene modeling toolkits and 3D model collections such as Google 3D Warehouse. Artists uploading a new 3D model could be asked to contextualize it by placement at appropriate locations within scenes thus demonstrating common contact surfaces and nearby objects. Contextualized 3D models can be processed using methods such as ours and may be used for any number of scene modeling tasks. In some ways, these model-specific examples may prove even more useful than complete environments: users contextualizing a specific model such as a computer monitor are likely to focus on positioning it relative to aspects of the environment that are most relevant, mitigating the need to guess which object relationships are important to the design patterns being learned.

7.4 Future Work

There are many exciting avenues of research that can be built upon the work described in this dissertation. We look at three specific directions: alternative sources of design patterns, understanding higher-order structures in scenes, and understanding the function of objects.

Alternative data sources. We have focused all our tools on learning patterns from 3D scene databases such as Google 3D Warehouse or virtual worlds. However, the information found in these scenes can be found in many other datasets. While a photograph of a computer desk contains different types of information than a 3D scene, there is significant overlap: learning basic facts such as what categories of objects constitute a computer desk can be done just as easily from photographs as from scenes. Likewise, sentences such as "put the toaster on the kitchen counter" contain information about the expected relationships between categories. We first need to better understand the information overlap between text, images, and 3D scenes, and then we can look at ways to transfer knowledge between these different modalities. This is especially important given that at present 2D image datasets and text corpora are significantly larger and more developed than 3D scene databases.

Higher-order structure. Even the more advanced tools we present, such as the generative model described in Chapter 6, focus on learning patterns between individual objects. This is appropriate for many design patterns. However, many large environments eventually contain so many objects that the relationships are best understood at the level of collections of objects. Such hierarchical patterns are quite common: restaurants are composed of booths which are composed of place settings which are composed of forks, plates and knives. Taken to the extreme, worlds (both real and virtual) are composed of a very complex hierarchy of entities such as nations, cities, skyscrapers, floors, offices, filing cabinets, and stacks of paper. In this dissertation we have explored only the lowest levels of this hierarchy. Understanding how to effectively learn higher-level structure from data enables the development of more powerful modeling tools. Artists might manipulate intelligent collections of entities

such as dragging a "5.1 speaker system" from one living room to another, and expect the corresponding components to transfer to viable locations and orientations within the new environment. Many important tasks involved in creating virtual worlds, such as designing intelligent agents and animations for dynamic environments, can be seen in a new light when the world is expressed at a level that is richer and more complex than an unorganized collection of objects.

Understanding object function. Geometric information reflects a fairly restricted type of information about an object. Although it is sufficient to render the object, much of the information pertaining to how the object can be used or how the object interacts with other objects is lost. You can insert a DVD into a DVD player and expect it to decode and transmit the signal to a TV, but trying to learn these functional aspects of a DVD player from even the most precise geometric reconstruction of the DVD player is not feasible. Many of the important relationships between objects are related to their function — picture frames are located at or above eye level because they need to be easily visible, while TV remotes need to be within arms reach of the couch people are sitting on to watch the TV. Understanding object function can greatly improve the object categorization described in Chapter 6. Unfortunately, scene databases that encode only geometry and texture make it difficult to learn the function of objects. To acquire functional knowledge, we need to look to other sources. One one hand we might extract this knowledge from a text corpus that contains sentences such as "heat the pizza in the oven". On the other hand, we might learn object function by observing users interact with objects in virtual worlds such as Second Life or even the real world. The possibilities for innovation in all these areas are large.

Chapter 8

Conclusion

Few virtual environments achieve the richness in object density and diversity present in the real world. Numerous experiments with virtual reality demonstrate that environments that lack this complexity fail to convey a sense of immersion [Bowman and McMahan 2007]. In applications such as video games and films, this can weaken the overall experience including the narrative and character development because participants have trouble becoming fully engaged. This lack of detail may occur for any number of reasons: graphics hardware may be unable to achieve high framerates when too many objects are present, or the time and cost of constructing such detailed environments may be prohibitive. The goal of this dissertation is to work towards lowering the barrier to creating these environments in terms of time and cost.

