AI Backed Crowd-Sourced How-to Manuals

by

Daniel J. Scarafoni

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Master's of Science

Supervised by

Professor Philip Guo

Department of Computer Science
Arts, Science & Engineering
Edmund A. Hajim School of Engineering & Applied Sciences

University of Rochester
Rochester, New York

2014

Abstract

So called unskilled jobs such as deli clerks and table waiting in fact require a number of skills to do well; these skills are often learned on the job through experience and mentorship situations. The process could be simplified with instruction written by professionals. Canned Mentorship is a system which allows small groups of professionals to create comprehensive instructions for day-to-day tasks in their jobs for use by newer staff, reducing the need for supervision. This program uses a front-end user interface which allows users to view the instructions as they are being formed and vote on editions and revisions. A back-end server collects the natural language data and parses it to coordinate the groups ideas. The end result is a list of instructions describing a particular task.

Table of Contents

1 Introduction

Crowdsourcing has shown to be an effective means of solving tasks that are computationally difficult, or otherwise impractical, for artificial intelligence programs to solve. (**?**). There are many instances, of course,where humans are more effective than machines and vice-versa. To pick the most obvious example: humans are more efficient at qualitative, ambiguous tasks, while artificial intelligence (AI) agents tend to be better at more quantitative, direct tasks. Naturally, most tasks do not fall cleanly into one category or another; there are many tasks which cannot be done perfectly by human or AI agents. For this reason, the merging of human and AI is necessary for the completion of a number of tasks.

Consider, for example, the task of building an instruction manual for a common profession. If a restaurant wanted to create a manual for a number of everyday tasks in the restaurant in order to train new employees quickly.

ADD SOMETHING ABOUT THIS BEING ABOUT SKILLED USERS

It would use data gathered from existing employees to solve this task, as the ambiguous and qualitative data of procedural tasks such making food and settling tables is far beyond the current capabilities of artificial intelligence. One experienced employee could be delegated to the task-instruction creation, but there are a number of issues with this method. First, an over-reliance on one individual leaves the system vulnerable to error. An individual may make mistakes when writing instructions. The only solution to this problem would be to have other users check the first users work, which automatically means that multiple users will be needed regardless. Second, many task, especially the qualitative, ambiguous ones previously mentioned, can be done in multiple ways. While the differences may not necessarily be erroneous in nature, they can pose problems. Certain delis, for example, have managers with different policies on the quality of meat used for sandwiches. If a trainee is taught the preference of one manager and not the other, issues can arise if the employee switches managers. Third, there are many instances where individuals have been shown to be slower than crowds (Lasecki and Bigham, 2013; **?**). It it likely that crowd-sourced users will also be more efficient than single users in this domain.

One major issue with gathering instructions from users is dealing with redundancy. Because many people have similar notions of instructions for various tasks, users tend to give similar instructions for a given step. This leads to redundancy. For example, if a group of 3 users were asked to write the first step for making a peanut-butter and jelly sandwich, the users might respond with

1. get two slices of bread
2. get two slices of bread, a knife, peanut butter, and some jelly
3. get the ingredients.

It is obvious in this situation that because suggestion 3 is described completely (and more thoroughly) by suggestion 2, that it is redundant and thus can be removed.

Failure to do so can impede the crowd-sourcing process. First, it introduces a form of noise into the system. With multiple identical sentences, if users are asked to vote on the suggestions, the voting process may be subverted. Consider the same situation as the last example, but the three inputs are

1. get two slices of bread
2. get the ingredients
3. get the ingredients.

If two users believe that option 2 (or equivalently, option 3) is correct, they could vote for either. If the third user votes for option 1, then there is no clear majority from voting, even though the consensus is clear.

This problem ultimately is an issue of natural language processing, and is unique in that it deals with very short documents in a very small corpus. A quick and efficient method of removing and/or minimizing redundancies in the context of this problem is essential to the completion of the task.

In total, the task of building a set of instructions (or a "how-to" manual) efficiently from a group of users requires efficient crowd-sourcing techniques as well as sufficiently sophisticated AI to augment this process. Although there has been much research on using crowds to answer single questions, there has not been much work done in the next level of abstraction: having users answer lists of questions (i.e. instructions) (Lasecki *et al.*, 2013; Bigham *et al.*, 2010).

