Scenic: Language-Based Scene Generation

Daniel Fremont
Xiangyu Yue
Tommaso Dreossi
Shromona Ghosh
Alberto L. Sangiovanni-Vincentelli
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-8
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-8.html

April 18, 2018

Scenic: Language-Based Scene Generation

Daniel J. Fremont
Logic and the Methodology of Science
University of California, Berkeley
dfremont@berkeley.edu

Xiangyu Yue
Electrical Engineering and Computer Sciences
University of California, Berkeley
xyyue@berkeley.edu

Tommaso Dreossi
Electrical Engineering and Computer Sciences
University of California, Berkeley
tommasodreossi@berkeley.edu

Shromona Ghosh
Electrical Engineering and Computer Sciences
University of California, Berkeley
shromona.ghosh@berkeley.edu

Alberto Sangiovanni-Vincentelli
Electrical Engineering and Computer Sciences
University of California, Berkeley
alberto@berkeley.edu

Sanjit A. Seshia
Electrical Engineering and Computer Sciences & Logic and the Methodology of Science
University of California, Berkeley
sseshia@berkeley.edu

Figure 1: Three scenes generated from a single Scenic scenario representing bumper-to-bumper traffic.

ABSTRACT

Synthetic data has proved increasingly useful in both training and testing machine learning models such as neural networks. The major problem in synthetic data generation is producing *meaningful* data that is not simply random but reflects properties of real-world data or covers particular cases of interest. In this paper, we show how a *probabilistic programming language* can be used to guide data synthesis by encoding domain knowledge about what data is useful. Specifically, we focus on data sets arising from *scenes*, configurations of physical objects: for example, images of cars on a road. We design a domain-specific language, Scenic, for describing *scenarios* that are distributions over scenes. The syntax of Scenic makes it easy to specify complex relationships between the positions and orientations of objects. As a probabilistic programming language, Scenic allows assigning distributions to features of the scene, as well as declaratively imposing hard and soft constraints over the scene. A Scenic scenario thereby implicitly defines a distribution over scenes, and we formulate the problem of sampling from this distribution as *scene improvisation*. We implement an improviser for Scenic scenarios and apply it in a case study generating synthetic data sets for a convolutional neural network designed to detect cars in road images. Our experiments demonstrate the usefulness of our approach by using Scenic to analyze and improve the performance of the network in various scenarios.

UC Berkeley EECS Technical Report, April 18, 2018
2018.

KEYWORDS

scenario description language, synthetic data, deep learning, probabilistic programming, automatic test generation, fuzz testing

1 INTRODUCTION

Machine learning (ML) is increasingly used in safety-critical applications, thereby creating an acute need for techniques to gain higher assurance in ML systems [1, 34, 35]. The traditional ML approach to this problem is to test the learned model[1] in its environment, gathering more data, and retraining if performance is inadequate. However, collecting real-world data can be slow and expensive, as it must be preprocessed and correctly labeled before use. Furthermore, it can be hard to observe and reproduce corner cases that are rare but nonetheless necessary to test against (for example, a car accident). As a result, recent work has investigated training and testing models with *synthetically generated data*, which can be produced in bulk with correct labels and giving the designer full control over the distribution of the data [17, 19, 39].

Generating *meaningful* synthetic data can be a nontrivial task since the input space of ML models is often large and unstructured. This is certainly true of the domain we consider in this paper: *scenes* comprising configurations of objects in space. Suppose we wanted a data set consisting of images of cars on a road. If we simply sampled uniformly at random from all possible configurations of, say, 12 cars, we would get data that was at best unrealistic, with cars facing sideways or backward, and at worst physically impossible, with cars intersecting each other. Instead, we want scenes like those

[1]The term "model" is commonly used in machine learning to refer to the learned classifier/predictor, e.g., a neural network.

in Fig. 1, where the cars are laid out in a consistent and realistic way. Furthermore, we may want scenes that are not only realistic but represent particular *scenarios* of interest, e.g., parked cars, cars passing across the field of view, or bumper-to-bumper traffic as in Fig. 1. In general, we need a way to *guide* data generation toward scenes that make sense for our application.

In this paper, we take a programming languages approach, designing a domain-specific *scenario description language*, Scenic. A scenario is a distribution over scenes. Scenic is thus a *probabilistic* programming language, allowing the user to programmatically specify a scenario of interest. Furthermore, it allows the user to both construct objects in a straightforward imperative style and impose hard and soft constraints declaratively. Scenic also provides readable, concise syntax for common geometric relationships that would otherwise require complex nonlinear expressions and constraints. In addition, Scenic provides a notion of classes allowing properties of objects to be given default values depending on other properties: for example, we can define a `Car` so that by default it faces in the direction of the road at its position. Finally, Scenic provides an easy way to generalize a concrete scene by automatically adding noise.

The variety of constructs in Scenic makes it possible to write scenarios anywhere on a spectrum from concrete scenes (i.e. individual test cases) to extremely broad classes of abstract scenes (see Fig. 2). A scenario can be reached by moving along the spectrum from either end: the top-down approach is to progressively constrain a very general scenario, while the bottom-up approach is to generalize from a concrete example (such as a known failure case), for example by adding random noise.

Probably most usefully, one can write a scenario in the middle which is far more general than simply adding noise to a single scene but has much more structure than a completely random scene: for example, the traffic scenario depicted in Fig. 1. We will illustrate all three ways of developing a scenario in this paper.

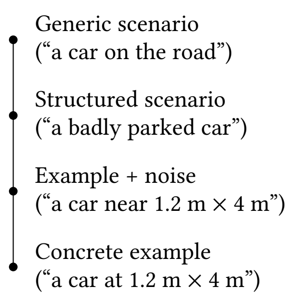

Figure 2: Spectrum of scenarios from general to specific.

Generating concrete scenes from a Scenic scenario requires sampling from the probability distribution it implicitly defines. This problem, while closely related to the general probabilistic programming inference problem [16], is theoretically interesting in its own right. We call it *scene improvisation*, as it can be viewed as a variant of *control improvisation* [10, 11], a class of problems involving the random generation of sequences subject to hard and soft constraints as well as distribution requirements.

Finally, we demonstrate the usefulness of Scenic with a case study testing and improving the reliability of a convolutional neural network designed to perform object detection in autonomous cars. We implemented an improviser for Scenic scenarios and used it to generate scenes which were rendered into images by Grand Theft Auto V (GTAV[2]), a high fidelity graphics videogame. Our experiments illustrate several ways Scenic can be used:

- generating specialized test sets to assess the accuracy of the ML system under particular conditions (e.g. in rain);
- generating instances of hard cases for the system so that they can be emphasized when retraining, improving accuracy in the hard case without impacting the typical case;
- generalizing a known failure case in many directions, exploring the sensitivity of the system to different features and developing a more general scenario for retraining.

These experiments show that Scenic is a very useful tool for understanding and improving the performance of an ML system.

In summary, the main contributions of this work are:

- Scenic, a domain-specific probabilistic programming language for describing *scenarios* that are distributions over configurations of physical objects;
- *scene improvisation*, an approach to generating a diverse set of concrete scenes from a Scenic scenario that draws inspiration from control improvisation [11];
- an implementation of an improviser for Scenic scenarios, with an interface to GTAV for producing realistic images;
- a case study showing how Scenic can be used in practice to analyze and improve the accuracy of SqueezeDet, a practical deep neural network for autonomous driving [42].

The paper is structured as follows: we begin by discussing related work in Sec. 2. Section 3 gives examples highlighting the major features of Scenic and motivating various choices in its design. In Sec. 4 we describe Scenic's syntax and semantics in detail, and in Sec. 5 we discuss the *scene improvisation* problem. Section 6 describes the experimental setup and results of our car detection case study. Finally, we conclude in Sec. 7 with a summary and directions for future work.

2 RELATED WORK

Data Generation and Testing for ML. There has been a large amount of work on generating artificial data sets for specific applications, including text recognition [19], text localization [17], robotic object grasping [39], and autonomous driving [8, 20]. Closely related is work on *domain adaptation*, which attempts to correct differences between synthetic and real-world input distributions. Domain adaptation has enabled synthetic data to successfully train models for several other applications including 3D object detection [23, 36], pedestrian detection [40], and semantic image segmentation [33]. Such work provides important context for our paper, showing that models trained exclusively on synthetic data sets (possibly domain-adapted) can achieve acceptable performance on real-world inputs. The major difference in our work is that we do not focus on any specific application but provide, through Scenic, a general way to specialize data generation for *any* application whose data derives from scenes.

Some works have also explored the idea of using adversarial examples (i.e. misclassified examples) to retrain and improve models [41, 43]. Some of these methods generate misclassifying examples by looking at the model gradient and by finding minimal input perturbations that lead to a misclassification [14, 26, 29, 38]. Other techniques assume the model to be gray/black-boxes and focus on

[2]GTAV: https://www.rockstargames.com/

input modifications or high-level properties of the model [5, 18, 21, 31]. Finally, Generative Adversarial Networks (GANs) [13], a particular kind of neural network able to generate synthetic data, have been used to augment training sets [22, 24]. The difference with Scenic is that GANs require an initial training set or pretrained model, while Scenic produces synthetic data involving only simulators.

Model-Based Test Generation. Techniques to guide generation towards useful outputs have been proposed in both automated testing and procedural generation [3]. A popular approach is to provide *example outputs*, as in mutational fuzz testing [37] and example-based scene synthesis [9]. While these methods are easy to use, they do not provide fine-grained control over the generated data. Another approach is to give *rules* or a *grammar* specifying how the data can be generated, as in generative fuzz testing [37] and procedural generation from shape grammars [27]. While grammars provide much greater control, they are hard to read and write and do not easily allow enforcing global properties. Readability and ease of use can be improved by instead providing a *program* in a domain-specific language with nondeterminism [7]. Conversely, directly writing *constraints* as in constrained-random verification [28] allows global properties but is even more difficult than writing a grammar. Scenic improves on these methods by simultaneously providing fine-grained control, enforcement of global properties, specification of probability distributions, and the ease of use of an imperative programming language.

Probabilistic Programming Languages. The semantics (and to some extent, the syntax) of Scenic are similar to other probabilistic programming languages such as Prob [16], Church [15], and BLOG [25]. In probabilistic programming the focus is usually on *inference* rather than *generation*, and in particular to our knowledge probabilistic programming languages have not previously been used for test generation. However, the most popular inference techniques are based on sampling and so could be directly applied to the scene improvisation problem, as we discuss in Sec. 5.

