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 with Honors Research

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

2 Background

John McCarthy was the first to formally identify the concept of commonsense reasoning in literature, and described it as

"the ability of an agent to deduce a sufficiently wide class of immediate consequences of anything if it is told what it already knows" (**?**).

For his seminal Comirit project, Dr. Benjamin Johnston completely rejected any definition of common sense period (**?**). Over the course of all surveyed literature, there are a number of consistencies in the various definitions (both implicit and explicit) of commonsense reasoning in artificial intelligence. Some common themes are:

1. The ability to quickly reason on a series of mundane topics and issues
2. Deduction capabilities on high-level, well known topics to which the system has been exposed previously
3. knowledge of a wide variety of topics and of relationships and interactions of entities within
4. an emphasis on rapidity of reasoning rather than accuracy or precision

The use of symbolic logic is also worth noting in the field of commonsense reasoning. Symbolic reasoning has been an integral part of a number of programs meant to emulate common sense. Dr Johnston's Comirit utilizes a tableaux reasoning which uses proof by

resolution in conjunctive normal form propositional logic in order to make logical deductions (**??**). the Soar project also utilizes a symbolic logic processing unit in addition to it 3D simulations in a very similar manner (**?**). McCarthy's original advice-taker and advice-giver programs were entirely symbolic, relying on forward inference in order to make deductions and reach conclusions based on the given data in its knowledge base and input from the user (**?**). Although statistical methods could potentially be used in this field of AI, the vast majority of literature utilizes symbolic or logical methods.

2.1 Commonsense Problem Formulation

Commonsense reasoning problems are approached from a different perspective from that of other AI problems. While many AI agents attempt to create an agent that can efficiently execute one task that supposedly requires "intelligence," commonsense reasoning problems are broader, and less deal with a less precise form of knowledge. For this reason, commonsense problems are often solved with general solutions with less precise metrics than other problems (**??**).

One of the first commonsense problems was the "Tweety problem." This involved a human feeding a story into an AI agent; this story progressively described an entity "Tweety." It was proposed that an agent with common sense would be able to infer that Tweety was a bird, and thus infer that it could likely fly. The agent would also be able to adjust it's mental preception of the bird, perhaps inferring it to be a flightless bird, such as a penguin, if it was later stated that Tweety could not fly (**??**). This problem requires a broad thinking over the general properties of a number of topics and objects, and the inference process is inexact and nebulous.

The "egg-cracking" problem is also one of the more famous lemmas of commonsense reasoning. In this problem, as with the Tweety problem, an agent reads a story about a chef making a cake (one of the steps of which is cracking an egg on the bowl and pouring the contents inside). The agent is then asked a series of questions about the scenario. These interrogations usually have a broad range, but would be considered trivial to a human ("why doesn't the chef throw the whole egg into the bowl?" "What will happen

if the chef does all the motions very slowly?") (**?**). In sum, the primary features of commonsense benchmark problems can be summarized in the following points:

- open problem domains
- evaluation based on the answering of naïve questions
- focus on "breadth" knowledge, rather than "depth"

2.2 Significance

The importance of a system to reason in common sense is readily apparent to the non-technical thinker. Nearly every task a human undertakes requires an implicit knowledge of everyday commonsense, and as such, the field of reasoning is almost inseparable from most of the end goals intelligent artificial agents (**??**). Tasks such as navigating a road during rush hour and pouring wine into a glass properly require naive knowledge of a variety of topics, such as traffic etiquette, physical properties of liquids and solids, acceleration and deceleration of automobiles, etc. There is also a suprising robust amount of existing data for commonsense systems. Johnston cites that many video games implement commonsense reasoning in the form of enemy AI (**?**). Further, common sense is, by definition, possessed by a large percentage of the human population. Because of this, there is a very large potential workforce for compiling or otherwise working with common sense reasoning systems. This has already been implemented with an online application for creating common 3D figures (**?**).

The generality of commonsense reasoning is one of its primary benifits. Almost any real world problem will have to deal with an open domain. It is clear that the world does not operate on anything even approximating the extremely controlled and limited worlds commonly evaluated in most computer environments. If an AI agent is to solve real-world problems, it must be able to solve more than a very narrow range of problems, and deal with the numerous contingencies that arise in real world scenarios. (**?????**).

2.3 3D Simulations

Hand in hand with commonsense reasoning is the role of 3D simulations. In an informal sense, a simulation is the ability to construct (or in some cases, reconstruct) physical scenes from symbolic data. More formally, a simulation is a type of analogous representation which begins with a description of a system which changes based on a causal description of the system (**?**). Simulations have been used many times in the course of both AI and cognitive science. McCarthy first suggested using simulation in his "advice taker" program. Since then, the concept has reappeared many times in literature. In particular, physical systems (those representing real-life physics) have implemented with simulation.

Previous examples of this in Gardin et al's use of atomic spheres to model liquids and their properties. This example implements spherical objects (analogous to atoms) which behave collectively as a liquid. The simulation of gravity along with these structures enabled a primitive AI to learn to correctly fill a glass. Johnston was able to accomplish a the same results with Comirit's (**??**). The Soar project also implemented 3D simulations; by creating a 3D environment and testing object positions within, the system was able to deduce rudimentary physical properties and solve related problems (**?**).

The relation between 3D simulations and commonsense reasoning is apparent. As mentioned earlier, commonsense reasoning's primary domain is real-life situations. Human beings are creatures who live in a environment with three spatial dimensions and one time dimension, physical laws are well defined and inescapable in the real world. This means that the most apparent and relevant environment for solving commonsense lemmas is one which incorporates the properties and aspects of the real world (**??**).