The remainder of this paper will be organized as follows:

- the Background section will discuss previous work done in both crowdsourcing and redundant sentence removal, as well the unresearched areas where this project will make its contributions
- The Methodology section will describe the Canned Mentorship system in detail, both at an algorithmic and implementation level. It will also descirbe the parameters tested and the experimental setups.
- The results section will describe the experimental results of the study, and discuss the any anomalies or other observations encountered during the testing process.

2 Background

2.1 Crowdsourcing

Crowdsourcing has shown itself to be a very effective computational resource. Formally, it is defined as

> the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call. (Brabham, 2008).

2.1.1 Current Scope and Usage of Crowdsourcing

Though crowdsourcing covers a very broad range of activities and fields, it is possible to define separate catagories. Brabham defined three separate categories:

1. crowdfunding
2. crowd labor
3. crowd research.

Crowdfunding is the use of crowd-based resources to fund projects. Websites such as gofundme.com, kickstarter, and patreon.com are all crowdfunding website which allow elicit funds from users. This method is particularly useful for freelance artists (Brabham, 2008).

Crowd labor utilizes the crowd for computation and other work-like tasks. Generally, this involves the users doing low effort tasks for low pay. In many situations, this can result in a much cheaper product than non crowdsourced, professional settings. Crowdsourcing is also often used for tasks which are prohibitively difficult or impossible with current computational means.

For example, the stock photo website iStockphoto collects images uploaded by users, and pays them a small compensation for their effort. This website is able to lease the rights to these images for $1, which undercuts professional photographers' prices by over 90%, Crowd labor also extends to a number of other fields, including industrial research (Howe, 2006).

2.1.2 Effective Crowdsourcing

3 Methodology

Our project began by creating a small webapp which coordinated the instruction generating process

4 Results

The system has shown remarkable progress since its first creation. The Blender Python environment has demonstrated its ability to act as an effective programming tool for creating and running 3D simulations in a quick and accurate manner. Our small library of functions and predications has given us a grounding from which a more extensible and complex system can be built.

Our project has demonstrated the ability to accurately place objects and query scenes based on given predications. Experiments in the past have demonstrated that the IMS can, given input objects and relations, generate appropriate scenes. The following three objects were used for the majority of the tests and examples for this project:

- ball "A"
- snowman "B"
- person "C"

these three objects were chosen because of their increasing number of mesh objects. The ball only has one mesh. The snowman has three meshes (the top, bottom, and middle spheres). The person, however, has fifteen meshes. This spectrum allows for a broad analysis of the different sized objects, which is important as certain methods' run-time is dependent on the number of mesh objects in the specified entity.

Figure 4.1 provides a visualization of the three objects used. Note, however, that in the majority of testing situations, only person and ball were used. With these objects,

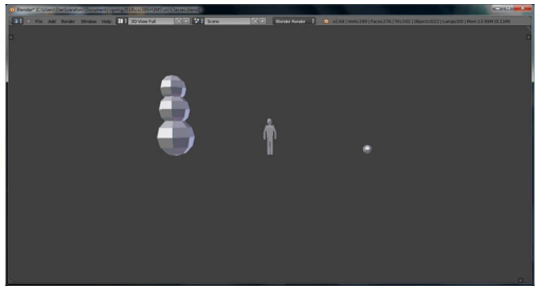

Figure 4.1: a visualization of the three objects used for basic testing. From left to right: the snowman, the person, and the ball. Entities are shown to scale.

the program will correctly an appropriate scene which is consistent with the constraints. The scene for this example is seen below in figure 4.2.

4.1 Binary Query Tests

The first set of tests examined the IMS's ability to place and query based on the simple binary predicate relations. For these tests, only two objects were needed, and as such the person, "B" and ball, "A" were implemented (as they utilized the most and least meshes respectively). Each predicate was tested ten times. In each test, the system placed the two objects were placed under the constraint <predicate >(A,B).

An objects focality determines, in a loose sense, its importance in a scene. For the intents and purposes of this experimentation, a non-focal object is placed around a focal object. For example, for the placement of "near(A,B)", if object "A" is focal, then it is left in its original location, and object "B" is placed around it. The focality was varied so that testing would not improperly bias certain conditions in the placement function.

Five tests were done with the person as the *focal object* (the stationary one), and five were done with the ball as the *focal object*. This gave an accurate view of the run-time of the predicates on objects of varying complexity. These ten tests were repeated not only for every predicate, but for varying positions as well.