3 OVERVIEW

We use several Scenic scenarios from our autonomous car case study to motivate and illustrate the main features of the language. Appendix A shows images rendered from the scenarios described in this section, as well as more complex scenarios such as that in Fig. 1, with complete Scenic code.

Basics: Objects, Regions, Vector Fields, Distributions, Defaults. To start, suppose we want scenes of one car viewed from another on the road. We can simply write:

```
1 import carLib
2 ego = Car
3 Car
```

First, we import a library containing everything specific to our case study: the definition of `Car` as a type of object, as well as information about locations and directions of roads (from now on we suppress this line). Scenic itself contains only general geometric operations and predicates that we will see below.

The second line creates a `Car` and assigns it to the special variable `ego` specifying the *ego object* which is the reference point for the scenario. In particular, rendered images from the scenario are from the perspective of the ego object.

Figure 3: A scene generated from a simple scenario.

Finally, the third line creates an additional `Car`. Notice that we have not specified anything about where the cars are or how they are oriented; despite this, the scenario will always yield reasonable scenes: an example is shown in Fig. 3. This is because Scenic enforces several *default requirements*: all objects must be contained in the workspace, must not intersect each other, and must be visible from the ego object. Furthermore, Scenic allows defining *default values* for all object properties. The definition of `Car` in `carLib` begins as follows (slightly simplified):

```
class Car:
    position: Point on road
    heading: roadDirection at self.position
```

Here `road` is a *region*, one of Scenic's primitive types, defined in `carLib` to specify which points in the workspace are on a road. Similarly, `roadDirection` is a *vector field* specifying the prevailing traffic direction at such points. The operator F `at` X simply gets the direction of the field F at point X, so the default value for a car's `heading` is the road direction at its `position`. The default `position`, in turn, is a `Point on road` (we will explain this syntax shortly), which means a *uniformly random* point on the road.

So our scenario, despite being so brief, will yield images where the cars are positioned realistically. In fact, the rest of the definition of `Car` in `carLib` specifies reasonable default distributions for car model and color, so these aspects of the cars will also be plausible.

On the other hand, our scenario is still extremely general ("one car, anywhere on the road"), and we might want to specify the cars' positions more precisely. To create a car that is 20–40 m ahead of the camera, for example, we could write:

```
Car offset by (-10, 10) @ (20, 40)
```

The interval notation `(`X`, `Y`)` creates a uniform distribution on the interval, and X `@` Y creates a vector from xy coordinates (as in Smalltalk [12]). This vector is then used as an offset to position the `Car`, and since no reference point is explicitly given the ego object is used. So the car is placed randomly up to 10 m left or right of the ego object and 20–40 m ahead (again, automatically conditioned on the fact that the car does not intersect buildings, etc.). This illustrates another general principle in Scenic, that *relative positions, headings, and distances are with respect to the ego object by default.* This makes it possible to write compact code by assigning an object to `ego` and building a local part of the scenario around it (possibly reassigning `ego` to construct another part, eventually leaving it assigned to the desired camera location).

Using `offset by` as above overrides the default position of the `Car`, but leaves the default orientation ("face along the direction of the road") unchanged. Suppose for greater realism we don't want to require the car to be *exactly* aligned with the road, but to be within say 5° of it. We could try:

```
Car offset by (-10, 10) @ (20, 40), \
    facing (-5, 5) deg
```

but this is not quite what we want, since this sets the orientation of the `Car` in *global* coordinates (i.e. within 5° of North). Instead we can use Scenic's general operator X `relative to` Y, which can interpret vectors and headings as being in a variety of local coordinate systems:

```
Car offset by (-10, 10) @ (20, 40), \
    facing (-5, 5) deg relative to roadDirection
```

Note that `roadDirection` is a vector field and so defines a different coordinate system at each point in space: here Scenic automatically uses the `position` of the `Car` being defined. If we wanted the heading to be up to 5° off of the ego car's orientation instead, we could simply write `(-5, 5) deg relative to ego`.

Readable Constructors. So far we have seen `offset by` X as a way of specifying relative positions, and `facing` Y for specifying orientations, but the syntax for these may seem unusual compared to typical constructors in object-oriented languages. There are two reasons why Scenic uses this kind of syntax: first, readability. The second reason is more subtle and based on the fact that in natural language there are many ways to specify positions, orientations, and other properties, some of which interact with each other. Consider the following ways one might describe the location of an object:

(1) "is at position X" (absolute position);
(2) "is just left of position X" (position based on orientation);
(3) "is 3 m left of the taxi" (a local coordinate system);
(4) "is one lane left of the taxi" (another local coordinate system);
(5) "appears to be 10 m behind the taxi" (relative to line of sight);

These are all fundamentally different from each other (e.g. (3) and (4) differ if the taxi is not exactly parallel to the lane), and in different situations can each be the most natural way to define a position.

Furthermore, these specifications combine other properties of the object in different ways: to place the object "just left of" a position, we must first know the object's `heading`; whereas if we wanted to face the object "towards" a location, we must first know the object's `position`. There can be chains of such *dependencies*: for example, "the car is 0.5 m left of the curb" means that the *right edge* of the car is 0.5 m away from the curb, not the car's `position`, which is its center. So computing the `position` requires knowing the car's `width`, which in turn depends on the car's `model`. In a typical object-oriented language, this might be handled by computing values for `position` and other properties and passing them to a constructor. For "a car is 0.5 m left of the curb" we might write:

```
m = Car.defaultModelDistribution.sample()
pos = curb.offsetLeft(0.5 + m.width / 2)
Car(pos, model=m)
```

Notice how `m` must be used twice, because `m` determines both the model of the car and (indirectly) its position. This is inelegant and breaks encapsulation because the default model distribution is used outside of the `Car` constructor. The latter problem could be fixed by having a specialized constructor,

```
CarLeftOfBy(curb, 0.5)
```

but these would proliferate since we would need to handle all possible combinations of ways to specify different properties (e.g. do we want to require a specific model? Are we overriding the width provided by the model for this specific car?). Instead of having such monolithic constructors, Scenic factors the definition of objects into potentially-interacting but syntactically-independent parts:

```
spot = curb offset by -0.5 @ 0
Car left of spot, with model BUS
```

Here `left of` X and `with model` M are *specifiers* which do not have an order, but which *together* specify the properties of the car. Scenic works out the dependencies between properties (here, `position` is provided by `left of`, which depends on `width`, whose default value depends on `model`) and evaluates them in the correct order. To use the default model distribution we would simply leave off `with model BUS`; keeping it affects the `position` appropriately without having to specify `BUS` more than once.

Specifying Multiple Properties Together. Recall that we defined the default `position` for a `Car` to be a `Point on road`: this is an example of another specifier, `on` *region*, which specifies `position` to be a uniformly random point in the given region. This specifier illustrates another feature of Scenic, namely that more complex specifiers can specify multiple properties at a time. Consider the following scenario, which creates a parked car given a region `curb` defined in `carLib`:

```
spot = OrientedPoint on visible curb
Car left of (spot offset by -0.25 @ 0)
```

The function `visible` *region* returns the part of the region that is visible from the ego object. The specifier `on visible curb` will then set `position` to be a uniformly random visible point on the curb. We create `spot` as an `OrientedPoint`, which is a built-in class that defines a local coordinate system by having both a `position` and a `heading`. The `on` *region* specifier can also specify `heading` if the region has a preferred orientation (a vector field) associated with it, as in our example, where `curb` is oriented by `roadDirection`, the nominal traffic direction. So `spot` is, in fact, a uniformly random visible point on the curb, oriented along the road. Then `spot offset by -0.25 @ 0` shifts `spot` 0.25 m left in its own local coordinate system, i.e. away from the curb, and finally we place the car to the left of the resulting position. So as desired we get a car parked on the side of the road at a random place, with a 0.25 m gap between it and the curb regardless of the type of car.

In fact, Scenic makes it easy to elaborate the scenario without needing to alter the code above. Most simply, we could specify a particular model or non-default distribution over models by just adding `with model` M to the definition of the `Car`. More interestingly, we could produce a scenario for *badly* parked cars by adding two lines:

```
spot = OrientedPoint on visible curb
badAngle = Uniform(1.0, -1.0) * (10, 20) deg
Car left of (spot offset by -0.5 @ 0), \
    facing badAngle relative to roadDirection
```

This will yield cars parked 10-20° off from the direction of the curb. An example is shown in Fig. 4.

Declarative Specification of Hard and Soft Constraints. Scenic also allows the user to define additional requirements on generated scenes beyond the default requirements avoiding object overlap and so forth. These requirements can check arbitrary conditions

Figure 4: A scene of a badly parked car.

built from various geometric predicates. For example, the following scenario produces a car that is 20–40 m ahead and headed roughly towards us, while still facing the nominal road direction:

```
car2 = Car offset by (-10, 10) @ (20, 40), \
           with viewAngle 30 deg
require car2 can see ego
```

Here we have used the *X* `can see` *Y* predicate, which in this case is checking that the ego car is inside the 30° view cone of the second car. If we only need this constraint to hold part of the time, we can use a *soft requirement* specifying the minimum probability with which the constraint must hold:

```
require[0.5] car2 can see ego
```

Such hard and soft requirements make it possible to easily generate scenes with properties that are difficult to ensure when constructing explicit distributions in a purely imperative fashion.

Mutations. Finally, Scenic provides an easy-to-use *mutation* system that adds variety to a scenario without changing its code. This is useful, for example, if we have a scenario encoding a single concrete scene obtained from real-world data and want to quickly generate variations on it. For instance:

```
taxi = Car at 120 @ 300, facing 37 deg, ...
...
mutate taxi
```

This will add Gaussian noise to the `position` and `heading` of `taxi`, while still enforcing all built-in and custom requirements. The standard deviation of the noise can be scaled by writing, for example, `mutate taxi by 2` (which adds twice as much noise as above), and we will see later that it can be controlled separately for `position` and `heading`. The mutation system is quite simplistic, and obviously gives far less control than writing a detailed scenario specifying how every property of every object can vary, but it allows quick exploration of the neighborhood around a scenario of interest.

4 THE SCENIC LANGUAGE

Scenic is a simple imperative probabilistic programming language, with no conditional control flow constructs or general data structures. Its syntax is largely devoted to expressing geometric relationships between objects in a concise yet readable manner. Figure 5 and its associated tables give a formal grammar for Scenic, which we now describe in detail.

