Language-Based Procedural Modeling for Randomized Scene Construction

Andy Scott
Chair: Dr. Hans-Peter Bischof
Reader: Dr. Carl Salvaggio
Observer: Rolando Raqueño

Rochester Institute of Technology
May 3, 2011

Contents

1 Problem Statement **4**
1.1 Background 4
1.2 Project Goal 4

2 Natural Language **5**
2.1 Scope and Strategy 5
2.2 Grammar Rules 6
2.3 Examples 7
2.4 Natural-ness 8

3 Project Models **8**
3.1 Functional Model 9
3.1.1 Use Cases 10
3.2 Data Model 10
3.2.1 Encapsulation Philosophy 11
3.2.2 Taxonomy of Scene Features 11
3.2.3 Session Data Structure 13
3.3 GUI Map 15

4 XML Representation **16**
4.1 Rationale 16
4.2 Structure 17
4.3 Doctype 17
4.4 Example 18

5 Environment **23**
5.1 Rationale for Blender 23
5.1.1 Robust Rendering Context 23
5.1.2 Extensibility via Python 24
5.1.3 GUI Toolkit 24
5.1.4 Deployment 26
5.2 Limitations 26

6 Implementation **27**
6.1 Data Flow 27
6.2 Grammar Parsing 29
6.3 XML Integration 31
6.4 GUI Scheme 33
6.4.1 GUI Stack 33
6.4.2 View Implementation Logistics 34
6.4.3 Clause Construction 34
6.4.4 Program Data View 36
6.4.5 Building-Block Views 37
6.5 Scene-Building Mechanics 40
6.5.1 Polygon Wrapping 40
6.5.2 Positioning Algorithms 44
6.5.3 Collision Avoidance 46
6.5.4 Randomization Integration 47
6.5.5 Object Recycling 48

7 Results **48**
7.1 Assessing the Natural Language . . . 48
7.2 Benefits of Data Structure . . . 50

8 Future Directions **52**
8.1 Terrain Integration . . . 52
8.2 Integration of Variables into the Natural Language . . . 53
8.3 Standard Geometry Library . . . 53
8.4 Street Networks . . . 54
8.5 Speech to Scene . . . 54

1 Problem Statement

1.1 Background

A portion of the goal of a research project belonging to the Imaging Science department at RIT is to develop a system capable of receiving a textual description of an outdoor, large-scale scene and producing a corresponding 3-dimensional rendering of it (I was hired to work on this endeavor beginning in the summer of 2010 as a research assistant). The benefits of such a system are in the ease and speed with which one could create scene renderings. The premise of the project as a whole involves running thermal-based simulations on sets of test scenes, which require 3-D renderings of these scenes as a basis. In the context of the research project, the intent of these simulations is associated with predictive modeling scenarios in connection with nuclear non-proliferation efforts (the project is funded by the nuclear security arm of the U.S. Department of Energy). However, the benefits of the rapid creation of such test scenes are not limited to the prior context. Conducting simulations on test scenes is a common process in several research efforts in the Imaging Science department alone. To that end, simulations are conducted by a program called DIRSIG, which implements ray-tracing techniques to ascertain the effects of different conditions on input scenery. Building scene renderings to use for DIRSIG simulations has been a tedious process which has consumed large amounts of man-hours. The ability to describe a scene in plain English should be applicable in those settings as well as the DOE project.

1.2 Project Goal

The specifications for the project pertaining to the text-scene translation requirement are vague. As it was explained to me, the vision was that of a user writing a paragraph of text which described the layout and features of a scene, and then using a software tool to translate it into an actual rendering. The basic premise is that a potential user would have a general, perhaps even vague idea of what features should be included in a scene and how they should be laid out - but either doesn't care about or doesn't have time to flesh out the details. For example, suppose Bob the user wants to model a typical college campus. Bob knows that the scene should include a cluster of academic buildings, a cluster of dorms nearby, some athletic fields, a large parking area, and some trees here and there. The role of the software tool would be to accept Bob's textual description of the campus and flesh out the details in a 3-D rendering.

The implication, from a development standpoint, is that the envisioned software would be able to understand the meaning behind any English-language construct and ascertain its contribution toward the specification of a feature of the scene. Such a system would necessitate a level of AI and machine-learning infrastructure rivaling that of IBM's "Watson". Given the one-year time frame which I was invited to work on the project, I decided it would be more prudent to design a language which would be natural-sounding and intuitive for a user to write, but for which it would be feasible to write a parser for (I will heretofore refer to this as my "natural language"). Thus, the specifications I have since been working under have been to develop a software tool which parses a script adhering to the natural language and automatically produces a rendering of the scene as per its natural-language specification.

As I proceeded, I added two major specifications aimed at increasing the set of potential use-cases: The first being an XML-based alternative to the natural-language to heighten the degree of precision when desired. An inherent shortcoming in my natural-language schema is the inability to specify positional and rotational characteristics in exact numerical terms, and an alternative XML-based specification is a much better vehicle to do that when the user wants to get technical.

Another benefit to the XML is portability: The exact specifications of any scenery a user is working on may be saved and loaded as an XML file wherever and whenever. The second additional specification is to develop a way to randomize certain features of a scene. I feel this feature will be useful in many contexts of use. The impetus behind conducting any simulation is the presence of variables for which differing permutations will yield uncertain results. If the physical features of a scene are subject to variation in a simulation-based setting, the user would most certainly appreciate the ability to define the parameters and scope of the variations, then letting the software produce random permutations; instead of implementing the randomizations by hand. For example, say the user would like to model a scene feature that is inherently dynamic, like a parking lot. For each space, it is uncertain whether a vehicle will occupy it. If a vehicle does happen to occupy it, it is uncertain what make/model it happens to be, whether it pulled into the space or backed in, and even how well the driver was able to center and orient it within the space. These are all variables that a user should be able to convey in terms of a randomization scheme which would be applied in the context of the parking space feature.

In summary, the goal of my Masters project was to implement a scene-building tool capable of processing a natural-language script into a scene rendering, applying randomization schemes to scene features, and reading/writing all program data in XML.

2 Natural Language

2.1 Scope and Strategy

The purpose of the "natural language", as I'm calling it, is to give the user a textual way to specify the features and layout of a scene in terms that feel natural, but are restrictive enough to write a parser for. In general, my strategy was that the language should allow a user to name features, specify their location, and specify their rotation in the context of the parent scene. There are several ways users may want to specify a location. They may want to specify it in a regional sense (i.e., the "north region") or they may want to specify it in a relational sense (i.e., "north of the main building"). The same applies for rotation: Users may want to say that "the car is facing east", or that "the car is facing the warehouse". Since the locational and rotational attributes of a feature may be specified in a relational context, this means that every feature should have a name that can be referenced as the base of the locational/rotational specification. This name should be unique in the context of the particular scene.

It occurred to me that when people express something's location as it relates to something else, they tend to think in polar terms (as opposed to Cartesian). If you asked someone where the city of Buffalo is in relation to the city of Rochester, the answer you'd get is that Buffalo is about 65 miles west-southwest of Rochester, which would be the polar coordinate specification. You might think it strange if the reply was given in Cartesian coordinates, that Buffalo is 60 miles west and 20 miles south of Rochester. Both answers are geographically correct, but most people find it more natural to express location in a direction and a distance rather than by magnitudes of two basis vectors. It was for this reason that I decided to structure the language to accept the specification of a location in polar terms (direction and distance). Direction is easily specified by compass keywords, but the notion of distance poses a problem in this setting: I can assume no unit-standard of measurement. This is because the software tool needs to be capable of dealing with any potential scale of scenes and the objects within. For this reason, distance must be expressed in general terms in the natural language, with terms like "near" and "far". Unit of measurement must be kept arbitrary in the context of the natural language if it is to be able to operate in any potential scale.

2.2 Grammar Rules

The set of grammar rules I devised to define the language are actually quite short. Fig. 1 is the full BNF specification of the language:

```
            SCRIPT -> SCENE*
             SCENE -> <scene name> { CLAUSE* }
            CLAUSE -> ENTITY [POSITIONAL_DATA] [ROTATIONAL_DATA] [, apply <scenario>]
            ENTITY -> [[GROUP of ]<integer>] <feature>[s] [as "<name>"]
             GROUP -> [tight | loose] (cluster | square | COMPASS/COMPASS row | <ref>)
   POSITIONAL_DATA -> LOC_RELATIVE | LOC_REGIONAL
   ROTATIONAL_DATA -> facing (COMPASS | [away from] <ref>)
      LOC_RELATIVE -> (HEAD_ANDOR_DIST | DIST_HEAD COMPASS of) <ref>
   HEAD_ANDOR_DIST -> DIST [and HEADING] | HEADING [and DIST]
         DIST_HEAD -> ([to [the]] [very] (near|far)) | ([very] (near|far) [to [the]])
              DIST -> ([very] (near | far from)) | inside
           HEADING -> [to the] COMPASS of
      LOC_REGIONAL -> in the [[very] (near | far)] COMPASS[ern] [region]
           COMPASS -> COMPASS_FULL[COMPASS_FULL] | COMPASS_ABBREV[COMPASS_ABBREV]
      COMPASS_FULL -> (N|n)orth | (S|s)outh | (E|e)ast | (W|w)est
    COMPASS_ABREV -> N | n | S | s | E | e | W | w
```

Figure 1: BNF Specification of natural langauge

The grammar is designed to allow for the optional specification of positional and rotational data to a named feature (entity) of the scene. Provisions are made for specifying this data both in a regional and relational context. An entity may be either a single feature or a grouping of features. Provisions are made for specifying a group as a cluster, a row, or a custom formation (which will be specified by the user). All directions are specified in compass-based terms, i.e. "north", "south", "northwest", etc. These may be abbreviated to "N", "S", "NW", respectively (case non-sensitive). While it is not an explicit requirement to name an entity, the writer must understand that entities whose positional or rotational specifications are defined relationally to another entity must refer to the latter entity by name. Any un-named entity that appears in the script will be assigned the name "feature_x", where "feature" is the definitive name of the feature being used and "x" is the first integer for which "feature_x" is a unique name in the context of the scene. Thus, failing to name an entity that is relied on as a relational base may result in the relation being unresolved (if the writer uses the wrong name to reference the base entity).

I have developed the grammar to be unambiguous, meaning that there is at most one token stream that can match any given input script. This made it relatively easy to design and implement an LR(1)-style parser for the language.

2.3 Examples

The full intent and style of the language may best be shown with some examples. The first script describes a very simple scene comprised of four features: A warehouse, parking lot, a shipping yard, and a cluster of trees:

```
Industrial_Park {
warehouse_bldg as "warehouse"
Shipping_Yard as "shipping" to the NE of and near warehouse
Parking_Lot as "parking" to the very near south of warehouse facing warehouse
tight cluster of 10 trees as "forest" in the far western region
}
```

With respect to the grammar, each line between the braces is a CLAUSE. The first clause is simply a fulfillment of an ENTITY (positional data, rotational data, and scenario application have been forgone). This entity references the feature "warehouse_bldg" and names it as "warehouse". In the second clause, we're referencing a feature called "shipping_yard" and naming it "shipping". Notice that we're also specifying positional data, defined by the rule POSITIONAL_DATA, which is really just an option between position defined in a relative context and position defined in a regional context. For this clause, we elect to define the position in the relative context, given by the rule LOC_RELATIVE, which gives several ways to define the heading and distance data associated with the relatively-specified location. In this clause, it is tokenized as "HEADING and DIST", since the heading specification "to the NE of" appears before the distance specification "near". The final piece of the LOC_RELATIVE rule is the required reference to the relational base, which in this case is the warehouse. Remember that the reason for naming the warehouse_bldg is so we could reference it by name in a relational position or rotation definition. We would not want to reference it by the name of the feature (warehouse_bldg) because we may want to use another instance of a warehouse_bldg in the scene. Naming each entity used in the scene allows us to maintain a unique reference to each feature, which avoids ambiguities. The structure of the grammar should be becoming clear: The third clause names a feature "parking_lot" as "parking", and specifies a relational locational scheme followed by a relational rotational scheme - In both schemes the base reference is the warehouse, the location is to the very near south of it, and the rotation is such that the parking lot is facing it. The fourth clause is an example of the grouped type of entity. Notice that the first portion of the clause, "tight cluster", is spawned by the rule GROUP. The GROUP rule is used to define the formation that the group of features should take. The option to describe it as a "tight" or "loose" group refers to the spacing that should exist between members of the group. A tight group spaces its members close together, a loose group; far apart, and using neither term implies that a moderate degree of spacing shall be used. It is important to note that the name "forest" is being given to the group itself, and in the context of the parent scene, the entire group is treated as a single entity. Subsequent entities in the same scene may not reference individual trees as a relational base - they must instead reference the forest itself. The rest of the fourth clause, "in the far western region", specifies a regional location (given by the LOC_REGIONAL rule). The forest is specified as being in the western area of the scene, and far from the center.

Let's take a look at one more example:

```
Parking_Lot {
N/S row of 10 parking_spaces as "row1" facing east
N/S row of 10 parking_spaces as "row2" to the near W of row1 facing W
N/S row of 10 parking_spaces as "row3" to the very near W of row2 facing E
N/S row of 10 parking_spaces as "row4" to the near W of row3 facing W
}
```

All four clauses are an example of the row type group specification. As you can see from the GROUP rule, the "row" designation must be preceded by an instance of COMPASS/COMPASS, i.e. N/S. It seemed to me that the best way to specify the geographic characteristics of a row of entities was to describe it as stretching from one direction to another, i.e. the row "stretches from north to south", or "it's a north/south row".

2.4 Natural-ness

On the spectrum between plain english and most programming languages, my natural language is somewhere in-between. The clauses are certainly written in broken english, but should be understandable to any english-speaking individual. Since the scope of the language is limited to the specification of the features and layout of a scene, my strategy was to accept the most natural-sounding english constructs *within* this scope. For example, if you were standing in a field looking north, and you saw a barn on the horizon, what would be the most natural way to describe the location of the barn as it relates to you? Obviously there's no right answer, but chances are you'd say something like "the barn is to the far north of me", or "the barn is far to the north of me", or "the barn is far from and to the north of me". The constructs employed to describe the location of the barn, in all of those fragments, are acceptable in the language (inspect the LOC_RELATIVE rule).

Naming an entity in this language may feel a little unnatural. To say "warehouse_bldg as warehouse" as the entity specification has more of a mechanical feel, but remember that explicitly naming an entity is optional. It is my hope that the writer would feel an intrinsic desire to assign a name to an entity for which they know is going to be used as a relational base to a sibling entity: "I'm going to place subsequent objects around this instance of warehouse_bldg, so I better name it something I can remember to reference it by". In this case, naming the entity would actually feel natural to the user.

Admittedly, the amount of keywords which exist in the grammar may be restrictive. For example, consider the two keywords associated with the specification of the spacing of a group: "tight" and "loose". Is "loose" the first adjective that comes to your mind when you want to describe a group with a large degree of spacing between its members? Maybe, but perhaps the word that comes to your mind is "sparse", or "scattered". You could grab a thesaurus, and for each keyword find dozens of alternatives which a writer may feel more inclined to use than the given keyword. My feeling is that the particular set of keywords in use in the grammar is small, and easy to remember. So instead of including a multitude of different keywords in the grammar, I am more inclined to simply let the set of keywords used in the grammar be the learning curve of the language. After using the language for a short while, it should start feeling natural for the writer to use.

3 Project Models

I drafted three models to describe the major aspects of the software tool. The first is a functional model, which describes the major conceptual facets of the tool's functionality. The second is a data model, serving as a specification of the structure of the tool's session data. The last is a GUI Map, which charts the network of GUI views the tool can shift between in order to facilitate the execution of all the potential use cases.

3.1 Functional Model

Fig. 2 is a diagram of the functional model, of which there are six facets.

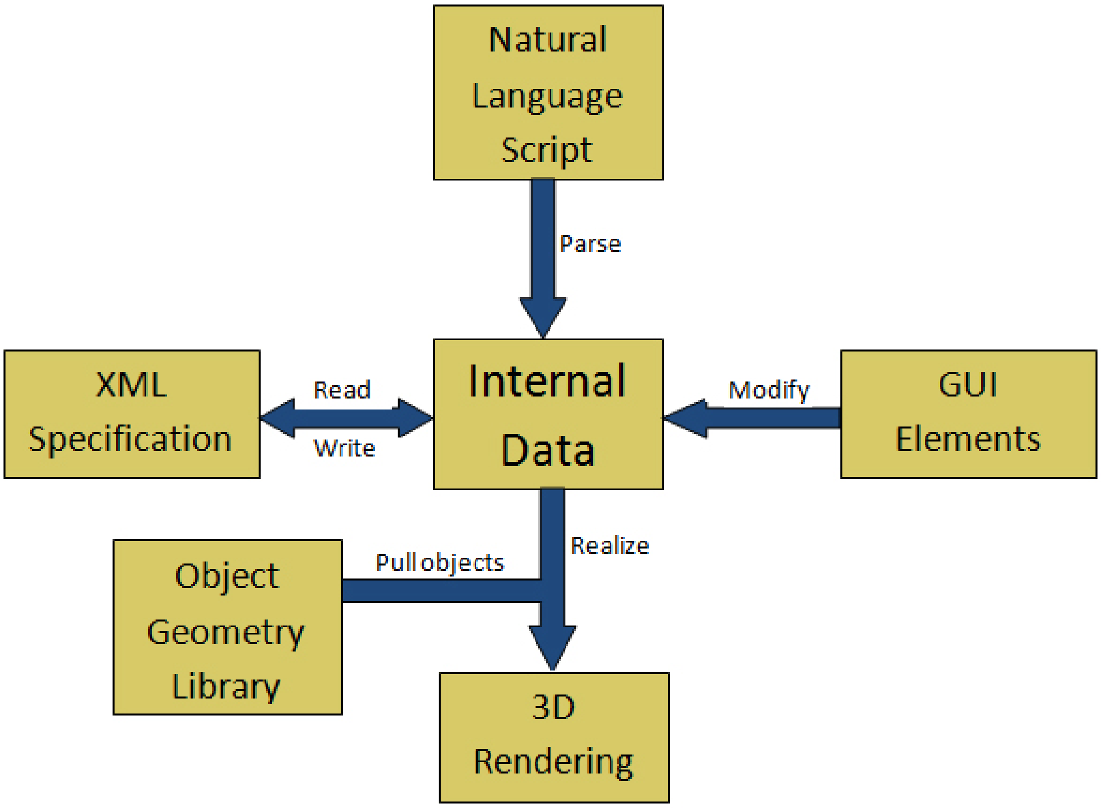


Figure 2: Functional Model of Tool

Internal Data, or ”session data”, is a representation of the stored data corresponding to the user’s session, the structure of which is defined in the second model. *Natural Language Script* is a conceptual representation of textual input which adheres to the grammar of the natural language. *XML Specification* represents the XML version of session data. *GUI Elements* simply refers to the GUI components of the tool which facilitate the creation and modification of session datum. This includes GUI schemes to assist the user in writing clauses which conform to the natural-language grammar and schemes to assist the user in developing other types of scene datum such as randomization schemes. *3D Rendering* refers to the rendering context in which scenes appear in their physical forms. *Object Geometry Library* is a conceptualization of the repository which contains the individual geometry files needed to render the scenery. The relationships and correspondences between the facets of the functional model are represented by the arrows: The *Parse* arrow is a reference to the parser which tokenizes and processes natural language texts. The parser accepts this text as input and creates corresponding scene data as per the structure of the data model. The *Read/Write* arrow is an indication of the correspondence between internal session data and its XML alternative. The implication here is of the existence of XML parsing and processing routines, which shall be discussed further in the following chapter. For now it is sufficient to know that there exists an XML representation of session data such that there is a 1-1 correspondence between a session data in its internal form and its XML specification; and that the tool is capable of converting between the two. The *modify* arrow simply indicates that there are GUI features which allow a user to create, modify, and delete datum within their session data. The *Realize/Pull Objects* arrow is a reference to the scene rendering function of the tool. Simply put, scenes specified in session data are realized in the 3-D rendering context using object geometries contained in the geometry repository. The basic dynamic of the functional model is the predication of an instance of session data, with several avenues for its modification.