The non-focal object was placed 1, 2, and 5 blender units on the x axis away from the *focal object*. The *focal object* was constrained to the origin. This was done to tests the predicates under multiple degrees of satisfiability. For predicates such as "above"

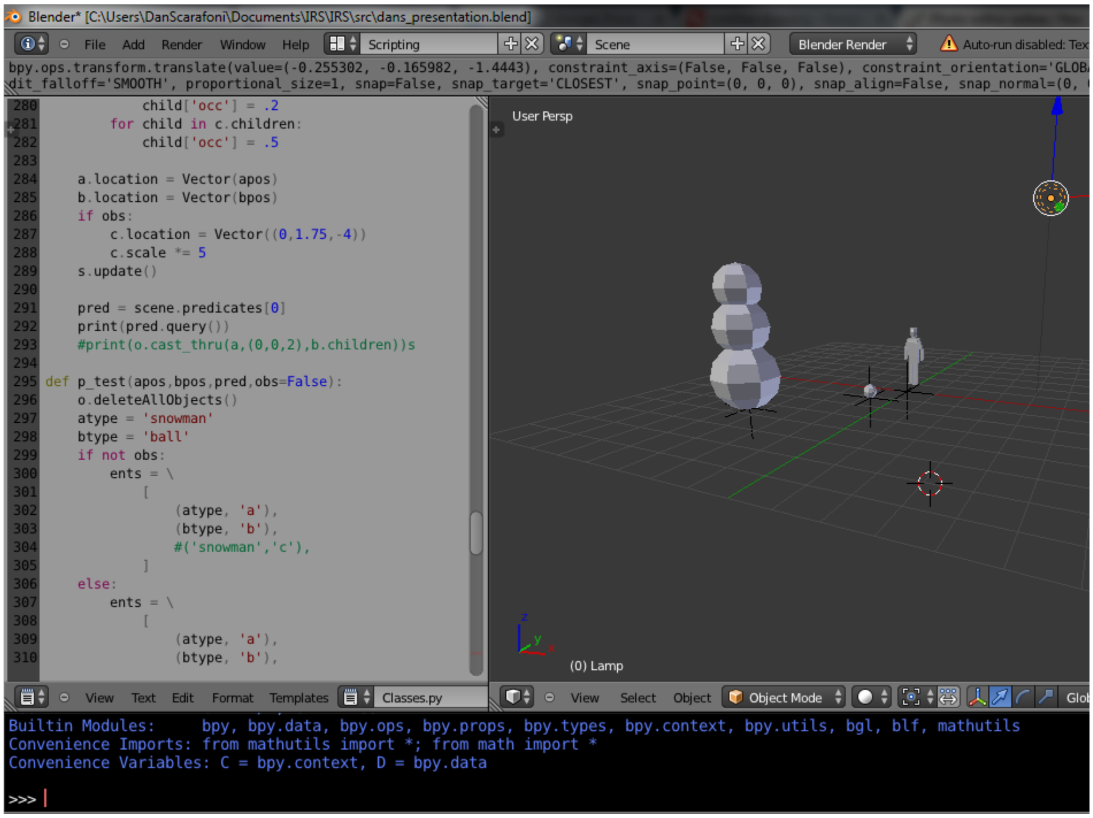

Figure 4.2: Given the above predicates, the IMS is able to generate an appropriate scene.

and "under" the non-focal object was also placed above and below the *focal object* respectively. If this was not done, then neither of these predicates would be satisfiable in any tests.

Because of the increased complexity of the "canSee" predicate, customized testing is necessary. To accurately gauge the efficacy of this predicate, the following scenario was used: a ball was placed in the coordinate origin. Around it, the person model was placed at one of eight locations: five blender units above or below on both the x and y axis. An obscuring snowman was placed at the coordinates `(3,0,0)` so that the line of sight between the person and the ball would be obscured when the person was placed in front of the snowman. The snowman was given an occluding factor of 0.5. This represents the somewhat absurd notion of a translucent snowman. This scenario was relevant, however, in that it provided a large, multi-object, obscuring entity which

predicate	distance	query	time: average	time: standard deviation
`near`	1	1	0.0142	0.005
	2	1	0.0186	0.006
	5	0.365	0.0165	0.006
`under`	1	0	0.0195	0.0
	2	0	0.0196	0.004
	5	0	0.0196	0.000
`in`	1	1	0.0235	0.005
	2	0	0.0196	0.000
	5	0	0.0227	0.005
`inside`	1	0.451	0.327	0.008
	2	0	0.3383	0.016
	5	0	0.3524	0.017
`above`	1	0	0.0255	0.007
	2	0	0.0264	0.009
	5	0	0.0257	0.007
`isTouching`	1	0.019	0.0057	0.004
	2	0	0.0066	0.006
	5	0	0.0079	0.006
`canSee`	1	1	0.0223	0.007
	2	1	0.0182	0.003
	5	1	0.0142	0.007