To achieve this goal we have developed tools that improve the scene modeling process by learning design patterns from a scene database. In Chapter 2, we start by showing how to transform existing databases, such as Google 3D Warehouse, into a format that is amenable to knowledge extraction tasks. This includes filtering the scene graph to achieve good segmentation and propagating text information to acquire good semantic labels. We also developed a new scene database with very good segmentation and labeling which we use to test our tools. In Chapter 3, we provide our algorithm for estimating the similarity between two objects in isolation by comparing properties such as name, tags, geometry, and texture, which is an important subcomponent of the arrangement-comparison techniques we present. In Chapter 4,

we developed a tool that suggests models at a given location in an environment by using kernel density estimation over the set of object co-occurrences observed in the database: a model is a good suggestion if it is reinforced by many observations in the database. In Chapter 5, we show how to transform scene graphs into relationship graphs that encode semantic relationships between objects, and how to compare two such relationship graphs using a graph kernel. We use this relationship graph kernel to support scene retrieval and suggestion, which can help artists rapidly find useful subcomponents or inspiration for their current environment. On one hand, these search and suggestion tools demonstrate that it is already possible to achieve good results when using data-driven tools on existing, real-world datasets. On the other hand, these tools still adhere to the standard object-by-object method of scene construction.

To develop tools that understand scene structure at a higher level, we need a very reliable way to transfer knowledge about observations between scenes. Towards this end, in Chapter 6 we show how to partition all objects in a scene collection into *contextual categories* which are sets of objects whose neighborhoods are largely interchangeable and allow for better generality when learning from examples. Finally, we use these categories to develop a generative model for 3D scenes. These generative models are learned from examples and encode a sophisticated understanding of a specific design pattern which enables very powerful modeling tools. Our generative models require very few examples and are consequently easy for artists to build; they make use of a much larger collection of scenes to fill in information missing from the small number of examples. Using such a model artists can perform operations which intelligently operate on large collections of objects at once.

One common observation throughout our work has been that human perception is very fast when compared against the time it takes to model environments by hand. Although data-driven tools can sometimes fail to consistently respect certain design principles, the time it takes to perceptually evaluate a candidate scene is orders of magnitude faster than the time it takes to create a scene from scratch. This means that even tools where only a small percentage of the results are desired by the artist have the potential to be useful. This is a very compelling motivation

for all our tools: we want to transition scene modeling from consisting of a long series of time-consuming interface manipulation tasks into a series of easy visual selection tasks. Because most humans have a lifetime of experience processing visual information, this will not only make scene modeling faster but also more accessible to casual users who do not want to invest the considerable time needed to learn complex modeling interfaces. One day these types of tools will dominate the process of modeling large environments, and will only become better as progressively more data becomes available. Moving forward with our scene modeling research, our focus is on designing tools that both have a high degree of automation which makes them fast and easy to use, while also allowing for a high degree of control, which makes it easy to produce high quality environments.

Bibliography

Akaike, H. 1973. Information theory and an extension of the maximum likelihood principle. In *Second International Symposium on Information Theory*, vol. 1, 267–281.

Bach, F. R., Lanckriet, G. R. G., and Jordan, M. I. 2004. Multiple kernel learning, conic duality, and the smo algorithm. In *Proceedings of the 21st international conference on machine learning*, ACM, New York, NY, USA, ICML '04.

Bagley, S., and Altman, R. 1995. Characterizing the microenvironment surrounding protein sites. *Protein Science 4*, 4, 622–635.

Bokeloh, M., Wand, M., and Seidel, H.-P. 2010. A connection between partial symmetry and inverse procedural modeling. In *ACM SIGGRAPH 2010 papers*, ACM, New York, NY, USA, SIGGRAPH '10, 104:1–104:10.

Borgwardt, K., Ong, C., Schönauer, S., Vishwanathan, S., Smola, A., and Kriegel, H. 2005. Protein function prediction via graph kernels. *Bioinformatics 21*, 47–56.

Bowman, D., and McMahan, R. 2007. Virtual reality: how much immersion is enough? *Computer 40*, 7, 36–43.