3.1.1 Use Cases

The functional model hints at the potential spectrum of use cases of the tool. If a user simply wanted to describe a scene in natural script form and have it be realized in a rendering (as per the original software specification), they would just parse their text and execute a rendering of their scene. But suppose they are concerned over the layout of a specific feature, and the natural script was unable to deliver the degree of precision they require for the placement of this feature. The XML specification offers exact numerical precision associated with the locational and rotational aspects of every entity of the scene, whereas the natural language offers only a limited set of keywords to express these characteristics. In this case, the user could parse their natural-language script into session data, and then execute an XML write-out of the session data. They could inspect the XML file and make whatever modifications they feel are necessary to achieve the desired degree of precision associated with the placement scheme of the features of their scenes.

The 1-1 correspondence between XML data and session data means that an XML file can be treated as a portable instance of session data. An XML specification of program data could be saved and passed around amongst users for collaborative purposes. A user could receive an XML file from a colleague and instantly re-create the session their colleague was working on. The ability to read in an XML file into session data also means that a user could simply write an XML file from scratch, which is easy enough to do once they have a grasp of the data structures.

The GUI elements referenced in the functional model further expand the use cases. Specifically, the tool includes a GUI scheme designed to assist the user in writing clauses conforming to the natural-language grammar rules. This is a GUI view that includes a series of widgets the user may poke around with. As they do this, a clause in the bottom of the view automatically updates itself based on the current states of the widgets. The purpose is to familiarize a newer user to the structure of the language, or simply as an alternative to having to write the clauses in text. Since the view operates on one clause at a time, it also allows a scene to be built incrementally, rather than all-at-once. If the user would rather work on one feature at a time, they could use this GUI view to do so. The tool also includes GUI views to facilitate the creation/modification/deletion of datum not related to feature positioning, such as randomization schemes and custom grouping formations. This is presented in GUI form as an alternative to doing it in an XML specification.

3.2 Data Model

From a high-level standpoint, the tool's session data maintains a collection of scene specifications and a collection of available features to use. Recall from the natural-language grammar that each clause of a scene specification references a feature to use as an entity. You may have guessed that the implication was that the tool maintains a list of features available for use in any scene the user designs. When looking at the clause examples, you may also have noticed that the referenced features seemed to be either singular objects, like a "warehouse_bldg", or what appeared to be an area or zone, like a "Parking_Lot" and "Shipping_Yard". These would actually be a type of feature which I call a "sub-scene". The name implies exactly what you'd expect - that a scene can be encapsulated as a feature and be inserted into another scene as such.

3.2.1 Encapsulation Philosophy

The theme of the data structure is that of encapsulation - the ability to define a scene and use it as a sub-scene in other contexts. Recall the two example scene specifications in the previous chapter. The first was Industrial_Park. The third clause in this scene referenced a feature named Parking_lot, which was the name of the second scene. Indeed, the entire Parking_Lot scene is being used as a sub-scene entity in the Industrial_Park scene. The benefit for the user is the ability to organize a scene in whichever way they want: In the Industrial_Park scene, the user has elected to treat the parking area as an abstract feature instead of drafting clauses describing the positions of each potential vehicle that could comprise the parking lot. This is a good way to keep scene specifications organized and uncluttered. An additional benefit is the ability to instantiate a general-purpose scene (like the parking lot) in a multitude of different parent scenes in the same session just by referencing it as a feature.

3.2.2 Taxonomy of Scene Features

Fig. 3 models the structure of a scene entity as it appears in session data:

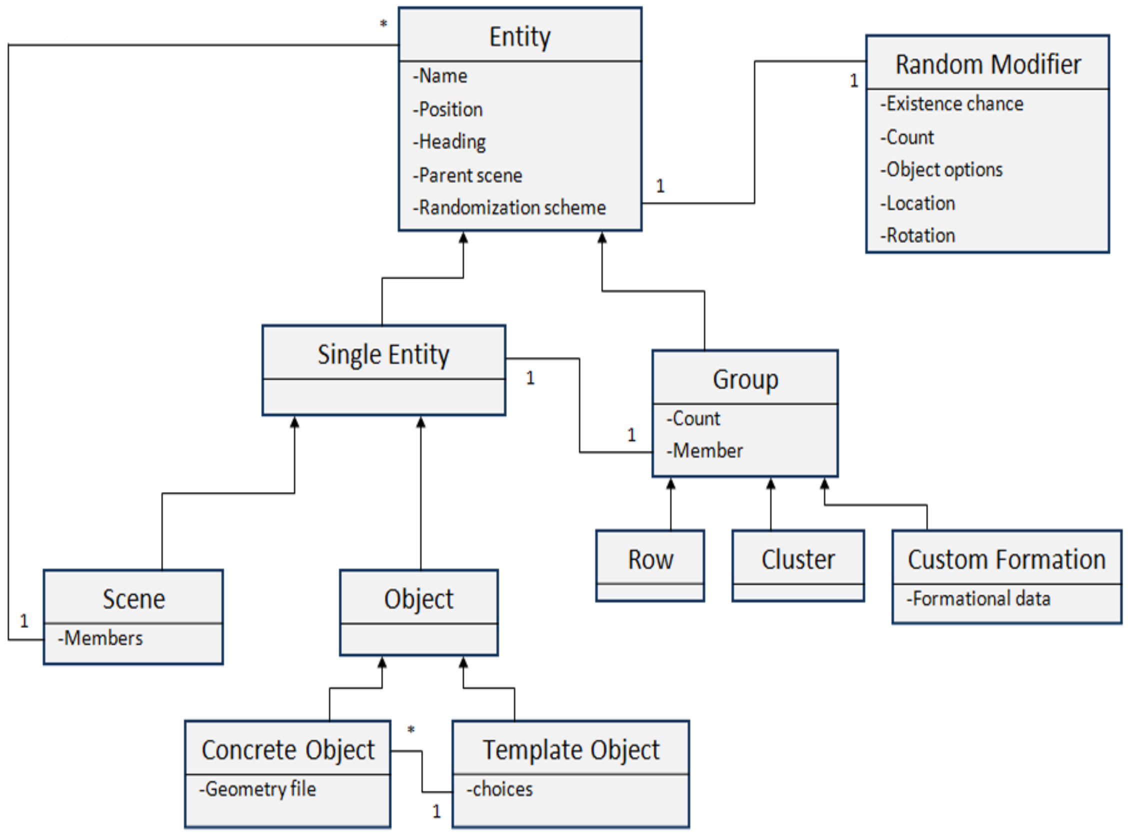


Figure 3: Data Structure of a scene "entity"

The encapsulation theme is self-evident in the UML-style model. A scene is defined in the existential sense as a collection of entity members. The base classification of an entity is as a single entity or a group. A single entity may either be a scene (reference to a sub-scene) or a single object. The following is a more technical definition of each feature of the data model:

- Entity: A scene element. May be an object, scene, or group of either.
  - Name: Must be unique in the context of the parent scene, as sibling elements may reference it as part of a relational position/rotation scheme.
  - Position: Desired position for the entity in the parent scene. May be specified regionally or relative to sibling elements.
  - Heading: Desired intrinsic rotation of the entity in the x-y (ground) plane. May be specified relative to or independently of sibling objects. i.e, truck facing east or truck facing warehouse, respectively.
  - Parent scene: Reference to parent scene. May be none if it is the root scene.
  - Randomization scheme: Reference to a randomization scheme. May be none. The entity will be randomized as specified by the scheme.
- Single Entity: Subclass of Entity. May be a scene or object. Basically, not a group.
- Group: Subclass of Entity. Defined as multiple instances of a particular Single Entity subject to a formation.
  - Count: Number of instances to include. May be randomized as part of an attached randomization scheme.
  - Member: Reference to a Single Entity. Base (copied) member of the group.
- Scene: Subclass of Single Entity. Comprised of multiple entities. Can be rendered.
  - Members: List of Entities which comprise the scene.
- Object: Subclass of Single Entity. A single object arising from a ”concrete” object or a ”template” object.
- Concrete Object: Subclass of Object. A certain object, backed by a geometry file.
  - Geometry file: URI reference to geometry file registered to the object.
- Template Object: Subclass of Object. An uncertain object; specific object will be chosen at render-time. The application of a randomization scheme which contains object choices is implicit.
  - Choices: Conceptually, list of concrete objects to choose from. Actually part of implied randomization scheme.
- Row: Subclass of Group. Forms the group as a line.
- Cluster: Subclass of Group. Forms the group as a cluster.
- Custom Formation: Subclass of Group. Applies a user-specified formation to the group.

  - Formational Data: User-specified formation. Consists of list of "spots" given in cartesian or polar coordinates which map out how the members should be placed relative to each other. If the count exceeds the defined spots, the excess members will be clustered in the periphery. If the count is less than the length of the list of spots, only the first count spots will be used.

- Random Modifier: Randomization Scheme. May be applied to any entity. Contains five optional facets: Existence chance, Count, Object options, Location, and Rotation.

  - Existence chance: Probability that the entity will appear. Between 0.0 and 1.0. May be specified as "reserved", which reserves the space the entity would have been placed so no other features may occupy it.
  - Count: Gaussian-based or range-based modifier for the number of members that appear in a group. Randomization schemes containing this may only be applied to a Group entity.
  - Object options: List of objects to choose from to use when rendering this entity. Probabilities may be weighted. Randomization schemes containing this may only be applied to a Template entity.
  - Location: Randomizes the entity's position in the scene. Defined as a "shift" and/or a choice of positions. If a choice of positions is specified, it trumps any positional data specified in the entity class. A shift will apply a Gaussian-based or range-based variation to the entity's position in a specified direction.
  - Rotation: Randomizes the entities' heading. Also defined as a "shift" and/or choice. If a choice of rotations is specified, it trumps any heading data specified in the entity class. A shift will apply a Gaussian-based or range-based variation to the entity's heading in the context of its 360 degree field of rotation.

Although the structure of a randomization scheme is laid out in the data model, it is not technically part of the entity taxonomy. As stated in the model, its conceptual role is as a modifier of the carrier entity.

3.2.3 Session Data Structure

Although the UML chart may make you think that sub-scene data is existentially encapsulated as a member entity of the parent scene, it is actually reference based. That is to say, the list of member entities registered to an instance of Scene is actually a list of references to available features (be they objects or scenes). At all times, the state of the tool's session data is defined by five lists of data sub-structures, named: Scenes, Objects, Templates, Scenarios, and CustomFormations.

Scenes This is the list of scenes which have been currently specified. Each member of this list contains a reference name of the particular scene, and a list of entities. Each entity contains a name, a reference to the feature it instantiates, positional data, rotational (heading) data, and optionally a randomization scheme (either natively or by reference to a Scenario - see the "Scenarios" section below). This set of information corresponds to the structure of the CLAUSE rule from the grammar: The three latter pieces of information determine how the entity is to be placed in the context of the parent scene.

The structure of locational and rotational data registered to an entity depends on whether they're specified relationally or not. In either case, a location is saved in polar coordinates. The specified compass direction evaluates to a theta coordinate between 0 and 2*pi, and the distance coordinate is obtained from the specified distance ("very near" maps to 0.25, "near" maps to 0.5, "moderate" - or no specification - maps to 1.0, "far" maps to 2.0, and "very far" maps to 4.0). If the location was specified relationally, the relational reference will be saved. At render time, the location of the relational reference will be treated as the origin. If the location was not specified relationally, the origin will be the origin of the scene itself. If the rotational data was specified relationally (i.e. "facing" or "facing away from" the relational reference), the rotational data is saved simply as the relational reference and a Boolean indicating whether it should face it or face away from it. If the rotational data was not specified relationally (just as a compass direction), it is saved as a theta in the same way as the polar coordinate from the locational data. It may seem unsettling that the distance coordinate from the locational data is unit-less, but remember we're working with an arbitrary metric. Don't worry - the distance value saved in locational data is just one input to an algorithm that determines the exact coordinates at render-time.

At all appropriate times, the tool performs non-cyclic sub-scene reference verification, implemented by a trivial recursive routine. This makes sure that the addition of a sub-scene which relies on the parent scene as part of its construction is never allowed.

It needs to be mentioned that there is an un-advertised feature (no way to specify via GUI or natural-language as of yet, only by XML) which I call "suggestions". A "suggestion" is simply a randomization scheme that may be applied to certain member(s) of a group. Applying a randomization scheme to a group-type entity will take effect in the context of the entire group being treated as a singular, randomized feature. For example, applying a random locational shift to a group-type entity will result in the picking of one shift which is applied to each member; instead of picking a new shift for each member. If the latter behavior is desired, a suggestion could be used to "suggest" the randomization to each member. In the event that a member carries a pre-existing randomization scheme (in the case of a template object or a previous suggestion), the schemes are merged (In a merge, the existing facets of the new scheme replace the corresponding facets of the old scheme).

Concrete Objects This is a list of single-object geometries the tool currently has access to. Each record is a tuple containing the name of the object and the URI at which the associated geometry file is stored on the local file system. At rendering time, the URI is used to load the geometry when the object is requested as a feature.

Templates This is a list of "template" objects which have been currently specified. A template is really just a named instance of a randomization scheme (which must contain a choice of objects to realize itself as). Like any other type of entity, a template may be used as a feature in any scene by referencing it by name.

Scenarios This is a list of named randomization schemes, but unlike Templates, a Scenario cannot be a physical feature of a scene (a scenario's randomization scheme may NOT include the "object options" facet and thus cannot be used to derive an actual object). The rationale behind the Scenario is convenience-based: If there's a particular randomization scheme which the user wishes to use on multiple entities, it's much easier for the user to define the randomization scenario, name it, and then reference it as an "application" to the entity (recall the last part of the CLAUSE rule in the grammar with the "apply" keyword). Scenarios can also be applied to other randomization schemes. Doing so merges the two, whereby the existing facets of the applied scenario replace the corresponding facets of the applicant scheme.

Custom Formations This is a list of named custom formations which are currently defined in session data. A custom formation can be used by referencing its name. In the natural language, this name would be used as the type of the group (instead of "row" or "cluster").

3.3 GUI Map

The tool's GUI is comprised of three views: A 3-D rendering context to display the scenes, a text panel to allow the user to view/edit the natural language script (or the XML specification of session data), and a "main" GUI view with widgets to drive the mechanisms. Fig. 4 is a network-style map which shows the states the views can transition between:

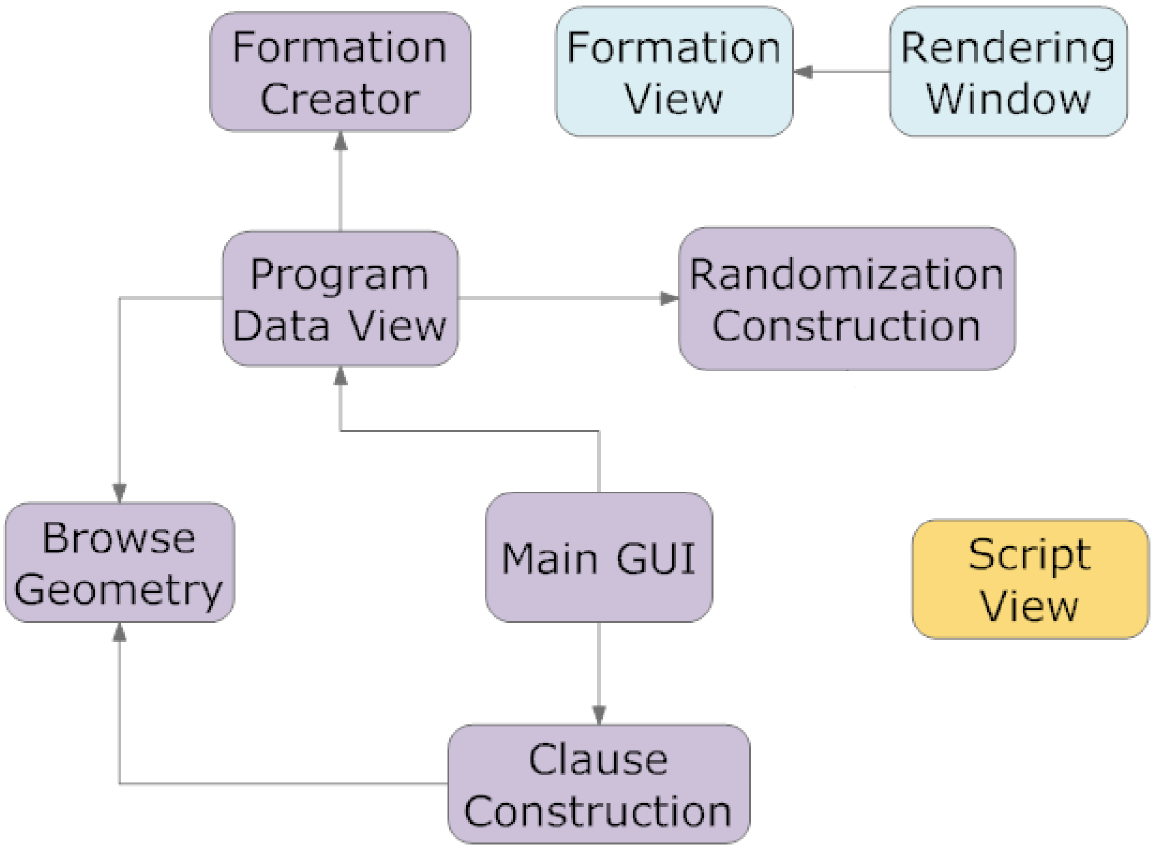


Figure 4: GUI Map

The starting state of the rendering context is "Rendering Window", the only state of the natural-language/XML script editor is "Script View", and the starting state of the main GUI view is "Main GUI". The rest of the boxes represent views that can be shifted to in order to facilitate different functions. The arrows indicate the direction that a view can shift. A view may only shift back along the path it shifted on. I.e. the transition "Clause Construction → Browse Geometry → Program Data View" is not allowed because the intent to browse for geometry was in the context of Clause Construction, not the Program Data View. This scheme was easily implemented by maintaining a GUI reference stack. The following is a conceptual description of each view:

- Rendering Window: The tool will start with this view as the 3-D rendering context. The current scene will be rendered in this viewer when the user requests a rendering.

- Script View: This is a pane that functions as a text editor. The user uses this view to load, save, and edit natural-language and XML scripts.
- Main GUI: The tool will start with the main GUI view in this state. Will contain links to the Program data view and clause construction view. Also will include a scene selector, which will allow the user to select which of the scenes defined in the program data to hook to the rendering view. Lastly, it will include widgets to invoke all the mechanisms associated with the tool's functionality.
- Program Data View: Nominal view of the current state of the program data. Will include links for defining new or editing existing randomization scenarios, templates, and custom formations. Also it will link to the file system browser view for hunting for new object geometries.
- Clause Construction: GUI with widgets to assist in the task of writing a clause adhering to the grammar rules. This should assist in the learning curve for the language. It will link to the browse geometry view if a new object is needed.
- Browse Geometry: This view acts as a file system browser which the user may use to find new object geometries.
- Randomization Construction: This view contains widgets to assist the user in creating/editing a randomization scheme, which can be used as part of a randomization scheme or template.
- Formation creator: GUI with widgets to assist the user in creating/editing a custom formation. When the user elects to construct/edit a custom formation, the main GUI will shift its state from "Program Data View" to "Formation creator", and the rendering context will shift its state from "Rendering View" to "Formation View" simultaneously. While in this state, the user can use both views to define a custom formation - the Formation Creator GUI gives a nominal view of the state of the formation, and the user is allowed to drag "marker" objects around in the rendering context to define the formation. Both views will shift back simultaneously as well.
- Formation view: Will shift from the rendering window simultaneously as the Program data view shifts to the Formation Creator. Will allow the user to drop "marker" objects onto an arbitrary x-y plane to define a formation. The Formation creator view will list the markers in coordinate terms and provide a link back to the Program Data view.