Table 4.1: Query Results Over Varying Predicates and Distances.

effected vision in the scene. The predicate “canSee(ball,person)” was then queried at each location ten times. The measurements of the query and the averages and standard deviations of the times were recorded. The results of this can be seen in Figure 4.3

As can be see in figure 4.3, all spots have perfectly clear sight of the ball in the center, except for the one which is blocked by the snowman, which returns a reduced sight capacity. This demonstrates the vision capabilities of the system, and the degree to which the occlusion feature accurately simulates reality.

4.2 Binary Placement Testing

Placement testing began with the placement of objects in a scene with one predication. All of the predications were tested, and the person and ball objects were used, so that the predications all took the form:

```
<predicate>(A,B)
```

Every predication was tested twenty times, ten times with each object as the focal object.

The placement function's rejection sampling was also tested for efficiency. Tests were done with no threshold (no rejection sampling), low threshold (placements had to return a non-zero query value), and high threshold (placements had to return a query satisfaction of at least 0.25). It was noted that for threshold values greater than this, the program failed to terminate in reasonable time (less than 5 minutes).

Figure 4.4 shows the results of the first round of tests, placement with no rejection sampling.

The results give a good indication of the satisfiability of the placement areas returned by the placement functions. Because the areas where the `near`, `in`, `isTouching`, and `canSee` predications hold are represented accurately (in a scene with two objects) by the valid placement returned by the placement function. The remaining predicates, however, are not very well represented by these placement areas, and as such their average satisfaction is much lower.

The test for a low threshold are represented in figure 4.5.

As can be seen by these results, there is a small but noticeable trade-off for all predications. As before there is a high amount of variance for predications whose placement areas do not match their query-satisfaction areas closely. For most, the difference between the results of this and the previous experiment are not significant, with the variations within one standard deviation.

The one exception with this is the time for the inside predicate. The average time for this predicate is significantly longer than the others, so much so that it alters the scale of the graph.This provides evidence that the querying function for this predication is notedly more inefficient than the others.

The final tests were done with a high threshold, these results are shown in figure 4.6. In this graph we see more noted improvements within the code, particularly for the predications with higher variation in previous tests. The trade-off for most predicates was expected. However, as before, the inside predication's query time was increased considerably. It is reasonable to state that this predication has the longest run-time of any in the library, and that it is likely the limiting factor that prevents the system from terminating in reasonable time when a high rejection threshold is used.

4.3 Full Scene Simulation

Because the most important part of the project was to simulate the story scene featuring a boy and a girl seeing a bird's nest this, this scene became the staple for testing scenes with multiple objects and predications. More specifically, this scene consisted of the following information:

- Entities
  - person "A"
  - tree "B"
  - nest "C"
  - egg "D"
- Predications
  - under(A,B)
  - canSee(A,C)
  - in(D,C)
  - in(C,B)

This corresponds to a simplified version of the story in the scene: one in which there is an egg in a nest (which is itself in a tree) and a child under the tree who can see the nest.

Once the scene was constructed, the scene was queried to see if the person could both see the nest and the egg. Note that the person was placed in the scene *without* the "canSee(A,D)" predication, and as such the scene was only configured such that the person could see the nest. Because of this, and because the person was to be placed under the tree, it was highly unlikely that the person should be able to see the egg in the nest. Since the egg was completely on top of the nest, the person could only look up from below, and the nest is a completely opaque object (having occlusion of -1).

canSee egg	canSee nest	time
0	0.6947030267	137.34
0	0.998120801	20.45
0	0.9678805377	0.45
0	0.8965050001	13.61
0	0.7376654109	710.13
0	0.9193492216	229.36
0.0940467865	0.8770202952	2.84
0	0.5460023627	11.99
0	0.9322889931	36.05
0	0.9973411794	23.23
averages		
0.0104496429	0.874685978	116.4566666667
standard deviation		
0.0297402052	0.1493400827	220.5594981582

Table 4.2: The testing results for the story placement scene.