Cognitive science has observed that humans process their physical environments as a series of simulations: forming an initial mental image and adjusting it over time in order to deduce properties about it (**?**). Thus, there is considerable evidence and motivation for utilizing simulations in AI, particularly in a program designed to model and reason in physical situations. The implementation of such a system as a specialist in the Epilog system also meshes well with previous experiments and evidence.

The incredibly complex spatial situations of three-dimensional simulations are far to computationally intense and convoluted for a symbolic logic system to represent by itself. The aid of a specialist provides a more narrow computational domain which can soundly and quickly compute responses to queries over a sub-domain of problems (in our case, problems pertaining to reasoning in spatial environments).

2.4 Specialist Programs

The usage of specialist programs (as opposed to stand alone programs) has shown to be an effective means of programming common sense. While many programs exist as single-component entities, a large proportion of AI programs, as mentioned earlier, deal with a much narrower scope than commonsense reasoning problems (**??**). Because of this, there is less of a need for specialist programs because the entire program is a specialist. However, commonsense reasoning requires breadth, not depth, and thus encounters the problems predicted by the clichè, though true, axiom of software engineering, and life in general: one cannot be good at everything.

Resolution theory provides a formal justification for specialist programs. The central premise of this is that if a given set of clauses each contains a set of sub-clauses which are mutually incompatible (contradictory), then at least one of those clauses need not be true. This allows a considerable narrowing down of the state space needed in order to solve a problem. Because of this, mutual incompatibility of logical atoms are essential to specialist construction. Schubert et al successfully implemented specialists for Epilog for time, color, and part relationships (**???**).

Thus, commonsense programs will likely need to be broken down into functional units. One of these units would specialize in 3D spatial reasoning. As mentioned above, it is infeasible and awkward to compute problems relating to 3D space in a purely symbolic environment. These computations, however, are very necessary for a true commonsense reasoning system. This lays the foundation for the motivation for the IMS project.

2.5 Usage in the IMS and Connection to Epilog

The Epilog Project is a natural language processing program which seeks to implement commonsense natural language understanding. The system is distinct from other, more traditional means of representing natural languages formally (such as first-order logic) in that it allows for complex and rich concepts such as belief systems, and, as the name of the system suggests, explicit references to time frames. It is from this feature that the name Epilog (EPIsodic LOGic) is derived (**?**).

It is apparent as to how this focus on time frames is closely related to simulation. Just as simulation environments involve updating scenes according to a set of defined rules and relationships as the passage of time occurs, Episodic Logic seeks to divide the world into distinct moments in time and create a logical framework from this (**??**).

Epilog does not, however, contain methods for accurately or quickly evaluating stories that exist in 3D environments, nor use knowledge based in this frame of reference. Epilog also has a well defined interface for creating and connecting specialist programs to the larger program. This high level of modularity means that specialist programs can be created easily for the greater Epilog project (**?**). The best solutions is, therefore, to utilize the "time-slice" centric approach of Episodic logic, as well as the existing specialist interface, and construct a specialist for reasoning in 3D environments.

In keeping with the established paradigm for commonsense program evaluation, it is most appropriate to base the evaluation of the specialist with a test of a broad range of knowledge that operates at a level that is considered "simplistic" by human standards. For this reason, childrens' stories provide an excellent source. The following passage, taken from from Lesson 32, Harris et al. 1889, provides information on a visually rich scene, one which requires a wide range of knowledge of spatial relations and laws in order to understand correctly:

1. Oh, Rosy! Do you see that nest in the apple tree?
2. Yes, yes, Frank; I do see it.
3. Has the nest eggs in it, Frank?
4. I think it has, Rosy.
5. I will get into the tree.

6. Then I can peep into the nest.
7. Here I am, in the tree.
8. Now I can see the eggs in the nest.
9. Shall I get the nest for you, Rosy?
10. No, no, Frank! Do not get the nest.
11. Do not get it, I beg you.
12. Please let me get into the tree, too.
13. Well, Rosy, here is my hand.
14. Now! Up, up you go, into the tree.
15. Peep into the nest and see the eggs.
16. Oh, Frank! I see them!
17. The pretty, pretty little eggs!
18. Now, Frank, let us go.

This story covers a number of physical scenarios and implicit information that can only be deduced with a sufficient understanding of the spatial laws of nature. For example, a specialist should be able to simulate a tree, two children (Rosie and Frank) the nest in the tree, and the fact that the eggs in the nest are visible from above the nest, but not below. This presents a broad knowledge scenario which tests the abilities of an agent to reason in an open-ended, commonsense environment. This is consistent with the evaluation methods for previous entries in literature, and as such is a good metric for the IMS (**???**). For the demonstrative purposes of this project, the 3D Specialist should be able to do the following:

1. construct a scene where
    - a person is under a tree
    - a nest is in a tree
    - an egg is in the nest
    - the person can see the nest
2. deduce that the person can see the nest (though the vision may be obscured by branches)
3. deduce that the person cannot see the egg (because the nest is in the way)

This scenario provides a diverse assortment of predications, objects, and conjectures about the story, such that the above tasks will be an important indication of whether or not the system is capable of larger, more complex inferences in commonsense reasoning.

2.6 Object Construction and Placement

From the more concrete aspect of 3D simulation, the methods by which objects are modeled in the 3D scene, and the methods by which they are placed to satisfy logical constraints, are a matter of considerable importance.

Most of the simulation work surveyed constructs objects in a very similar fashion. 3D object made of polygons bound together are the norm for 3D simulation (**???**). Further, there is a plethora of existing software that utilizes this "face, mesh, vertex" paradigm. Thus it makes the most sense to utilize this model in order to capitalize on the fruits of previous research.