4 XML Representation

As previously stated, the tool is capable of converting an instance of session data to an equivalent XML-based representation (and vice-versa).

4.1 Rationale

The major impetus behind the inclusion of the XML specification was not as an alternative to the natural language, but as a complete-transparency front-end to session data. Ultimately, the locational and rotational data specified in clause form is processed into numerical data, which coming fresh from a natural language description will be imprecise (remember, there are only 8 compass directions - N,NE,E,SE,S,SW,W,NW - and 5 distances - "very near", "near", "moderate" A.K.A. no specification, "far", and "very far"). If this set of choices is too limited, the user could go to the XML specification and edit the corresponding values in their numerical form to achieve as much precision as necessary. The decision to make it XML-based was a natural choice given its popularity as a format to represent data structures and, given its widespread use, the relatively good chance that a potential user of the tool has some familiarity with XML.

4.2 Structure

As you may expect, the structure of the XML specification follows the structure of the session data extremely closely. An instance of program data is defined as a collection of Scenes, Objects, Templates, Scenarios, and Custom Formations. In the XML, the contents of the root node "<data>" are zero-to-many "<scene>", "<obj>", "<template>", "<scenario>", and "<custom_formation>" nodes. I structured the sub-nodes in the most logical way I could with the goal of making it as analogous as possible to the structure of session data.

4.3 Doctype

Below is the full Doctype specification of the XML representation of session data. It is presented here for posterity, but reading through the example in the next sub-section may be a faster way to get a feel for its structure.

```
<!DOCTYPE DATA [

<!ELEMENT DATA (OBJ*,TEMPLATE*,SCENARIO*,CUSTOM_FORMATION*,SCENE*)>
<!ELEMENT OBJ (#PCDATA)>
<!ELEMENT TEMPLATE (RANDOM)>
<!ELEMENT SCENARIO (RANDOM)>
<!ELEMENT CUSTOM_FORMATION ((CARTESIAN|POLAR)*)>
<!ELEMENT SCENE (ENTITY)*>
<!ELEMENT ENTITY ((SCENE_REF|GROUP|OBJECT),POSITION,HEADING,RANDOM)>
<!ELEMENT SCENE_REF (#PCDATA)>
<!ELEMENT GROUP (BASE,FORMATION,SUGGEST?)>
<!ELEMENT BASE (#PCDATA)>
<!ELEMENT FORMATION (CLUSTER|SQUARE|ROW|CUSTOM)>
<!ELEMENT CLUSTER (#PCDATA)>
<!ELEMENT SQUARE (#PCDATA)>
<!ELEMENT ROW (#PCDATA)>
<!ELEMENT CUSTOM (#PCDATA)>
<!ELEMENT OBJ_REF (#PCDATA)>
<!ELEMENT POSITION (CARTESIAN|POLAR)>
<!ELEMENT HEADING (#PCDATA)>
<!ELEMENT RANDOM (APPLY*,EXIST?,COUNT?,OBJ_CHOICES?,LOCATION?,ROTATION?)>
<!ELEMENT SUGGEST (RANDOM)>
<!ELEMENT APPLY (#PCDATA)>
<!ELEMENT EXIST (#PCDATA)>
<!ELEMENT COUNT (GAUSS|RANGE)>
<!ELEMENT OBJ_CHOICES (OBJ_CHOICE+)>
<!ELEMENT OBJ_CHOICE (#PCDATA)>
<!ELEMENT LOCATION (LOC_CHOICE*,LOC_SHIFT?)>
<!ELEMENT LOC_CHOICE (POSITION,LOC_SHIFT?)>
<!ELEMENT LOC_SHIFT (DIRECTION,DISTANCE)>
<!ELEMENT DIRECTION (ROT_CHOICE+|#PCDATA)>
<!ELEMENT DISTANCE (GAUSS|RANGE|#PCDATA)>
<!ELEMENT ROTATION (ROT_CHOICE+|#PCDATA)>
<!ELEMENT ROT_CHOICE (GAUSS|RANGE|#PCDATA)>
<!ELEMENT CARTESIAN (#PCDATA)>
<!ELEMENT POLAR (#PCDATA)>
<!ELEMENT GAUSS (#PCDATA)>
<!ELEMENT RANGE (#PCDATA)>

<!ATTLIST OBJ URI CDATA #REQUIRED>
<!ATTLIST TEMPLATE NAME CDATA #REQUIRED>
<!ATTLIST SCENARIO NAME CDATA #REQUIRED>
<!ATTLIST CUSTOM_FORMATION NAME CDATA #REQUIRED>
<!ATTLIST SCENE NAME CDATA #REQUIRED>
<!ATTLIST ENTITY NAME CDATA #REQUIRED>
<!ATTLIST EXIST RESERVE CDATA #IMPLIED>
<!ATTLIST GROUP COUNT CDATA #REQUIRED>
<!ATTLIST GROUP SPACING CDATA #REQUIRED>
<!ATTLIST ROW FROM CDATA #REQUIRED>
<!ATTLIST ROW TO CDATA #REQUIRED>
<!ATTLIST CARTESIAN X CDATA #REQUIRED>
<!ATTLIST CARTESIAN Y CDATA #REQUIRED>
<!ATTLIST POLAR THETA CDATA #REQUIRED>
<!ATTLIST POLAR DIST CDATA #REQUIRED>
<!ATTLIST SUGGEST NUM CDATA #REQUIRED>
<!ATTLIST OBJ_CHOICES EQUAL_CHANCE CDATA #IMPLIED>
<!ATTLIST LOCATION EQUAL_CHANCE CDATA #IMPLIED>
<!ATTLIST DIRECTION EQUAL_CHANCE CDATA #IMPLIED>
<!ATTLIST ROTATION EQUAL_CHANCE CDATA #IMPLIED>
<!ATTLIST OBJ_CHOICE CHANCE CDATA #IMPLIED>
<!ATTLIST LOC_CHOICE CHANCE CDATA #IMPLIED>
<!ATTLIST ROT_CHOICE CHANCE CDATA #IMPLIED>
<!ATTLIST RANGE WINDOW CDATA #REQUIRED>
<!ATTLIST RANGE MEAN CDATA #REQUIRED>
<!ATTLIST GAUSS STDEV CDATA #REQUIRED>
<!ATTLIST GAUSS MEAN CDATA #REQUIRED>

]>
```

4.4 Example

The following is an XML specification of an actual instance of session data which the tool wrote out. In the session, I had resolved the URIs of 9 object geometries, and specified 3 randomization scenarios, a template, a custom formation, and 2 scenes: "parking_lot" and "MainScene".

```
<data>
   <obj uri='C:/geometries/warehouse.obj'>warehouse</obj>
   <obj uri='C:/geometries/aerodyne_red_subaru.obj'>subaru</obj>
   <obj uri='C:/geometries/stwagon.obj'>stwagon</obj>
   <obj uri='C:/geometries/pickup_fixed.obj'>pickup</obj>
   <obj uri='C:/geometries/suv.obj'>suv</obj>
   <obj uri='C:/geometries/infiniti.obj'>infiniti</obj>
```

```
<obj uri='C:/geometries/Red_Maple_1.obj'>maple</obj>
<obj uri='C:/geometries/ship_cnt_20.obj'>ship_cont</obj>
<obj uri='C:/geometries/minivan.obj'>minivan</obj>
<scenario name='loc_jitter'>
   <random>
      <location>
         <loc_shift>
            <direction>
               <range window='360.0' mean='0.0'/>
            </direction>
            <distance>
               <gauss mean='0.0' stdev='0.025'/>
            </distance>
         </loc_shift>
      </location>
   </random>
</scenario>
<scenario name='jitter'>
   <random>
      <apply>loc_jitter</apply>
      <rotation>
         <range window='360.0' mean='0.0'/>
      </rotation>
   </random>
</scenario>
<scenario name='parking'>
   <random>
      <apply>loc_jitter</apply>
      <rotation>
         <rot_choice chance='0.75'>
            <gauss mean='0.0' stdev='4.5'/>
         </rot_choice>
         <rot_choice chance='0.25'>
            <gauss mean='180.0' stdev='4.5'/>
         </rot_choice>
      </rotation>
   </random>
</scenario>
<template name='parking_space'>
   <random>
      <apply>parking</apply>
      <obj_choices equal_chance='true'>
         <obj_choice>stwagon</obj_choice>
         <obj_choice>minivan</obj_choice>
         <obj_choice>pickup</obj_choice>
         <obj_choice>infiniti</obj_choice>
         <obj_choice>suv</obj_choice>
      </obj_choices>
   </random>
</template>
```

```
<custom_formation name='four_by_two'>
   <cartesian y='0.0' x='0.0'/>
   <cartesian y='5.0' x='2.5'/>
   <cartesian y='-5.0' x='2.5'/>
   <cartesian y='-5.0' x='0.0'/>
   <cartesian y='0.0' x='2.5'/>
   <cartesian y='5.0' x='0.0'/>
   <cartesian y='10.0' x='0.0'/>
   <cartesian y='10.0' x='2.5'/>
</custom_formation>
<scene name='parking_lot'>
   <entity name='1st_row'>
      <group count='10' spacing='0.15'>
         <base>parking_space</base>
         <formation>
            <row to='270.0' from='90.0'/>
         </formation>
      </group>
      <position>
         <polar theta='0.0' dist='0.0'/>
      </position>
      <heading>0.0</heading>
      <random/>
      <suggest num='1-10'>
         <random>
            <exist reserve='True'>0.95</exist>
         </random>
      </suggest>
   </entity>
   <entity name='2nd_row'>
      <group count='10' spacing='0.15'>
         <base>parking_space</base>
         <formation>
            <row to='270.0' from='90.0'/>
         </formation>
      </group>
      <position>
         <polar theta='180.0' dist='0.4'/>
      </position>
      <heading>180.0</heading>
      <random/>
      <suggest num='1-10'>
         <random>
            <exist reserve='True'>0.75</exist>
         </random>
      </suggest>
   </entity>
   <entity name='3rd_row'>
      <group count='10' spacing='0.15'>
         <base>parking_space</base>
```

```
            <formation>
               <row to='270.0' from='90.0'/>
            </formation>
         </group>
         <position>
            <polar theta='180.0' dist='0.6'/>
         </position>
         <heading>0.0</heading>
         <random/>
         <suggest num='1-10'>
            <random>
               <exist reserve='True'>0.5</exist>
            </random>
         </suggest>
      </entity>
      <entity name='4th_row'>
         <group count='10' spacing='0.15'>
            <base>parking_space</base>
            <formation>
               <row to='270.0' from='90.0'/>
            </formation>
         </group>
         <position>
            <polar theta='180.0' dist='1.0'/>
         </position>
         <heading>180.0</heading>
         <random/>
         <suggest num='1-10'>
            <random>
               <exist reserve='True'>0.25</exist>
            </random>
         </suggest>
      </entity>
   </scene>
   <scene name='MainScene'>
      <entity name='main_bldg'>
         <obj_ref>warehouse</obj_ref>
         <position>
            <polar theta='0.0' dist='0.0'/>
         </position>
         <heading>0.0</heading>
         <random/>
      </entity>
      <entity name='parking'>
         <scene_ref>parking_lot</scene_ref>
         <position base='main_bldg'>
            <polar theta='270.0' dist='0.25'/>
         </position>
         <heading facing='True' base='main_bldg'/>
         <random/>
```

```
        </entity>
        <entity name='trees'>
           <group count='5' spacing='0.5'>
              <base>maple</base>
              <formation>
                 <row to='180.0' from='0.0'/>
              </formation>
           </group>
           <position base='parking'>
              <polar theta='270.0' dist='0.25'/>
           </position>
           <heading>0.0</heading>
           <random/>
        </entity>
        <entity name='ship_conts'>
           <group count='8' spacing='0.5'>
              <base>ship_cont</base>
              <formation>
                 <custom>four_by_two</custom>
              </formation>
           </group>
           <position base='main_bldg'>
              <polar theta='90.0' dist='0.2'/>
           </position>
           <heading>0.0</heading>
           <random>
              <apply>jitter</apply>
           </random>
        </entity>
     </scene>
</data>
```

The "parking_lot" scene is of particular interest here as it shows how templates and scenarios can be leveraged to model the variance of a high-level dynamic (like that of a parking lot). The scene itself is structured as four rows of ten of the template object "parking_space". Notice that each entity node associated with each row of parking_spaces has a suggestion applied to it. For each row, the "suggestion" is to assign the specified existence chance to all 10 members (and indicate that the space should be reserved). The parking_space template is defined with a randomization scheme which contains 5 object choices (stwagon, minivan, pickup, infiniti, and suv) each with an equal (20%) chance of occurring, and the application of the randomization scenario "parking". "parking" is defined with a randomization scheme which specifies two rotational options and the application of the randomization scenario "loc_jitter". The first rotational option, which is 75% likely to be the choice, specifies a gaussian context with a mean of 0 degrees and a standard deviation of 4.5 degrees. The second rotational option is similar, but is only 25% likely and has a Gaussian mean of 180 degrees. "loc_jitter" is a randomization scenario defined by a locational shift with a 360-degree directional range and a small Gaussian-based distance.

An understanding of the nuts-and-bolts of the structure of the parking_lot and associated randomizations should give an understanding of the high-level behavior being modeled: That the parking lot features rows which are more desirable than others (the "first_row" bearing a higher existence rate and being located in the native "front" of the parking_lot), that nobody parks their vehicle perfectly (represented by the small locational and rotational variance), and that some choose to back into a parking space rather than pull in (represented by the two rotational choices with a base of 0 degrees and 180 degrees).

5 Environment

The conceptual areas of the tool's functionality have been presented in full. At certain points in the prior sections you may have wondered how various aspects have actually been implemented. Specifically, you may be wondering how it's getting access to this 3-D rendering context I keep mentioning. The answer to this is that the tool has been developed as an extension to the rendering software Blender [1], which is a free download and is extensible via python scripts.

5.1 Rationale for Blender

There are numerous advantages to using Blender as a platform for the tool. It is already a popular choice amongst the Imaging Science department to prepare scenery for simulations (thus the research staff has familiarity with it). In addition, it was a near-perfect environment to develop the tool in because of the multitude of features which can be seamlessly integrated into the tool's functionality.

5.1.1 Robust Rendering Context

Blender's marquee feature is the ability to render 3-D geometry. Object geometries may be imported and exported from and to a variety of different formats, or created from scratch. Blender maintains a list of Scene instances (distinct from the "Scene" class in the tool's session data model!) which object geometries may "link" to. The "current" scene is the one which appears in the rendering window (the user may switch which scene is "current" any time they want). Every object which is "linked" to the current scene will appear in the rendering view. Fig. 5 is a screenshot of a blender session featuring a scene containing four objects: A cube, cone, sphere, and a monkey head.

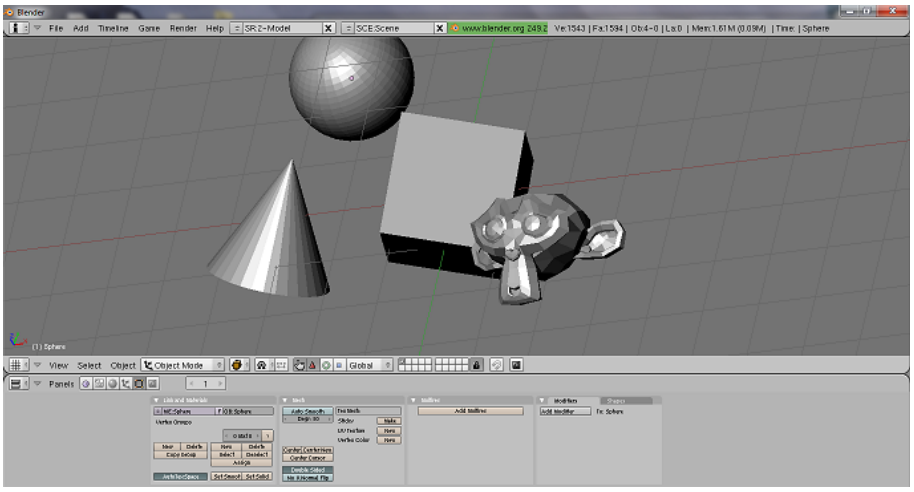


Figure 5: A Blender Session

The user may interact with the rendering view by changing the perspective and position of the viewing camera, and can drag objects around in the scene using the mouse. The rotation and size of objects can also be edited easily via a mouse-based interface.

Each object instance existing in a blender session has a (unique) name, locational coordinates, and rotational coordinates. The locational coordinates (LocX,LocY,LocZ) are specified in 3-dimensional Cartesian format with arbitrary units. The rotational coordinates (RotX,RotY,RotZ) are in radians and represent the object's Euler (intrinsic) rotational state (i.e. RotX describes an object's rotational state with respect to the X-axis). It is the same concept as the roll, pitch, and yaw of an aircraft. These attributes can be accessed and modified via python scripts.

5.1.2 Extensibility via Python

Blender contains an on-board python interpreter which can be used as a shell or for running embedded python scripts. References to scenes and objects can be obtained and operated on from the "Blender" module in the python environment. For example, entering the following commands in the python shell inside a blender session (or by putting them in a script and executing it)

```
cube = Blender.Object.Get("cube")
cube.LocX = cube.LocX + 10
cube.RotZ = 3.14
```

will move the object named "cube" (assuming it exists) to the right by 10 units and set it's intrinsic rotation around the z-axis to 3.14 radians. The simple commands involved with setting an object's locational and positional state form the basis (and essentially, the extent) of the tool's interface to the rendering window.

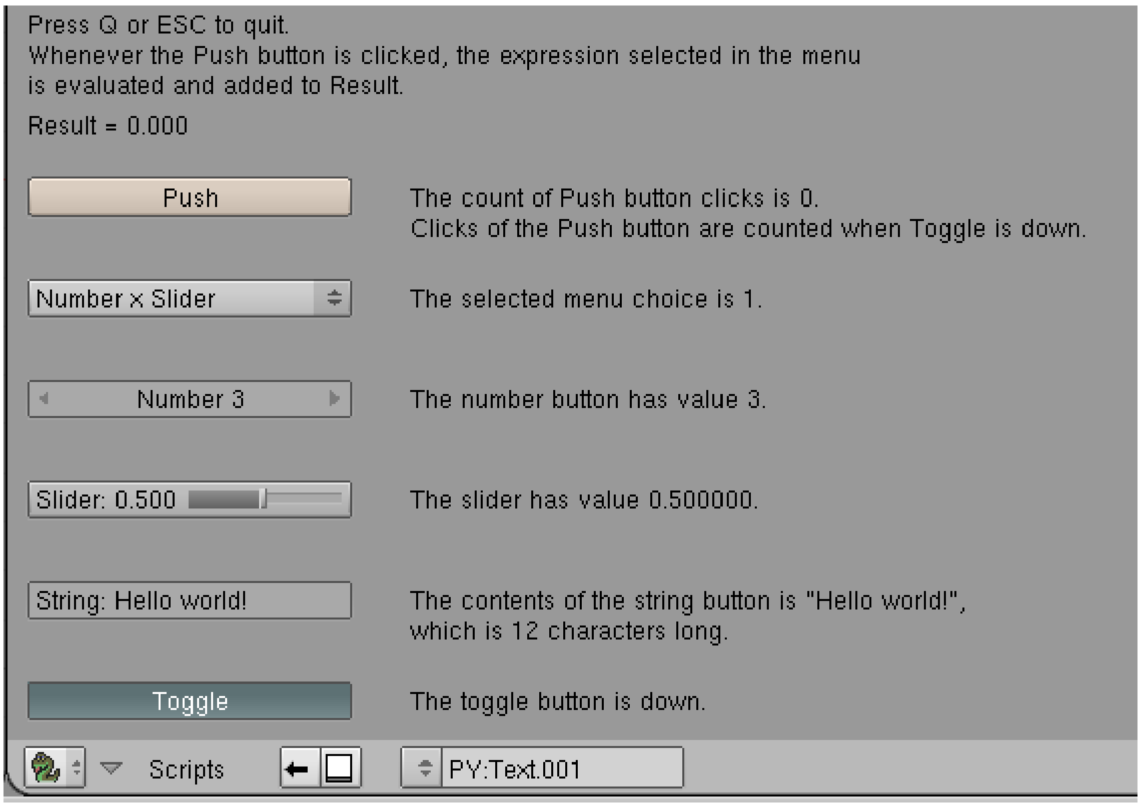
The ability to access and modify a 3-D scene rendering from a programming environment is the crux of the tool. The ease with which one can do this in the Blender platform is by far the biggest reason for its role as the platform of the tool.