Brants, T., Popat, A., Xu, P., Och, F., and Dean, J. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, Association for Computational Linguistics, 858–867.

Chaudhuri, S., and Koltun, V. 2010. Data-driven suggestions for creativity support in 3D modeling. *ACM Transactions on Graphics 29* (December), 183:1–183:10.

Chaudhuri, S., Kalogerakis, E., Guibas, L., and Koltun, V. 2011. Probabilistic reasoning for assembly-based 3D modeling. *ACM Transactions on Graphics 30*, 4.

Chen, X., Golovinskiy, A., and Funkhouser, T. 2009. A benchmark for 3D mesh segmentation. In *ACM SIGGRAPH 2009 papers*, ACM, 73.

Cristianini, N., and Shawe-Taylor, J. 2000. *An introduction to support vector machines: and other kernel-based learning methods*. Cambridge University Press.

Deng, J., Berg, A. C., and Fei-Fei, L. 2011. Hierarchical semantic indexing for large scale image retrieval. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 0*, 785–792.

Diez, Y., and Sellarès, J. A. 2007. Efficient colored point set matching under noise. In *Proceedings of the international conference on Computational science and its applications*, 26–40.

Feist, M. 2000. *On in and on: An investigation into the linguistic encoding of spatial scenes*. UMI, Ann Arbor, Michigan.

Fellbaum, C., et al. 1998. *WordNet: An electronic lexical database*. MIT press Cambridge, MA.

Fisher, M., and Hanrahan, P. 2010. Context-based search for 3D models. *ACM Transactions on Graphics 29* (December), 182:1–182:10.

Fisher, M., Savva, M., and Hanrahan, P. 2011. Characterizing structural relationships in scenes using graph kernels. In *ACM SIGGRAPH 2011 papers*, SIGGRAPH '11.

Fisher, M., Ritchie, D., Savva, M., Funkhouser, T., and Hanrahan, P. 2012. Example-based synthesis of 3d object arrangements. *ACM Transactions on Graphics (TOG) 31*, 6, 135.

Funkhouser, T., Kazhdan, M., Shilane, P., Min, P., Kiefer, W., Tal, A., Rusinkiewicz, S., and Dobkin, D. 2004. Modeling by example. In *ACM Transactions on Graphics*, vol. 23, ACM, 652–663.

Gal, R., and Cohen-Or, D. 2006. Salient geometric features for partial shape matching and similarity. *ACM Transactions on Graphics 25*, 1, 150.

Galleguillos, C., Rabinovich, A., and Belongie, S. 2008. Object categorization using co-occurrence, location and appearance. In *CVPR*, 1–8.

Gartner, T., Flach, P., and Wrobel, S. 2003. On graph kernels: Hardness results and efficient alternatives. In *Proceedings of the 16th Annual Conference on Learning Theory*, 129–143.

Goldfeder, C., and Allen, P. 2008. Autotagging to improve text search for 3D models. In *JCDL '08: Proceedings of the 8th ACM/IEEE-CS Joint Conference on Digital Libraries*, ACM, New York, NY, USA, 355–358.

Gupta, A., Efros, A., and Hebert, M. 2010. Blocks world revisited: Image understanding using qualitative geometry and mechanics. *Computer Vision–ECCV*, 482–496.

Habegger, B., and Debarbieux, D. 2006. Integrating Data from the Web by Machine-Learning Tree-Pattern Queries. *On the Move to Meaningful Internet Systems*, 941–948.

Harchaoui, Z., and Bach, F. 2007. Image classification with segmentation graph kernels. In *CVPR*, 1–8.

Hays, J., and Efros, A. A. 2007. Scene completion using millions of photographs. *ACM Transactions on Graphics (SIGGRAPH 2007) 26*, 3.

Henrich, A., and Morgenroth, K. 2003. Supporting collaborative software development by context-aware information retrieval facilities. In *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, IEEE Computer Society, Washington, DC, USA, DEXA '03, 249–.

Horn, B. 1984. Extended gaussian images. *Proceedings of the IEEE 72*, 12, 1671–1686.