Scenic provides several primitive data types:

scenario := (`import` *file*)* (*statement*)*
boolean := `True` | `False` | *booleanOperator*
scalar := *number* | *distribution* | *scalarOperator*
distribution := *baseDist* | `resample(`*distribution*`)`
vector := *scalar* `@` *scalar* | *Point* | *vectorOperator*
heading := *scalar* | *OrientedPoint* | *headingOperator*
direction := *heading* | *vectorField*
value := *boolean* | *scalar* | *vector* | *direction*
| *region* | *instance* | *instance*`.`*property*
classDefn := `class` *class*[`(`*superclass*`)`]`:`
(*property*`:` *defaultValue*)*
instance := *class* *specifier*`,` …
specifier := `with` *property* *value* | *posSpec* | *headSpec*

Figure 5: Simplified Scenic grammar. *Point* and *OrientedPoint* are instances of the corresponding classes. See Tab. 5 for statements, Fig. 7 for operators, Tab. 1 for *baseDist*, and Tables 3 and 4 for *posSpec* and *headSpec* respectively.

Table 1: Distributions. All parameters *scalar* except *value*.

Syntax	Distribution
`(`*low*`,` *high*`)`	uniform on interval of $\mathbb{R}$
`Uniform(`*value*`,` …`)`	uniform over given values
`Discrete({`*value*`:` *wt*`,` …`})`	discrete with given weights
`Normal(`*mean*`,` *stdDev*`)`	normal with given μ, σ

Booleans expressing truth values of requirements to satisfy.
Scalars floating-point numbers, which can be sampled from various distributions (see Table 1).
Vectors representing positions and offsets in space, constructed from coordinates with the Smalltalk [12] syntax `X @ Y`.
Headings representing orientations in space. Conveniently, in 2D these are a single angle (in radians, anticlockwise from North). We use the convention that the heading of a local coordinate system is the heading of its y-axis, so, for example, `-2 @ 3` means 2 meters left and 3 ahead.
Vector Fields associating an orientation to each point in space. For example, the shortest paths to a destination or (in our case study) the nominal traffic direction.
Regions representing sets of points in space. These can have a vector field associated with them so that points in the region have preferred orientations (e.g. the surface of an object could have normal vectors, so that objects placed randomly on the surface face outward by default).

In addition, Scenic provides a lightweight notion of *objects*, mainly used to represent the physical objects present in a scene. Objects are simply immutable maps from *properties* to values; for example, each object has a `position` property whose value is a vector storing the object's position. Objects are instances of *classes*, which specify a set of properties their instances must have, together with corresponding default values (see Fig. 5). The classes form a single-inheritance hierarchy, where subclasses may provide new default values for properties defined in superclasses.

Default value expressions are evaluated each time an object is created. Thus in a class definition if we write `weight: (1, 5)` then each instance of this class will have a `weight` drawn *independently*

Table 2: Properties of `Point`, `OrientedPoint`, and `Object`.

Property	Default	Meaning
`position`	(0, 0)	position in global coordinates
`viewDistance`	50	distance for `can see` predicate
`positionStdDev`	1	mutation σ for `position`
`heading`	0	heading in global coordinates
`viewAngle`	360°	angle for `can see` predicate
`headingStdDev`	5°	mutation σ for `heading`
`width`	1	width of bounding box
`height`	1	height of bounding box
`allowCollisions`	false	collisions with objects allowed
`requireVisible`	true	must be visible from ego

from `(1, 5)`. Furthermore, default values may use the special syntax `self.`*property* to refer to one of the other properties of the object, which is then a *dependency* of this default value. In our case study, for example, the `width` and `height` of a `Car` are by default derived from its `model`.

Physical objects in a scene are instances of `Object`, which is the default superclass when none is specified. `Object` descends from the two other built-in classes: its superclass is `OrientedPoint`, which in turn subclasses `Point`. `Point` and `OrientedPoint` represent locations in space, without and with an orientation respectively, and so provide the fundamental properties `position` and `heading`. `Object` extends this by defining a bounding box with the properties `width` and `height`. Table 2 lists all the properties of the built-in classes and their default values.

To allow cleaner notation, `Point` and `OrientedPoint` are automatically interpreted as vectors or headings in contexts expecting these (as shown in Fig. 5). For example, we can write `taxi offset by 1 @ 2` and `30 deg relative to taxi` instead of `taxi.position offset by 1 @ 2` and `30 deg relative to taxi.heading`. Ambiguous cases (e.g. `taxi relative to limo`) are illegal and the more verbose syntax must be used instead.

The most interesting aspect of objects is that they are defined using the system of flexible *specifiers* illustrated above: as shown in Fig. 5, an object is created by writing the class name followed by a (possibly empty) comma-separated list of specifiers. Arbitrary properties (including user-defined properties with no meaning in Scenic) can be specified with the generic specifier `with` *property value*. There are many more specifiers for the built-in properties `position` and `heading`, shown in Tables 3 and 4 respectively. Note that some `position` specifiers can also specify `heading`, but *optionally*, meaning other specifiers will override them: for example, if `road` is a region with a preferred orientation as in our case study, `Object on road` will create an object at a position uniformly random in `road` and with the preferred orientation there, but to set the orientation ourselves we can for example write `Object on road, facing 20 deg` (whereas `Object on road, at 3 @ 4` would be illegal because `position` is specified twice).

In general, the semantics of object creation is as follows. Let P be the set of properties defined in the object's class and superclasses, together with any properties specified with the `with` *property value* specifier. The object will have exactly these properties, and the value of each $p \in P$ is determined as follows. If p is specified

Table 3: Specifiers for `position`. Those in the second group also optionally specify `heading`.

Specifier	Dependencies
`at` *vec*	—
`offset by` *vec*	—
`offset along` *direction* `by` *vec*	—
`( left | right ) of` *vec*	`heading`, `width`
`( ahead of | behind )` *vec*	`heading`, `height`
`beyond` *vec* `by` *vec* `[from` *vec*`]`	—
`visible [from (` *Point* `|` *OrientedPoint* `)]`	—
`( in | on )` *region*	—
`( left | right ) of` *OrientedPoint*	`width`
`( ahead of | behind )` *OrientedPoint*	`height`
`following` *vecField* `[from` *vec*`] for` *scalar*	—

Table 4: Specifiers for `heading`.

Specifier	Dependencies
`facing` *heading*	—
`facing` *vectorField*	`position`
`facing ( toward | away from )` *vector*	`position`
`apparently facing` *heading* `[from` *vector*`]`	`position`

non-optionally by multiple specifiers the scenario is ill-formed. If p is only specified optionally, and by multiple specifiers, this is ambiguous and we also declare the scenario ill-formed. Otherwise, the value of p will be determined by its unique non-optional specifier, unique optional specifier, or the most-derived default value, in that order: call this specifier/default s_p. Construct a directed graph with vertices P and edges to p from each of the dependencies of s_p (if a dependency is not in P, then a specifier or default value references a nonexistent property and the scenario is ill-formed). If this graph has a cycle, there are cyclic dependencies and the scenario is ill-formed (e.g. `Car left of 0 @ 0, facing roadDirection`: the `heading` must be known to evaluate `left of` *vector*, but `facing` *vectorField* needs `position` to determine `heading`). Otherwise, topologically sorting the graph yields an evaluation order for the specifiers and default values so that all dependencies are available when needed. The properties of the object are determined by evaluating the specifiers in this order.

As the semantics of the specifiers in Tables 3 and 4 are mostly evident from their syntax, we defer exact definitions to Appendix B. We briefly discuss some of the more complex specifiers, referring to the examples in Fig. 6:

- `behind` *vector* means the object is placed with the midpoint of its front edge at the given vector, and similarly for `ahead/left/right of` *vector*.
- `beyond` *A* `by` *O* `from` *B* means the position obtained by treating *O* as an offset in the local coordinate system at *A* oriented along the line of sight from *B*. In this and other specifiers, if the `from` *B* is omitted, the ego object is used by default. So for example `beyond taxi by 0 @ 3` means 3 m directly behind the taxi as viewed by the camera (see Fig. 6 for another example).

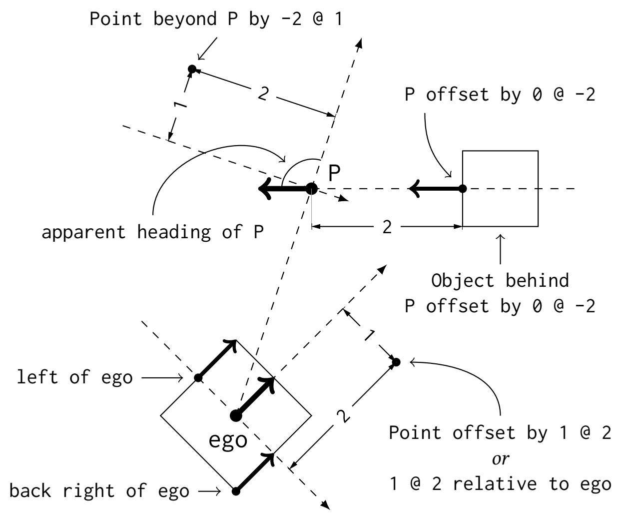

Figure 6: Various Scenic operators and specifiers applied to the ego object and an `OrientedPoint` `P`. Instances of `OrientedPoint` are shown as bold arrows.

- `in` *region*, yielding a uniformly random `position` in the region as seen above, optionally specifies `heading` if the region has a preferred orientation.
- The `heading` optionally specified by `left of` *OrientedPoint*, etc. is that of the `OrientedPoint` (thus in Fig. 6, we see that `P offset by 0 @ -2` yields an `OrientedPoint` facing the same way as `P`). Similarly, the `heading` optionally specified by `following` *vectorField* is that of the vector field at the specified `position`.
- `apparently facing` H `from` V means the object has heading H with respect to the line of sight from V (as above, the ego object if V is omitted). For example, `apparently facing 90 deg` would orient the object so that the camera views its left side head-on.

