# Supplementary Materials for "ThreeDWorld: A Platform for Interactive Multi-Modal Physical Simulation"

We confirm the usage of MIT license, as our data can be created with the released code. The authors state that we bear all responsibility in case of violation of rights, etc. We will maintain the platform, dataset, and codebase for long-term accessibility. A download of TDW's full codebase and documentation is available at: https://github.com/threedworld-mit/tdw; the code for creating datasets described in this paper are available at: TDW-Image, TDW-Sound, and TDW-Physics.

In this supplement, we start by discussing the broader impact of TDW. Section 2 discusses implementation details of the TDW image dataset, and explains the setup of each scenario in the Advanced Physical Prediction Benchmark dataset described in the paper. Section 3 introduces the details of training the HRN [14] and new proposed DRHRN for physical dynamics predictions. We then elaborate on the lighting model used in TDW, and in Section 5 discuss in more detail how TDW compares to other simulation environments. Lastly, Section 6 provides a detailed overview of TDW's system architecture, API, benchmarks and code examples showing both back-end and front-end functionality.

## 1 Broader Impact

As we have illustrated, TDW is a completely general and flexible simulation platform, and as such can benefit research that sits at the intersection of neuroscience, cognitive science, psychology, engineering and machine learning / AI. We feel the broad scope of the platform will support research into understanding how the brain processes a range of sensory data – visual, auditory and even tactile – as well as physical inference and scene understanding. We envision TDW and PyImpact supporting research into human – and machine – audio perception, that can lead to a better understanding of the computational principles underlying human audition. This understanding can, for example, ultimately help to create better assistive technology for the hearing-impaired. We recognize that the diversity of "audio materials" used in PyImpact is not yet adequate to meet this longer-term goal, but we are actively addressing that and plan to increase the scope significantly. We also believe the wide range of physics behaviors and interaction scenarios TDW supports will greatly benefit research into understanding how we as humans learn so much about the world, so rapidly and flexibly, given minimal input data. While we have made significant strides in the accuracy of physics behavior in TDW, TDW is not yet able to adequately support robotic simulation tasks. To support visual object recognition and image understanding we constantly strive to make TDW's image generation as photoreal as possible using today's real-time 3D technology. However, we are not yet at the level we would like to be. We plan to continue improving our rendering and image generation capability, taking advantage of any relevant technology advances (e.g. real-time hardware-assisted ray tracing) while continuing to explore the relative importance of object variance, background variability and overall image quality to vision transfer results.

35th Conference on Neural Information Processing Systems (NeurIPS 2021) Track on Datasets and Benchmarks.

## 2 Dataset Details

### 2.1 TDW-image Dataset

To generate images, the controller runs each model through two loops. The first loop captures camera and object positions, rotations, etc. Then, these cached positions are played back in the second loop to generate images. Image capture is divided this way because the first loop will "reject" a lot of images with poor composition; this rejection system doesn't require image data, and so sending image data would slow down the entire controller.

The controller relies on IdPassGrayscale data to determine whether an image has good composition. This data reduces the rendered frame of a segmentation color pass to a single pixel and returns the grayscale value of that pixel. To start the positional loop, the entire window is resized to $32 \times 32$ and render quality is set to minimal, in order to speed up the overall process. There are then two grayscale passes: One without occluding objects (by moving the camera and object high above the scene) and one with occluding scenery, but the exact same relative positions and rotations. The difference in grayscale must exceed 0.55 for the camera and object positions and rotations to be "accepted". This data is then cached. In a third pass, the screen is resized back to $256 \times 256$, images and high-quality rendering are enabled, and the controller uses the cached positional/rotational data to iterate rapidly through the dataset.


Figure 1: Examples from the TDW pre-training dataset, to be released as part of the TDW package.

### 2.2 Advanced Physical Prediction Benchmark

Individual descriptions of each of the physics dataset scenarios as mentioned in the paper and shown in the Supplementary Material video. Note that additional scenarios are included here that were not mentioned in the paper; some are included in the video.