Both Soar, Wordseye, and CAPS utilized similar placement methods for objects. Soar utilized the concept of "legal areas:" distinct areas of three-dimensional space that represented potential locations for object placement that satisfied predications (**?**). Similarly, the CAPS project implemented "legal polygons," the two dimensional equivalent (**?**). Wordseye uses a combination of the two: both three-dimensional placement areas and two-dimensional legal polygons, as a means of specifying areas where object can be placed in order to satisfy predications. In addition, Wordseye also object "tagging," labeling certain parts of objects as relevant areas for certain predications, as a means to aid in object placement. For example, the hand of a person model is tagged with as being able to hold certain objects. If the scene requires the person to be holding an apple, the area specified by this tag becomes the critical point.

3 Methodology

The Blender software (www.blender.org) was chosen as the simulation environment for our project. Blender is a free, open source program designed for 3D modeling, physics simulation, and game design. Blender provides a means of easily creating 3D objects, attaching properties and constraints to those objects, and developing scripts that interact with the software and objects.

Models in Blender are stored as "objects" with associated "meshes." A mesh is a data structure with a series of faces, edges, and vertex locations in space for the object. Blender also contains methods for adding custom properties to objects, which serves to make the system quite extensible. Native support for 3D modeling and operations is highly desirable as well. 2D approaches were also considered, but possessed limited support for modeling and manipulating scenes in 3D. With Blender's existing 3D support and extensive library utilities for 3D object management, the task of developing a simulation environment is eased considerably.

Blender contains a very accessible Python API. This allows Python scripts to access methods in Blender, and allows the system to create and change objects, measure properties of these objects, and even instantiate physical laws (such as gravity) in the scene. The direct purpose of this system is to allow Epilog to communicate with a built in series of Python scripts, which then, in turn, will utilize Blender to construct a 3D environment, which can then be examined to derive information about the scene to be fed back as input into Epilog. These functions are to be performed incrementally, i.e.,

in support of a sentence-by-sentence story understanding process.

The IMS exists as a series of Python files and a database of objects which interface with Blender. Please note that, from this point on, direct references to structures in the code will be indicated by `courier` font.

3.1 High Level Overview

The source folder contains three Python files:

1. `Classes.py`
2. `predMethods.py`
3. `obutils.py`

A single directory, `ObjectData`, is also included.

`Classes.py` file is the highest level Python file, and serves as the top level of the system's organization and control. `Classes.py` contains three classes:

1. `Scene`
2. `Entity`
3. `Predication`

The `Scene` class serves to hold the information currently being modeled in the Blender scene, to provide methods for adding entities and predications to the scene, and for querying the scene for how well a given predication is satisfied. Although object placement in the system is meant to ensure that all the predications in a given scene are satisfied, this is not necessarily the case. The predications may very well be mutually unsatisfiable, and in keeping with the central purpose of this project, a quick and slightly less accurate placement mechanism is preferable to a more accurate, but much slower one.

Each instantiation of the `Entity` class stands for an entity in the domain of the scene being modeled. An entity is any object in the scene which can be modeled as a Blender mesh object. The `Entity` class contains attributes for storing information about the given entity - including a pointer to the object being modeled in the Blender scene - and methods for interacting with the object in the Blender scene.

Each instantiation of the `Predication` class is a predication active in the current scene. A `Predication` instance contains references to the Blender [parent] objects bound by the given predication, and a method for returning placement constraints for either of the objects relative to the other, and one for querying how well the predication is satisfied given the current locations/orientations of the `Predication`'s objects in the Blender scene. The `Predication` class is generic, and imports placement and querying methods specific to a given predication (e.g. "near", "above") from the `Classes.py` file.

Note that the actual functions for the predication are stored as variables in each class instantiation. Once the methods are copied from `predMethods.py`, `predMethods.py` is no longer called directly by the predication.

`predMethods.py` contains placement and query methods for each predicate, for example (respectively) `nearP` and `nearQ` for the "near" predicate. Many of these predicates involve complex functions in Blender's three-dimensional space, which are stored in `obutily.py`. `obutily.py` consists of various methods for operating on Blender objects, and is used by both `predMethods.py` and `predMethods.py`. The full table of the methods for `predMethods.py` and `obutils.py` can be viewed in final section. This ontology was chosen because of how well it fit in with existing Epilog framework. Epilog's internal ontology already defines the domain of discourse into subcategories denoting objects, events, predications, etc. (**?**). Thus, it makes sense to partition the IMS's ontology similarly. Entities (individuals existing in real space) and predications acting on these entities are explicitly defined in the current model. The other relevant features of the scenes, such as episodes, are built into the framework of the system. For example, an episode is defined as the current arrangement of objects and predications at a given moment within the simulation. This utilizes the built-in

model of the IMS as a specialist is utilized.

3.2 Objects and Entities

The ObjectDatabase directory contains two sub-directories:

1. OBJ
2. XML

These correspond, respectively, to the two types of information being stored for each object: the three-dimensional model of the object as used by Blender, and other information about the object that is not directly used by Blender.

In the IMS, entities are modeled in Blender not with a single mesh, but with separate meshes corresponding to each of the entity's parts. When an entity's meshes are imported to the scene, the `Entity` class imports entity <.obj >-format object files from the <item name >subdirectory in the OBJ directory, where "item name" stands for the entity's key (e.g. "person" or "house"). In the Blender scene, these part meshes are bound as children under an empty-point Blender object that is the parent. In the `Entity` class, the pointer to the Blender object is a pointer to this empty parent object (from which children/parts are easily accessed). Figure 3.1 illustrates the flow of information and interaction between components.

The XML directory contains `<item name>.xml` files for each entity for which there exist Blender models, where `<item name>` stands for the entity's key. Each XML file contains the following information: a meronomy graph for the entity, a list of the parts with meshes, and finally bounding box coordinates for the object. Periodic updates, predominantly introduction of new objects and predications, as well as changes to existing objects and predications, are sent to the specialist by Epilog.