The results of the test, including the placement and subsequent querying, are displayed in figure 4.2. In all but one test, the system was able to correctly place the objects in the scene in such a way that the person was able to see the nest, but not the egg inside. This indicates a very high rate of success for the IMS's predication placement and querying setup. It should also be noted that variability in the visibility of the nest is a result of the nest being placed inside the tree canopy, which has an opacity of 0.5.

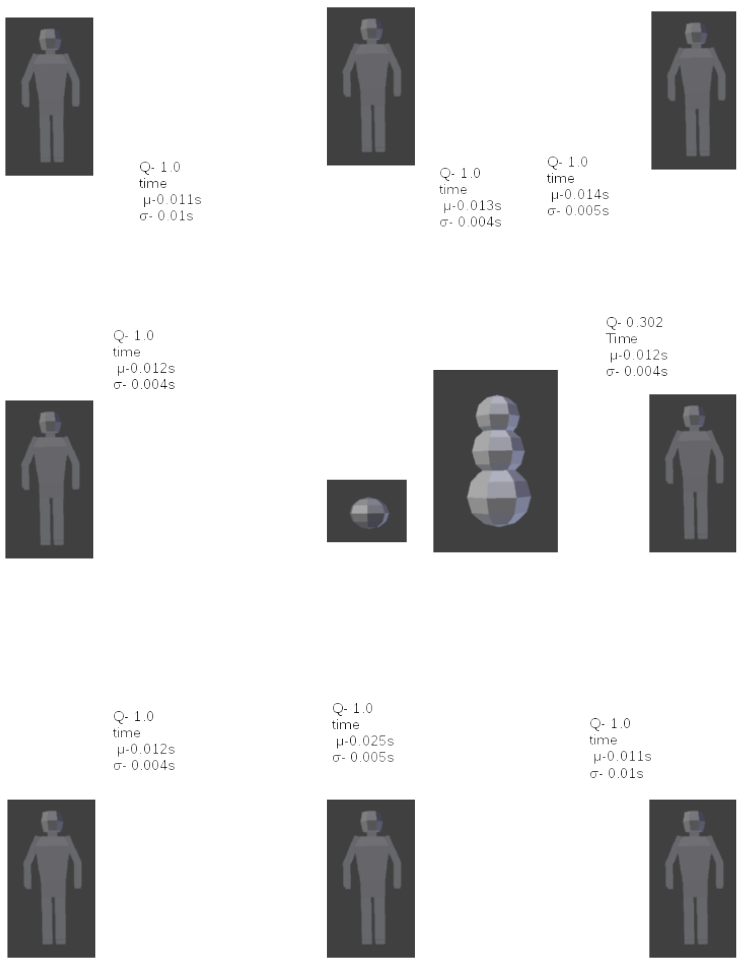

Figure 4.3: In a variety of locations, IMS is able to gauge visibility given the presence of obscuring objects. The relative locations of the snowman, ball, and person represent their location in the testing on the xy plane

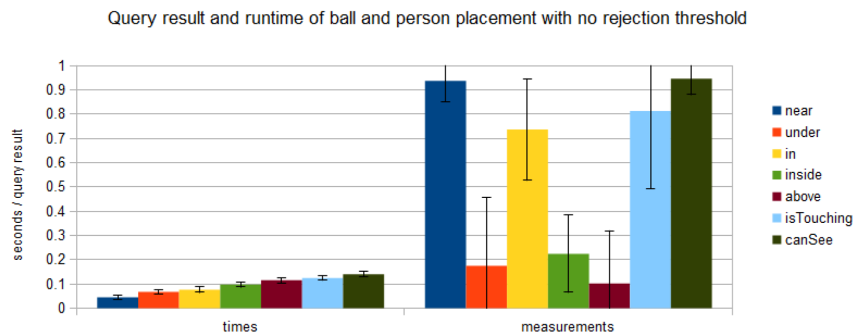

Figure 4.4: Placement with no rejection sampling. `Near`, `isTouching`, `canSee`, and `in` all show accurate placement, while others do not. Considerable variation is present amongst all predicates.