5.1.3 GUI Toolkit

The Blender module contains a sub-module "Draw", which can be used to create various GUI schemes inside blender. From its documentation page in the Blender API: "This module provides access to a windowing interface in Blender. Its widgets include many kinds of buttons: push, toggle, menu, number, string, slider, scrollbar, plus support for text drawing" [2]. Fig. 6 depicts a GUI window including (from top to bottom) examples of the Push-button, Menu, Number-selector, Slider, Text-entry, and Toggle-button widgets [3].

An additional useful GUI feature are pop-up menus and pop-up blocks, also located in the Draw module. The executing thread can be paused at any time to ascertain the return value of either by invoking "Draw.PupMenu" or "Draw.PupBlock". A pop-up menu presents a list of text-based options; once the user clicks one of them it goes away. Fig. 7 depicts an example pop-up menu, spawned from a simple script.

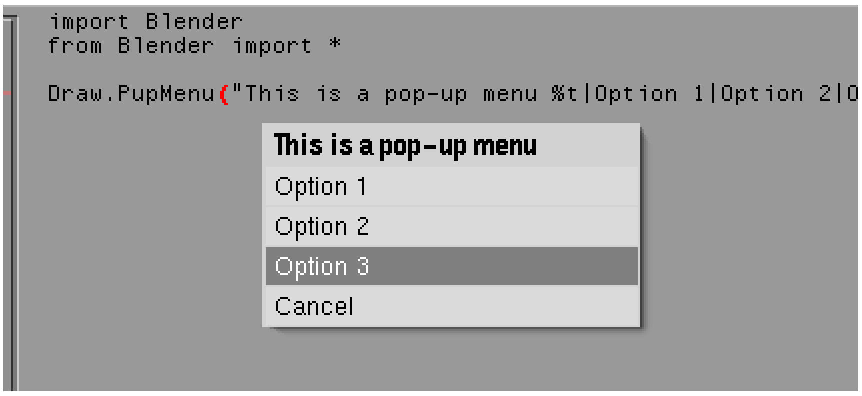
Alternatively, a pop-up block can present multiple different widgets in a self-contained pop-up view (references to the widgets used in a pop-up block must be saved in order to access their associated data once users indicate they are done).



Press Q or ESC to quit.
Whenever the Push button is clicked, the expression selected in the menu
is evaluated and added to Result.
Result = 0.000

Push	The count of Push button clicks is 0. Clicks of the Push button are counted when Toggle is down.
Number x Slider	The selected menu choice is 1.
Number 3	The number button has value 3.
Slider: 0.500	The slider has value 0.500000.
String: Hello world!	The contents of the string button is "Hello world!", which is 12 characters long.
Toggle	The toggle button is down.

Figure 6: A Sample GUI Window in Blender



```
import Blender
from Blender import *

Draw.PupMenu("This is a pop-up menu %t|Option 1|Option 2|O
```

Figure 7: A Blender Pop-up Menu

A GUI view is instantiated by "registering" a gui-drawing function with corresponding event-handler functions. The GUI window can transition from one view to another simply by registering a different set of functions. This can be done from an event handler, which makes it easy to design a plugin in an event-driven paradigm. If you recall the GUI model of the tool, it was presented as a "map" of views; each view facilitating a different facet of the user interaction scheme. Each transition in the GUI map is achieved simply by registering the new view from the appropriate place in the old view's event handler. In the tool's implementation, all event-handler functions have access to the global gui reference stack, so the most recent gui can be popped off the stack and re-registered whenever appropriate.

5.1.4 Deployment

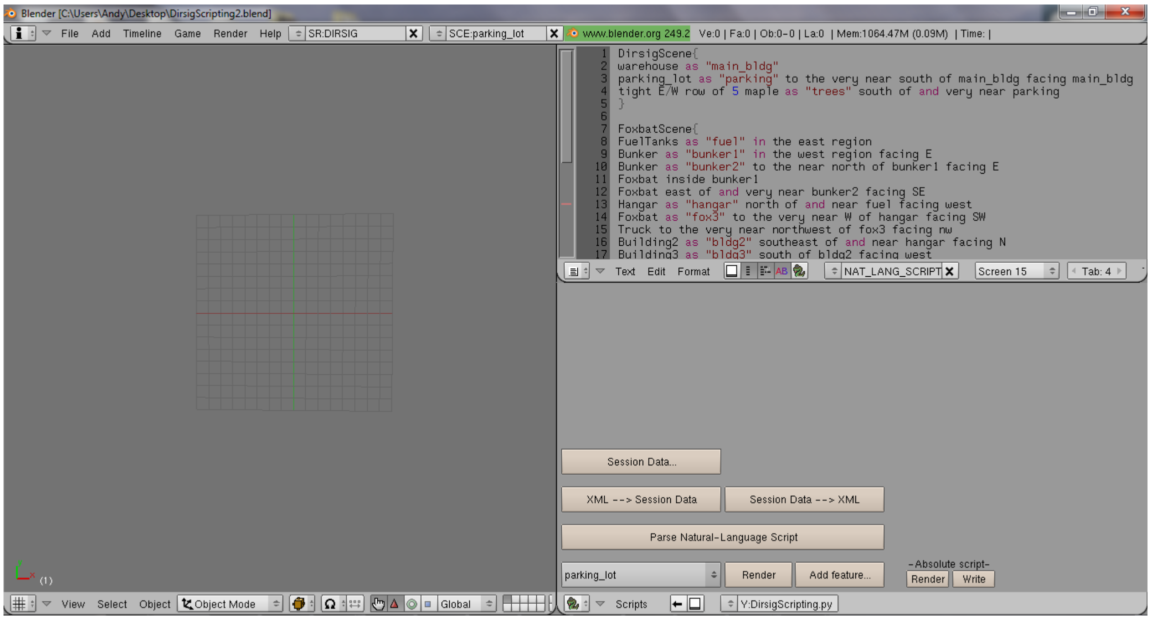
A major advantage to using Blender as the tool's functional platform is the ease with which the tool can be deployed. As previously mentioned, Blender is free software, so access to the tool's environment is just a download away. And since Blender is multi-platform, this means the tool is automatically multi-platform as well because it's a plugin. Indeed, Blender distributions are available for Windows 2000, XP, Vista, 7; Mac OS X (PPC and Intel); Linux (i386); and FreeBSD (i386) [4].

The text-editor view in blender can display any of the text "objects" which are a part of the current session. A text "object" is simply a text-based file which has been loaded or created. A blender plugin is typically executed by loading the necessary source code file into a text object and executing it. If the source code imports modules which are not a part of the standard python library, the module's source file can be pasted into a separate text object and the on-board interpreter takes care of the rest. If at any point the executed script registers a GUI view, the text pane which contained the source transforms into that GUI view and subsequent view transitions take place within that pane. A blender session may even be saved in the midst of an event-driven GUI script (i.e. whilst displaying a GUI view waiting to process a user-spawned event), and be in the exact same state of execution when re-loaded. This means that in addition to posting a source-code distribution of the tool, I will also post an executable version in the form of a .blend file, which when loaded will automatically display the main GUI of the tool. The .blend file distribution will also be automatically set up to display the rendering context and script view alongside the main GUI in a 3-panel display, as seen in Fig. 8.

All a potential user will need to do to use the tool will be to download and install Blender (and a compatible version of Python if need be - the Blender installation process prompts the user to install Python if it does not detect a suitable version on their machine), then download the .blend file distribution of the tool. Opening the .blend file version of the tool resumes the session I have saved as the starting state of the tool. For the source code distribution, I will offer a .zip of all source files which the tool uses. In this case, the setup process still starts with installing Blender. Then, a new blender window would be opened and every source file would have to be pasted into a separate Text object. Lastly the main file would be executed and the tool would start.

5.2 Limitations

Developing the tool as a Blender extension instead of as a standalone application is not without its disadvantages. Since everything needs to happen inside a blender session, the features of the tool are limited to the features Blender can provide. For example, it would be nicer to have a pop-up text area to write natural-language scripts in rather than having to confine it to a Blender Text panel which can be shared by all other Text objects existing in the session.



```
DirsigScene{
warehouse as "main_bldg"
parking_lot as "parking" to the very near south of main_bldg facing main_bldg
tight E/W row of 5 maple as "trees" south of and very near parking
}

FoxbatScene{
FuelTanks as "fuel" in the east region
Bunker as "bunker1" in the west region facing E
Bunker as "bunker2" to the near north of bunker1 facing E
Foxbat inside bunker1
Foxbat east of and very near bunker2 facing SE
Hangar as "hangar" north of and near fuel facing west
Foxbat as "fox3" to the very near W of hangar facing SW
Truck to the very near northwest of fox3 facing nw
Building2 as "bldg2" southeast of and near hangar facing N
Building3 as "bldg3" south of bldg2 facing west
```

Figure 8: Starting State of Tool

The biggest limitation was simply the language restriction - that everything had to be written in Python. Given the tool's heavy predication on the data structure, I would have preferred to develop it in a more traditional object-oriented language like java. While it's true that Python has class support, it is not structured specifically in the object-oriented paradigm and on numerous occasions I found it awkward to define and work with a rigid data structure. And perhaps it was because I was new to the language as I began implementing the tool, but I found the text parsing abilities of the standard library to be rather limited. Building a formal parser for the natural language using only the standard library would have been nightmarish. Fortunately, I was able to make use of a free and open-source module called pyparsing [5] which spared me a great deal of work.

6 Implementation

Although I had spent the later months of 2010 developing earlier, rudimentary versions of the tool, formal implementation of the current version occurred between mid-December 2010 and mid-April 2011. A look at a data-flow diagram will help set the context for a discussion of the tool's implementation.

6.1 Data Flow

Fig. 9 is a data-flow diagram which charts the exchange of data throughout the tool. In the diagram, ovals represent things which the user sees/interacts with, and rectangles represent internal data storage and processes. Here, Session data is split into two conceptual types: "Scene Data" and "Building Blocks". "Scene Data" represents the portion of the data structure which represents scene specifications, i.e. the "Scenes" list of session data. "Building Blocks" represents the rest of the data structure, i.e. Objects, Templates, Scenarios, and Custom Formations, which conceptually are the building blocks which can be used to specify scene features. However, it also

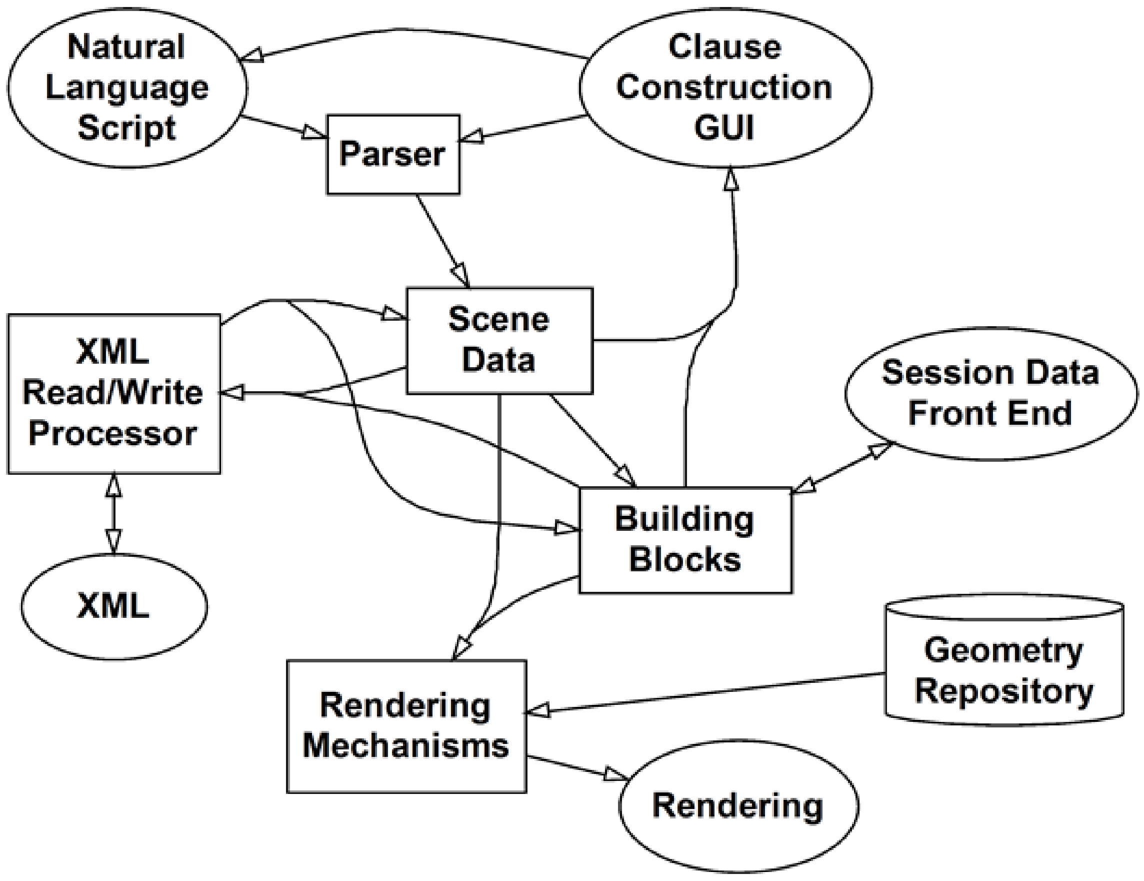


Figure 9: Data Flow Diagram

represents Scenes in the context of them being available for use as a sub-scene. Think of it as "Building Blocks" containing a reference list of scenes which may be used as sub-scenes. This is the reason for the arrow from "Scene Data" to "Building Blocks". Conceptually, references to defined scenes are pulled from Scene Data and registered as usable features. The basic concept is that scene data is created using the features defined in Building Blocks. The other nodes in the diagram are generally defined by their interaction with Scene Data and Building Blocks.

- Natural Language Script: Text-based scene specification data originates from the user and is pulled into the Natural-Language Parser.
- Parser: Representation of the parsing and processing mechanisms which the tool employs (found in the file ParseNL.py). Pulls Text-based data from the Natural Language Script and pushes properly-formatted scene data to session data (conceptually, the Scene Data node). Also accepts individual text-based clauses from the Clause Construction GUI.
- XML: XML-based specification of all session data. XML data can be pulled (read) or pushed (written) from/to the XML Read/Write Processor.
- XML Read/Write Processor: Representation of the XML processing abilities of the tool. Conceptual merging of the module "ElementTree" [6] - which carries out conversions between

text-based XML and a representation of an XML tree defined in python classes - and a file I wrote called XmlAccess.py, which carries out conversions between session data and the XML representation ElementTree works with. The net result is that this node acts as a ”black box” which converts between text-based XML and session data (session data being represented by Scene Data and Building Blocks).

- Clause Construction GUI: This node corresponds to the view which bears the same name in the GUI map. Recall that its intent is to aid the user in writing acceptable clauses. To do this, it must have reference access to the features available to build a clause with; thus you have the pulling of data from Building Blocks. It must also have access to the names of the entities which already exist in the scene for which a clause is being developed (for relational-based positioning); thus you have the pulling of data from Scene Data. Once the user has defined a clause using this view, the clause can be used in one or both of the following ways: It can be parsed, processed, and sent to Scene Data, and/or sent in text form to the natural script under the proper scene heading.

- Session Data Front End: This node corresponds to the ”Program Data View” in the GUI map. Recall that this view presents the user with a way to create/modify/delete randomization scenarios, template objects, and custom formations; and add/remove object geometries and scenes (but NOT to specify scenes). So conceptually it acts as a front-end to the ”Building Blocks” portion of session data, which is the reason for the bi-directional data flow between this node and Building Blocks.

- Rendering Mechanisms: This node represents the mechanisms used to produce a physical rendering of a scene, the details of which shall be discussed later in this section. Obviously, these mechanisms need to pull from Scene Data. Data also needs to be pulled from Building Blocks because the full specifications of non-scene data reside there (whereas they are only referenced in Scene Data). For now, think of Rendering Mechanisms as a black box which pulls from session data and interfaces with Blender’s rendering context to produce scene renderings with referenced objects from the geometry repository.

- Geometry Repository: A purely conceptual representation of where the object geometry files available to the tool are stored. For now, the repository is simply the user’s local file system and the geometry files are in .obj (and corresponding .mtl format) [7]. Each object should be represented in the repository by an .obj file to define the physical vertices and faces of the object, and by a .mtl (material) file to define the material (color) assigned to each face.

- Rendering: Represents Blender’s 3-D rendering context. Rendering Mechanisms pulls objects from the repository and positions them as per the specification of the scene being rendered.

6.2 Grammar Parsing

The parsing and processing of natural-language text takes place in the file ParseNL.py. As previously mentioned, I made use of a free and open-source text-parsing module called ”pyparsing” [5], the source of which is included in the blend-file distribution of the tool as a separate text object. With this module, a programmer may define a grammar, rule by rule, in terms of parser elements. In general, there is a parser element corresponding to every special character in BNF. Most prominently, ”Or” for ”—”, ”Option” for ”[...]”, and ”ZeroOrMore” for ”*”. For a better sense of this, recall the COMPASS, COMPASS FULL, and COMPASS ABBREV rules from the grammar:

```
      COMPASS -> COMPASS_FULL[COMPASS_FULL] | COMPASS_ABBREV[COMPASS_ABBREV]
 COMPASS_FULL -> (N|n)orth | (S|s)outh | (E|e)ast | (W|w)est
COMPASS_ABREV -> N | n | S | s | E | e | W | w
```

After importing all parser elements from the pyparsing module with the command

```
from pyparsing import *
```

The compass rules can be defined like so:

```
n = Or([Literal("N"),Literal("n")])
s = Or([Literal("S"),Literal("s")])
e = Or([Literal("E"),Literal("e")])
w = Or([Literal("W"),Literal("w")])
comp_abbr = Or([n,s,e,w])
north = n + Literal("orth")
south = s + Literal("outh")
east = e + Literal("ast")
west = w + Literal("est")
comp_full = Or([north,south,east,west])
compass = Or([comp_full+Optional(comp_full), comp_abbr+Optional(comp_abbr)])
```

A string may be parsed in the context of any defined parser element, and the parsing definitions included in the module will find a token stream that matches the input. I have carefully devised the grammar to be unambiguous in this sense; meaning that there cannot be more than one token stream that matches any given input.