Although a very extensible sotware system, Blender was not designed specifically for the complex scripting that this project utilized. As such, several object and code issues were encountered with the Blender system that needed to be documented. First,

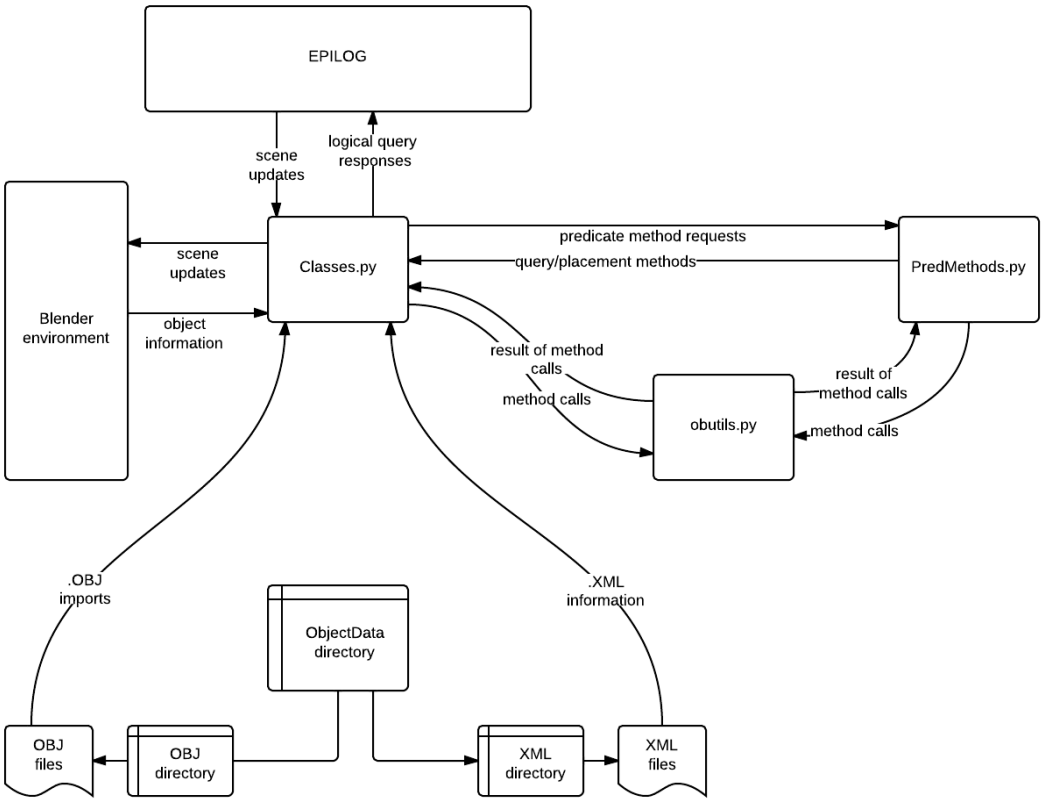

Figure 3.1: The basic flow of information in the IMS.

many of the objects in the system did not have *well-formed* meshes. *Well-formed* in this case has very specific meaning. In this case, a blender mesh is said to be *well-formed* if it contains closed objects with no holes in the mesh, edge loops (which define faces) which are connected and form a closed loop, and no disconnected components. Unfortunately, several objects in the system (notably the chair object) contain some glitch in their meshes which renders them not *well-formed*. This can cause several errors when intersection and difference algorithms (or other methods which involve mesh operations) are utilized by the system. These changes were made, and the objects were able to function correctly.

There are further limiting factors that were noted when utilizing Blender to place and measure objects. Due to the algorithms used in the volume calculation:

- Faces and Edges cannot be placed directly on top of one another, an offset (even as small as 0.001 BU) of one of the objects must be used
- Even if an object is deleted from Blender's scene environment, it's mesh information will still remain. This can slow down the computation of built-in functions in blender whose run-time scales with the total number of meshes in the scene. As such, scenes must be restarted for multiple rounds of testing to ensure accurate, releiable data.

3.3 Predications

My particular work has focused on the construction of the predicate class, as well as the helper methods in `obutily.py`, and management and organization of the program files. Each `Predication` instance contains two methods: `Place()` and `Query()`. These predicates are stored as `<predicate name>P` and `<predicate name>Q`. As shown above, the placement function of the near predication is `nearP` and the query function is `nearQ`. Query methods evaluate the scene and return a value for how well the given predicate is satisfied in the current scene. This can be done continuously as a value between 0.0 and 1.0, or as a boolean. For a boolean value, the continuous answer is still evaluated: 1.0 is returned if the answer is above 0.5, and 0.0 if below. The purpose of the `BinaryFlag` attribute in the `Predication` class is to determine whether the predicate is evaluated as a boolean value or not.

3.3.1 Vision Predication

By far, the largest and most complicated predicate was `canSee(A,B)`. The implementation of this function relied heavily on ray-casting and went through several iterations before a sufficient model and algorithm were developed. The querying method for this predication originally was to be implemented as a single ray-cast, from one object's center to another.

A number of different attempts to construct a sufficient `canSee(A,B)` function were tested, yet none were of the quality needed for the project. Examples include the use of a conical object expanding from entity A's "eye" to B and noting the percent of B's volume that was encompassed inside. Several functions in Blender's Game Engine were also implemented. Early on, a function existed which ray-casted from the center-point of A to the center point of B. This function was naively simple, but proved itself to be surprisingly difficult to implement. For a graphical representation, Figure 3.2 below illustrates the ideal for this model. Note that in this (and future) diagrams, a solid line represents the actual ray cast, and a dotted line represents the remainder distance that the cast would have traveled if there had not been an object in the way acting as an interrupting agent.