**Binary Collisions** Randomly-selected "toys" are created with random physics values. A force of randomized magnitude is applied to one toy, aimed at another.

**Complex Collisions** Multiple objects are dropped onto the floor from a height, with randomized starting positions and orientations.

**Object Occlusion** Random "big" and "small" models are added. The small object is at random distance and angle from the big object. The camera is placed at a random distance and rotated such that the "big" model occludes the "small" model in some frames. Note – not included in video.

**Object Permanence** A ball rolls behind an occluding object and then reemerges. The occluder is randomly chosen from a list. The ball has a random starting distance, visual material, physics values, and initial force.

**Shadows** A ball is added in a scene with a randomized lighting setup. The ball has a random initial position, force vector, physics values, and visual materials. The force vectors are such that the ball typically rolls through differently-lit areas, i.e. a bright spot to a shadowy spot.

**Stability** A stack of 4-7 objects is created. The objects are all simple shapes with random colors. The stack is built according to a "stability" algorithm; some algorithms yield more balanced stacks than others. The stack falls down, or doesn't.

**Containment** A small object is contained and rattles around in a larger object, such as a basket or bowl. The small object has random physics values. The bowl has random force vectors.

**Sliding/Rolling** Objects are placed on a table. A random force is applied at a random point on the table. The objects slide or roll down.

**Bouncing** Four "ramp" objects are placed randomly in a room. Two to six "toy" objects are added to the room in mid-air and given random physics values and force vectors, such that they will bounce around the scene. Note – not included in video.

**Draping/Folding** A cloth falls, 80 percent of the time onto another rigid body object. The cloth has random physics values.

**Dragging** A rigid object is dragged or moved by pulling on a cloth under it. The cloth and the object have random physics values. The cloth is pulled in by a random force vector.

**Squishing** Squishy objects deform and are restored to original shape depending on applied forces (e.g. squished when something else is on top of them or when they impact a barrier). Note – not included in video.

**Submerging** Objects sink or float in fluid. Values for viscosity, adhesion and cohesion vary by fluid type, as does the visual appearance of the fluid. Fluids represented in the video include water, chocolate, honey, oil and glycerin.

## 3 Training a Learnable Intuitive Physics Simulator

**HRN Architecture.** We re-implemented the HRN architecture as published [14], using the Tensorflow-2.1 library. To predict the future physical state, the HRN resolves physical constraints that particles connected in the hierarchical graph impose on each other. Graph convolutions are used to compute and propagate these effects. Following [3], the HRN uses a *pairwise graph convolution* with two basic building blocks: (1) A pairwise processing unit $\phi$ that takes the sender particle state $p_s$, the receiver particle state $p_r$ and their relation $r_{sr}$ as input and outputs the effect $e_{sr} \in \mathbb{R}^E$ of $p_s$ on $p_r$, and (2) a commutative aggregation operation $\Sigma$ which collects and computes the overall effect $e_r \in \mathbb{R}^E$. In our case this aggregation is a simple summation over all effects on $p_r$. Together these two building blocks form a convolution on graphs. The HRN has access to the Flex particle representation of each object, which is provided at every simulation step by the environment. From this particle representation, we construct a hierarchical particle relationship scene graph representation $G_H$. Graph nodes correspond to either particles or groupings of other nodes and are arranged in a hierarchy, whereas edges represent constraints between nodes. The HRN as the dynamics model takes a history of hierarchical graphs $G_H^{(t-T,t]}$ as input and predicts the future particle states $P^{t+1}$. The model first computes collision effects between particles ($\phi_C^W$), effects of external forces ($\phi_F^W$), and effects of past particles on current particles ($\phi_F^W$) using pairwise graph convolutions. The effects are then propagated through the particle hierarchy using a hierarchical graph convolution module $\eta^W$. First effects are propagated from leaf to ancestor particles (L2A), then within siblings (WG), and finally from ancestors to descendants (A2D). Finally, the fully-connected module $\psi^W$ computes the next particle states $P^{t+1}$ from the summed effects and past particle states.