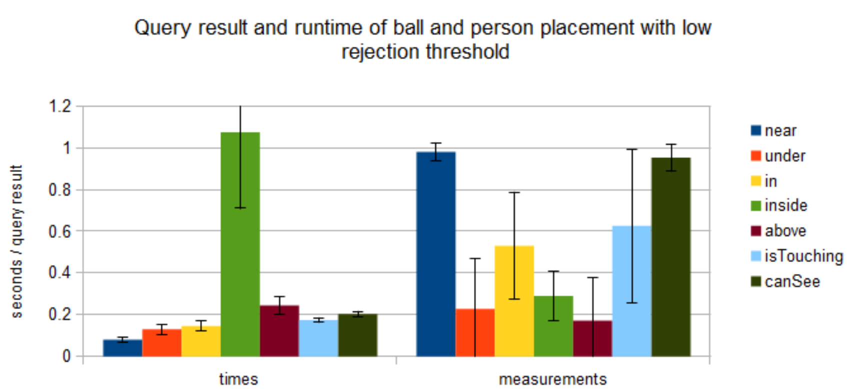

Figure 4.5: Placement with low rejection sampling. Slight but not significant improvements are shown across all predicates, as well as a dramatic increase in run-time for the `inside` predicate.

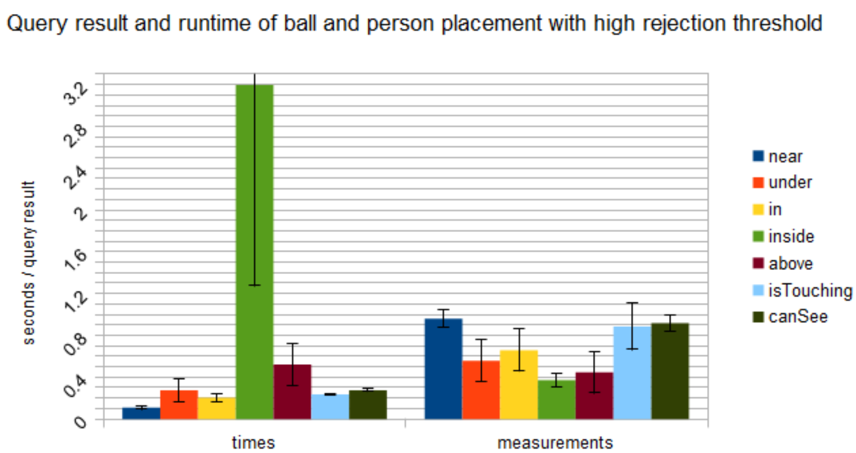

Figure 4.6: Placement with high rejection sampling. Improvements are noticed in placement accuracy, though the run-time of the near predication increases even more dramatically then before.

5 Conclusion and Discussion

As shown in figure 4.1, the system is able to handle the simple case for object querying very reasonably. The fastest calculating predicate is `isTouching`, followed closely by `near`. Because these two predicates operate on similar algorithms, it make sense that the two operate in similar time. One potential significance is the fact that `isTouching` operates faster than `near`, and the only tangible difference between the two algorithms is the fact that `isTouching` has a smaller threshold than `near`. It was first speculated that this meant that the internal Blender functions relied on by these predication functions' run-time increased with object distance. However, the more likely explanation is the calculation of the threshold between the two functions. In `near`, the threshold is calculated with a function that operates in linear time. In `isTouching`, it is a constant, predetermined value, and thus requires no overhead to calculate.

The program worked into the Cornell Cup's Haptek team, where it was used to simulate a person navigating a virtual maze. This project set out to equip a room with several Kinect cameras which monitored the movement of a person in the room. A program, receiving input from the cameras, would build a virtual maze in a virtual representation of the room. The person's virtual avatar would be inserted into this simulation.

The person's motion would be tracked with the cameras, and the avatar would move with the user. If the avatar came into contact with one of the virtual walls in the maze, then a buzzing feedback mechanism would be triggered. This allows the user to navigate a virtual maze. Because this is a simulation in 3D space, the project utilized many of the aspects of the IMS project, including several of the predicates, the models for "person" and "wall," and the temporal updating feature of the 3D scene.

In a similar note, the use of Blender has opened up a number of options for future endeavors. Because Blender is a widely-used program with a broad range of applications and widespread compatibility, IMS has the potential to be utilized in any number of studies and situations. For example, many 3D printers currently accept model input in file formats supported by Blender. At this year's Rochack Hackathon, we were able to 3D print one of the models. This model can be seen in figure 5.1.