Iyer, N., Jayanti, S., Lou, K., Kalyanaraman, Y., and Ramani, K. 2005. Three-dimensional shape searching: state-of-the-art review and future trends. *Computer-Aided Design 37*, 5, 509–530.

Kalogerakis, E., Chaudhuri, S., Koller, D., and Koltun, V. 2012. A probabilistic model for component-based shape synthesis. *ACM Transactions on Graphics 31*, 4.

Kalvin, A., and Taylor, R. 1996. Superfaces: Polygonal mesh simplification with bounded error. *Computer Graphics and Applications, IEEE 16*, 3, 64–77.

Kashima, H., Tsuda, K., and Inokuchi, A. 2004. Kernels for graphs. *Kernel methods in computational biology*, 155–170.

Kazhdan, M., Funkhouser, T., and Rusinkiewicz, S. 2003. Rotation invariant spherical harmonic representation of 3D shape descriptors. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Eurographics Association, 156–164.

Kazhdan, M. 2007. An approximate and efficient method for optimal rotation alignment of 3D models. *IEEE Transactions on Pattern Analysis and Machine Intelligence 29*, 7, 1221 –1229.

Kokare, M., Chatterji, B., and Biswas, P. 2003. Comparison of similarity metrics for texture image retrieval. In *TENCON 2003. Conference on convergent technologies for Asia-Pacific region*, vol. 2, IEEE, 571–575.

Kuhn, H. 1955. The hungarian method for the assignment problem. *Naval research logistics quarterly 2*, 1-2, 83–97.

Lawson, A., Linderman, M., Leonard, M., Stauffer, A., Pokines, B., and Carlin, M. 2009. Perturbation and pitch normalization as enhancements

to speaker recognition. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, IEEE, 4533–4536.

Lazebnik, S., Schmid, C., and Ponce, J. 2006. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR*, vol. 2, 2169 – 2178.

Lee, B., Srivastava, S., Kumar, R., Brafman, R., and Klemmer, S. 2010. Designing with interactive example galleries. In *Proceedings of the 28th international conference on Human factors in computing systems*, ACM, 2257–2266.

Levandowsky, M., and Winter, D. 1971. Distance between sets. *Nature 234*, 5323, 34–35.

Mahé, P., Ueda, N., Akutsu, T., Perret, J.-L., and Vert, J.-P. 2004. Extensions of marginalized graph kernels. In *21st international conference on machine learning*, ACM, New York, NY, USA, ICML '04, 70–77.

Malisiewicz, T., and Efros, A. A. 2009. Beyond categories: The visual memex model for reasoning about object relationships. In *NIPS*.

Medin, D., and Schaffer, M. 1978. Context theory of classification learning. *Psychological review 85*, 3, 207–238.

Merrell, P., Schkufza, E., and Koltun, V. 2010. Computer-generated residential building layouts. *ACM Transactions on Graphics 29* (December), 181:1–181:12.

Merrell, P., Schkufza, E., Li, Z., Agrawala, M., and Koltun, V. 2011. Interactive furniture layout using interior design guidelines. In *ACM SIGGRAPH 2011 papers*, 87:1–87:10.

Min, P. 2004. A 3D model search engine. Princeton University.

Murtagh, F. 1984. Complexities of hierarchic clustering algorithms: state of the art. *Computational Statistics Quarterly 1*, 2, 101–113.

Neisser, U. 1967. *Cognitive Psychology*. Appleton-Century-Crofts, New York.

Novotni, M., and Klein, R. 2003. 3D Zernike descriptors for content based shape retrieval. In *8th ACM symposium on solid modeling and applications*, ACM, 216–225.

Papadakis, P., Pratikakis, I., Trafalis, T., Theoharis, T., and Perantonis, S. 2008. Relevance feedback in content-based 3D object retrieval: A comparative study. *Computer-Aided Design and Applications Journal 5*, 5, 753–763.

Paraboschi, L., Biasotti, S., and Falcidieno, B. 2007. 3D scene comparison using topological graphs. *Eurographics Italian Chapter, Trento (Italy)*, 87–93.