Next, we describe Scenic's operators, shown in Fig. 7. Again, many are self-explanatory and we defer exact definitions to Appendix B. Several are illustrated in Fig. 6. Various points to note:

- X `can see` Y uses a simple visibility model where a `Point` can see out to a certain radius, and an `OrientedPoint` restricts this to the circular sector along its `heading` with a certain angle (see Table 2 for the relevant properties). An `Object` is visible if part of its bounding box is.
- X `relative to` Y interprets X as an offset in a local coordinate system defined by Y. Thus `-3 @ 0 relative to` Y yields 3 m left of Y if Y is an `OrientedPoint`, and 3 m West of Y if Y is a vector. If defining a heading inside a specifier, either X or Y can be a vector field: it is interpreted as a heading by evaluating it at the `position` of the object being specified. So we can write for example `Car at 120 @ 70, facing 30 deg relative to roadDirection`.
- `visible` *region* yields the part of the region visible from the ego object, so we can write `Car on visible road`. The operator *region* `visible from` X does the same, but viewed from X.

scalarOperator := `max(`*scalar*`, …)` | `min(`*scalar*`, …)`
  | `-`*scalar* | `abs(`*scalar*`)` | *scalar* (`+` | `*`) *scalar*
  | `relative heading of` *heading* [`from` *heading*]
  | `apparent heading of` *OrientedPoint* [`from` *vector*]
  | `distance` [`from` *vector*] `to` *vector*
booleanOperator := `not` *boolean*
  | *boolean* (`and` | `or`) *boolean*
  | *scalar* (`==` | `!=` | `<` | `>` | `<=` | `>=`) *scalar*
  | (*Point* | *OrientedPoint*) `can see` (*vector* | *Object*)
  | *vector* `is in` *region*
headingOperator := *scalar* `deg`
  | *vectorField* `at` *vector*
  | *direction* `relative to` *direction*
vectorOperator := *vector* `relative to` *vector*
  | *vector* `offset by` *vector*
  | *vector* `offset along` *direction* `by` *vector*
regionOperator := `visible` *region*
  | *region* `visible from` (*Point* | *OrientedPoint*)
orientedPointOperator :=
  vector `relative to` *orientedPoint*
  | *orientedPoint* `offset by` *vector*
  | `follow` *vectorField* [`from` *vector*] `for` *scalar*
  | (`front` | `back` | `left` | `right`) `of` *Object*
  | (`front` | `back`) (`left` | `right`) `of` *Object*

Figure 7: Operators by result type.

Table 5: Statements.

Syntax	Meaning
identifier `=` *value*	variable assignment
`ego =` *Object*	ego object assignment
`param` *identifier* `=` *value*`, …`	parameter assignment
classDefn	class definition
instance	object definition
`require` *boolean*	hard requirement
`require[`*number*`]` *boolean*	soft requirement
`mutate` *instance*`, …` [`by` *number*]	enable mutation

- `front of` *Object*, `front left of` *Object*, etc. yield the corresponding points on the bounding box of the object, oriented along the object's `heading`.

Finally, we discuss Scenic's statements, listed in Table 5. The semantics of class and object definitions, and the role of the ego object have been discussed above. Variable assignment is as in a typical imperative probabilistic programming language: when the *value* is a distribution, the value assigned is a *sample* from the distribution. So for example

```
x = (0, 1)
y = x @ x
```

does not produce a uniform distribution over the unit box, but rather over its diagonal. For convenience in producing multiple samples from a (potentially complex) distribution, Scenic provides a `resample` function which returns an independent sample from the same distribution as the given value.

The statement `param` *identifier* `=` *value* assigns values to global parameters of the scenario. These parameters have no semantics in Scenic but provide a general-purpose way to encode arbitrary scene-wide information. For example, in our case study we used parameters `time` and `weather` to put distributions on the time of day and the weather conditions during the scene.

The `require` *boolean* statement requires that the given condition hold in all instantiations of the scenario (equivalently to *observe* statements in other probabilistic programming languages [16]). As mentioned above, Scenic automatically includes implicit requirements that `Objects` cannot intersect and must be visible from the ego object (these can be disabled on a per-object basis: see Table 2). The variant statement `require[`*p*`]` *boolean* adds a *soft* requirement that need only hold with some probability p (which must be a constant). We will discuss the precise semantics of soft requirements in the next section.

Lastly, Scenic provides an easy way to add variety to a scenario (possibly encoding a single concrete scene obtained from real-world data) with the `mutate` *instance*`, ... by` *number* statement. This causes the listed objects (or every `Object`, if no list is given) to have Gaussian noise added to their `position` and `heading` properties. The standard deviation of this noise is the `positionStdDev`/`headingStdDev` property (see Table 2) of the object multiplied by the number (if given) in the `mutate` statement. So for example `mutate taxi by 2` would add twice as much noise as `mutate taxi`, and we could keep the `heading` of the taxi fixed by adding `with headingStdDev 0` when defining it.

5 SEMANTICS AND SCENE GENERATION

Each time one runs a Scenic program, its output is a *scene* consisting of an assignment to all the properties of each `Object` defined in the scenario, plus any global parameters defined with `param`. Since Scenic allows sampling from distributions, the imperative part of a scenario actually induces a *distribution over scenes*, following the semantics described in the previous section. The declarative part of a scenario, consisting of its `require` statements, *modifies this distribution.* As mentioned above, hard requirements are equivalent to "observations" in other probabilistic programming languages (e.g. [16, 25]), *conditioning* the distribution on the requirement being satisfied.

The semantics of *soft* requirements is less clear-cut. We choose the natural definition that `require[`p`]` C is equivalent to a hard requirement `require` C that is only enforced with probability p. More precisely, if D is the distribution induced by a Scenic scenario S, then the distribution induced by the scenario S' defined as S; `require[`p`]` C is $p\cdot(D|C)+(1-p)\cdot D$. This ensures in particular that scenes generated from S' will satisfy C with probability at least p, as desired.

This reduction of soft to hard requirements makes the problem of sampling scenes from the distribution defined by a Scenic scenario essentially a special case of the sampling problem for imperative probabilistic programming languages with observations. While as we discuss below we could apply general techniques for such problems, there are several features that make the special case of Scenic interesting in its own right: soft requirements, lack of conditional control flow (except for that implicit in soft requirements), and complex hard requirements (even simple geometric relations can involve trigonometric functions, for example). Therefore we propose *scene improvisation*, the task of generating scenes from a Scenic scenario, be studied as a new theoretical problem.

We note that scene improvisation is related to *control improvisation*, an abstract framework capturing various problems requiring synthesis under hard, soft, and randomness constraints (and which inspired the term scene improvisation) [10, 11]. Scene improvisation can be viewed as an extension with a more detailed randomness constraint given by the imperative part of the scenario.

Algorithms to solve control improvisation problems based on counting and sampling solutions to constraints with SAT and SMT solvers have been proposed, and it would be interesting to see whether these could be adapted to scene improvisation. However, sampling from the highly nonlinear geometric constraints of Scenic has not yet been studied. A more promising approach would be to adapt the Markov Chain Monte Carlo (MCMC) methods that have been successfully used in probabilistic programming (see, e.g., [25, 30]).

In our implementation we use a rejection sampling approach where scenes are generated from the imperative part of the scenario until all requirements are satisfied. While this samples from exactly the desired distribution, it has the drawback that a huge number of samples may be required to yield a single valid scene (in the worst case, when the requirements have probability zero of being satisfied, the algorithm will not even terminate). However, we found in our experiments that all reasonable scenarios we tried required only several hundred iterations at most, yielding a sample within a few seconds. Although not needed for rejection sampling, our implementation does maintain symbolic representations of all distributions so that more intelligent heuristics and sampling methods could easily be added in the future.

6 EXPERIMENTS

We performed experiments illustrating three different uses of Scenic: assessing the accuracy of an ML system under particular conditions, retraining the system to improve accuracy in hard cases, and exploring the system's behavior around a known failure case. We begin by describing the general experimental setup.

6.1 Experimental Setup

We generated scenes in the virtual world of the video game Grand Theft Auto V (GTAV)[1]. We wrote a Scenic library defining `Regions` representing the roads and curbs in (part of) this world, as well as a type of object `Car` providing two additional properties:

- `model`, representing the type of car. Over 300 models are supported by GTAV; we used a set of 13 diverse models, with the default distribution over these being uniform.
- `color`, representing the car color. The default distribution was based on car color statistics for North America [6].

In addition, we implemented two global scene parameters:

- `time`, representing the time of day. The default distribution was uniform over all 24 hours.
- `weather`, representing the weather as one of 14 discrete types supported by GTAV (e.g. "clear" or "snow"). The default distribution gave all types positive probability, biased towards less extreme weather.

[1] The publisher of GTA allows non-commercial use of footage of gameplay [32].

Unfortunately, GTAV does not provide an explicit representation of its map. We obtained an approximate map by processing a bird's-eye schematic view of the game world[3]. To identify points on a road, we converted the image to black and white, effectively turning roads white and everything else black. We then applied the Sobel filter to detect edges, identifying points on the curb. Finally, we computed the nominal traffic direction by finding for each curb point X the nearest curb point Y on the other side of the road, and assuming traffic flows perpendicular to the segment XY, in opposite directions on either side of its midpoint (this was more robust than using the directions of the edges in the image). Since the resulting road information was imperfect, some generated scenes placed cars in undesired places such as sidewalks or medians. We had to manually filter the generated images to remove these. With a more open simulator this would not be necessary.

Our implementation's interface to GTAV is based on DeepGTAV[2]. To render a scene, we use a series of API calls to create the cars and set the time of day and weather.

Our experiments were done using squeezeDet [42], a convolutional neural network real-time object detector for autonomous driving. We used a batch size of 20 and trained all models for 10,000 iterations. Images captured from GTAV with resolution 1920×1200 were resized to 1248×384, that is the resolution used by squeezeDet. All models were trained and evaluated on NVIDIA TITAN Xp GPUs.

We now define the metrics used to measure the performance of our models. Let $\hat{\mathbf{y}} = f(\mathbf{x})$ be the prediction of the model f for input $\mathbf{x}$. In usual object detection tasks, $\hat{\mathbf{y}}$ encodes bounding boxes, scores, and categories predicted by f for the image $\mathbf{x}$. Let B_{gt} be a ground truth box (i.e. a bounding box from the label of a training sample that indicates the position of a particular object) and $B_{\hat{\mathbf{y}}}$ be a prediction box (i.e. the box returned by the model). The *Intersection over Union* (IoU) is defined as $IoU(B_{gt}, B_{\hat{\mathbf{y}}}) = \mathcal{A}_{B_{gt}} \cap \mathcal{A}_{B_{\hat{\mathbf{y}}}} / \mathcal{A}_{B_{gt}} \cup \mathcal{A}_{B_{\hat{\mathbf{y}}}}$, where $\mathcal{A}_B$ is the area of a box B. IoU is a common evaluation metric used to measure how well predicted bounding boxes match ground truth boxes. We adopt the common practice of considering $B_{\hat{\mathbf{y}}}$ a *detection* for B_{gt} if $IoU(B_{gt}, B_{\hat{\mathbf{y}}}) > 0.5$.

