A Model implementation details and hyperparameters.

We discuss the architectures and learning hyperparameters of our various models in the subsections below.

A.1 Physical Dynamics Model

We implemented our dynamics model with three Transformer layers for both the encoder and the decoder, and a hidden dimension of 256 for objects and actions. The resulting model has 17 million parameters. We pretrained the model for 20 epochs over 280k state transitions, with a batch size of 1024. We use an Adam optimizer (Kingma and Ba, 2014) with a learning rate of $1e^{-3}$.

A.2 Ordering attributes in decoding.

Recall that we use a left-to-right transformer to decode into an attribute representation, predicting attributes one-by-one from c_1 to c_n. Our model is agnostic to the actual order, as no matter what the order is, it still is modeling a decomposition of the joint probability of generating that object. However, we implemented this by using the name as the first attribute c_1 that is predicted, and ordered the rest in a descending way by vocabulary size so as to predict harder attributes first.

A.3 Optimization Hyperparameters chosen

We finetuned PIGLeT for both tasks with an Adam optimizer (Kingma and Ba, 2014). We did a small grid search for hyperparameter values, choosing the best learning rate $\{2e^{-5}, 1e^{-5}, 1e^{-6}\}$ by accuracy on the development set, and likewise the best batch size 16 or 32. We considered freezing the physical dynamics backbone as another hyperparameter. We found it slightly boosted performance on PIGPeN-NLG when we froze the physical dynamics backbone, but not so for PIGPeN-NLU. We trained our model for 80 epochs on paired data.

We trained the baseline models with the same backbone in the same way, using similar hyperparameters. However, we found that after 80 epochs, the baseline models without pretrained dynamics failed to converge, so we finetuned them for 200 epochs total. For T5, we used similar identical hyperparameter ranges as the other models. However, because T5 uses a different optimizer (AdaFactor; Shazeer and Stern (2018)), which operates on a slightly different scale, we used a different set of learning rates. We chose the best one over $\{1e^{-4}, 2e^{-4}, 4e^{-4}\}$.

Search. Both of our tasks involve left-to-right decoding. We used argmax (greedy) search for PIGPeN-NLU, finding that it worked well as a 'closed-ended generation' style task. On the other hand, we used Nucleus Sampling for PIGPeN-NLG as there are often several ways to communicate a state transition; here we set $p = 0.8$.

A.4 Encoding the input for text-to-text models

Text-to-text models, needless to say, can only handle text. We encode the world states into a representation suitable for these models by formatting the object states as a JSON-style dictionary of keys and values. We had to make several modifications to the encoding however from a default JSON, because we handle a lot of attributes in this task, and JSON has quote characters '' that take up a lot of space in a BPE encoding. We thus strip the quote characters and lowercase everything (with this also helping BPE-efficiency). We put parentheses around each object and give names to all 'binned' attributes.

An example encoding might be:

```
Predict next object states: (objectname: bowl, parentreceptacles: cabinet, containedobjects: none, distance: 6 to 8 ft, mass: .5 to 1lb, size: medium, temp: roomtemp, breakable: true, cookable: false, dirtyable: true, broken: false, cooked: false, dirty: false, filledwithliquid: false, open: false, pickedup: false, sliced: false, toggled: false, usedup: false, moveable: false, openable: false, pickupable: true, receptacle: true, sliceable: false, toggleable: false, materials: glass) (objectname: egg, parentreceptacles: none, containedobjects: none, distance: 2 to 3ft, mass: .1 to .2lb, size: tiny, temp: cold, breakable: true, cookable: true, dirtyable: false, broken: false, cooked: false, dirty: false, filledwithliquid: false, open: false, pickedup: true, sliced: false, toggled: false, usedup: false, moveable: false, openable: false, pickupable: true, receptacle: false, sliceable: true, toggleable: false, materials: food) (action: throwobject10)
```

We have models decode directly into this kind of format when predicting state changes. Though the T5 models usually get the format right, we often have to sanitize the text in order for it to be a valid object state in our framework. This is espe-

cially an issue with GPT3, since it is given limited supervision (we squeeze 3 examples into the 2048-BPE token context window) and often makes up new names and attributes. Thus, for each word not in an attribute's vocabulary, we use a Levenstein distance heuristic to match the an invalid choice with its closest (valid) option. If the model fails to generate anything for a certain attribute key – for example if it does not include something like `openable` somewhere, we copy the representation of the input object for that attribute, thereby making the default assumption that attributes do not change.

B All THOR attributes

We list a table with all of the attributes we used for this work in Table 4.

C Turk Annotation Details

We followed crowdsourcing best practices, such as using a qualification exam, giving feedback to workers, and paying workers well (above $15 per hour). Each of our HITs required writing three sentences, and we paid Mechanical Turk workers 57 cents per HIT. We used three workers per example, allowing us to have multiple language references for evaluation. A screenshot of our user interface is shown in Figure 6.

D Our Pretrained Language Model

We use our own pretrained language model primarily because it allows us to investigate the impact of data on model performance. We trained a prefix-masked language model (Dong et al., 2019) on Wikipedia and Book data, mimicing the data used by the original BERT paper (Devlin et al., 2019). We trained the model for 60000 iterations, at a batch size of 8192 sequences each of length 512. This corresponds to 50 epochs over the dataset. We masked inputs in the bidirectional prefix with SpanBERT masking (Joshi et al., 2020). Since BERT-style 'masked' out inputs are easier to predict than tokens generated left-to-right, we reduced the loss component of left-to-right generation by a factor of 20; roughly balancing the two loss components.