Figure 5.1: The 3D printed model of the person from the story scene. A quarter is shown nearby for scale

The Blender file for the person was uploaded with minimal changes to the 3D printer at the school, and printed. The entire process of converting the file, uploading it to the printer, and printing the figure took less than an hour. Given the growing importance of 3D printing in the present, and the many applications to 3D environments that IMS offers, the ability to quickly model objects that can be 3D printed could prove to be a valuable feature of the system.

An area of concern is the run-time of the “inside” predication. The run-time of this predication is significantly worse than the others. Because this function runs in $O(n^2)$ time, it is most likely not due to an inefficient algorithm. Rather, this lack of satisfiability is likely related to the legal placement area generated by the placement function in predMethods.py.

Inside is similar to in, differing only in that it requires one object to be inside the mesh (rather than bounding box) of the other. The legal area, however, is generated to encompass the entire bounding box. As such, it may overestimates the legal area by a considerable amount. Shrinking the placement area would potentially solve this problem, though it may prevent placement in legal areas.

Our project was able to successfully query and place entities under predicate constraints. Because the scene was able to pass the original test of constructing the story-based image and deduce the implicit, spatial information in the scene (the person could not see the egg in the nest), the study was a success. The efficacy of rejection sampling in the placement function requires further investigation. Current testing indicates that it improves scene construction by making placements more accurate, though the key issue in this endeavor is the efficiency of the inside predicate.

The placement system creates a complex relationship between predications and entities during placement. An “optimal” order of placement emerges in complex scenes. Deviation from the optimal layout was shown to increase placement time dramatically.

The “optimal” configuration is one in which the entities are placed in order of decreasing number of predication constraints. For example, in the placement of the story scene, the required predications are:

- a person is under a tree
- a nest is in a tree
- an egg is in the nest
- the person can see the nest

In this scene, the nest is the most constrained object, because it is used in a predication with the person (can see), the egg (in), and the tree (in). The least constrained entities are the tree and the person, each involved in only one predication. This forms a "constraint hierarchy."

For an optimal placement run-time, the entities in the scene should be placed from the most to least constrained. Placing the most constrained entities first allows the most legal placement area for the subsequent entities (which, because they are lower in the hierarchy, have naturally less constrained placement areas), which means that the system has to do less sampling and backtracking on average in order to place them. Note that a constraint hierarchy is not necessarily unique for a given scene because multiple entities can be involved in the same number of predications.

The greatest flaw in the system, and the biggest boundary to continued expansion of the project, is the ad-hoc nature of the predicate and object database. Both entities and predicates are created in an ad-hoc fashion; there exist no templates for either, although some share similar features. Because of this, adding new members to either library can be cumbersome. Further, because the predicate functions of each are based on qualitative semantic interpretations of the predicates, it can be quite difficult to write functions that objectively represent the predicates they are meant to.

Our project's success in creating and querying 3D scenes demonstrates both its usability as a situation for reasoning in 3D spaces, and the expressive power of the system in general. Our program was able to quickly and efficiently manage and query over a small library of entities and predicates. The system is black-boxed to outside input, and as such can be used for more than just Epilog. Preliminary work with the 3D printed model and the collaboration with Haptek has shown this. The system, in its current form, holds potential to be of use in a number of projects and experiments that involve 3D space. We hope that the IMS will be put to use as a specialist program, both in Epilog and beyond.

6 Code Tables

`obutily.py`	
Method	Description
`cast_thru(Object start_obj,Vector endpt, Object end_obj)`	Casts repeated rays from `start_obj` to `endpt`, noting the degradation due to occlusion that occurs along the way stopping when `end_obj` is hit. Returns a value 0-1 indicating the occlusion encountered on the way
`nudge(Object a, Vector pt)`	Returns a points several thousandths closer to a from pt, used in repeated ray casting experiments
`rayCast(Object a, Vector b)`	Shoots a ray from the center of a to b, returns the same as `Scene.ray_cast(start,end)`
`alignMeasure(a, b, top=float(inf), right=float(inf), bottom=float(-inf), left=float(-inf))`	Creates a rectangular prism mesh on top of b that can be used for intersection testing. If no maximum top/bottom/etc... points are specified it will use the most outward points on b's bounding box
`highest(Object obj, char dim)`	Returns the highest vertex in the object (in the dimension (x,y,z) specified)
`lowestPt(Object obj,char dim)`	Returns the lowest vertex in the object (in the dimension (x,y,z) specified)
`getIntersection`	Returns the part of m's mesh that is intersecting with a