This takes care of the tokenization of the language, but the pyparsing module may also be leveraged to assist in processing the language. This can be achieved by assigning a "parsing action" to a parser element. Returning to the compass example, my goal is to be able to parse a string describing a compass direction in the context of the compass element and receive a corresponding degree value. I achieve this by defining functions to return the proper float values, and then registering these functions as the "parseAction" associated with the corresponding parser element:

```
def n_dir(): return 90.0
n.setParseAction(n_dir)
north.setParseAction(n_dir)
def s_dir(): return 270.0
s.setParseAction(s_dir)
south.setParseAction(s_dir)
def e_dir(): return 0.0
e.setParseAction(e_dir)
east.setParseAction(e_dir)
def w_dir(): return 180.0
w.setParseAction(w_dir)
west.setParseAction(w_dir)
```

Now, for instance, when the parser detects a "n" or "north" token, it will add the float value 90.0 to the result set of the higher context. However, we must make sure that we deal with all 8 compass directions which can be specified by the language. Since the compass element consists of one and optionally another element that will each evaluate to a float, we can assign a parser action to the compass element to deal with these accordingly:

```
def resolveDir(oneOrTwoFloats):
    if len(oneOrTwoFloats) == 1: return oneOrTwoFloats[0]
    if abs(oneOrTwoFloats[0]-oneOrTwoFloats[1]) == 270.0: return 315.0
    return (oneOrTwoFloats[0] + oneOrTwoFloats[1]) / 2.0

compass.setParseAction(resolveDir)
```

Now, for example, the command compass.parseString(”Northwest”) will return the float value 135.0. The rest of the natural-language grammar is parsed and processed in the same fashion. If you are interested in the specifics, take a look at the ParseNL.py file.

Interfacing to the parsing process is bottle-necked into one function defined in ParseNL.py, named ”add_clause_to_scn_data”, which receives a reference to session data, a reference to the particular scene being specified, and a clause in string form. The function processes the clause, creates the entity as per its specification, and inserts it as a member of the given scene. Along the way, the reference to session data and the names of the entities which currently exist in the scene are used to verify the references used in the clause. For example, if the clause references a feature which is not defined in session data, the clause cannot be accepted. If this occurs, or no suitable token stream is found, the user is alerted of the error and that no action has occurred. The function is called once from the main script after the user commits to a clause they have created from the clause construction GUI, and however many times as necessary when parsing an entire natural-language script (i.e, however many clauses appear in total). In the latter case the script goes through a degree of pre-processing. First, the text is gathered from the text object assigned to the natural language with the following command:

```
lines = Blender.Text.Get("NAT_LANG_SCRIPT").asLines()
```

This creates a list of strings corresponding to each line of the script. Using the standard python library, each line is processed to ascertain whether or not it is intended as a clause, scene header, or blank. If a line is determined to be a clause, it is also determined which scene it belongs to based on the last scene header seen. The add_clause_to_scn_data function is then called for each clause with the proper scene. The reason for the pre-processing, as opposed to processing an entire script with the pyparsing module, is my uncertainty of the ability of pyparsing to handle carriage returns in the proper way.

6.3 XML Integration

In the discussion of the data flow model, I mentioned that the mechanisms which carry out the reading and writing of XML data are found in a module called ElementTree and I file I wrote called XmlAccess.py. With respect to how these files are integrated into the tool, it is a similar setup to the pyparsing module and the file ParseNL.py. The source files associated with the ElementTree module, ElementTree.py and ElementPath.py, are stripped from the ElementTree source-code distribution and placed into the Blender session as Text objects, and interfacing to them is done in XmlAccess.py.

The ElementTree module defines its own data structure to represent an XML tree. ”Each Element instance can have an identifying tag, any number of attributes, any number of child element instances, and an associated object (usually a string)” [6]. For example, the XML data

```
<scene name='MainScene'>
    <entity name='main_bldg'>
```

```
        <obj_ref>warehouse</obj_ref>
        <position>
            <polar theta='0.0' dist='0.0'/>
        </position>
        <heading>0.0</heading>
    </entity>
</scene>
```

Can be defined in ElementTree format with the following code:

```
from ElementTree import Element
scn = Element("scene")
scn.attrib["name"] = "MainScene"
ent = Element("entity")
ent.attrib["name"] = "main_bldg"
obj = Element("obj_ref")
obj.text = "warehouse"
ent.insert(obj)
pos = Element("position")
polar = Element("polar")
polar.attrib = {"theta":"0.0", "dist":"0.0"}
pos.insert(polar)
head = Element("heading")
head.text = "0.0"
ent.insert(head)
scn.insert(ent)
```

As you would expect, ElementTree has provisions for converting between an XML tree in string format and an XML tree in its data structure. From string format, an XML tree is returned by the function ElementTree.fromstring(xml_string) given that xml_string is a reference to some XML string. Going the other way, a tree can be written out to an actual XML file, but I needed to send the text to a Blender Text object. I was able to write a fairly simple recursive function which takes an instance of ElementTree.Element and returns a string representation of that node. I called this function "pretty_xml" because it adds carriage returns where appropriate, adds the appropriate amount of tabs, and writes nodes with only text content on one line. Below is the function in its entirety:

```
def pretty_xml(elem,tabs):
    txt = ""
    i = 0
    while i<tabs:
        txt = txt + "\t"
        i = i + 1
    txt = txt+"<"+elem.tag
    for atr in elem.attrib.keys():
        txt = txt+" "+atr+"='"+str(elem.attrib[atr])+"'"
    if len(elem.getchildren()) == 0:
        if elem.text == None: return txt+"/>\n"
        else: return txt+">"+str(elem.text)+"</"+elem.tag+">\n"
    txt = txt + ">\n"
    for c in elem.getchildren(): txt = txt + pretty_xml(c,tabs+1)
    i = 0
    while i<tabs:
        txt = txt + "\t"
        i = i + 1
    txt = txt+"</"+elem.tag+">\n"
    return txt
```

That is the function I use to convert XML data stored in ElementTree format into string format. The returned string is dumped into the Text object named "XML_DATA".

The preceding example where I defined the "MainScene" node in ElementTree format was only to show the structure of the ElementTree.Element class. The process the tool uses to convert data between the session data format and ElementTree format is not so trivial. In fact, the entire purpose of the lengthy file XmlAccess.py is to implement this 2-way conversion. There are two logical sections of this file: The first containing functions to implement conversion from session data to tree format, and the second; vice-versa. With a few exceptions, there are two functions per class definition in the data structure: a "read" function and a "make" function, for example "read_scenario" and "make_scenario". The "read" functions read in an instance of ElementTree.Element and return a corresponding instance of a class in session data, while the "make" functions do the opposite. In general, the name of the class is used as the tag name.

From the rest of the tool, the interface to the XML processor are by the functions XmlAccess.read_data() and XmlAccess.write_data(data). "read_data" loads the text-based XML tree from the Text object named "XML_DATA" into ElementTree format and converts it to an instance of session data, which is returned. "write_data" takes the argument "data" (which should be a reference to an instance of session data), converts it to ElementTree format, then writes it to the Text object "XML_DATA" in text form. The other functions are called as the processing efforts travel through the XML or program data. For example, "read_data" will call "read_scene" to read each scene element it finds as a sub-element of the data element, "read_scene" will call "read_entity" to read each entity element it finds as a sub-element of the scene element, etc...

At this time there is no mechanism for verification by a Doctype. This may be implemented in the future - but for now, missing or superfluous tags/data will not cause errors in the reading/writing process, respectively.

6.4 GUI Scheme

The tool is event-driven: The user interacts with GUI views to create/edit/modify session data, parse natural-language scripts, read/write XML, and interactively create clauses.

6.4.1 GUI Stack

In the discussion of the GUI Toolkit in the preceding section, it was mentioned that the GUI map was going to be implemented by a GUI reference stack. A stack is a natural choice to represent the path through the GUI network which the user has taken; given the need to "walk it back" to the last GUI view after the user is done with the current one. Every event handler associated with a GUI view has access to the gui reference stack. Wherever a view shift occurs in the event handler of one view, a reference to the current view is pushed onto the stack and the new view is registered. Wherever appropriate in an event handler (at a logical done/accept/cancel event), a reference to the last GUI view is popped off the GUI stack and registered as the new view. To the best of my knowledge, there is no data structure in the standard python library which explicitly represents a stack, but a list may be *treated* as a stack. When treating a python list as a stack, the "append" function in the list class is logically equivalent to a push (adds item to the end of a list), and the list function "pop" can remove and return the item at the end of the list.

The references to the GUI views are actually pairs of function names. Each pair is a reference to a GUI view whereby the first member is the name of the function which controls the layout of the view and the second being the name of the function that serves as the event handler for the view. There are actually three functions that need to be registered as callbacks for a GUI view: The "draw" function (which draws the view), the "event" function (which handles keyboard and mouse input events), and the "button" function (which handles Draw Button events) [2]. The latter of a GUI reference pair is actually the "button" function instead of the "event" function. This is because I handle all user interaction with the GUI views entirely with various types of Blender Button widgets instead of keystrokes and non-button mouse clicks. Every view in the tool is registered with the same "event" function (imaginatively named "event"), which simply prompts the user with a quit-confirm if the "q" or "esc" key is pressed. So, a new view is registered by the command "Draw.Register(draw_func, event, button_handler)", where "draw_func" is the drawing function associated with the view and "button_handler" is the button event handler associated with the view. This can be seen in the utility function I wrote to roll the GUI back to the previous view:

```
def goto_last_gui():
    global gui_stack
    lg = gui_stack.pop(len(gui_stack)-1)
    Draw.Register(lg[0],event,lg[1])
```

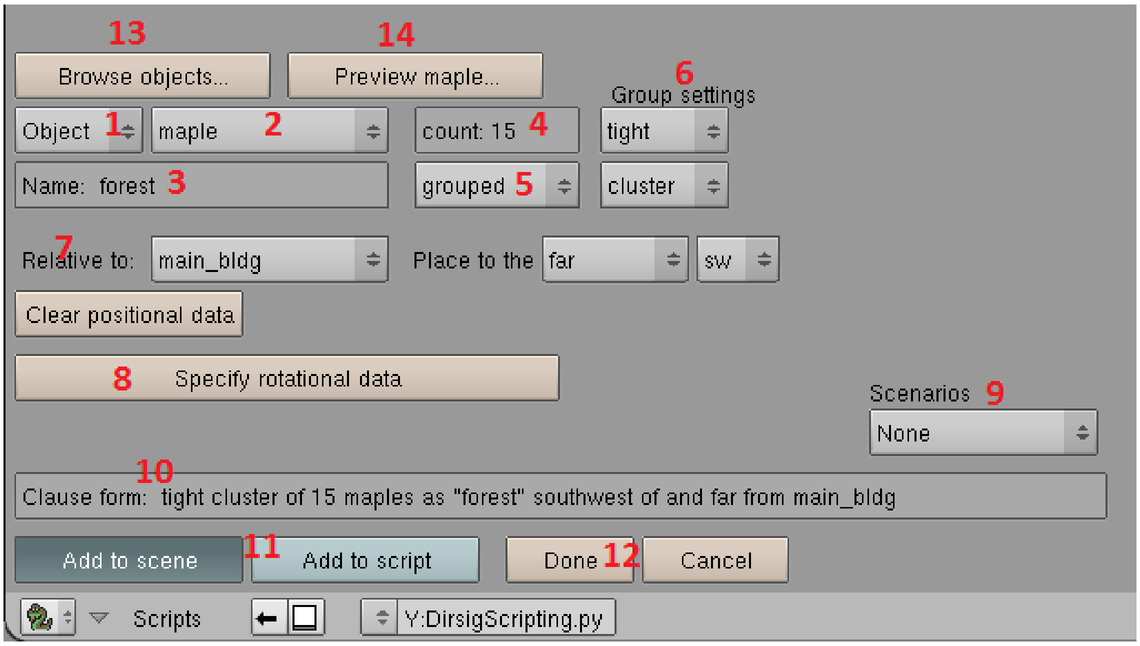
6.4.2 View Implementation Logistics

When registered, a GUI view will be drawn as specified by its "draw" function. The responsibility of the draw function is to set the proper states of its GUI widgets and place them in the proper layout. Setting the layout of the widgets is a no-frills procedure, which I actually appreciate when it comes to laying out GUI components. For each widget, the width, height, and coordinates of its lower-left coordinates must be specified. No support for abstract layout schemes is provided. Widgets can be declared globally and imported into the draw function for initialization, or declared and initialized solely in the scope of the draw function. The difference between the two methods is that the corresponding event handler can be given access to the state of the widget if declared globally, but has no way to access it if not. Obviously, any widget whose state is modifiable needs to be declared globally so the user's particular indication may be processed. With simple labels and pushable buttons, it does not matter. In either case, every widget must be initialized to the proper state in the draw function. The reason for this is that the draw function is called after every captured event (whether the particular event is processed or not) as a way to update the state of all widgets. As far as how the event handler function works, part of the initialization of a widget is the assignment of an event number to each widget. The corresponding event handler receives the event number of the widget which the user just interacted with and the proper action can be determined by a conditional construct on the event number.

6.4.3 Clause Construction

In the discussion of the GUI map, it was mentioned that there would be a view named "Clause Construction GUI" which would facilitate the process of writing clauses conforming to the natural language grammar. In the data flow diagram, you saw that this view pulls data from "Scene Data" and "Building Blocks". Remember that the distinction between the two was conceptual, and that both are contained in an instance of session data. Thus, a global reference to session data is made available in the callbacks registered to this view.

The Clause Construction GUI is invoked in the context of a particular scene. Fig. 10 is an annotated screenshot of the clause construction view.



Clause form: tight cluster of 15 maples as "forest" southwest of and far from main_bldg

Figure 10: Clause Construction GUI

1. Menu selector for the type of entity the clause will specify. Options are Object, Template, or Scene.

2. Menu selector to pick the actual entity the clause will specify. Options are the set of entities of the selected type which exist in session data.

3. "String" button (which the user may type in) to assign a name to the entity.

4. String button to indicate the entity count. While a number greater than 1 is entered, the widget at annotation 5 appears.

5. Menu selector to indicate whether or not the fact that the count is greater than 1 indicates that the entity should be considered a group (as opposed to an iterative clause). Options are "grouped" and "ungrouped". While "grouped" is selected, the widgets under annotation 6 appear.

6. Grouping of widgets to specify the formation of the group. Remember that a group can be a tight/loose/normal spaced cluster, square, row, or custom formation. As long as the row type is selected, menus appear which allow the user to select the "from" and "to" compass directions. As long as the custom type is selected, a menu appears which allows the user to select from the custom formations existing in session data.

7. Grouping of widgets to specify locational data. This section of the view has two states; the first being that locational data is not specified and the second being that it is. While the former case applies, the section is just one button which the user can select to switch to the latter case, which is the situation in the screenshot. The first menu allows the user to select the relational base, of which the options are the names of the entities currently specified in the contextual scene plus "none". While "none" is selected, the widgets to the right accommodate regional placement specification. While a pre-existing feature is selected, they accommodate relational placement. The button at the bottom of the group takes the state of the widget group back to no-specification state.

8. Grouping of widgets to specify rotational data. Also operates with two specification states; in the screenshot there is no specification. While rotational data is specified, the user is allowed to pick a relational base, which could be none. While a relational base is selected, a selector menu appears with the options "facing" and "facing away". While "none" is selected, a selector menu appears with compass directions.

9. Menu selector to pick a randomization scenario to apply. Options are the names of the set of scenarios which exist in session data plus "none".

10. The string button displaying the current text of the clause being built. This updates every time any widget is modified. The user is allowed to write directly into it, but it will be re-written if a widget is modified afterwards. The intent of this widget is to help the user learn the structure of the natural language.

11. The toggle buttons "Add to Scene" and "Add to Script". The state of these switches determine the behavior of the tool after the user accepts the clause. If "Add to Scene" is selected, the clause will be parsed, added to session data as the newest entity of the contextual scene, and a rendering of the scene will be executed. If "Add to Script" is selected, the clause is added to the Text object "NAT_LANG_SCRIPT" under the proper scene. Both options may be selected. Selecting neither would be pointless.

12. The "Done" button accepts the clause in its current form and applies the proper actions based on the states of the toggle buttons. "Cancel" applies no action with the clause. The end result of both buttons is that the GUI is walked back to the last view.

13. The "Browse objects" buttons takes the user to the File selector view to browse for new geometry files. This transition can be seen in the GUI map.

14. The "preview" button allows the user to preview the currently selected object.

6.4.4 Program Data View

The Program Data View (Fig. 11) provides a nominal representation of the current state of session data. The user is able to transition to this view directly from the main GUI when they want to add/edit/delete the "Building Blocks" (as conceptualized in the data flow model) portion of session data.

The program data view is separate from the clause construction view. My philosophy behind separating the two views should be clear at this point: You use the program data view to define the features available for use in a scene, and then use the clause construction GUI to specify how to include the features in a scene.

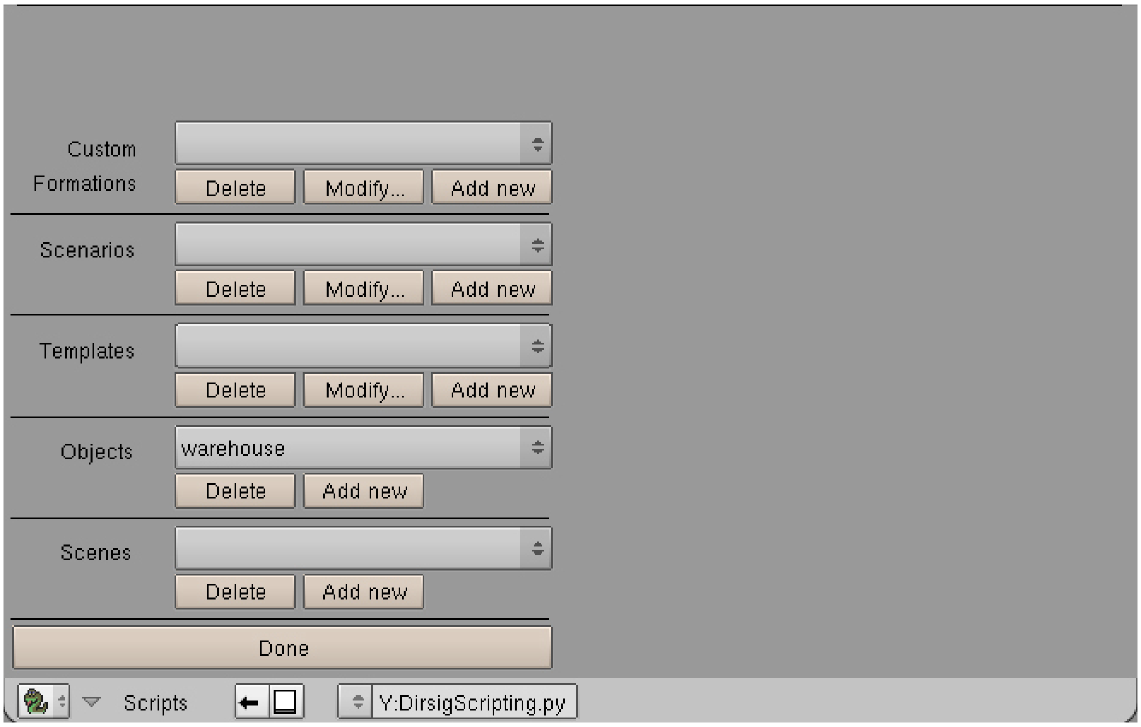


Figure 11: Program Data View

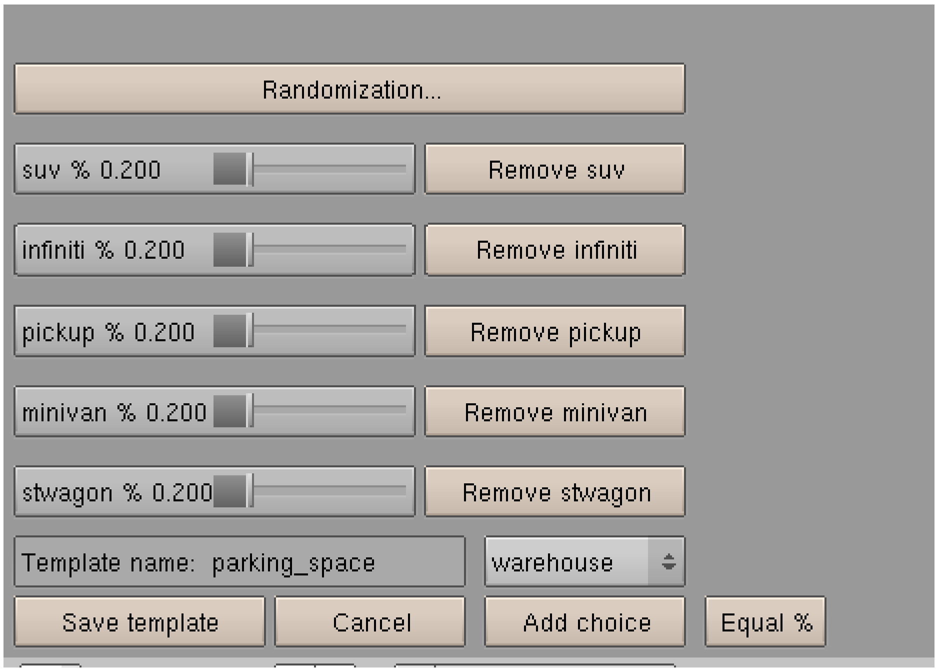
From the program data view, if the user elects to add or modify a template, a scenario, or a custom formation, the viewer will shift to a "creator" GUI for the specific feature. For instance, Selecting "Add new" in the "Scenario" section will present the user with a separate GUI to assist the user in defining a new randomization scenario.