**Dynamic Recurrent HRN (DRHRN) for large environments.** Representing environment components (floor, walls) at the particle resolution of small objects is inefficient. Decreasing the resolution is problematic as small objects might miss environment interactions. Instead, we propose to initially model environment components as a sparse triangular mesh. At any given time, we compute each object particle's contact point with the environment by intersecting a ray originating from the particle in the direction of the mesh surface normal. If the contact point is closer than distance $d$, we spawn particles onto the triangle surface at the resolution of the small object. We dynamically add these particles to the graph $G_H^t$ and connect them to the object particle. Conversely, we delete environment nodes from the graph when objects move away from the environment. With this novel dynamic resolution method, which we call the Dynamic Recurrent HRN (DRHRN), we can efficiently represent large environments that can be modeled with TDW.

DRHRN also builds on the original HRN by introducing an improved training loss and recurrent component. Specifically, the DRHRN loss predicts the position delta between current and next state $\Delta p = p^{t+1} - p^t$ using $L_2$ loss ($L_{Delta}$). To preserve object structure and shape, we additionally match the pairwise distance between predicted particle positions within each object $d_{ij} = ||p_i - p_j||$ to the

ground truth particle distances via $L_2$ loss ($L_{Structure}$). The total loss is equal to the $\alpha$ weighted sum of both loss terms: $L = \alpha L_{Structure} + (1 - \alpha) L_{Delta}$.

Iterative physics prediction models accumulate errors exponentially. Naively trained one-step physics predictors only operate on ground truth input and do not see their own predictions as input during training, despite being tested via unrolling the model. To make DRHRN robust against its own prediction errors, we therefore train the model recurrently in a state-invariant way, i.e. without using a hidden state, as physical dynamics is state-free. The overall loss is then defined as the sum of all per time step losses.

## 4 TDW Lighting Model

The lighting model for both interior and exterior environments utilizes a single primary light source that simulates the sun and provides direct lighting, affecting the casting of shadows. In most interior environments, additional point or spot lights are also used to simulate the light coming from lighting fixtures in the space.

General environment (indirect) lighting comes from "skyboxes" that utilize High Dynamic Range images (HDRI). Skyboxes are conceptually similar to a planetarium projection, while HDRI images are a special type of photographic digital image that contain more information than a standard digital image. Photographed at real-world locations, they capture lighting information for a given latitude and hour of the day. This technique is widely used in movie special-effects, when integrating live-action photography with CGI elements.

TDW's implementation of HDRI lighting automatically adjusts:

- The elevation of the "sun" light source to match the time of day in the original image; this affects the length of shadows.
- The intensity of the "sun" light, to match the shadow strength in the original image.
- The rotation angle of the "sun" light, to match the direction shadows are pointing in the original image .

By rotating the HDRI image, we can realistically simulate different viewing positions, with corresponding changes in lighting, reflections and shadowing in the scene (see the Supplementary Material video for an example).

TDW currently provides over 100 HDRI images captured at various locations around the world and at different times of the day, from sunrise to sunset. These images are evenly divided between indoor and outdoor locations.

## 5 Related Simulation Environments

Recently, several simulation platforms have been developed to support research into embodied AI, scene understanding, and physical inference. These include AI2-THOR[11], HoME[23], VirtualHome[16], Habitat[17], Gibson[25], iGibson [24], Sapien [26] PyBullet [7], MuJuCo [20], and Deepmind Lab [4]. However none of them approach TDW's range of features and diversity of potential use cases.