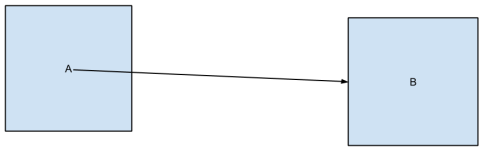

Figure 3.2: an idealized model of the naive implementation of `canSee(A,B)`.

One issue that came up with this design was the collision nature of ray-casting. Namely, a given ray-cast in Blender will return when it hits an object, any object. This means that casts from A's center would always stop upon hitting A's outer mesh. One early attempt to solve this was to simply implement repeated casting: the method would cast continuous rays from the end of one to the start of the other until the end object was met. Figure 3.3 shows the idealized version of the function of this implementation of the function.

This revealed yet another problem, however, as casting from the face of one object (which was the collision point of the fist ray-cast in the above) would simply return the

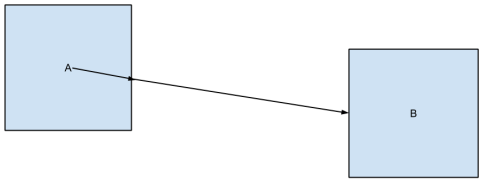

Figure 3.3: A second implementation of ray-casting

starting point of the cast. The ray-cast could repeat infinite times and still be stuck on the same point. The third iteration of this function was meant to fix this. This iteration involved the addition of another method in `obutily.py`, `nudge`, which would return an infinitesimally small distance closer to the end point. This function allowed repeated ray-casting to continue unabated and as planned. The new model for ray-casting in this simple iteration is illustrated in figure 3.4.

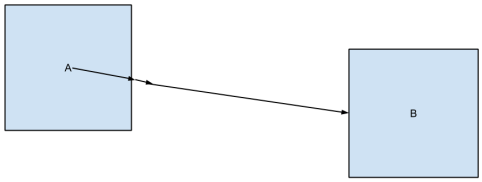

Figure 3.4: The nudge addition to the naive model

This model was able to accurately cast a ray from the center of A to B, stopping when it hit B's outer mesh. However, the naive model was, as the name should suggest, not sufficient and could easily return erroneous answers. For example, if there was an object in between A and B which could block the cast, then the method would conclude

that A could not see B, even if it were obvious that a line of sight existed between the two. Figure 3.5 illustrates this.

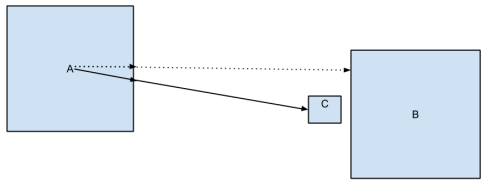

Figure 3.5: C blocks the site of A, even though there exist obvious casts (dashed) that would indicate that B is visible

It was obvious early on that this implementation would only serve as a template for future endeavors. The second model cast a slew of rays from A to the vertices of B. This improved over the naive implementation in two ways. First, it allowed a varying degrees of visibility. Several of the rays may reach their target, and several may not, which allowed for a sense of partial obscurity which was not allowed in the original model. This ran into a small technical issue early on, however, as rays would not return if they made contact with the vertices of an object. Due to a technicality in Blender, casts would only return if they made contact with the face of an object. As such, a modification of the nudge function was implemented which pushed the end point of the casts closer to the center. This way, there was a guarantee that the casts, if unobstructed, would hit the object properly. Figure 3.6 illustrates this model.

This model was not perfect, however, in that it could give bias to a section of an object if there were a large number of vertices concentrated at a given section. In figure 3.7, we show how this could be problematic. In this example, A would cast a disproportionate amount of rays to B's bottom side, which would collide with C, giving the impression that B is much less visible than it actually is.

One suggested solution to this situation involved weighting the vertices depending

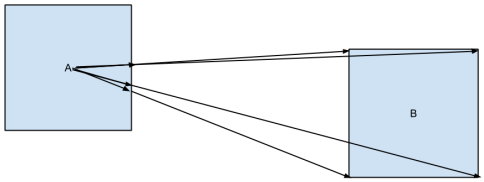

Figure 3.6: The second iteration of the `canSee(A,B)` algorithm

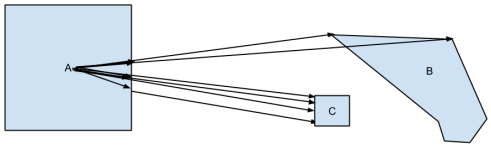

Figure 3.7: The majority of the casts are blocked by C even though B is visible.

on how close they were. In this interpretation, casts to close-by vertices would be worth less. If these successfully hit their target, they would return a number less than one. The exact weighting system was not decided. This system, however, proved to be unnecessary, as the final algorithm removed the need for vertices entirely.

The final system utilized an obscure library in Blender Python (`BPY_extras`) to grasp random points on the meshes faces. Rays were cast to these points, and the percent of casts that returned were used to evaluate the visibility of the target object. This method was both fair and correct. The number of rays cast to a given face was proportional to the face's percent of the total surface area of the object, which ensured that the most visible (largest) faces would receive the most casts. Because the number

of samples is proportional to the size of the object's faces, no section of the object will over represent itself in testing. Figure 3.8 shows how the new method solves the problems of the old.

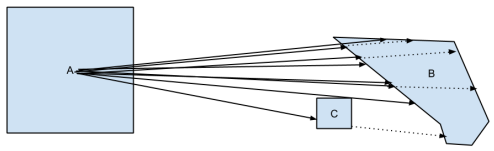

Figure 3.8: Rays are cast evenly across object B.