6.4.5 Building-Block Views

As you saw in the GUI map, the Program Data View can transition to the Browse Geometry view (file selector), the Formation Creator view, and the Randomization Construction view. Electing to add a new object will take the user to the Browse Geometry view, adding/modifying a custom formation takes them to the Formation Creator view, and adding/modifying a template or scenario takes the user to the Randomization Creator view.

Conceptually, templates and scenarios both lead to the randomization view, but remember that the difference between the two is that object options are allowed in a template's randomization scheme but not in a randomization scenario's. This difference manifests itself in a subtle difference between the view for a template and the view for a scenario: The template view (Fig. 12) presents the user with a scheme to add/remove objects to/from the options list (and tweak their probabilities), and a button link to the actual randomization scheme creator view.

Hitting the "Randomization…" button, or electing to add/modify a scenario takes the user to the randomization scheme view. The randomization scheme view provides the user with a way to specify all facets of a randomization scheme except for object options, as seen in Fig. 13.



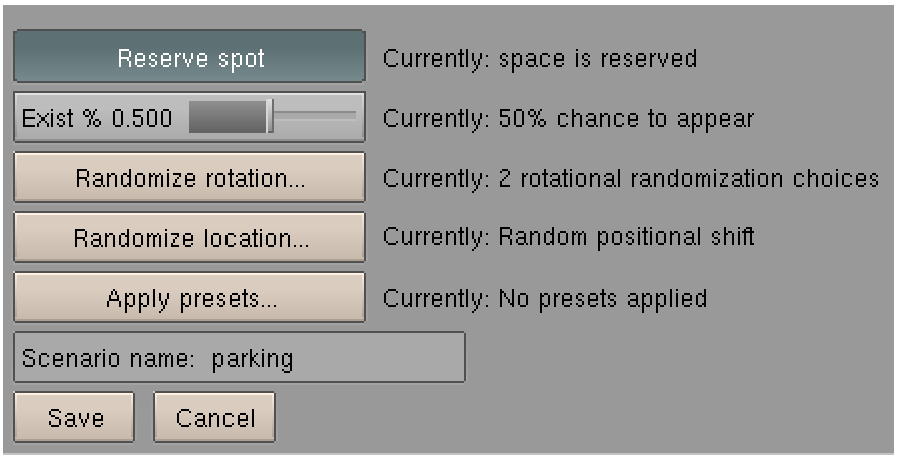
Template name: parking_space

Figure 12: Template Creator View

When adding/modifying a randomization scenario, the ”Scenario name” widget must be specified to name the scenario. Coming from the template view, the naming widget does not appear because the template is named in the template view.

The ”Exist %” slider widget allows the user to specify the existence probability facet (by sliding the bar or typing). While the probability is less than 1, the ”Reserve spot” toggle button appears, allowing the user to indicate whether the space should be reserved should the applicant entity be picked to not exist in a given rendering (i.e. ”parking_space” reserves the spot). The other widgets lead the user into sub-views for specifying the other facets of the randomization scheme. Because they would clutter the diagram, these sub-views do not appear in the GUI map; think of them as being under the conceptual heading of the Randomization view. The ”Apply presets. . . ” button presents a view which lets the user add/remove pre-existing scenarios as presets to the randomization scheme, as seen in Fig. 14.

The ”Randomize location” and ”Randomize rotation” widgets lead into a cascade of sub-views which handle the specifics of the locational and rotational facets of a randomization scheme. If you recall from the discussion of the structure of a randomization scheme, the locational and rotational facets are structured similarly as a series of positional and rotational ”choices”, respectively. A ”choice” is a positional or rotational specification, respectively. In the locational facet, a locational ”shift” may be applied to an individual choice, or may be applied globally on top of all choices. There are separate views to specify a position and a shift in this context. In the position specification sub-view, the user can select either ”cartesian” or ”polar”, and enter the coordinates (x,y) or (theta,dist), respectively. In the locational shift sub-view, the user can specify the direction and distance of the shift. The distance of the shift can be specified exactly, or in the context of a ”range” or ”gaussian” window. Specifying the distance of a shift actually leads into the same view used for specifying the rotational facet of a randomization scheme, since the data



Exist % 0.500

Currently: space is reserved

Currently: 50% chance to appear

Currently: 2 rotational randomization choices

Currently: Random positional shift

Currently: No presets applied

Scenario name: parking

Figure 13: Randomization GUI

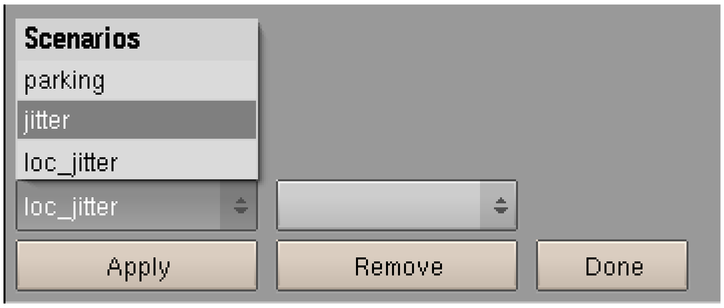


Figure 14: Scenario Application GUI

structure representing the direction of a randomized shift is the same as the rotational facet. In the rotational facet, each rotational choice takes the form of a rotational value, optionally in the context of a range or gaussian window. Figures 15-18 show the layout of the views which allow the user to specify the locational facet, positional specification, locational shift, and rotational facet, respectively.

Since all of the ”Building Block” views add/modify a certain piece of the data structure of session data, the ”draw” and ”button” functions registered with each view are given access to an actual instance of that piece of data to operate on (i.e. the scenario view is given an instance of a Scenario, the locational shift view is given an instance of a locational shift, etc.). The ”draw” functions access the characteristics of the instance to set up the current widget states in the proper way, and the ”button” (event handler) function updates the characteristics of the instance based on the state of the widgets. If the particular feature is being added, a new instance is created to use. If however the particular feature is being modified, a deep copy of the already existing feature is used. This is so that modification of a feature can be canceled (you saw the cancel buttons in the screenshots) and returned to its original state. If a new feature is cancelled, it’s simply thrown out. In both cases, pressing ”Done” is treated as the user’s acceptance of the state of the feature, and the instance being operated on is committed to the proper place in session data.

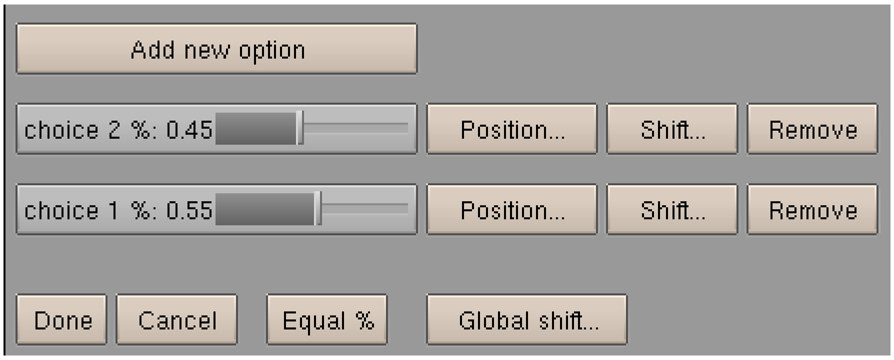


Figure 15: Locational Facet Specification View

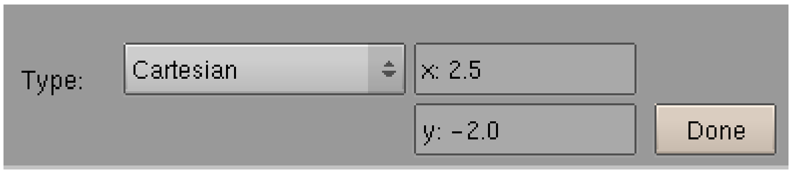


Figure 16: Position Specification View (Randomization)

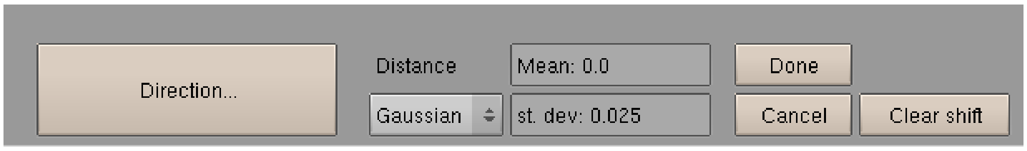
6.5 Scene-Building Mechanics

The obvious portion of implementation which has yet to be discussed is the actual mechanics of the scene rendering process (the "Rendering Mechanisms" block of the data flow diagram). The responsibilities of the rendering process for a scene are intuitive: Render each child entity of the given scene and place it as per its locational, rotational, and randomization data. Indeed, since a scene is defined as a collection of child entities, this will assemble the scene. The algorithms which carry out the process are structured recursively; reflective of the recursive nature of the data structure. Basically there are functions for creating and placing each type of feature, and there are type-independent post-conditions which must be satisfied. Believe it or not, most of the post-conditions pertain not to the actual feature, but to a polygon which is wrapped around the feature.

6.5.1 Polygon Wrapping

Although the rendering context and the object geometries are three-dimensional, the placement paradigm is two-dimensional. Regarding this, you may have noticed that all locational and rotational schemes are implemented two-dimensionally as well (directional specification limited to compass directions, no "above" or "below" keywords in the grammar). As you may have already guessed, all objects are laid out on the Z-plane, i.e. ground. This behavior is fine for simulations which assume a perfectly flat terrain (which is often the case), but not for simulations which require a scene to be laid out on un-even terrain. Fortunately, I believe that terrain integration is a very extractable problem and can be worried about later on. This topic will be re-visited in the Future Directions section, but for now it is a digression.

The point is that the z-coordinate has been taken out of the equation when it comes to feature placement, which means that the positional characteristics of any entity can be wholly represented by a 2-dimensional polygon. The polygon of an entity can be created as soon as the entity has been

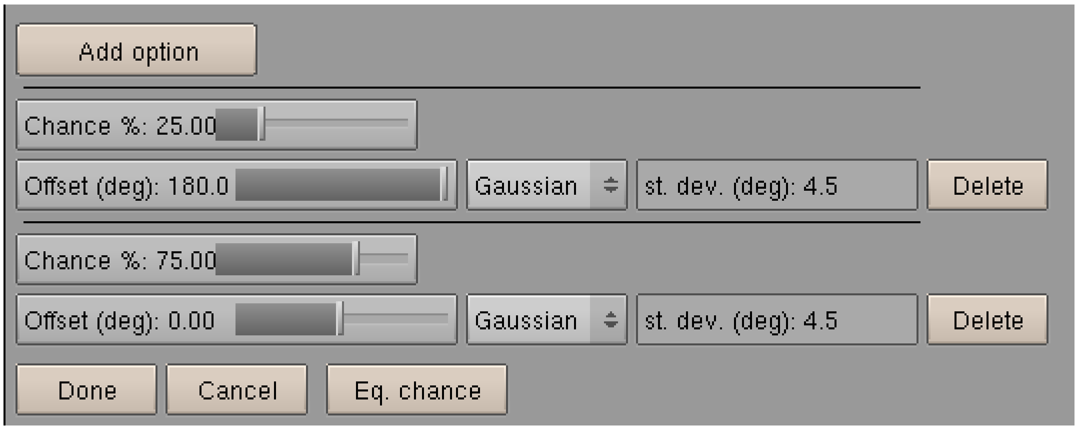

Direction...

Distance: Gaussian

Mean: 0.0

st. dev: 0.025

Figure 17: Locational Shift Specification View (Randomization)



Chance %: 25.00

Offset (deg): 180.0 | Gaussian | st. dev. (deg): 4.5

Chance %: 75.00

Offset (deg): 0.00 | Gaussian | st. dev. (deg): 4.5

Figure 18: Rotational Facet Specification View

assembled. An entity's polygon is defined as the convex polygon which encompasses the polygons of all its child entities, or which wraps around a single object. The former is the recursive case - where the entity is a scene or a group. The latter is the conceptual base case - where the entity is a single object (whether concrete or template).

Defining the polygon around a single object is just a matter of finding the points of the object's "bounding box". Blender maintains a "bounding box" for each object it currently has in memory. Mathematically it is the rectangular box formed by the minimum and maximum values, in each axis, achieved by the object's set of vertices. This is depicted in Fig. 19.

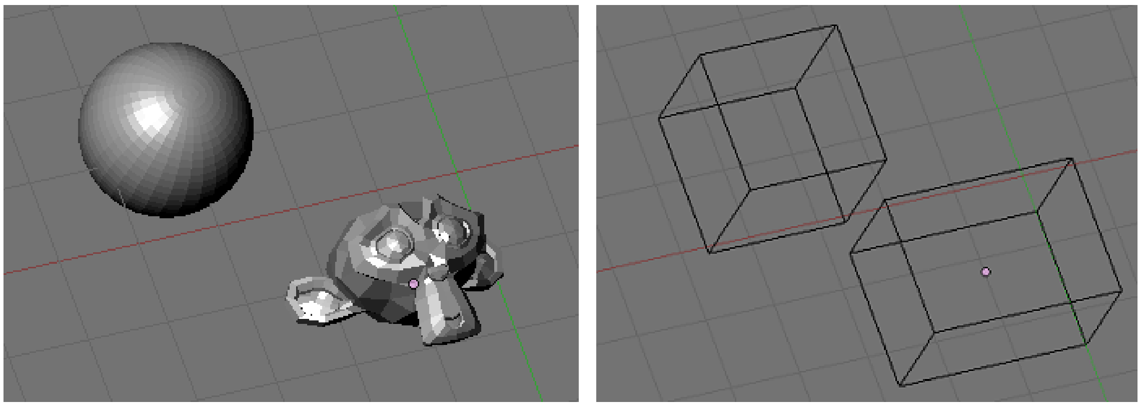


Figure 19: Object View (left) and Bounding-Box View (right)

The polygon is defined using the (x,y) coordinates of the lower rectangle of the object's bounding box.

Finding the polygon for a group-type or a scene-type entity requires an algorithm for finding the convex polygon around a set of points. For this, I decided to implement the "Gift wrapping" algorithm [8]. The only subroutine I needed to define for this was a simple function to determine which "side" (+/- by the right-hand rule) of a line a point is on.

An entity's polygon defines its characteristics in the context of a placement scheme. Geographically, the entities of a scene are placed in such a way as to avoid intersections between their polygons (except if the two entities are in a locational relationship). Another characteristic that is used is the center point of the entity, which in all cases is defined as the centroid of the polygon. An additional characteristic derived from the polygon is the "size" of the entity, which I define as the greatest length between a pair of points in the polygon (furthest pair).

Finding the centroid of a polygon is done by implementing a few summation equations [9]. Finding the size (as I have defined it) is a trivial, however $O(n^2)$ algorithm. The centroid acts as the center point of an entity, and entity movement and rotation is based on its centroid.

I ended up defining a class to represent an entity's polygon. The constructor accepts a rendered (but not yet positioned) entity, computes and saves the convex polygon as a list of points, then computes and saves the centroid and size. In addition to being a good place to put the functions associated with finding a convex polygon, centroid, and size, the class also contains functions used for moving and rotating a polygon. Whenever an entity is moved or rotated, its polygon must be moved or rotated by the same amount to stay representative of it. Moving a polygon to a specified (x,y) is achieved by moving its centroid to that point, and every member of its list of points by the same distance. Rotating a polygon by a certain theta is achieved by rotating each member of the list of points around the centroid (by applying a rotational matrix).

Figs. 20 and 21 show a scene rendering in its actual form followed by a conceptualization of each entity as a polygon. The scene contains a warehouse, a custom formation of shipping containers to the north of the warehouse, a "parking lot" sub-scene to the south of it, and a row of trees south of that.