**Rendering and Scene Types.** Research in computer graphics (CG) has developed extremely photo-realistic rendering pipelines [12]. However, the most advanced techniques (e.g. ray tracing), have yet to be fully integrated into real-time rendering engines. Some popular simulation platforms, including Deepmind Lab [4] and OpenAI Gym [5], do not target realism in their rendering or physics and are better suited to prototyping than exploring realistic situations. Others use a variety of approaches for more realistic visual scene creation – scanned from actual environments (Gibson, Habitat), artist-created (AI2-THOR) or using existing datasets such as SUNCG [19] (HoME). However all are limited to the single paradigm of rooms in a building, populated by furniture, whereas TDW supports real-time near-photorealistic rendering of both indoor and outdoor environments. Only TDW allows users to create custom environments procedurally, as well as populate them with custom object configurations for specialized use-cases. For example, it is equally straightforward with TDW to arrange a living room full of furniture (see Fig. 1a-b), to generate photorealistic images of outdoor

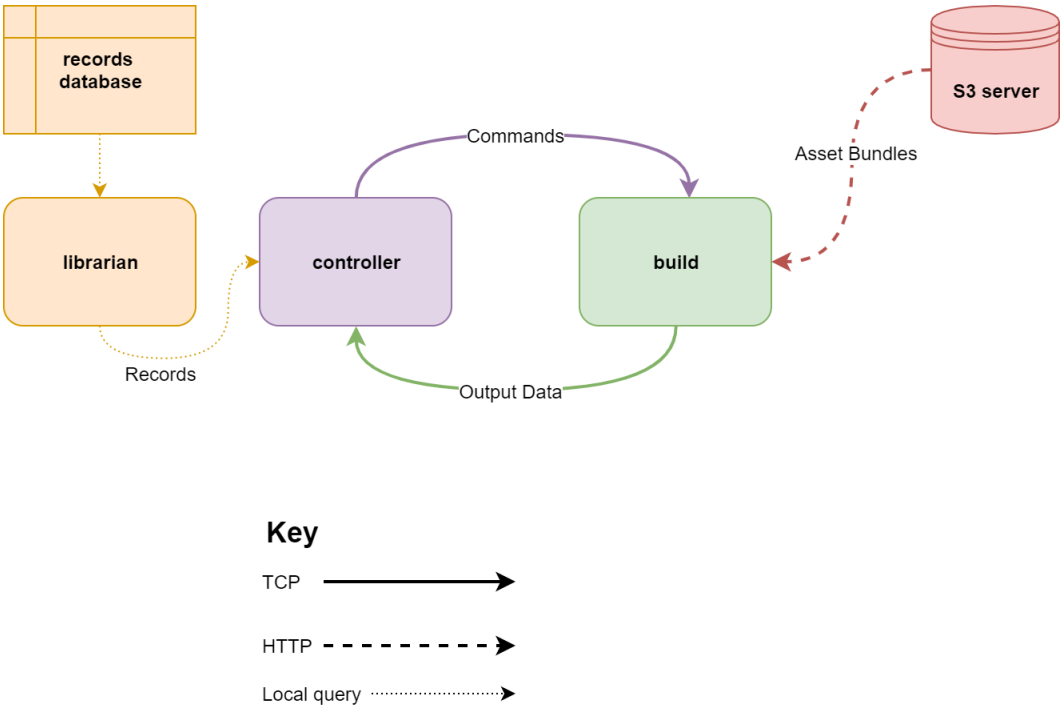
scenes (Fig. 1c) to train networks for transfer to real-world images, or to construct a "Rube Goldberg" machine for physical inference experiments (Fig. 1h).

**Physical Dynamics.** Several stand-alone physics engines are widely used in AI training, including PyBullet and MuJuCo which support a range of accurate and complex physical interactions. However, these engines do not generate high-quality images or audio output. Conversely, platforms with real-world scanned environments, such as Gibson and Habitat, do not support free interaction with objects. HoME does not provide photorealistic rendering but does support rigid-body interactions with scene objects, using either simplified (but inaccurate) "box-collider" bounding-box approximations or the highly inefficient full object mesh. AI2-THOR provides better rendering than HoME or VirtualHome, with similar rigid-body physics to HoME. In contrast, TDW automatically computes convex hull colliders that provide mesh-level accuracy with box-collider-like performance (Fig. 2). This fast-but-accurate high-res rigid body (denoted "RF" in Table 1) appears unique among integrated training platforms. Also unique is TDW's support for complex non-rigid physics, based on the NVIDIA FLeX engine (Fig. 1d). Taken together, TDW is substantially more full-featured for supporting future development in rapidly-expanding research areas such as learning scene dynamics for physical reasoning [28, 27] and model-predictive planning and control [2, 13, 9, 3, 6, 1, 18, 8, 15].

