Extracting Implicit Knowledge from Text

Benjamin D. Van Durme

March 31, 2014

Benjamin David Van Durme was born in Dansville, New York on November 13th, 1979. He began studies at the University of Rochester in 1997, graduating in 2001 with a Bachelor of Arts degree in the area of Cognitive Science and a Bachelor of Science degree in the area of Computer Science. From 2002 to 2004, he attended Carnegie Mellon University, and graduated with a Master of Science degree in Language Technologies. Benjamin returned to the University of Rochester in the Fall of 2004, pursuing research in the subjects of Computer Science and Linguistics, under the direction of Professor Lenhart K. Schubert. He received the Master of Science degree in Computer Science from the University of Rochester in 2006. During the Summer of 2006, as well as 2007, Benjamin performed research at Google Inc., under the direction of Marius Paşca.

To my committee as a whole, William Cohen, Gregory Carlson, Daniel Gildea, and Lenhart K. Schubert: thank you for your advice and guidance.

Chapter **??** is the result of extensive discussions with Len Schubert on a variety of semantic phenomena. Chapter **??** derives from joint work with Len Schubert [**?**]. Chapter **??** is the result of joint work with Ting Qian and Len Schubert [**?**]. Material from Chapter **??** is based on collaboration with Marius Paşca while at Google Inc. [**?**]. Chapter **??** is based on joint work with Phillip Michalak and Len Schubert [**?**]. Chapter **??** is based on work performed with Dan Gildea [**?**].

This material is based upon research supported by National Science Foundations awards, IIS-0328849 entitled "Deriving General World Knowledge from Texts by Abstraction of Logical Forms", IIS-0535105 entitled "Knowledge Representation and Reasoning Mechanisms for Explicitly Self-Aware Communicative Agents" and CCF–0910415 entitled "RI: Small: General Knowledge Bootstrapping from Text", in addition to a University of Rochester Provost's Multidisciplinary Award (2008), entitled "Computational Psycholinguistics: Integrating Computational and Behavioral Methods to Study Human Language Processing". Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of above named organizations.

Abstract

The everyday intelligence of both humans and machines relies on a large store of background, or common-sense, knowledge. That such a knowledge base is not yet available to machines helps partially explain the community's inability to provide society with the sort of synthetic intelligence described by futurists such as Turing, or Asimov.

In response, there have emerged a variety of methods for automated *Knowledge Acquisition (KA)* that are now being actively explored. Here I consider the extraction of knowledge that is conveyed *implicitly*, both within everyday texts and queries posed to internet search engines. Through recognizing certain forms of existential predicative patterns, and abstracting from these to more strongly quantifiable statements, I show that a significant amount of general knowledge can be gleaned based on how we talk about the world. I provide experimental results both for the direct extraction and strengthening of such knowledge, and for the automatic acquisition of supporting resources for this task.

In addition, I draw attention to the relationship between automatically acquired background knowledge and natural language *generic* sentences. Humans use generics when they wish to directly assert the same sorts of "rules of the world" that are of concern to the KA community. And yet, there has been little recognition in applied circles that decades of work from formal linguistic semantics may have a role to play in the representation, and perhaps even the acquisition, of common knowledge.

Contents

List of Tables

List of Figures

Commonsense reasoning is the field of artificial intelligence dedicated to simulating and replicating high level intuition in human beings. Because many higher level human reasoning processes involve knowledge (intuitive or otherwise) about three-dimensional objects, scenes, and situations, and because significant evidence exists indicating that human minds utilize simulations in three-dimensions when thinking, it follows that reasoning in spatial environments is an intuitive and important part of artificial intelligence [**?**].

Dr. Lenhart Schubert's EPILOG project is a natural language processing system which is capable of symbolic reasoning of English utterances. The system has shown success in a number of tests, however, does not have a system for reasoning in in three dimensions [**?**]. Although the system could theoretically reason as such, the act of doing so would require an extensive library of ad-hoc information and costly computation. For example, consider the phrase it is unwise to play leap-frog with a unicorn. Reasoning about this phrase requires knowledge of the anatomy of both a human and unicorn, the basics of the game leap-frog (i.e. one entity moves from the anterior to posterior of the other, usually passing over the head), and that sharp object cause pain when in contact with certain body parts. Rather than hand-code all of these (and more) properties into the system, it makes much more sense to utilize a 3D simulation, where many of these properties are ingrained into the framework.

Luckily, EPILOG has a built in framework for integrating so-called specialists. These programs are additions to the EPILOG framework which are able to quickly and efficiently solve logical lemmas over a subset of the overall domain of the larger system [**?**, **?**, **?**] As a means of supplementing and improve the reasoning capabilities of the EPILOG project, and to bring it closer to its goal of accurate simulation of human thought, our group of undergraduates chose to create a specialist for reasoning in three dimensions (subbed the Image Reasoning Specialist, name pending). Our project began over the summer and has continued into this semester. With our combined skills and contributions, we have been able to create and implement a system which can successfully place and query entities over a small library of objects and predications.

McCarthy was the first to formally identify the concept 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 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 cracking an egg in a bowl 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. It is clear that common sense capabilities are of great use for large variety of AI problems.

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.

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 [?, ?].

In cognitive science, it has been 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. 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 is essentially to specialist construction. Schubert et al successfully implemented specialists for EPILOG for time, color, and part relationships [**?**, **?**, **?**].