Precision and *recall* are metrics used to measure the accuracy of a prediction on a particular image. Intuitively, precision is the fraction of predicted boxes that are correct, while recall is the fraction of objects actually detected. Formally, precision is defined as $tp/(tp + fp)$ and recall as $tp/(tp + fn)$, where *true positives* tp is the number of correct detections, *false positives* fp is the number of predicted boxes that do not match any ground truth box, and *false negatives* is the number of ground truth boxes that are not detected. We use *average precision* and *recall* to evaluate the performance of a model on a collection of images constituting a test set.

6.2 Generating Specialized Test Sets

When testing a model, one might be interested in a particular application domain. For instance, in the case of autonomous cars, a manufacturer might be more interested in certain road conditions than others (e.g. desert or forest roads) depending on where its cars will be mainly used. SCENIC provides a systematic way to describe different scenarios of interest and construct corresponding test sets.

To demonstrate this, we first wrote very general scenarios describing scenes of 1–4 cars (not counting the camera), specifying only that the cars face within 10° of the road direction. We generated 1,000 images from each scenario, yielding a training set of 4,000 images, and used these to train a model M_{generic} as described in Sec. 6.1. We also generated an additional 50 images from each scenario to obtain a generic test set T_{generic} of 200 images.

Next, we specialized the general scenarios in opposite directions: scenarios for good/bad road conditions fixing the time to noon/midnight and the weather to sunny/rainy respectively. We used these to generate specialized test sets T_{good} and T_{bad}.

Evaluating M_{generic} on the three test sets T_{generic}, T_{good}, and T_{bad}, we obtained average precisions of 86.13%, 88.48%, and 78.93%, respectively, and average recalls of 94.46%, 96.08%, and 95.00%. These results show that, as might be expected, the model tends to perform better on bright days than on rainy nights. This illustrates how specialized test sets can highlight the weaknesses and strengths of a particular model.

6.3 Retraining to Improve Performance on Hard Cases

In the synthetic data setting, we are limited not by data availability but by the cost of training. The natural question is then how to generate a synthetic data set that as effective as possible given a fixed size. In this section we show that *over-representing* a type of input that may occur rarely but is difficult for the model can improve performance on the hard case without compromising performance in the typical case. SCENIC makes this possible by allowing the user to write a scenario capturing the hard case specifically.

For our car detection task, an obvious hard case is when one car substantially occludes another. We wrote a scenario generating such scenes by placing one car behind the other as viewed from the camera, offset left or right so that it is at least partially visible. Generating 1,000 images from this scenario yielded a training set X_{overlap}. We also generated 1,000 images from the generic two-car scenario above, obtaining a training set X_{twocar}.

Note that X_{twocar} did contain images of overlapping cars, since the generic two-car scenario does not constrain the cars' locations. However, the average overlap was much lower than that of X_{overlap}, as seen in Fig. 8 (note the log scale): thus the overlapping car images are highly "untypical" of generic two-car images. We would like to ensure the network performs well on these difficult images by emphasizing them in the training set. Therefore we constructed various *mixtures* of the two training sets, fixing the total number of images but using different ratios of images from X_{twocar} and X_{overlap}. We trained the network on each of these mixtures and evaluated their performance on 400-image test sets T_{twocar} and T_{overlap} from the two-car and overlapping scenarios respectively.

To reduce the effect of randomness in training, we used the maximum precision and recall obtained when training for 4,000 through 5,000 steps in increments of 250 steps (in training, it is common to save different model weights and keep the best ones with respect to desired metrics [2], in our case average precision and recall). Additionally, we repeated each training 8 times, using a random mixture each time: for example, for the 90/10 mixture of X_{twocar} and X_{overlap}, each training used an independent random choice of which 90% of X_{twocar} to use and which 10% of X_{overlap}.

As the results in Tab. 6 show, the model trained purely on generic two-car images has high precision and recall on T_{twocar} but has drastically worse recall on T_{overlap}; essentially, the network has

[3]https://www.gtafivemap.com/
[2]https://github.com/aitorzip/DeepGTAV

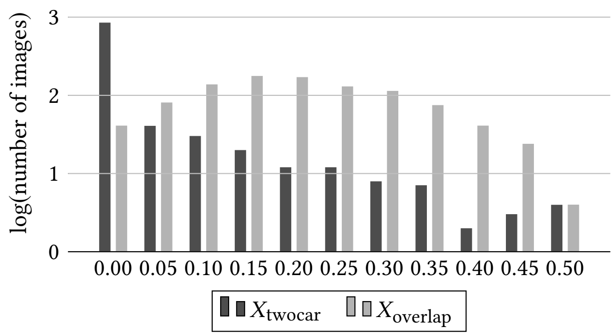

Figure 8: Intersection Over Union (IOU) distribution for two-car and overlapping training sets (log scale).

Table 6: Performance of models trained on mixtures of T_{twocar} and T_{overlap} and tested on both, averaged over 8 training runs. 90/10 indicates a 9:1 mixture of $T_{\text{twocar}}/T_{\text{overlap}}$.

	T_{twocar}		T_{overlap}	
Mixture	Precision	Recall	Precision	Recall
100/0	96.5 ± 1.0	95.7 ± 0.5	94.6 ± 1.1	**82.1** ± 1.4
90/10	95.3 ± 2.1	96.2 ± 0.5	93.9 ± 2.5	86.9 ± 1.7
80/20	96.5 ± 0.7	96.0 ± 0.6	96.2 ± 0.5	**89.7** ± 1.4
70/30	96.5 ± 0.9	96.5 ± 0.6	96.0 ± 1.6	90.1 ± 1.8

difficulty detecting the partially-occluded car. However, devoting 20% of the training set to overlapping cars gives a large 8% improvement to recall on T_{overlap} while leaving performance on T_{twocar} essentially the same. This demonstrates that we can improve the performance of a network on difficult corner cases by using Scenic to increase the representation of such cases in the training set.

6.4 Generalizing from a Known Failure Case

In our final experiment, we show how Scenic can be used to generalize a single input on which a model behaves badly. Scenic makes it easy to explore the neighborhood of a given scene in a variety of different directions, giving insight into which features of the scene are responsible for the undesired behavior. The original misclassification can then be generalized to a broader scenario describing a class of inputs on which the model misbehaves, and this scenario can be used for retraining.

We selected one scene from our first experiment, consisting of a single car viewed from behind at a slight angle, on which M_{generic} had only 33.3% precision (and 100% recall). We wrote several scenarios which left most of the features of the scene fixed but allowed others to vary. Specifically, scenario (1) varied the model and color of the car, (2) left the position and orientation of the car relative to the camera fixed but varied the absolute position, effectively changing the background of the scene, and (3) used the mutation feature of Scenic to add a small amount of noise to the car's position, heading, and color. For each scenario we generated 150 images and evaluated the performance of M_{generic} on them. As seen in Tab. 7, changing the model and color improved performance the most, suggesting they were most relevant to the misclassification,

Table 7: Performance of M_{generic} on different scenarios representing variations of a single misclassified image.

Scenario	Precision	Recall
(1) varying model and color	**88.4**	100
(2) varying background	63.7	100
(3) varying local position and orientation	74.2	100
(4) varying position but staying close	68.8	99.3
(5) any position, same apparent angle	74.3	98.6
(6) any position and angle	81.2	100
(7) varying background, model, and color	74.4	100
(8) staying close, same apparent angle	64.1	100
(9) staying close, varying model	71.7	100

while local position and orientation were less important and global position (i.e. the background) was least important.

To investigate these possibilities further, we wrote a second round of variant scenarios, also shown in Tab. 7. The results confirmed the importance of model and color (compare (2) to (7)), as well as angle (compare (5) to (6)), but also suggested that being close to the camera could be the relevant aspect of the car's local position. We confirmed this with a final round of scenarios (compare (5) and (8)), which also showed that the effect of car model is small among scenes where the car is close to the camera (compare (4) and (9)).

Having established that car model, closeness to the camera, and view angle all contribute to poor performance of the network, we proceeded to capture these features in broader scenarios. To avoid overfitting, and since our experiments indicated car model was not critical to misclassification when the car is close to the camera, we decided not to fix the car model. Instead, we specialized the generic one-car scenario from our first experiment to produce only cars close to the camera. We also created a second scenario specializing this further by requiring that the car be viewed at a shallow angle.

Finally, we used these scenarios to retrain M_{generic}, hoping to improve performance on its original test set T_{generic} (to better distinguish small differences in performance, we increased the test set size to 400 images). To keep the size of the training set fixed as in the previous experiment, we replaced 400 one-car images in T_{generic} (10% of the whole training set) with images generated from our scenarios. We also used images produced with classical image augmentation techniques implemented in `imgaug`[1], a Python library for image augmentation. Specifically, we modified the original misclassified image by randomly cropping 10%–20% on each side, flipping horizontally with probability 50%, and applying Gaussian blur with $\sigma \in [0.0, 3.0]$.

The results of retraining M_{generic} on the resulting data sets are shown in Tab. 8. Interestingly, classical augmentation actually *hurt* performance, indicating that such techniques may not make sense in the plentiful-data regime where training set size is fixed. On the other hand, replacing part of the data set with specialized images of cars close to the camera significantly improved performance (the improvement for the "shallow angle" scenario was less, perhaps due to overfitting to the restricted angle range). This demonstrates how Scenic can be used to improve performance by generalizing individual misclassifications into scenarios that capture the essence

[1] `imgaug`: https://github.com/aleju/imgaug

Table 8: Performance of M_{generic} after retraining, replacing 10% of T_{generic} with different data.

Replacement Data	Precision	Recall
Original (no replacement)	85.9	94.8
Classical augmentation	82.6	94.4
Close car	89.8	94.0
Close car at shallow angle	87.6	94.8

of the problem but are broad enough to prevent overfitting during retraining.