**Audio.** As with CG, advanced work in computer simulation has developed powerful methods for physics-based sound synthesis [10] based on object material and object-environment interactions. In general, however, such physics-based audio synthesis has not been integrated into real-time simulation platforms. HoME and PyBullet are the only other platforms to provide audio output, generated by user-specified pre-placed sounds. TDW, on the other hand, implements a physics-based model to generate situational sounds from object-object interactions (Fig. 1h). TDW's PyImpact Python library computes impact sounds via modal synthesis with mode properties sampled from distributions conditioned upon properties of the sounding object [21]. The mode distributions were measured from recordings of impacts. The stochastic sound generation prevents overfitting to specific audio sequences. In human perceptual experiments, listeners could not distinguish our synthetic impact sounds from real impact sounds, and could accurately judge physical properties from the synthetic audio[21]. For this reason, TDW is substantially more useful for multi-modal inference problems such as learning shape and material from sound [22, 29].

**Interaction and API** All the simulation platforms discussed so far require some form of API to control an agent, receive state of the world data or interact with scene objects. However not all support interaction with objects within that environment. Habitat focuses on navigation within indoor scenes, and its Python API is comparable to TDW's but lacks capabilities for interaction with scene objects via physics (Fig. 1e), or multi-modal sound and visual rendering (Fig. 1h). VirtualHome, iGibson and AI2-THOR's interaction capabilities are closer to TDW's. In VirtualHome and AI2-THOR, interactions with objects are explicitly animated, not controlled by physics. TDW's API, with its multiple paradigms for true physics-based interaction with scene objects, provides a set of tools that enable the broadest range of use cases of any available simulation platform.

# 6 System overview and API




## 6.1 Core components

1. **The build** is the 3D environment application. It is available as a compiled executable.
2. **The controller** is an external Python script created by the user, which communicates with the build.
3. **The S3 server** is a remote server. It contains the binary files of each model, material, etc. that can be added to the build at runtime.
4. **The records databases** are a set of local .json metadata files with records corresponding to each asset bundle.
5. A **librarian** is a Python wrapper class to easily query metadata in a records database file.

## 6.2 The simulation pattern

1. The controller communicates with the build by sending a list of **commands**.
2. The build receives the list of serialized Commands, deserializes them, and executes them.
3. The build advances 1 physics frame (simulation step).
4. The build returns **output data** to the controller.

Output data is always sent as a list, with the last element of the list being the frame number:

```
[data, data, data, frame]
```

## 6.3 The controller

All controllers are sub-classes of the `Controller` class. Controllers send and receive data via the `communicate` function:

```
from tdw.controller import Controller

c = Controller()
```

```
# resp will be a list with one element: [frame]
resp = c.communicate({"$type": "load_scene", "scene_name": "ProcGenScene"})
```

Commands can be sent in lists of arbitrary length, allowing for arbitrarily complex instructions per frame. The user must explicitly request any other output data:

```
from tdw.controller import Controller
from tdw.tdw_utils import TDWUtils
from tdw.librarian import ModelLibrarian
from tdw.output_data import OutputData, Bounds, Images

lib = ModelLibrarian("models_full.json")
# Get the record for the table.
table_record = lib.get_record("small_table_green_marble")

c = Controller()

table_id = 0

# 1. Load the scene.
# 2. Create an empty room (using a wrapper function)
# 3. Add the table.
# 4. Request Bounds data.
resp = c.communicate([{"$type": "load_scene",
                       "scene_name": "ProcGenScene"},
                      TDWUtils.create_empty_room(12, 12),
                      {"$type": "add_object",
                       "name": table_record.name,
                       "url": table_record.get_url(),
                       "scale_factor": table_record.scale_factor,
                       "position": {"x": 0, "y": 0, "z": 0},
                       "rotation": {"x": 0, "y": 0, "z": 0},
                       "category": table_record.wcategory,
                       "id": table_id},
                      {"$type": "send_bounds",
                       "frequency": "once"}])
```