The final change to the vision predication involved compensation for opacity. It was desired that the visibility of an entity not simply depend on line of sight, but also on the presence of partially occluding objects passed through en route to the goal object. This was solved through custom properties and repeated casting. Each ray-cast was given an initial value of 1. The cast traveled through space as in the above model. When another object was encountered, the behavior of the cast differed depending on the object hit. If the object was the target object, then the cast returned it's value (initially set to one). If it encountered a different object with no occlusion property (or an occlusion value of one) then the hit object was assumed to be completely opaque and the cast returned zero. A completely opaque object (one that would cause any ray to return 0 if it made contact), was specified with an occlusion of -1. If the object hit an object with an occlusion value between zero and one, then the object recast towards its target, but the value of the occlusion property was added to an occlusion counter for the cast. This occlusion counter would would be reduced by the occlusion amount of the object when the cast left the object.

As the cast traveled through an object, the value would degrade proportional to the current counter. If the return value was reduced to zero, then the cast would fail and

the function would return zero. Thus, the further through an obscuring object (such as a fog or the leaves of a tree) an object was, the more obscured it became. The occlusion counter was necessary in order to allow multiple obscuring factors affect the object. This corresponds to situations such as man looking at a bird through a tree in a fog.

The addition of occlusion necessitated the introduction of a new function in `obutily.py`: `cast_thru`. This function performed all of the above functions, and took care of the repeated casting (including calls to ray_cast and nudge). The initial cast was handled by the `predicateMethods.py` file, and subsequent casts were all done in the `cast_thru method`. This led `canSee` to a slightly different larger dependence on `obutily.py` functions than other predications.

Originally, the starting point for `canSee` was the location of the first object in the predication. Typically, this was the lowest point in the object, in the center of the xy plane. This presents an obvious issue for calculating visibility: human beings do not see with their feet. Thus, for models which were capable of vision, an additional "eye" was added to list of meshes. The `canSee` function was changed to begin ray tracing from the location of the eye, which was located just in front of the face of the figures.

3.3.2 Placement Methods

`Place()` returns the areas in the Blender scene where the predicate holds for one entity relative to the other. This makes the assumption that the second object has already been placed; for example in the predication "near(A,B)", if `Place(A)` was called, it would return an acceptance area of "nearness" for A, such that the predication is satisfied. At present, the placement locations are returned as a tuple of pairs, with each pair containing the minimum and maximum values for the x, y, and z dimensions. This was meant to build of previous (similar) methods in Wordseye and Soar, which used similar methodology. (**??**). The placement functions went through several iterations and different ideas, and were the subject of much revision over the course of the project. For the first build of the project, the current system was implemented. Figure 3.9 provides a visualization for this method.

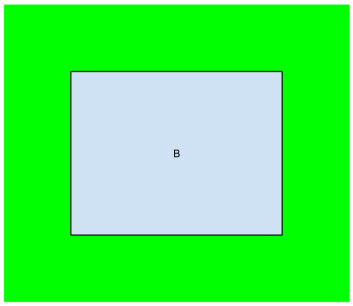

Figure 3.9: A visualization of the first and current placement mechanism.

This system suffices for many predicates; however, it is not easily extensible to predications which have disjoint or disconnected valid placement areas. As such, there were concerns about whether a better placement paradigm was necessary. In particular, the vision predicates can generate placement areas of this sort. Figure 3.10 illustrates the issues that the system raised.

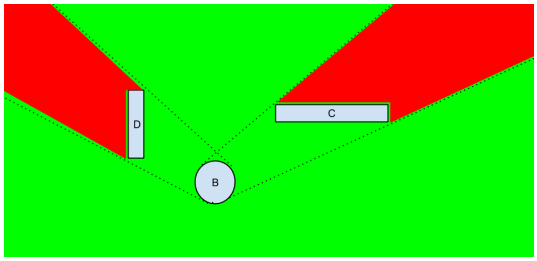

Figure 3.10: The theoretical placement areas for the predicate `canSee`. Valid areas are colored green, invalid areas are red.

Several solution have been researched and proposed. Wordseye utilizes a the concept of "valid placement polygons." This concept represents the valid placement areas of objects as polygons, and, with this heuristic, is able to choose the best area for inserting an object in a scene (**?**). Another similar solution is proposed in Xu et al, where Minkowsky sums are utilized. This technique has been shown to be efficient in certain special cases of polygons, and was able to produce fruitful results in experimental trials (**?**).

In this vein, I proposed the idea of utilizing multiple of our current placement

rectangular-prisms (and their combinations) in order to represent the valid placement areas of a given polygon. This idea has been worked on and refined this idea into several possible solutions. One solution is the idea of the "grid-box technique." In this technique, the geometry of placement areas is represented as an assembly of discrete 'boxes', one of which is sampled for a valid placement location of the given object. Each box is represented a 3-tuple of pairs, corresponding to x, y, and z coordinates of a cube in three-dimensional space. These boxes, unlike their counterparts in the current placement system, are of a discrete and unchanging size, and are arranged in a tessellating fashion similar to a grid. In this method, a square is considered a valid placement area if all of it's points satisfy a the respective predication. Figure 3.11 below provides an illustration.

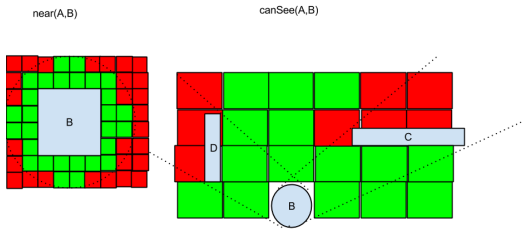

Figure 3.11: One proposed method of placing objects using by using a uniform grid of placement areas.