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).

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.

Object 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, as 2D approaches were also considered, but possessed limited support for modeling and manipulating three-dimensional environments. With Blender's existing 3D support and extensive library utilities for 3D object management, the task of developing a simulation environment is eased considerably. explain 2d options and why they were bad

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 Mental Imagistic System, "" exists as a series of Python files and a database of objects which interface with Blender. The source folder contains three Python files:

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

A single directory, ObjectData, is also included.

0.0.1 High Level Software

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, and 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.

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 stands for 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 predMethods.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 obutils.py. obutils.py consists of various methods for operating on Blender objects, and is used by both predMethods.py and Classes.py.

show why this ontology is good: what inspiration did we get it from?

0.0.2 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 , 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 0.0.2 illustrates the flow of information and interaction between components.

r0.5

[width=0.48]figures/mirs$_{f}$*low.png* The basic flow of information in the .

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 (for example, a person's "arm" may not have a corresponding mesh, while their "forearm" and "upper arm" do), and finally bounding box coordinates for the object.

Periodic updates, predominantly introduction so new objects and predications, as well as changes to existing objects and predications, are sent to the specialist by EPILOG. more stuff on objects, changes recently, etc...

There were several glitches in the system that required resolution. First, 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.

0.0.3 Predications

My particular work has focused on the construction of the predicate class, as well as the helper methods in obutils.py, and management and organization of the program files.

— p5cm — p10cm —

obutils.py

Methods	Descriptions
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 object A from pt, used in repeated ray casting experiments
rayCast(Object a,Vector b)	Shoots a ray from the center of object 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 highest vertex in the object (in the dimension (x,y,z) specified)
Returns the highest vertex in the object (in the dimension (x,y,z) specified)	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.

def glob2Loc(Vector pt,object obj) Returns point pt in object 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

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.

— p2.5cm — p2.5cm — p7cm — p4cm —

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. obutils.maxDim() obutils.closestPoints() obutils.isArbitrarilyLarge()

---

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 obutils.alignMeasure() obutils.getIntersection() obutils.getVolume() obutils.deleteObject()

---

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. obutils.getVolume() obutils.deleteObject()

---

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 obutils.getIntersection() obutils.getVolume() obutils.deleteObject()

---

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. obutils.alignMeasure() obutils.getIntersection() obutils.getVolume() obutils.deleteObject()

---

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. obutils.distance()

---

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 obutils.cast_thru()

By far, the largest and most complicated predicate was the canSee(A,B) predication. The implementation of this function relied heavily on ray-casting (see above) 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 1 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 case would have traveled if there had not been an object in the way acting as an interrupting agent.

[width=0.48]figures/vision1.png

Figure 1: 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 2 shows the idealized version of the function of this implementation of the function.

[width=0.48]figures/vision2.png

Figure 2: A second implementation of ray-casting

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 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 obutils.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.

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 [width=0.48]figures/vision3.png

Figure 3: The nudge addition to the naive model

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 4 illustrates this.

[width=0.48]figures/vision4.png

Figure 4: 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 5 illustrates this model.

[width=0.48]figures/vision5.png

Figure 5: The second iteration of the canSee(A,B) algorithm

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 6, 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 on how close they were. In this interpretation, casts to close-

[width=0.48]figures/vision6.png

Figure 6: The majority of the casts are blocked by C even though B is visible.

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 7 shows how the new method solves the problems of the old.

[width=0.48]figures/vision7.png

Figure 7: 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. 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 obutils.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 obutils.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.

0.0.4 Placement methods

Place() returns the areas in the Blender scene where the predicate holds true 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, our 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. 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. 8 provides a visualization for this method.

[width=0.48]figures/square$_{placement.png}$

Figure 8: 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. 9 illustrates the issues that the system raised.

[width=0.48]figures/vision$_placement_theoretical.png$

Figure 9: 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. Eric Bigelow and I have worked together 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. 0.0.4 below provides an illustration.

[width=0.48]figures/grid$_placement.png$

Figure 10: 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 by myself 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. 11 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, it is unclear whether or not this methodology is computationally feasible. Aside from these issues, however, 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. We hope that future work will lead to a complete and functional program capable of supporting robust array of symbolic and 3D reasoning.

[width=0.48]figures/rejection$_{s}ampling_{p}lacement.png$

Figure 11: Rejection sampling was also considered as a placement mechanism.

Ultimately, however, these proposed systems turned out to be unnecessary. The additional overhead of all of these algorithms were a concern, and was not compensated for by an increased

enumerate algorithms of placement algorithms

Our project's success in creating and querying 3D scenes demonstrates both its usability as a situation for reasoning in 3D spaces, but the expressive power of a system in general. Our program was able to quickly and efficiently manage and query over a small library of objects and predicates.

As shown in Figure **??**, 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 system functions well as a specialist for a larger system. In this respect, it is a very well performing program. The system is black-boxed to outside input, and as such can be used for more than just EPILOG. The program worked into the Cornell Cup's Haptek team, where it was used to simulate a person navigating a virtual maze.

The greatest flaw in the system, and the biggest boundary to continued expansion of the project, is the lack of extensibility of the predicate and object database. Both Objects 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 quite difficult. 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.

talk about the importance of this project in the context of future endeavors