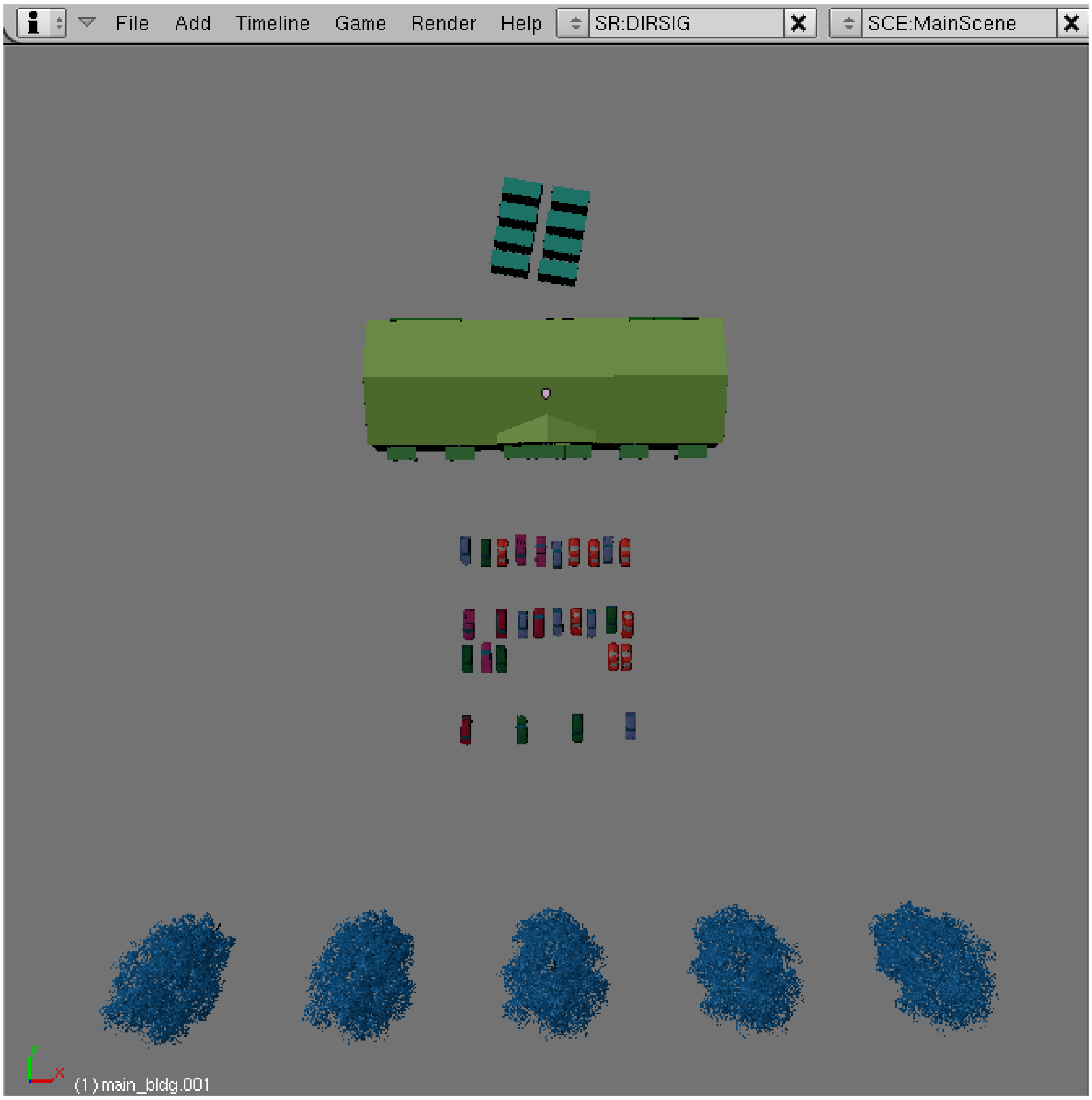


Figure 20: A Scene Rendering in its actual form

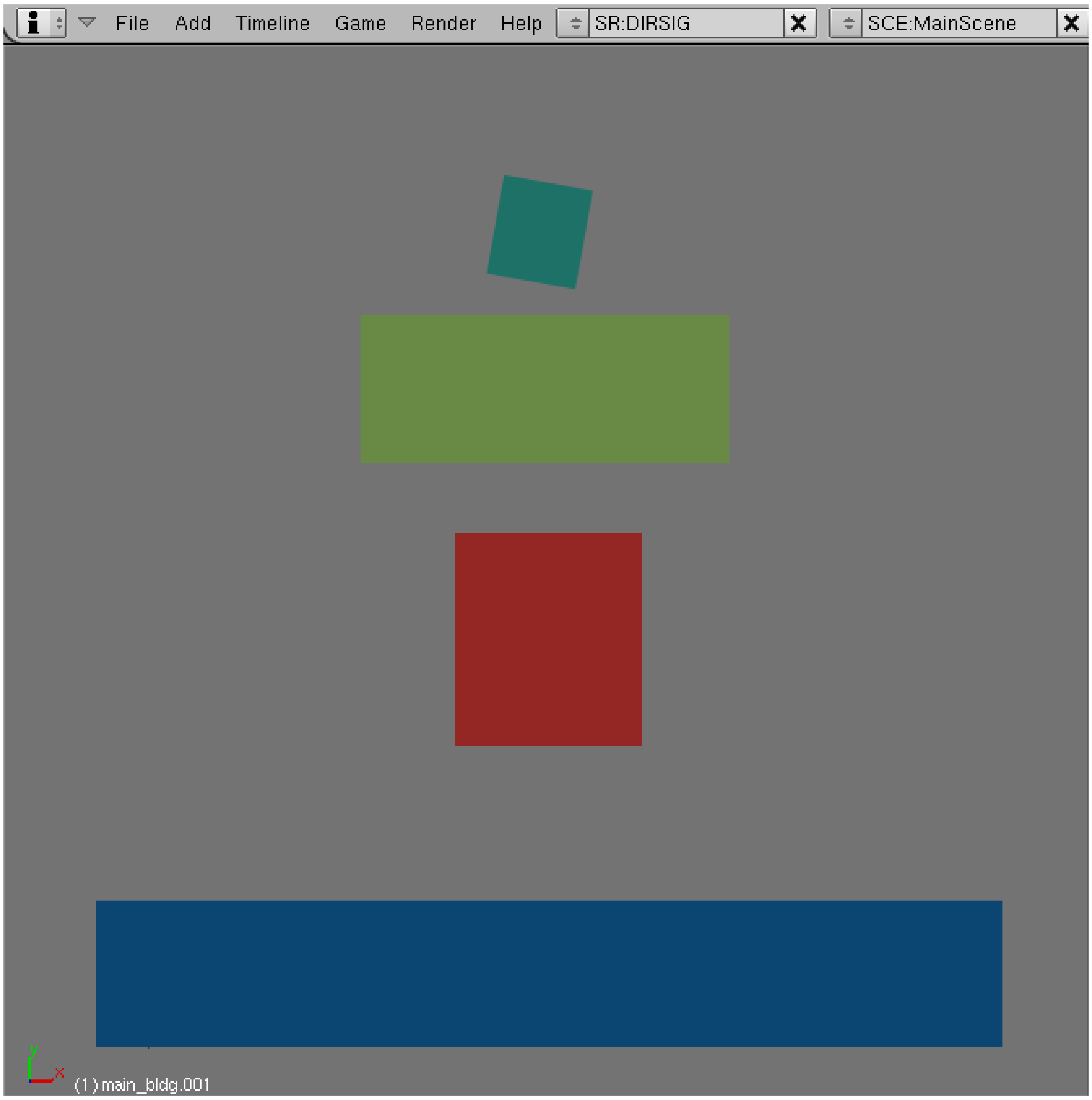


Figure 21: A Scene Rendering, conceptualized in polygonal form

6.5.2 Positioning Algorithms

For each type of entity (scene, concrete object, template object, group), there is a function whose responsibility it is to render it (render_scene, render_conc_obj, render_temp_obj, render_group). Each function adheres to three post-conditions:

1. All objects associated with the entity have been rendered in the rendering context and placed in their proper positions and rotations relative to each other. Every singular object contained in the entity has been given a reference to the actual object which appears in the rendering.
2. A polygon (instance of the polygon class) for the entity has been created, and a reference to it is given to the entity.

3. The entity is located at the origin (centroid is located at (0,0) in the rendering context) and its rotational state is set to 0 degrees (facing east).

Think of each rendering function as rendering the particular entity in a vacuum - for example, when rendering a single object, all you would do is create the object and place it at the origin in its native rotation (facing east). Only by containing the object in the scope of a scene can you logically apply locational and rotational information to it. In a vacuum, an object is just an object, but in the context of a scene (or group), location and rotation take meaning.

You might already see how this is going to work. The entry point to the rendering process is at render_scene, and the scene is built in a depth-first recursive process: Assembling the entities as it travels down, and placing them in the proper way in their parent context as it travels back up.

Think of the render_conc_obj function as something of a base case, where the goal is simply to render a single object. This is, after all the ONLY place where actual geometry files are accessed and imported into the rendering context. All the other rendering functions are abstractions which eventually find their way here. The command to import an object from an .obj file is

```
Blender.import_obj.load_obj(uri)
```

where "uri" is the full path of the desired file. Immediately after it is imported, the object will be the only one in the rendering that is in a "selected" state. A reference to the actual object is obtained by polling Blender for a list of currently selected objects and selecting the first (only) one:

```
obj = Blender.Object.GetSelected()[0]
```

As mentioned in the discussion of the environment, the actual interfacing to the rendering context is done by setting the locational and rotational coordinates of references to actual objects. Here, the object is set at the origin with a rotation of zero degrees. Lastly, the reference to the physical object is saved, and a polygon is created for the object and saved (as variables registered to the entity). The render_temp_obj function simply polls its randomization scheme for a choice of object to use, and from there renders it as a concrete object.

The render_scene and render_group function are very similar. For each child entity they need to include, it is rendered in a recursive call to the proper rendering function, and then placed in the proper way. For a group, it is simply a realization of the formational data of the group. For a scene, the proper way to place a child entity is to access its locational, rotational, and randomization information and determine its proper location and rotation in the context of the scene. If you recall the discussion of the structure of session data, it was mentioned that distance values are stored as unit-less float values - and that these would be used to determine the exact coordinates of the entity at render-time. Well, here we are. To get the actual distance value, the float value stored in the locational scheme is multiplied to the size of the entity, as derived from its polygon. So, for example, if a tree is specified as being a distance of "1.0" from its base (either a base entity or the origin), it will be placed one "tree"-length to the north of its base. If specified as "0.5", it will be placed half a tree-length away, etc. (this scheme is also applied to the "spacing" attribute of a group to determine the space between group members). This was my solution to working in an environment with arbitrary units - to use the size of an entity as a frame of reference.

Determining the proper position and rotation for an entity in its parent context is just half the battle. Actually moving and rotating them in the rendering context is the second half. I wrote functions to execute the moving and rotating of each type of entity. As you might expect, the movement and rotation functions for entities which aren't a single object are abstractions built from the ability to move and rotate one object at a time in the rendering context. An object's (whether concrete or template) move function simply sets the location of its physical object in the rendering context (as well as calling its polygon's move function), and the rotation function does the same for its rotation with respect to the z-axis (as well as call its polygon's rotate function). For a scene or group, the move and rotate functions are based on their existential definition as a collection of child entities. For these, the move function is still simple: just compute the delta of the desired move, and call the move function on each entity with whichever coordinates that move the entity by the delta. Also, call the move function of the parent entity's polygon. The rotate function is a little more complex. In concept, each child entity needs to be rotated around the centroid of the parent entity's polygon. To achieve this, each child entity must be moved to the spot given by a rotation matrix applied to the child entity's centroid to rotate it around the parent's centroid. Next, the child entity must be rotated by the exact amount the parent entity is being rotated by. Lastly, the rotation function of the parent entity's polygon must be called.

6.5.3 Collision Avoidance

As a rule, no pair of child entities of a scene may be placed such that their polygons intersect; except in the special case that one serves as the other's positional base. If the placement routines happen to place an entity in this way, it will be moved to a nearby spot. To do this, the tool uses a function to detect polygonal intersections - if found, the tool uses a function to place the offending entity in a nearby spot.

At a technical level, an intersecting pair of polygons is defined as two polygons for which there is a side s1 from the first polygon that intersects a side s2 from the second; OR as one polygon which completely envelops the other. The detection function operates on two polygons, and checks each possible pair of sides between the two for intersections. If an intersection is found, the function immediately returns "True", indicating the polygons collide. If it gets through each possible pair without finding an intersection, it proceeds to check for envelopment. Polygonal envelopment is defined as the case where all points of one polygon are on the same side of all lines of the other. If this is found to be the case, "True" is returned. If not, "False" is returned. Thus in the worst case, the detection routine is O(m*n1*n2), where "m" is the number of entities which exist in the scene (the detection function must be run on each established entity with the new entity), "n1" is the number of points of the first polygon, and "n2" is the number of points of the second polygon. Fortunately, the requirement that every polygon must be a convex wrapper around its entity keeps the number of points of each polygon relatively low in the typical case.

If a collision is detected, it is compensated for in a function which spirals the entity outward until there is no collisional state between it and the set of entities it needs to avoid (given as a parameter). It starts by picking a random direction and tries to move the entity in that direction by 1/20th of the offending entity's size. If that fails, it adds 90 degrees to the direction and tries again. If that fails, it adds 90 degrees and tries again. If that fails, it adds 90 degrees and tries one more time. If that fails, it adds another 1/20th of the entity's size to the distance and tries four more directions, repeating the entire process until there is no collision.

It may seem that collision detection and compensation could become quite computationally taxing, but again I would say that the amount of vertices of a typical polygon is kept low by the convex restriction. It's been my experience that the vast bulk of the computation involved in rendering a scene is the actual importing of the geometries from the .obj files.

6.5.4 Randomization Integration

Instead of being applied all at once, an entity's randomization scheme is "sprinkled in" to the rendering process. For example, recall that the first thing the template-rendering function accessed was the randomization scheme to determine which object of the available choices to use. What actually happens is that the "pick()" function of the "object options" facet of the template's randomization scheme is called, and returns an object picked at random based on its list of object options and associated weighted probabilities. Each of the other four facets of the randomization scheme have "pick()" functions as well, which return context-appropriate information. Fortunately, the Blender module contains a sub-module "Mathutils", which has a function "Rand()" that returns a random float between 0.0 and 1.0. This is called whenever one of the pick() functions require a random contribution.

Now you know how the "object options" facet of a randomization scheme is integrated. The following is a summary of where and how the other four are worked into the rendering process:

- Count: The count facet represents a randomization of the number of objects which appear in a group. Thus, it only comes into play in the render_group function. The first thing this function does is ascertain how many members should be in the group, so it can set up an iterative loop to produce them. If there is a randomization scheme with a specified count facet applied to the group, the count facet's "pick()" function is called and returns an integer derived from the numerical randomization context, whether it be range-based or gaussian-based. If not, the count integer specified to the group is used (it was trumped in the other case).

- Exist: Before a recursive call to render a child entity is even made (from rendering a scene or a group), the tool checks to see if the child entity has a randomization scheme with a specified "Exist" facet. If it does, the Exist facet's pick() function is called, and returns "True" or "False" as to whether the entity should even appear. Recall that part of the Exist facet is the "reserve" Boolean, indicating whether the space where the entity would have gone (if it's picked to not exist) should be reserved. If the space should not be reserved, the call to render the child entity is simply never made and it is never rendered. If the space should be reserved, then the child entity is rendered as usual, but marked for deletion. The last step of both the render_scene and render_group function is to remove the condemned entities.

- Location: Recall that a locational randomization can be specified as a weighted-probability list of positional options, each with an optional locational shift; and/or a global locational shift to be applied on top. In the case that there is a list of positional options, one of the options is picked to trump any positional information assigned to the entity. Whether this was the case or not, a locational shift is "picked" (either from just the shift applied to the choice, just the global shift, or from a conglomerate of both if they both exist) as a (dx,dy) where "dx" is the distance to shift the entity along the x-axis, and "dy" is the distance to shift along the y-axis. The pick() function for this facet returns the positional choice and/or the shift coordinates, depending on what exists in the facet's specification. The information received from this pick() function is used in render_scene to determine the coordinates to place the child entity at. It is also used in render_group to determine if a shift should be applied to the member that was just placed (but not to derive its actual position - this MUST be derived from the formation information assigned to the group).

- Rotation: In both the render_scene and render_group functions, the rotational state of a child entity is resolved immediately after its location. After the child entity is moved to the proper location, it checks to see if the child entity has a randomization scenario with a specified rotation facet. If not, the rotational information assigned to the entity is used. If it does, the rotation is picked from the rotation facet. The rotation facet is defined is a list of weighted-probability rotational options, each of which could either be an exact degree value, a range-based window, or a gaussian-based window. The pick() function will select one of the options, and select a rotation (in degrees) from that option to use. The return value is the amount of degrees the child entity should be rotated in the parent context.

6.5.5 Object Recycling

The object geometries I've been working with have varying degrees of detail. Most of the objects contain enough vertices to cause a noticeable lag while loading them into a scene. This results in a somewhat less than desirable wait time when rendering a scene. Subsequent re-renderings of the same scene not only took the same amount of time, but also forced Blender to allocate more memory to hold the new objects (the old objects may be unlinked from a scene, but still exist in memory until the entire session is terminated). Thus, rendering a scene too many times would eventually crash the program as it ran out of memory. My solution was to code in a book-keeping service to keep track of which objects came in under which URI so they can be re-used in subsequent re-renderings. Re-renderings are thus faster than primary renderings and do not waste memory.

7 Results

Empirically, the tool is a success insofar as it does everything which was in the specifications I drafted for it: It's capable of processing a natural-language script into a scene rendering, applying randomization schemes to scene features, and reading/writing all program data in XML. At a higher level, the tool's success can only be gauged by an assessment of its "helpfulness" in the various settings in which it may be used.

7.1 Assessing the Natural Language

Separating the natural-language processing from all other features of the tool is perhaps the only way to get an empirical assessment of the efficiency of the natural language for building scenes. This means no sub-scene encapsulation, templates, or randomizations of any kind (the usefulness of these features is difficult to quantify, but I will argue to their benefits later). What we are left with is an assessment of the efficiency of using the language to describe the position and rotation of every single object in a scene (a clause for each object). For this, I present the following example scene (Fig. 22), which models a small airport-type area for MiG-25 "Foxbat" fighter jets [10].

The scene contains 2 "bunker" objects, a hangar, two other generic buildings, a "fueltanks" object, a shack, four foxbats, three small trucks, and a helicopter. This is one of the "stock" scenes that are distributed with the stock DIRSIG installation. Using the singular object geometries and a screenshot of the scene as a reference, it took me about 10 minutes to assemble the scene in Blender by hand. When I say "by hand", I mean by using Blender's own interface to import the objects into the rendering context, then dragging them about and rotating them using the mouse. Next, I attempted to construct the same scene using the tool to process the following object-by-object natural-language description of the scene:

```
FoxbatScene {
FuelTanks as "fuel" in the east region
Bunker as "bunker1" in the west region facing E
```

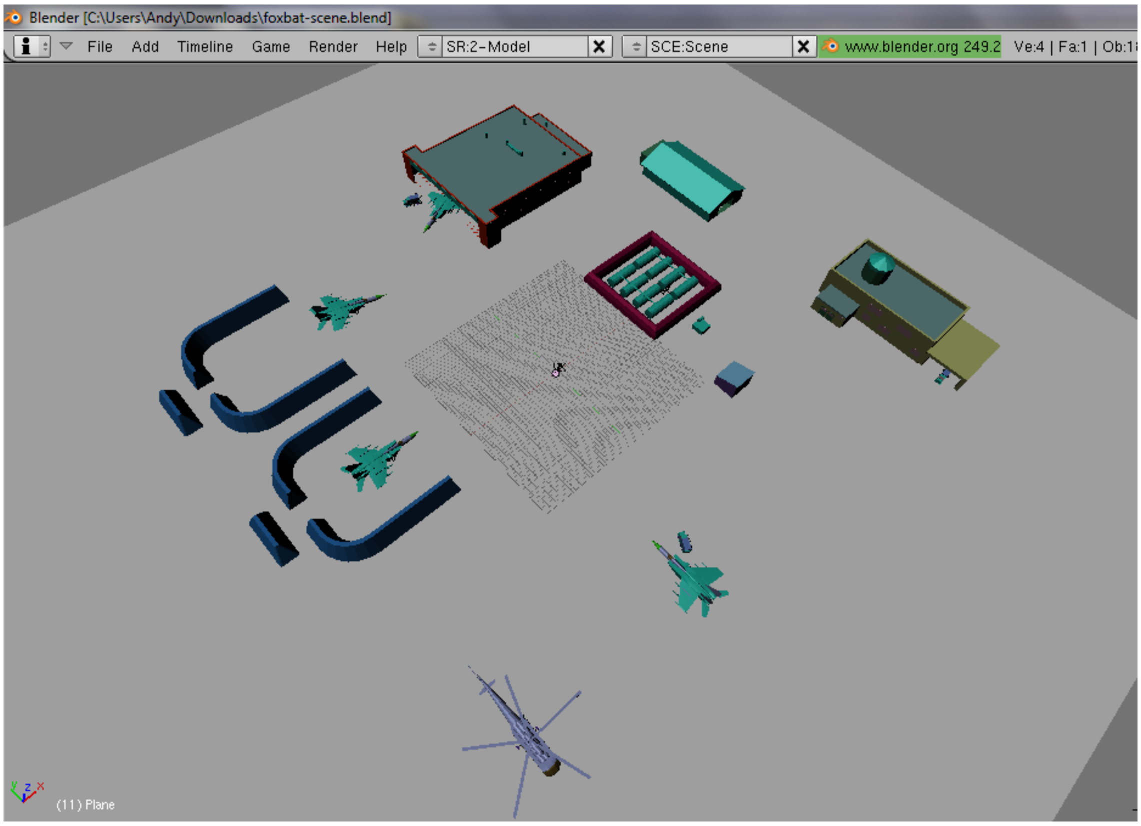


Figure 22: Foxbat Scene, Constructed by Hand

```
Bunker as "bunker2" to the very near north of bunker1 facing E
Foxbat inside bunker1
Foxbat east of and very near bunker2 facing SE
Hangar as "hangar" north of and near fuel facing west
Foxbat as "fox3" to the very near W of hangar facing SW
Truck to the very near northwest of fox3 facing nw
Building2 as "bldg2" southeast of and near hangar facing N
Building3 as "bldg3" south of bldg2 facing west
Truck very near to the southwest of bldg3 facing W
Shack south of and near fuel
Helicopter as "heli" south of bunker1 facing south
Foxbat as "fox4" east of heli facing N
Truck northeast of and very near fox4 facing ne
}
```

Processing the script and rendering the scene produced the rendering in Fig. 23.