7 CONCLUSION

In this paper, we introduced Scenic, a probabilistic programming language for specifying distributions over configurations of physical objects. We showed how Scenic can be used to generate synthetic data sets useful for deep learning tasks. Specifically, we used Scenic to generate specialized test sets, improve the effectiveness of training sets by emphasizing difficult cases, and generalize from individual failure cases to broader scenarios suitable for retraining.

In future work we intend to interface Scenic with other simulators such as CARLA [4], an open-source simulator for autonomous driving. We also plan to extend Scenic in several directions: allowing user-defined specifiers, describing 3D scenes, and encoding dynamics to enable generation of videos instead of static scenes.

ACKNOWLEDGMENTS

The authors would like to thank Ankush Desai and Jonathan Ragan-Kelley for helpful discussions. This work is supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1106400, NSF grants CNS-1646208 and CNS-1739816, DARPA under agreement number FA8750-16-C0043, the DARPA Assured Autonomy program, and TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. 2016. Concrete Problems in AI Safety. *CoRR* abs/1606.06565 (2016). arXiv:1606.06565 http://arxiv.org/abs/1606.06565

[2] Sylvain Arlot and Alain Celisse. 2010. A survey of cross-validation procedures for model selection. *Statist. Surv.* 4 (2010), 40–79. https://doi.org/10.1214/09-SS054

[3] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. 2005. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[4] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Conference on Robot Learning, CoRL.* 1–16.

[5] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. 2017. Compositional Falsification of Cyber-Physical Systems with Machine Learning Components. In *NASA Formal Methods, NFM.* 357–372. https://doi.org/10.1007/978-3-319-57288-8_26

[6] DuPont. 2012. Global Automotive Color Popularity Report. (2012). https://web.archive.org/web/20130818022236/http://www2.dupont.com/Media_Center/en_US/color_popularity/Images_2012/DuPont2012ColorPopularity.pdf

[7] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. 2013. CONCURRIT: a domain specific language for reproducing concurrency bugs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 153–164.

[8] Artur Filipowicz, Jeremiah Liu, and Alain Kornhauser. 2017. *Learning to recognize distance to stop signs using the virtual world of Grand Theft Auto 5.* Technical Report. Princeton University.

[9] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. 2012. Example-based Synthesis of 3D Object Arrangements. In *ACM SIGGRAPH 2012 (SIGGRAPH Asia '12).*

[10] Daniel J. Fremont, Alexandre Donzé, and Sanjit A. Seshia. 2017. Control Improvisation. arXiv preprint. (2017). arXiv:1704.06319

[11] Daniel J. Fremont, Alexandre Donzé, Sanjit A. Seshia, and David Wessel. 2015. Control Improvisation. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS) (LIPIcs)*, Vol. 45. 463–474.

[12] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts.

[13] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems.* 2672–2680.

[14] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and Harnessing Adversarial Examples. *CoRR* abs/1412.6572 (2014). arXiv:1412.6572 http://arxiv.org/abs/1412.6572

[15] Noah Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A universal language for generative models. In *Uncertainty in Artificial Intelligence 24 (UAI).* 220–229.

[16] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *FOSE 2014.* ACM, 167–181.

[17] Ankush Gupta, Andrea Vedaldi, and Andrew Zisserman. 2016. Synthetic Data for Text Localisation in Natural Images. In *Computer Vision and Pattern Recognition, CVPR.* 2315–2324. https://doi.org/10.1109/CVPR.2016.254

[18] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Computer Aided Verification, CAV.* 3–29. https://doi.org/10.1007/978-3-319-63387-9_1

[19] Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Synthetic Data and Artificial Neural Networks for Natural Scene Text Recognition. *CoRR* abs/1406.2227 (2014). arXiv:1406.2227 http://arxiv.org/abs/1406.2227

[20] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. 2017. Driving in the Matrix: Can virtual worlds replace human-generated annotations for real world tasks?. In *International Conference on Robotics and Automation, ICRA.* 746–753. https://doi.org/10.1109/ICRA.2017.7989092

[21] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I.* 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

[22] Xiaodan Liang, Zhiting Hu, Hao Zhang, Chuang Gan, and Eric P Xing. 2017. Recurrent Topic-Transition GAN for Visual Paragraph Generation. *arXiv preprint arXiv:1703.07022* (2017).

[23] Joerg Liebelt and Cordelia Schmid. 2010. Multi-view object class detection with a 3D geometric model. In *Computer Vision and Pattern Recognition, CVPR.* 1688–1695. https://doi.org/10.1109/CVPR.2010.5539836

[24] Marco Marchesi. 2017. Megapixel Size Image Creation using Generative Adversarial Networks. *arXiv preprint arXiv:1706.00082* (2017).

[25] Brian Milch, Bhaskara Marthi, and Stuart Russell. 2004. BLOG: Relational modeling with unknown objects. In *ICML 2004 workshop on statistical relational learning and its connections to other fields.* 67–73.

[26] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks. In *Computer Vision and Pattern Recognition, CVPR.* 2574–2582. https://doi.org/10.1109/CVPR.2016.282

[27] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural modeling of buildings. In *ACM Transactions On Graphics*, Vol. 25. ACM, 614–623.

[28] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. 2006. Constraint-Based Random Stimuli Generation for Hardware Verification. In *Proc. of AAAI.* 1720–1727.

[29] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Computer Vision and Pattern Recognition, CVPR.* 427–436. https://doi.org/10.1109/CVPR.2015.7298640

[30] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs.. In *AAAI.* 2476–2482.

[31] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Symposium on Operating Systems Principles, SOSP.* 1–18. https://doi.org/10.1145/3132747.3132785

[32] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. 2016. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision.* 102–118.

[33] Germán Ros, Laura Sellart, Joanna Materzynska, David Vázquez, and Antonio M. López. 2016. The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes. In *Computer Vision and Pattern Recognition, CVPR.* 3234–3243. https://doi.org/10.1109/CVPR.2016.352

[34] Stuart Russell, Tom Dietterich, Eric Horvitz, Bart Selman, Francesca Rossi, Demis Hassabis, Shane Legg, Mustafa Suleyman, Dileep George, and Scott Phoenix. 2015. Letter to the Editor: Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter. *AI Magazine* 36, 4 (2015).

[35] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. 2016. Towards Verified Artificial Intelligence. *CoRR* abs/1606.08514 (2016). http://arxiv.org/abs/1606.08514
[36] Michael Stark, Michael Goesele, and Bernt Schiele. 2010. Back to the Future: Learning Shape Models from 3D CAD Data. In *British Machine Vision Conference, BMVC*. 1–11. https://doi.org/10.5244/C.24.106
[37] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley.
[38] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *CoRR* abs/1312.6199 (2013). arXiv:1312.6199 http://arxiv.org/abs/1312.6199
[39] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 2017. Domain randomization for transferring deep neural networks from simulation to the real world. In *International Conference on Intelligent Robots and Systems, IROS*. 23–30. https://doi.org/10.1109/IROS.2017.8202133
[40] David Vazquez, Antonio M Lopez, Javier Marin, Daniel Ponsa, and David Geronimo. 2014. Virtual and real world adaptation for pedestrian detection. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2014), 797–809.
[41] Sebastien C Wong, Adam Gatt, Victor Stamatescu, and Mark D McDonnell. 2016. Understanding data augmentation for classification: when to warp?. In *Digital Image Computing: Techniques and Applications (DICTA), 2016 International Conference on*. IEEE, 1–6.
[42] Bichen Wu, Forrest N. Iandola, Peter H. Jin, and Kurt Keutzer. 2017. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. In *Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops*. 446–454. https://doi.org/10.1109/CVPRW.2017.60
[43] Yan Xu, Ran Jia, Lili Mou, Ge Li, Yunchuan Chen, Yangyang Lu, and Zhi Jin. 2016. Improved relation classification by deep recurrent neural networks with data augmentation. *arXiv preprint arXiv:1601.03651* (2016).

A GALLERY OF SCENARIOS

This section presents Scenic code for a variety of scenarios, along with images rendered from them. The scenarios range from simple examples used above to illustrate different aspects of the language, to those representing interesting road configurations like platoons and lanes of traffic.

CONTENTS

A.1 The `carLib` Module

All the scenarios below begin with a line (not shown here) importing the `carLib` module, which as explained above contains all definitions specific to our autonomous car case study. These include the definitions of the regions `road` and `curb`, as well as the vector field `roadDirection` giving the prevailing traffic direction at each point on the road. Most importantly, it also defines `Car` as a type of object:

```
class Car:
    position: Point on road
    heading: (roadDirection at self.position) \
        + self.roadDeviation
    roadDeviation: 0
    width: self.model.width
    height: self.model.height
    viewAngle: 80 deg
    visibleDistance: 30
    model: CarModel.defaultModel()
    color: CarColor.defaultColor()
```

Most of the properties are inherited from `Object` or are self-explanatory. The property `roadDeviation`, representing the heading of the car with respect to the local direction of the road, is purely a syntactic convenience; the following two lines are equivalent:

```
Car facing 10 deg relative to roadDirection
Car with roadDeviation 10 deg
```

The `carLib` library also defines a few convenience subclasses of `Car` with different default properties. For example, `EgoCar` overrides `model` with the fixed car model we used for the ego car in our interface to GTA V.

A.2 The Simplest Possible Scenario

This scenario, creating a single car with no specified properties, was used as an example in Sec. 3.

```
ego = Car
Car
```

Figure 9: Scenes generated from a Scenic scenario representing a single car (with reasonable default properties).

A.3 A Single Car

This scenario is slightly more general than the previous, allowing the car (and the ego car) to deviate from the road direction by up to 10°. It also specifies that the car must be visible, which is in fact redundant since this constraint is built into Scenic, but helps guide the sampling procedure. This scenario was also used as an example in Sec. 3.

```
wiggle = (-10 deg, 10 deg)
ego = EgoCar with roadDeviation wiggle
Car visible, with roadDeviation resample(wiggle)
```

Figure 10: Scenes generated from a Scenic scenario representing a single car facing roughly the road direction.

A.4 A Badly-Parked Car

This scenario, creating a single car parked near the curb but not quite parallel to it, was used as an example in Sec. 3.

```
ego = Car
spot = OrientedPoint on visible curb
badAngle = Uniform(1.0, -1.0) * (10, 20) deg
Car left of (spot offset by -0.5 @ 0), \
    facing badAngle relative to roadDirection
```

Figure 11: Scenes generated from a Scenic scenario representing a badly-parked car.