Another proposed method is the rejection-sampling model. A short mockup of it was even built earlier this year. This model randomly selects points given a heuristic for the model, and rejects them if they fails to satisfy the specified predication or predications. 3.12 below illustrates the system.

As previously noted, this system does allow for a very simple implementation (forgoing the placement prism entirely), though because it rejects so many potential points, The feasibility of this algorithm was a subject of investigation.

A combination of rejection sampling and the standard "block placement" algorithm was used. The collective blocks of all the predications of an object are overlayed, their intersection being the new legal placement area. Rejection sampling is then used to sample locations from this space. The resulting configuration is queried. If all the

predications are satisfied, the next object is placed. If it is not, then another spot is sampled, and the process begins again.

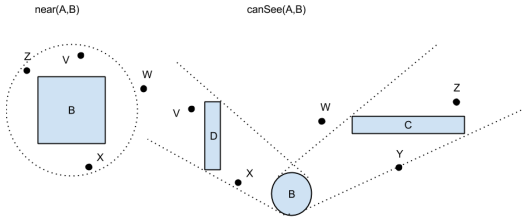

Figure 3.12: Rejection sampling was also considered as a placement mechanism.

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,

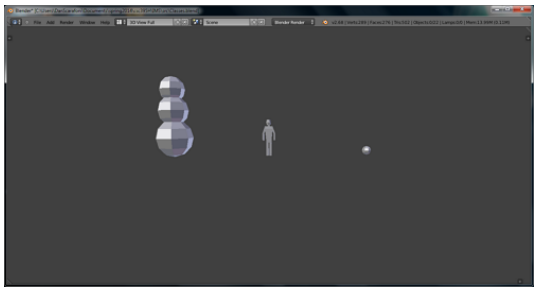

Figure 4.1: a visualization of the three objects used for basic testing. From left to right: the snowman, the person, and the ball. Entities are shown to scale.

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"

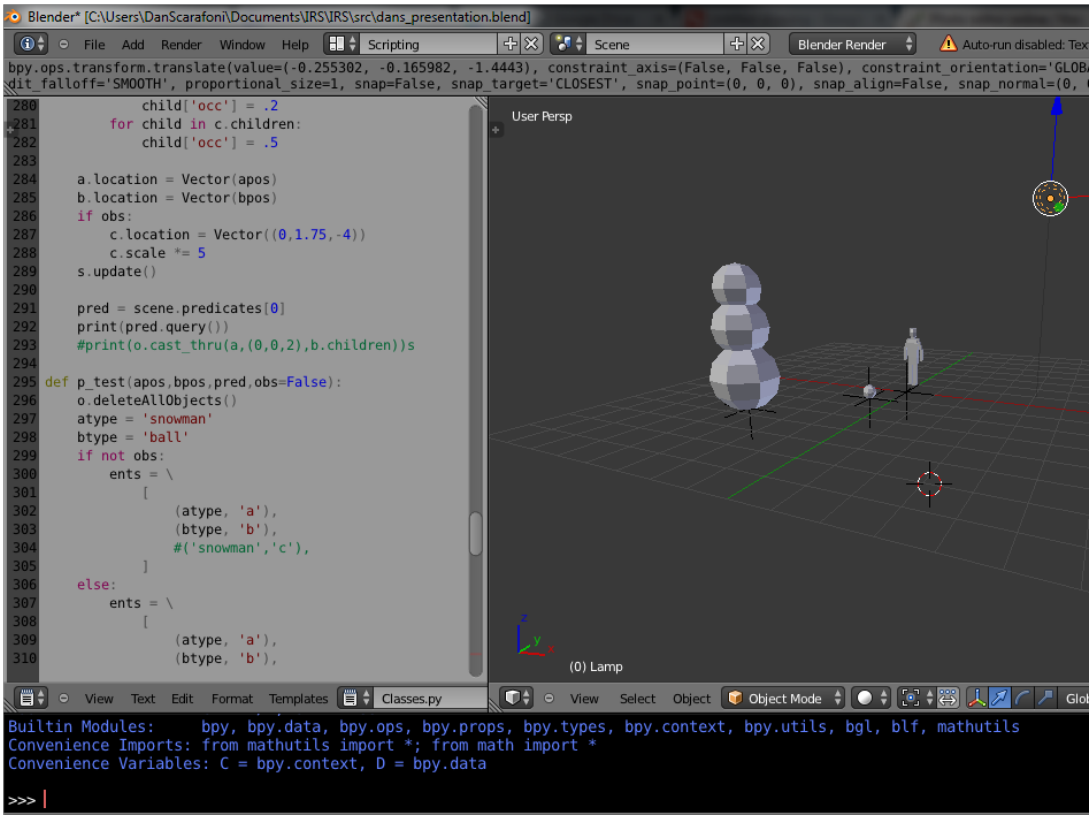

Figure 4.2: Given the above predicates, the IMS is able to generate an appropriate scene.

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	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.