Using the tool to locate the necessary geometries and writing the scene in the natural language took about 3.5 minutes (the time the tool took to process the script and produce the rendering was negligible). This is by no means evidence that the tool makes the scene building process 65% more efficient. For one thing, I obviously know my way around the tool's GUI, and I'm obviously

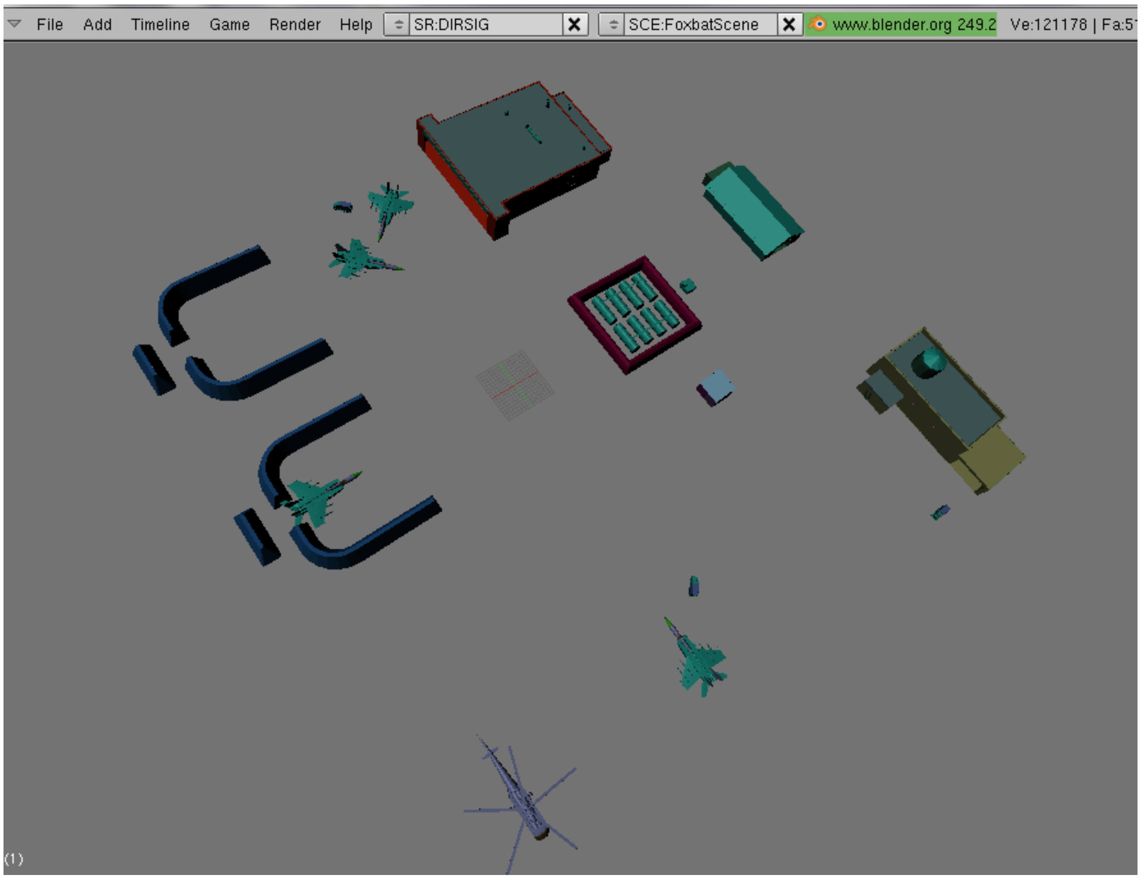


Figure 23: Foxbat Scene, Constructed by Tool

familiar with the keywords associated with the natural language. Learning-curve aside, you'll also notice that the positions of the three foxbats in the northwest area (near the hangar and bunkers) aren't quite where they appear in the previous screenshot. In a typical setting, I would expect the user to use Blender's mouse interface to the rendering context to fine-tune the positions of various objects to their liking. Here, I would take an additional minute or two to slide the foxbats into their proper places.

At this point, you may or may not be convinced of the tool's usefulness as an aid to build scenery, but I feel the real value is in the benefits of the features afforded by the data structure.

7.2 Benefits of Data Structure

When the project was presented to me in the summer of 2010, the first functional model I drafted for the tool had a text-based scene specification being sucked into a black box and output as a scene rendering. Basically, the tool would just be one mechanism to convert text to a rendering. The biggest evolutionary step in the tool's design was to insert the intermediary data structure between scene specification and scene rendering. The persistence of scene data not only allowed for editing via XML and GUI elements, but also allowed for the inclusion of what I feel are two major motifs of the tool: Sub-scene encapsulation, and randomization.

The ability to use a scene as a child entity in a parent scene allows the user to logically organize the features of a scene in whatever way they see fit. If the user wants to include a parking lot in a scene, but not have to worry about the vehicle-level detail of it in the context of this scene, this can be achieved by defining the parking lot as its own scene. If the user wishes to randomize the parking lot, the randomizations can be set up in the context of the parking lot scene and will still apply in whatever parent context the parking lot scene is used.

Fig. 24 is a screenshot is of such a scene, built and rendered with the tool.

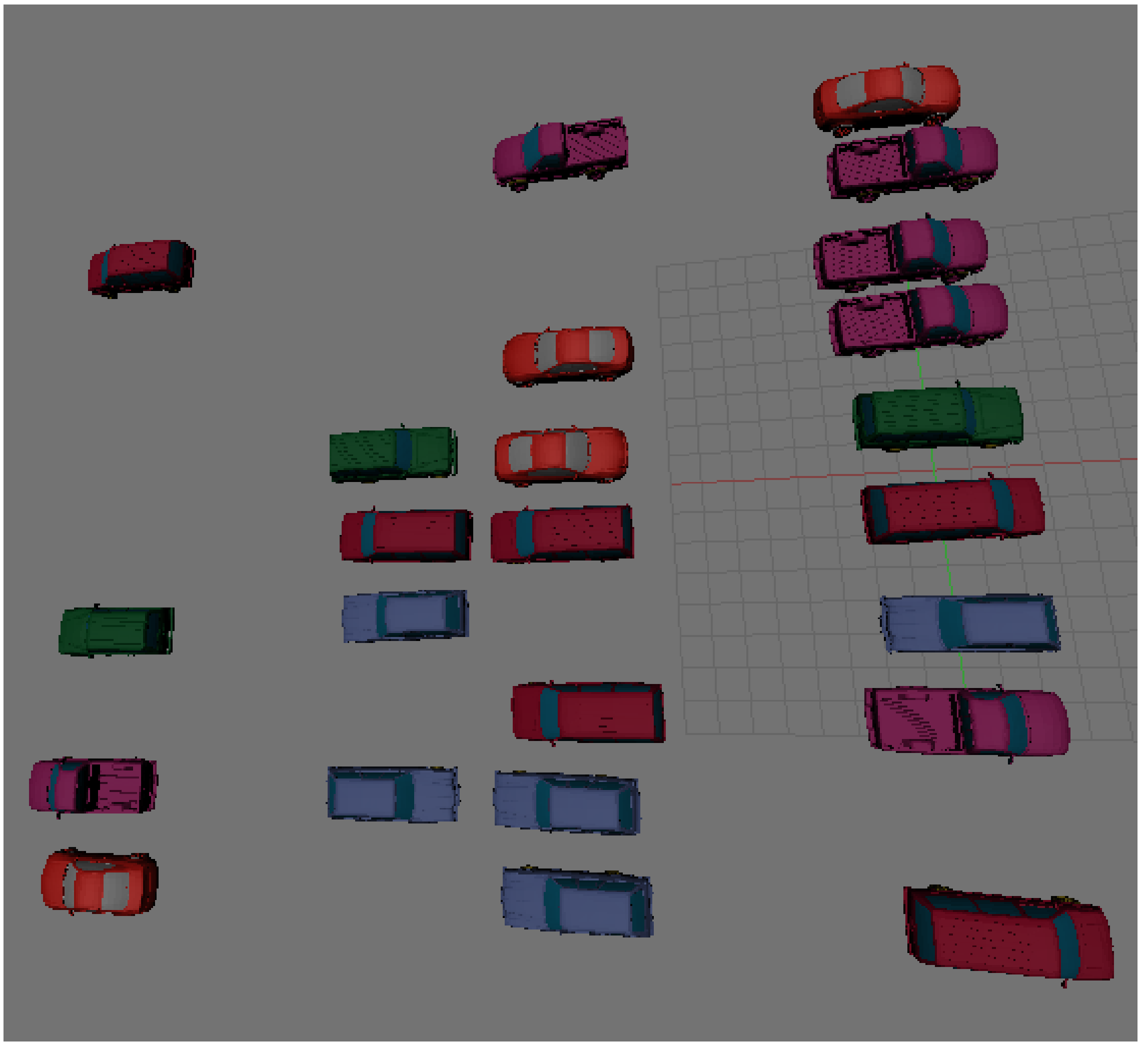


Figure 24: A Randomized Parking Lot, Rendered with the Tool

This scene, named "parking_lot", demonstrates the power that randomization schemes have to model high-level concepts such as the dynamics of a parking lot. To that, you'd expect a random assortment of vehicles to occupy the spaces, that the spaces toward the front of the lot would be more popular, that some might back in rather than pull in, and that no one parks perfectly.

All these concepts were realized in randomization schemes applied to the features of the parking lot. The first step was to create the template object "parking_space", defined as a randomization scheme with an equal choice between five vehicle objects (a station wagon, minivan, pickup, suv, and infiniti). Next, a scenario was created which specified a rotational choice of 75% for 0 degrees and 25% for 180 degrees (to represent pulling in/backing in), and a small guassian locational and rotational shift (to represent imperfect parking). This scenario was applied to the randomization scheme of the parking_lot to complete the definition of the parking_space. The actual parking_lot scene was built as four rows of ten parking_spaces each. To represent desirability of the spaces toward the front (east) end of the lot, different existence probabilities were suggested to each row. The first row got 95%, the second got 75%, the third got 50%, and the fourth got 25%.

When used in conjunction, sub-scene encapsulation and randomization could be leveraged to model just about anything. With sub-scenes, a scene specification can be built from any amount of abstractions needed, and since randomizations may be applied at any layer, the amount of behaviors which can be modeled are limited only by the imagination of the user.

As previously mentioned, the benefits of sub-scenes and randomizations are apparent but difficult to quantify. After all, abstracting scene components as potentially multi-object entities is a paradigm-shift from procedurally building a scene one object at a time. Also, the benefits of subtle randomizations in the context of a simulation may not be enough to waste time and endure the monotony of creating parallel scenes by hand. For example, the effort required to create 10 different-looking parking lots by hand as part of a scene to run 10 simulations on would probably outweigh the potential impact the subtle randomizations would have on the simulation. However, with a way to specify what a parking lot CAN be rather than just what a parking lot IS, it becomes worth the effort. That is a good summary of the benefit of randomization schemes - that they are a way to describe what a scene CAN be rather than what it IS. Once a scene is encapsulated and randomized in such a way as to represent the variability of the area being modeled, the tool can automatically render as many realizations of that scene as needed for the simulations.

8 Future Directions

8.1 Terrain Integration

It may have been un-satisfying that none of the screenshots of the scene renderings which you have seen included a terrain. Currently, all objects are snapped to the x-y plane (z=0) and their euler rotations are set to zero with respect to the x-axis and the y-axis (standing straight up). If a flat, perhaps paved surface is assumed, you could simply create a flat plane object along the x-y plane and use that as the terrain. However, in some more rural scenes you'd expect an uneven terrain - and that objects would snap to that. Conceptually, this is a behavior which is easily separable from the existing placement routines.

Recall that the tool has only been concerned with resolving locations in 2-dimensional coordinates (x,y). If a terrain can be thought of as simply an elevation map, i.e. $f(x,y) \Rightarrow z$, then any (x,y) coordinates can be plugged into the elevation map function and receive the proper elevation.

As I see it, there are two cases involved with placing an object with respect to a terrain. The first case is for the object to be placed "on" the terrain, and the second is to place it "in" the terrain. Say you want to place a house and a car with respect to a terrain. You'd expect the car to be resting on the terrain, but you'd expect the house to be embedded into the terrain. In the first case, the challenge is in determining the proper z-coordinate and the proper set of euler rotations to apply to the object so that it "settles" properly on the terrain. In the second case, the challenge would be in automating the process of extending the foundation of the object down until there are no gaps between it and the terrain.

8.2 Integration of Variables into the Natural Language

One of the limitations of the natural language is the lack of support for variables. For example, which of the following scene scripts do you think would be more useful?

```
Parking_Lot {
N/S row of 10 parking_spaces as "row1" facing east
N/S row of 10 parking_spaces as "row2" to the near W of row1 facing W
N/S row of 10 parking_spaces as "row3" to the very near W of row2 facing E
N/S row of 10 parking_spaces as "row4" to the near W of row3 facing W
}

Parking_Lot(X) {
N/S row of X parking_spaces as "row1" facing east
N/S row of X parking_spaces as "row2" to the near W of row1 facing W
N/S row of X parking_spaces as "row3" to the very near W of row2 facing E
N/S row of X parking_spaces as "row4" to the near W of row3 facing W
}
```

Basically, a scene definition could accept parameters to act as or modify the numerical details associated with a scene. When referencing a parking_lot to use as a sub-scene, you could reference it as "parking_lot(20)" for a parking lot with 20 spaces per row. This behavior could be extended to templates and randomization schemes as well. For example, a parking space template could be referenced as parking_space(0.5) to get a parking space which is 50% likely to produce a vehicle.

Continuing this line of thinking to its completion results in the natural language becoming a programming language, and this would be a significant departure from the original project specification. To make sure the natural language stays accessible to everyone, variable integration would have to be done carefully.

8.3 Standard Geometry Library

Building the geometries of single objects is something which I have always treated as being out-of-scope of the tool's functionality. I have always thought of the tool as being for object placement, not object creation. Thus, a pre-condition of the tool's deployment has to be the existence of a library of geometry files which the tool can access.

This falls under the maintenance category of the software development process. Any distribution of the tool would have to include a standard repository of geometry files, or an online repository would be created and set up so that the tool can access it remotely. I envision the latter as the more desirable case. A single repository could be more easily maintained and expanded on.

There are some standards which object geometries must adhere to. Objects which are meant to be used in the same context MUST be of reasonable sizes as they relate to each other (a car should not be as big as an office building, for example). Also, since the tool operates under the assumption that the east direction is zero degrees (north being 90, west being 180, south being 270), object geometries must be saved in such a way that they're facing east - this is so that their "native" zero-degree rotational state has them facing east. For some objects, this won't matter (which way is a tree facing?). Part of the vetting process for including an object in the repository would be for the librarian to use their best judgment to ascertain if the geometry has a logical, intrinsic "heading", and if so, save it so that it's facing east.

On the implementation side of things, the tool could be expanded to accept multiple formats of object geometries. There are more formats the Imaging Science dept. uses for object geometries than the "wavefront" .obj format, several for which scripts have already been written to import/export them to/from Blender. It would really just be a matter of plugging these scripts into the tool.

8.4 Street Networks

A particularly useful feature to integrate into the tool would be the ability to build a scene based on a street network. Given the frequency in which urban and suburban themed areas are created for simulation purposes, this will most certainly be looked in to in the near future. The biggest challenge here is how to model the implications which arise from carving the workable area into lots separated by the streets. This would be something of a paradigm-shift in the structure of a scene: Currently, the entities of a scene are self-deterministic (they know how to produce themselves). The implication involved with introducing a street network is that there should be some oversight in the production of individual entities so that they interact with the network in a logical way. For example, you could create a scene called "city_block", which could produce a single block of an urban area, but you wouldn't be able to simply place an instance of city_block in every lot created by the network because not every lot is going to take the same shape. The "oversight" would need to include mechanisms for things like snapping the fronts of buildings along the edges of lots.

Fortunately, there is a software program called CityEngine [11] which solves almost the exact same problem. CityEngine uses fractal and recursive techniques to define how a lot can be constructed while taking into account the variability of the shape and size of the lot. I will assuredly use this program for inspiration when crafting my own solution into the tool.

8.5 Speech to Scene

One neat idea which was mentioned to me is to have the tool accept not only text-based natural-language input, but also speech-based natural-language input. After all, talking is faster than typing. In this context, it would probably be best to devise a scheme to dynamically update a scene as the user talks, instead of simply saving their spoken clauses as scene data. The user's speech would be interpreted more informally, perhaps with the tool simply listening for references to usable features and placement-specific keywords - then making its "best guess" of the user's intent. I'd imagine that if a sentence starting with the word "No" is received, it would undo its last placement action.

A possible lead toward implementation is the python module "speech.py", which provides "a clean interface to Windows speech recognition and text-to-speech capabilities" [12].

References

[1] *Blender*, http://www.blender.org/. 2 May 2011.

[2] *Blender's Draw Module*, http://www.blender.org/documentation/248PythonDoc/Draw-module.html. 2 May 2011.

[3] *Creation of a GUI (Graphical User Interface) for Blender for your python scripts*, http://www.cs.indiana.edu/classes/a202-dger/fall2006/bpxi06/blender-lg-02-en.pdf. 2 May 2011.

[4] *Blender.org - Features*, http://www.blender.org/features-gallery/features/. 2 May 2011.

[5] *pyparsing*, http://pyparsing.wikispaces.com/. 2 May 2011.

[6] *ElementTree*, http://effbot.org/zone/element-index.htm. 2 May 2011.

[7] *Wavefront .obj file*, http://en.wikipedia.org/wiki/Wavefront_.obj_file. 2 May 2011.

[8] *Gift Wrapping Algorithm*, http://en.wikipedia.org/wiki/Gift_wrapping_algorithm. 2 May 2011.

[9] *Centroid*, http://en.wikipedia.org/wiki/Centroid#Centroid_of_polygon. 2 May 2011.

[10] *Mikoyan-Gurevich MiG-25*, http://en.wikipedia.org/wiki/Foxbat. 2 May 2011.

[11] *CityEnginge*, http://www.procedural.com/. 2 May 2011.

[12] *Speech 0.5.2*, http://pypi.python.org/pypi/speech/. 2 May 2011.