A.5 An Oncoming Car

This scenario, creating a car 20–40 m ahead and roughly facing towards the camera, was used as an example in Sec. 3. Note that since we do not specify the orientation of the car when creating it, the default `heading` is used and so it will face the road direction. The `require` statement then requires that this orientation is also within 15° of facing the camera (as the view cone is 30° wide).

```
ego = Car
car2 = Car offset by (-10, 10) @ (20, 40), \
    with viewAngle 30 deg
require car2 can see ego
```

Figure 12: Scenes generated from a Scenic scenario representing a car facing roughly towards the camera.

A.6 Adding Noise to a Scene

This scenario, using Scenic's mutation feature to automatically add noise to an otherwise completely-specified scenario, was used in the experiment in Sec. 6.4 (it is Scenario (3) in Table 7). The original scene, which is exactly reproduced by this scenario if the `mutate` statement is removed, is shown in Fig. 14.

```
param time = 12 * 60    # noon
param weather = 'EXTRASUNNY'

ego = EgoCar at -628.7878 @ -540.6067, \
    facing -359.1691 deg

Car at -625.4444 @ -530.7654, \
    facing 8.2872 deg, \
    with model CarModel.models['DOMINATOR'], \
    with color CarColor.byteToReal([187, 162, 157])

mutate
```

Figure 14: The original misclassified image in Sec. 6.4.

Figure 13: Scenes generated from a Scenic scenario adding noise to the scene in Fig. 14.

A.7 Two Cars

This is the generic two-car scenario used in the experiments in Secs. 6.2 and 6.3.

```
wiggle = (-10 deg, 10 deg)
ego = EgoCar with roadDeviation wiggle
Car visible, with roadDeviation resample(wiggle)
Car visible, with roadDeviation resample(wiggle)
```

Figure 15: Scenes generated from a Scenic scenario representing two cars, facing close to the direction of the road.

A.8 Two Overlapping Cars

This is the scenario used to produce images of two partially-overlapping cars for the experiment in Sec. 6.3.

```
wiggle = (-10 deg, 10 deg)
ego = EgoCar with roadDeviation wiggle

c = Car visible, with roadDeviation resample(wiggle)

leftRight = Uniform(1.0, -1.0) * (1.25, 2.75)
Car beyond c by leftRight @ (4, 10), \
    with roadDeviation resample(wiggle)
```

Figure 16: Scenes generated from a Scenic scenario representing two cars, one partially occluding the other.

A.9 Four Cars, in Poor Driving Conditions

This is the scenario used to produce images of four cars in poor driving conditions for the experiment in Sec. 6.2. Without the first two lines, it is the generic four-car scenario used in that experiment.

```
param weather = 'RAIN'
param time = 0 * 60   # midnight

wiggle = (-10 deg, 10 deg)
ego = EgoCar with roadDeviation wiggle
Car visible, with roadDeviation resample(wiggle)
Car visible, with roadDeviation resample(wiggle)
Car visible, with roadDeviation resample(wiggle)
Car visible, with roadDeviation resample(wiggle)
```

Figure 17: Scenes generated from a Scenic scenario representing four cars in poor driving conditions.

A.10 A Platoon, in Daytime

This scenario illustrates how Scenic can construct structured object configurations, in this case a platoon of cars. It uses a helper function provided by `carLib` for creating platoons starting from a given car, which is shown in Fig. 18. If no argument `model` is provided to `createPlatoonAt`, as in this case, all cars in the platoon have the same `model` as the starting car; otherwise, the given model distribution is sampled independently for each car. The syntax for functions and loops supported by our Scenic implementation is inherited from Python.

```
param time = (8, 20) * 60        # 8 am to 8 pm

ego = Car with visibleDistance 60
c2 = Car visible
platoon = createPlatoonAt(c2, 5, dist=(2, 8))
```

```
def createPlatoonAt(car, numCars, model=None, dist=(2, 8), shift=(-0.5, 0.5), wiggle=0):
  lastCar = car
  for i in range(numCars-1):
    center = follow roadDirection from (front of lastCar) for resample(dist)
    pos = OrientedPoint at (center offset by shift @ 0), facing resample(wiggle) relative to roadDirection
    lastCar = Car ahead of pos, with model (car.model if model is None else resample(model))
```

Figure 18: Helper function for creating a platoon starting from a given car.

Figure 19: Scenes generated from a Scenic scenario representing a platoon of cars during daytime.

A.11 Bumper-to-Bumper Traffic

This scenario creates an even more complex type of object structure, namely three lanes of traffic. It uses the helper function `createPlatoonAt` discussed above, plus another function for placing a car ahead of a given car with a specified gap in between, shown in Fig. 20.

```
depth = 4
laneGap = 3.5
carGap = (1, 3)
laneShift = (-2, 2)
wiggle = (-5 deg, 5 deg)

def createLaneAt(car):
  createPlatoonAt(car, depth, dist=carGap, \
    wiggle=wiggle, model=modelDist)

ego = Car with visibleDistance 60
modelDist = CarModel.defaultModel()
leftCar = carAheadOfCar(ego, laneShift + carGap, \
  offsetX=-laneGap, wiggle=wiggle)
createLaneAt(leftCar)

midCar = carAheadOfCar(ego, resample(carGap), \
  wiggle=wiggle)
createLaneAt(midCar)

rightCar = carAheadOfCar(ego, \
  resample(laneShift) + resample(carGap),
  offsetX=laneGap, wiggle=wiggle)
createLaneAt(rightCar)
```

```
def carAheadOfCar(car, gap, offsetX=0, wiggle=0):
  pos = OrientedPoint at (front of car) offset by (offsetX @ gap), \
    facing resample(wiggle) relative to roadDirection
  return Car ahead of pos
```

Figure 20: Helper function for placing a car ahead of a car, with a specified gap in between.

Figure 21: Scenes generated from a Scenic scenario representing bumper-to-bumper traffic.

B DETAILED SEMANTICS OF SPECIFIERS AND OPERATORS

This section provides precise semantics for Scenic's specifiers and operators, which were informally defined above. In the following figures, S indicates a *scalar*, V a *vector*, H a *heading*, F a *vectorField*, R a *region*, P a `Point`, and OP an `OrientedPoint`. Figure 22 defines notation used in the rest of the semantics. In *forwardEuler*, N is an implementation-defined parameter specifying how many steps should be used for the forward Euler approximation when following a vector field (we used a fixed $N = 4$).

Figure 23 gives the semantics of the `position` specifiers. The figure writes the semantics as a vector value; the semantics of the specifier itself is to assign the `position` property of the object being specified to that value. Several of the specifiers refer to properties of `self`: as explained in Sec. 4, this refers to the object being constructed, and the semantics of object construction are such that specifiers depending on other properties are only evaluated after those properties have been specified (or an error is raised, if there are cyclic dependencies).

Figure 24 gives the semantics of the `position` specifiers that can also optionally specify `heading`. The figure writes the semantics as an `OrientedPoint` value; if this is OP, the semantics of the specifier is to assign the `position` property of the object being constructed to `OP.position`, and the `heading` property of the object to `OP.heading` if `heading` is not otherwise specified (see Sec. 4 for a discussion of optional specifiers). If the `heading` of OP is given as $\perp$, then `heading` is not optionally specified.

Figure 25 gives the semantics of the `heading` specifiers. As for the `position` specifiers above, the figure indicates the heading value assigned by each specifier.

Finally, Figures 26–31 give the semantics for Scenic's operators, broken down by the type of value they return.

$$
\begin{aligned}
\langle x, y\rangle &= \text{point with the given XY coordinates}\\
\mathit{rotate}(\langle x, y\rangle, \theta) &= \langle x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta\rangle\\
\arctan(v_1 - v_2) &= \text{arctangent of } v_1 - v_2 \text{ in the correct quadrant, i.e. the heading from } v_2 \text{ to } v_1\\
\mathit{Disc}(c, r) &= \text{set of points in the disc centered at } c \text{ and with radius } r\\
\mathit{Sector}(c, r, h, a) &= \text{set of points in the sector of } \mathit{Disc}(c, r) \text{ centered along } h \text{ and with angle } a\\
\mathit{boundingBox}(O) &= \text{set of points in the bounding box of object } O\\
\mathit{visibleSet}(X) &= \begin{cases} \mathit{Sector}([\![\texttt{X.position}]\!], [\![\texttt{X.viewDistance}]\!], [\![\texttt{X.heading}]\!], [\![\texttt{X.viewAngle}]\!]) & X \in \texttt{OrientedPoint}\\ \mathit{Disc}([\![\texttt{X.position}]\!], [\![\texttt{X.viewDistance}]\!]) & X \in \texttt{Point}\end{cases}\\
\mathit{uniformPointIn}(X) &= \text{uniformly random point in the set of points } X\\
\mathit{orientation}(R) &= \text{preferred orientation of the region } R \text{ if any; otherwise } \perp\\
\mathit{forwardEuler}(x, d, F) &= \text{result of iterating the map } y \mapsto y + \mathit{rotate}(\langle 0, d/N\rangle, [\![F]\!](y)) \text{ a total of } N \text{ times on } x\\
\texttt{OrientedPoint}(V, H) &= \text{an } \texttt{OrientedPoint} \text{ with the given } \texttt{position} \text{ and } \texttt{heading}
\end{aligned}
$$

Figure 22: Notation used to define the semantics.

$$
\begin{aligned}
[\![\texttt{at } V]\!] &= [\![V]\!]\\
[\![\texttt{offset by } V]\!] &= [\![V \texttt{ relative to ego.position}]\!]\\
[\![\texttt{offset along } H \texttt{ by } V]\!] &= [\![\texttt{ego.position offset along } H \texttt{ by } V]\!]\\
[\![\texttt{left of } V]\!] &= [\![V]\!] + \mathit{rotate}(\langle -[\![\texttt{self.width}]\!]/2, 0\rangle, [\![\texttt{self.heading}]\!])\\
[\![\texttt{right of } V]\!] &= [\![V]\!] + \mathit{rotate}(\langle [\![\texttt{self.width}]\!]/2, 0\rangle, [\![\texttt{self.heading}]\!])\\
[\![\texttt{ahead of } V]\!] &= [\![V]\!] + \mathit{rotate}(\langle 0, [\![\texttt{self.height}]\!]/2\rangle, [\![\texttt{self.heading}]\!])\\
[\![\texttt{behind } V]\!] &= [\![V]\!] + \mathit{rotate}(\langle 0, -[\![\texttt{self.height}]\!]/2\rangle, [\![\texttt{self.heading}]\!])\\
[\![\texttt{beyond } V_1 \texttt{ by } V_2]\!] &= [\![\texttt{beyond } V_1 \texttt{ by } V_2 \texttt{ from ego.position}]\!]\\
[\![\texttt{beyond } V_1 \texttt{ by } V_2 \texttt{ from } V_3]\!] &= [\![V_1]\!] + \mathit{rotate}([\![V_2]\!], \arctan([\![V_1]\!] - [\![V_3]\!]))\\
[\![\texttt{visible}]\!] &= [\![\texttt{visible from ego}]\!]\\
[\![\texttt{visible from } P]\!] &= \mathit{uniformPointIn}(\mathit{visibleSet}(P))
\end{aligned}
$$