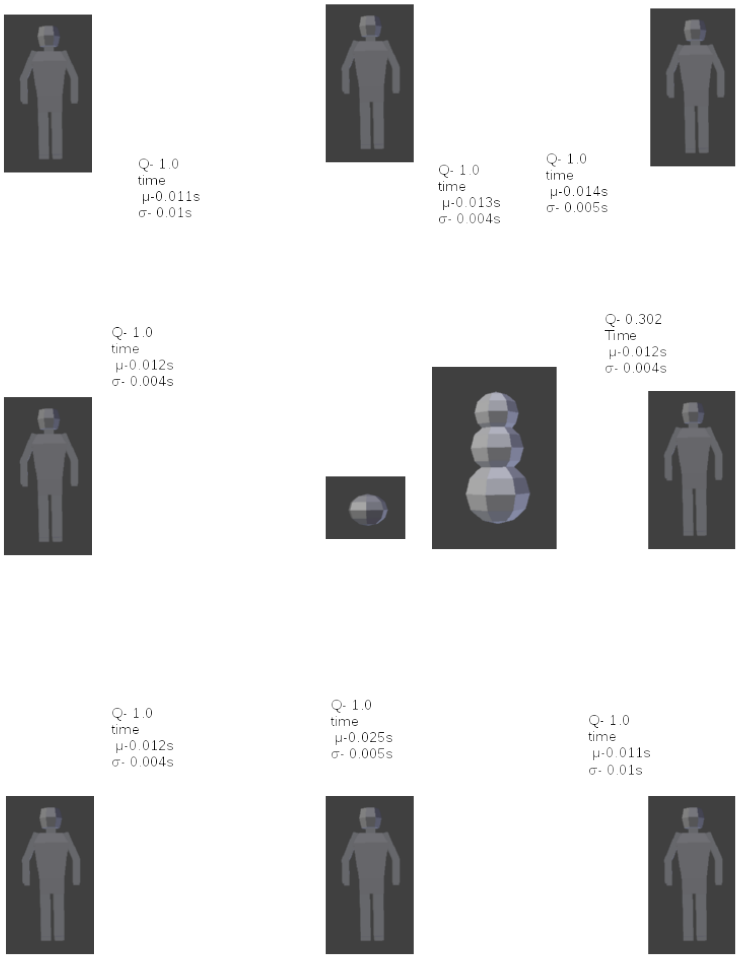

Figure 4.3: In a variety of locations, IMS is able to gauge visibility given the presence of obscuring objects. The relative locations of the snowman, ball, and person represent their location in the testing on the xy plane

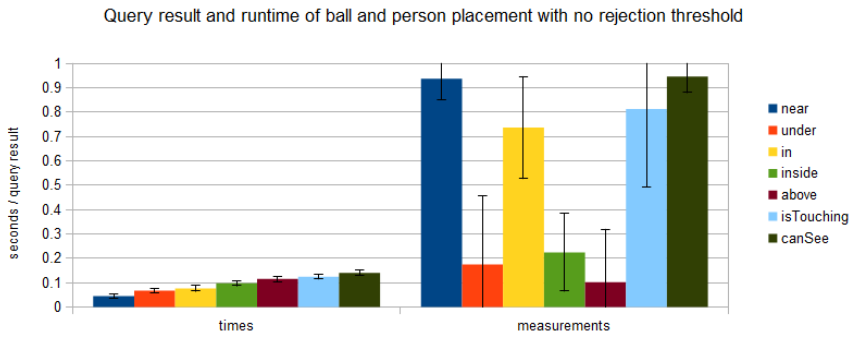

Figure 4.4: Placement with no rejection sampling. `Near`, `isTouching`, `canSee`, and `in` all show accurate placement, while others do not. Considerable variation is present amongst all predicates.

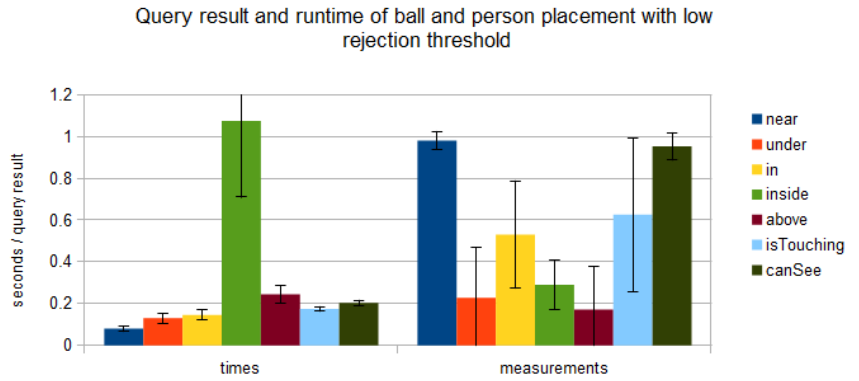

Figure 4.5: Placement with low rejection sampling. Slight but not significant improvements are shown across all predicates, as well as a dramatic increase in run-time for the `inside` predicate.

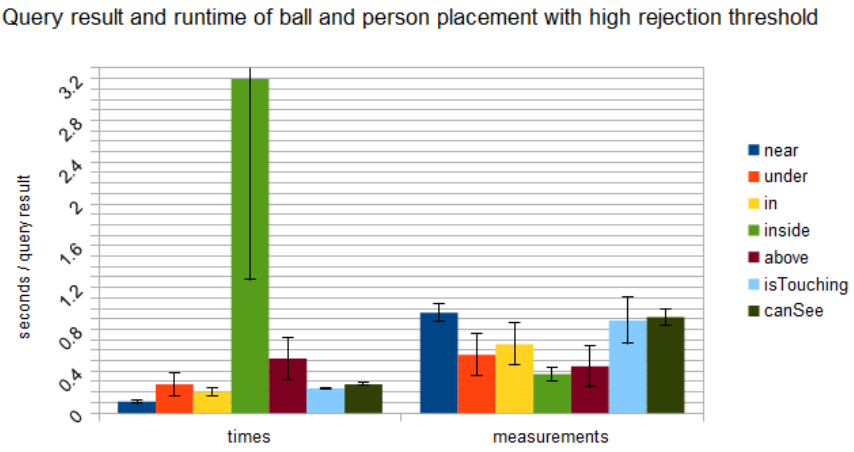

Figure 4.6: Placement with high rejection sampling. Improvements are noticed in placement accuracy, though the run-time of the near predication increases even more dramatically then before.

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