`getDiff(Object m, Object a)`	Returns the part of m's mesh that is not intersecting with a
`closest_points(Scene scene, Object a, Object b)`	Returns a tuple containing the closest point on the mesh of a to b, and the closest point to a on b's mesh, in that order
`maxDim(Object [] ary)`	Returns the largest dimension (x,y,z) among all objects in ary
`distance(Vector a, Vector b)`	Returns the distance between a and b (will not work if an object is in the way). Measured without pathfinding.
`glob2Loc(Vector pt, Object obj)`	Returns point pt in `obj`'s local space
`vertsGlob(Object obj)`	Returns an array of `obj`'s vertices in global coordinates
`locs2Glob(Vector[] pts, Object obj)`	Returns the location of the coordinates in `pts` in global coordinates (assuming `pts` are originally in `obj`'s local space)
`loc2Glob(Vecotr pts, Object obj)`	Same as `locs2Glob`, but for only one point
`aInBSpace(Vector pt, Object a, Object b)`	Returns pt (which is in `a`'s local space at the start) in `b`'s local space; works through matrix multiplication
`getVolume(Object obj)`	Returns the volume in BV^3 of the object; underneath this wrapper method are numerous helper methods

Table 6.1: Obutils Methods

predMethods.py			
Predicate	Description	`Query`	`Place()`
`near(A,B)`	determines whether A is close to B	Iterates through the children of a and b and selects the two with the shortest distance between the two meshes. the value returned is the distance relativized to the sizes of the two entities. The result is not proportional to distance, and returns true up to a certain distance and then a result from a steeply graded exponential function thereafter.	Returns a box bounded by the x and y location of the focal object plus and minus the largest dimensions of the two objects.
`under(A,B)`	determines to what extent A is underneath B	draws a temporary object directly under B; the percent volume of A that intersects with this is the return value	If B is the focal object, it returns a bounding area spanning from the bottom of B to 5 BU under B. If A is the focal object, the z boundary is from the top of A to 5 BU above A. In either case, the x any y boundaries are the maximum of the two objects in the respective dimensions.

`in(A,B)`	determines whether A is inside B's bounding box	draws a temporary object around B's bounding box and returns the percent volume of A, or true if more than half of A's volume is in B.	The boundaries in all dimensions are A's location plus or minus B's size in the given dimension.
`inside(A,B)`	determines the extent to which A is inside B	same as in(A,B), but the temporary object is drawn around B's mesh rather than bounding box	The boundaries in all dimensions are A's location plus or minus B's size in the given dimension.
`above(A,B)`	determines the extent to which A is above B	Draws a temporary object in the space above B and returns the percent of A's volume that intersects.	If A is the focal object, it returns a bounding area spanning from the bottom of A to 5 BU under A. If B is the focal object, the z boundary is from the top of B to 5 BU above B. In either case, the x any y boundaries are the maximum of the two objects in the respective dimensions.
`isTouching(A,B)`	determines the extent to which A is touching B	This function works the same as near(A,B), but does not adjust with relative object size and requires objects to be much closer.	Returns a box bounded by the x and y location of the focal object plus and minus the largest dimensions of the two objects divided by ten.

`canSee(A,B)`	determines whether A can see B	(this assumes an "eye" object on A). Ray casting is done from A's eye to points on the meshes of B's children. The percent of successful casts (that reach B) is the value returned. If the cast hits a translucent object, it will continue but will return a reduced value	returns a bounding box surrounding the focal object's position plus or minus 10 BU in all dimensions

Table 6.2: predMethods Methods

Bibliography

Jeffrey P Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, et al. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 333–342. ACM, 2010.

Daren C Brabham. Crowdsourcing as a model for problem solving an introduction and cases. *Convergence: the international journal of research into new media technologies*, 14(1):75–90, 2008.

Jeff Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

Walter S Lasecki and Jeffrey P Bigham. Interactive crowds: real-time crowdsourcing and crowd agents. In *Handbook of Human Computation*, pages 509–521. Springer, 2013.

Walter S Lasecki, Rachel Wesley, Jeffrey Nichols, Anand Kulkarni, James F Allen, and Jeffrey P Bigham. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 151–162. ACM, 2013.