Figure 23: Semantics of `position` specifiers.

$$\llbracket \texttt{in } R \rrbracket = \llbracket \texttt{on } R \rrbracket = \begin{cases} \texttt{OrientedPoint}\,(x, \llbracket \mathit{orientation}\,(R) \rrbracket(x)) & \mathit{orientation}\,(R) \neq \bot \\ \texttt{OrientedPoint}\,(x, \bot) & \text{otherwise} \end{cases} \quad \text{with } x = \mathit{uniformPointIn}\,(\llbracket R \rrbracket)$$

$$\llbracket \texttt{left of } OP \rrbracket = \texttt{OrientedPoint}\,(\llbracket \texttt{OP.position} \rrbracket + \mathit{rotate}(\left\langle -\llbracket \texttt{self.width} \rrbracket/2, 0 \right\rangle, \llbracket \texttt{OP.heading} \rrbracket), \llbracket \texttt{OP.heading} \rrbracket)$$

$$\llbracket \texttt{right of } OP \rrbracket = \texttt{OrientedPoint}\,(\llbracket \texttt{OP.position} \rrbracket + \mathit{rotate}(\left\langle \llbracket \texttt{self.width} \rrbracket/2, 0 \right\rangle, \llbracket \texttt{OP.heading} \rrbracket), \llbracket \texttt{OP.heading} \rrbracket)$$

$$\llbracket \texttt{ahead of } OP \rrbracket = \texttt{OrientedPoint}\,(\llbracket \texttt{OP.position} \rrbracket + \mathit{rotate}(\left\langle 0, \llbracket \texttt{self.height} \rrbracket/2 \right\rangle, \llbracket \texttt{OP.heading} \rrbracket), \llbracket \texttt{OP.heading} \rrbracket)$$

$$\llbracket \texttt{behind } OP \rrbracket = \texttt{OrientedPoint}\,(\llbracket \texttt{OP.position} \rrbracket + \mathit{rotate}(\left\langle 0, -\llbracket \texttt{self.height} \rrbracket/2 \right\rangle, \llbracket \texttt{OP.heading} \rrbracket), \llbracket \texttt{OP.heading} \rrbracket)$$

$$\llbracket \texttt{following } F \texttt{ for } S \rrbracket = \llbracket \texttt{following } F \texttt{ from ego.position for } S \rrbracket$$

$$\llbracket \texttt{following } F \texttt{ from } V \texttt{ for } S \rrbracket = \llbracket \texttt{follow } F \texttt{ from } V \texttt{ for } S \rrbracket$$

Figure 24: Semantics of `position` specifiers that optionally specify `heading`.

$$\llbracket \texttt{facing } H \rrbracket = \llbracket H \rrbracket$$

$$\llbracket \texttt{facing } F \rrbracket = \llbracket F \rrbracket(\llbracket \texttt{self.position} \rrbracket)$$

$$\llbracket \texttt{facing toward } V \rrbracket = \arctan\,(\llbracket V \rrbracket - \llbracket \texttt{self.position} \rrbracket)$$

$$\llbracket \texttt{facing away from } V \rrbracket = \arctan\,(\llbracket \texttt{self.position} \rrbracket - \llbracket V \rrbracket)$$

$$\llbracket \texttt{apparently facing } H \rrbracket = \llbracket \texttt{apparently facing } H \texttt{ from ego.position} \rrbracket$$

$$\llbracket \texttt{apparently facing } H \texttt{ from } V \rrbracket = \llbracket H \rrbracket + \arctan\,(\llbracket \texttt{self.position} \rrbracket - \llbracket V \rrbracket)$$

Figure 25: Semantics of `heading` specifiers.

$$\llbracket \texttt{relative heading of } H \rrbracket = \llbracket \texttt{relative heading of } H \texttt{ from ego.heading} \rrbracket$$

$$\llbracket \texttt{relative heading of } H_1 \texttt{ from } H_2 \rrbracket = \llbracket H_1 \rrbracket - \llbracket H_2 \rrbracket$$

$$\llbracket \texttt{apparent heading of } OP \rrbracket = \llbracket \texttt{apparent heading of } OP \texttt{ from ego.position} \rrbracket$$

$$\llbracket \texttt{apparent heading of } OP \texttt{ from } V \rrbracket = \llbracket \texttt{OP.heading} \rrbracket - \arctan\,(\llbracket \texttt{OP.position} \rrbracket - \llbracket V \rrbracket))$$

$$\llbracket \texttt{distance to } V \rrbracket = \llbracket \texttt{distance from ego.position to } V \rrbracket$$

$$\llbracket \texttt{distance from } V_1 \texttt{ to } V_2 \rrbracket = |\llbracket V_2 \rrbracket - \llbracket V_1 \rrbracket|$$

Figure 26: Scalar operators.

$$\llbracket P \texttt{ can see } O \rrbracket = \mathit{visibleSet}\,(\llbracket P \rrbracket) \cap \mathit{boundingBox}\,(\llbracket O \rrbracket) \neq \emptyset$$

$$\llbracket V \texttt{ is in } R \rrbracket = \llbracket V \rrbracket \in \llbracket R \rrbracket$$

Figure 27: Boolean operators.

$$\llbracket F \texttt{ at } V \rrbracket = \llbracket F \rrbracket(\llbracket V \rrbracket)$$

$$\llbracket F_1 \texttt{ relative to } F_2 \rrbracket = \llbracket F_1 \rrbracket(\llbracket \texttt{self.position} \rrbracket) + \llbracket F_2 \rrbracket(\llbracket \texttt{self.position} \rrbracket)$$

$$\llbracket H \texttt{ relative to } F \rrbracket = \llbracket H \rrbracket + \llbracket F \rrbracket(\llbracket \texttt{self.position} \rrbracket)$$

$$\llbracket F \texttt{ relative to } H \rrbracket = \llbracket H \rrbracket + \llbracket F \rrbracket(\llbracket \texttt{self.position} \rrbracket)$$

$$\llbracket H_1 \texttt{ relative to } H_2 \rrbracket = \llbracket H_1 \rrbracket + \llbracket H_2 \rrbracket$$

Figure 28: Heading operators.

$$\begin{aligned}
[\![V_1 \texttt{ offset by } V_2]\!] &= [\![V_1]\!] + [\![V_2]\!] \\
[\![V_1 \texttt{ offset along } H \texttt{ by } V_2]\!] &= [\![V_1]\!] + \mathit{rotate}([\![V_2]\!], [\![H]\!]) \\
[\![V_1 \texttt{ offset along } F \texttt{ by } V_2]\!] &= [\![V_1]\!] + \mathit{rotate}([\![V_2]\!], [\![F]\!]([\![V_1]\!]))
\end{aligned}$$

Figure 29: Vector operators.

$$\begin{aligned}
[\![\texttt{visible } R]\!] &= [\![R \texttt{ visible from ego}]\!] \\
[\![R \texttt{ visible from } P]\!] &= [\![R]\!] \cap \mathit{visibleSet}([\![P]\!])
\end{aligned}$$

Figure 30: Region operators.

$$\begin{aligned}
[\![OP \texttt{ offset by } V]\!] &= [\![V \texttt{ relative to } OP]\!] \\
[\![V \texttt{ relative to } OP]\!] &= \texttt{OrientedPoint}([\![\texttt{OP.position}]\!] + \mathit{rotate}([\![V]\!], [\![\texttt{OP.heading}]\!]), [\![\texttt{OP.heading}]\!]) \\
[\![\texttt{follow } F \texttt{ for } S]\!] &= [\![\texttt{follow } F \texttt{ from ego.position for } S]\!] \\
[\![\texttt{follow } F \texttt{ from } V \texttt{ for } S]\!] &= \texttt{OrientedPoint}(y, [\![F]\!](y)) \text{ where } y = \mathit{forwardEuler}([\![V]\!], [\![S]\!], [\![F]\!]) \\
[\![\texttt{front of } O]\!] &= [\![\left\langle 0, [\![\texttt{O.height}]\!]/2 \right\rangle \texttt{ relative to } O]\!] \\
[\![\texttt{back of } O]\!] &= [\![\left\langle 0, -[\![\texttt{O.height}]\!]/2 \right\rangle \texttt{ relative to } O]\!] \\
[\![\texttt{left of } O]\!] &= [\![\left\langle -[\![\texttt{O.width}]\!]/2, 0 \right\rangle \texttt{ relative to } O]\!] \\
[\![\texttt{right of } O]\!] &= [\![\left\langle [\![\texttt{O.width}]\!]/2, 0 \right\rangle \texttt{ relative to } O]\!] \\
[\![\texttt{front left of } O]\!] &= [\![\left\langle -[\![\texttt{O.width}]\!]/2, [\![\texttt{O.height}]\!]/2 \right\rangle \texttt{ relative to } O]\!] \\
[\![\texttt{back left of } O]\!] &= [\![\left\langle -[\![\texttt{O.width}]\!]/2, -[\![\texttt{O.height}]\!]/2 \right\rangle \texttt{ relative to } O]\!] \\
[\![\texttt{front right of } O]\!] &= [\![\left\langle [\![\texttt{O.width}]\!]/2, [\![\texttt{O.height}]\!]/2 \right\rangle \texttt{ relative to } O]\!] \\
[\![\texttt{back right of } O]\!] &= [\![\left\langle [\![\texttt{O.width}]\!]/2, -[\![\texttt{O.height}]\!]/2 \right\rangle \texttt{ relative to } O]\!]
\end{aligned}$$

Figure 31: `OrientedPoint` operators.