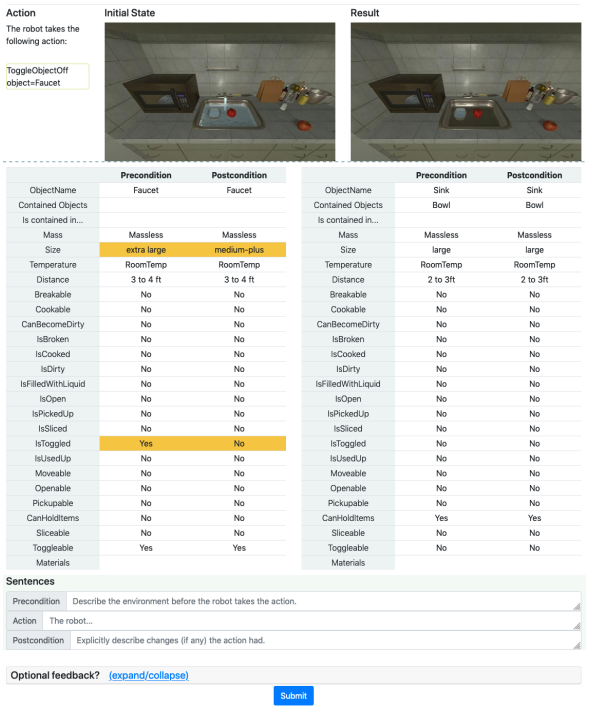

Figure 6: Our user interface for Mechanical Turk annotation.

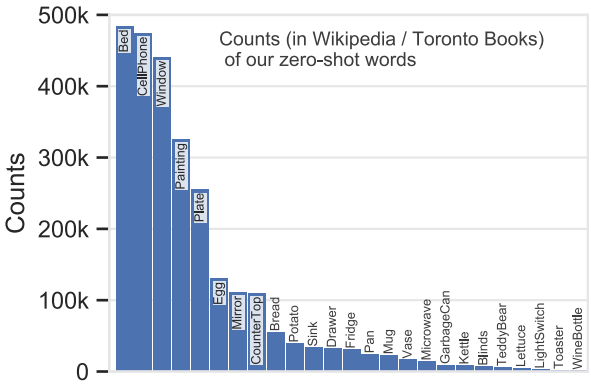

Figure 7: Counts of zero-shot words that appear in BERT's training data (Wikipedia and Toronto Books). For example, in the 4 billion words BERT is trained on, it sees the word 'Bed' almost 500k times. This might allow it to perform superficially well at answering questions about beds – while not necessarily possessing deep physical knowledge about them.

Attribute Name	Vocab size	Values
objectName	126	One per object type, along with `None`
parentReceptacles	126	One per object type, along with `None`
receptacleObjectIds	126	One per object type, along with `None`
mass	8	8 bins
size	8	8 bins
distance	8	8 bins
ObjectTemperature	3	`Hot`, `Cold`, `RoomTemp`
breakable	2	
canBeUsedUp	2	
canFillWithLiquid	2	
cookable	2	
dirtyable	2	
isBroken	2	
isCooked	2	
isDirty	2	
isFilledWithLiquid	2	
isOpen	2	
isPickedUp	2	
isSliced	2	
isToggled	2	
isUsedUp	2	
moveable	2	
openable	2	
pickupable	2	
receptacle	2	
sliceable	1	
toggleable	2	
salientMaterials_Ceramic	2	
salientMaterials_Fabric	2	
salientMaterials_Food	2	
salientMaterials_Glass	2	
salientMaterials_Leather	2	
salientMaterials_Metal	2	
salientMaterials_None	2	
salientMaterials_Organic	2	
salientMaterials_Paper	2	
salientMaterials_Plastic	2	
salientMaterials_Rubber	2	
salientMaterials_Soap	2	
salientMaterials_Sponge	2	
salientMaterials_Stone	2	
salientMaterials_Wax	2	
salientMaterials_Wood	2	

Table 4: All attributes that we consider for this work in THOR. We list the attribute's name, the size of the attribute vocabulary, and the range of values the attribute can take on. For attributes like 'mass', 'size', and 'distance', we note that the underlying simulator stores them as floats; we bin them to 8 values for this work. All the values for attributes with a vocabulary size of 2 are boolean.

Generator	Description
`put_X_in_Y`	Samples an object `X` from the scene, and a receptacle `Y`. Tries to put it in `Y`.
`throw_X_at_Y`	Samples two objects `X` and `Y` from the scene. Picks up `X`, moves to face `Y`, and throws it forward with variable intensity.
`toggle_X`	Samples an object `X`, and turns it on or off.
`slice_X`	Samples an object `X` and a surface `Y`. Picks up `X`, places it on `Y`, and cuts it.
`dirty_X`	Samples an object `X`, and makes it dirty.
`clean_X`	Samples a dirty object `X`. Finds a `Sink` nearby a `Faucet`, and places `X` inside. Turns on/off the `Faucet`, cleaning `X`.
`toast_bread`	Finds some `Bread`, slicing it if necessary, places it in a `Toaster`, then turns it on.
`brew_coffee`	Picks up a `Mug`, places it under a `CoffeeMachine`, and turns the machine on.
`fry_X`	Picks up a food `X`, slices it if necessary, and puts it in a `Pot` or `Pan`. Brings it to a `StoveBurner` and turns the burner on.
`microwave_X`	Picks up an object `X` and slices it if necessary. Places it in a `Microwave`, closes it, and then turns it on.
`fill_X`	Picks up an object `X` places it under a `Faucet`. Turns on/off the `Faucet`, then pours out the liquid.

Table 5: Trajectory generation functions that we used to sample 'interesting' physical interactions, such as the effects that actions will have on objects, and which actions will succeed or not.