Platt, J. 1999. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods*, MIT press, 185–208.

Rabinovich, A., Vedaldi, A., Galleguillos, C., Wiewiora, E., and Belongie, S. 2007. Objects in context. *IEEE 11th International Conference on Computer Vision*, 1–8.

Ramon, J., and Gärtner, T. 2003. Expressivity versus efficiency of graph kernels. In *1st International Workshop on Mining Graphs, Trees and Sequences*, 65–74.

Rosch, E. 1973. Natural categories. *Cognitive psychology 4*, 3, 328–350.

Russell, B., and Torralba, A. 2009. Building a database of 3d scenes from user annotations. *CVPR*, 2711–2718.

Russell, B., Torralba, A., Murphy, K., and Freeman, W. 2008. LabelMe: a database and web-based tool for image annotation. *International Journal of Computer Vision 77*, 1, 157–173.

Salton, G., and Buckley, C. 1988. Term-weighting approaches in automatic text retrieval. In *Information Processing and Management*, 513–523.

Shawe-Taylor, J., and Cristianini, N. 2004. *Kernel methods for pattern analysis*. Cambridge University Press.

Shechtman, E., and Irani, M. 2007. Matching local self-similarities across images and videos. In *CVPR*, 1–8.

Shilane, P., Min, P., Kazhdan, M., and Funkhouser, T. 2004. The Princeton shape benchmark. In *Shape Modeling International*.

Silverman, B. 1986. *Density estimation for statistics and data analysis*, vol. 26. Chapman & Hall/CRC.

Steinbach, M., Karypis, G., and Kumar, V. 2000. A comparison of document clustering techniques. In *KDD workshop on text mining*, vol. 400, 525–526.

Sundar, H., Silver, D., Gagvani, N., and Dickinson, S. 2003. Skeleton based shape matching and retrieval.

Torralba, A., 2010. The context challenge. `http://web.mit.edu/torralba/www/carsAndFacesInContext.html`.

Torsello, A., Albarelli, A., and Pelillo, M. 2007. Matching relational structures using the edge-association graph. In *14th International Conference on Image Analysis and Processing*, 775 –780.

Tung, T., and Schmitt, F. 2005. The augmented multiresolution Reeb graph approach for content-based retrieval of 3D shapes. *International Journal of Shape Modeling 11*, 1, 91–120.

van Rijsbergen, C., Robertson, S., and Porter, M. 1980. New models in probabilistic information retrieval. *British Library Research and Development Report No. 5587*.

Varga, T., and Bunke, H. 2003. Generation of synthetic training data for an hmm-based handwriting recognition system. In *Document Analysis and Recognition, 2003. Proceedings. Seventh International Conference on*, IEEE, 618–622.

Wolfson, H., and Rigoutsos, I. 1997. Geometric hashing: an overview. *Computational Science Engineering 4*, 4, 10 –21.

Xu, Y., and Kemp, C. 2010. Constructing spatial concepts from universal primitives. *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, 1–6.

Xu, K., Stewart, J., and Fiume, E. 2002. Constraint-based automatic placement for scene composition. In *Graphics Interface 2002*, 25–34.

Xu, K., Zhang, H., Cohen-Or, D., and Chen, B. 2012. Fit and diverse: Set evolution for inspiring 3d shape galleries. *ACM Transactions on Graphics 31*, 4.

Yeh, Y.-T., Yang, L., Watson, M., Goodman, N. D., and Hanrahan, P. 2012. Synthesizing open worlds with constraints using locally annealed reversible jump mcmc. *ACM Transactions on Graphics 31*, 4.

Yu, L.-F., Yeung, S.-K., Tang, C.-K., Terzopoulos, D., Chan, T. F., and Osher, S. J. 2011. Make it home: automatic optimization of furniture arrangement. In *ACM SIGGRAPH 2011 papers*, 86:1–86:12.

Matthew Fisher

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Pat Hanrahan) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Scott Klemmer)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Barbara Tversky)

Approved for the University Committee on Graduate Studies