The `resp` object is a list of byte arrays that can be deserialized into output data:

```
# Get the top of the table.
top_y = 0
for r in resp[:-1]:
    r_id = OutputData.get_data_type_id(r)
    # Find the bounds data.
    if r_id == "boun":
        b = Bounds(r)
        # Find the table in the bounds data.
        for i in range(b.get_num()):
            if b.get_id(i) == table_id:
                top_y = b.get_top(i)
```

The variable `top_y` an be used to place an object on the table:

```
box_record = lib.get_record("iron_box")
box_id = 1
c.communicate({"$type": "add_object",
               "name": box_record.name,
               "url": box_record.get_url(),
               "scale_factor": box_record.scale_factor,
```

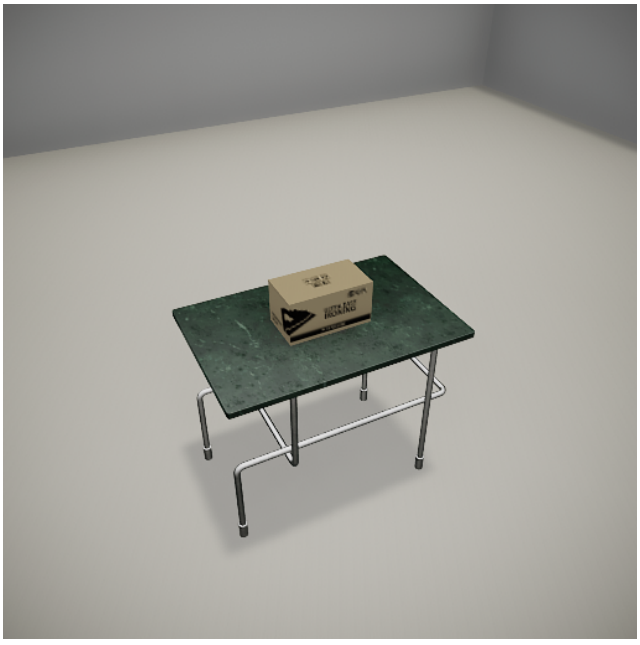
```
                "position": {"x": 0, "y": top_y, "z": 0},
                "rotation": {"x": 0, "y": 0, "z": 0},
                "category": box_record.wcategory,
                "id": 1})
```

Then, an “avatar" can be added to the scene. In this case, the avatar is a just a camera. The avatar can then send an image:

```
avatar_id = "a"
resp = c.communicate([{"$type": "create_avatar",
                       "type": "A_Img_Caps_Kinematic",
                       "avatar_id": avatar_id},
                      {"$type": "teleport_avatar_to",
                       "position": {"x": 1, "y": 2.5, "z": 2}},
                      {"$type": "look_at",
                       "avatar_id": avatar_id,
                       "object_id": box_id},
                      {"$type": "set_pass_masks",
                       "avatar_id": avatar_id,
                       "pass_masks": ["_img"]},
                      {"$type": "send_images",
                       "frequency": "once",
                       "avatar_id": avatar_id}])

# Get the image.
for r in resp[:-1]:
    r_id = OutputData.get_data_type_id(r)
    # Find the image data.
    if r_id == "imag":
        img = Images(r)
```

This image is a numpy array that can be either saved to disk or fed directly into a ML system.Put together, the example code will create this image:



### 6.4 Benchmarks

**CPU:** Intel i7-7700K @4.2GHz **GPU:** NVIDIA GeForce GTX 1080

| Benchmark | Quality | Size | FPS |
|---|---|---|---|
| Object data | N/A | N/A | 850 |
| Images | low | 256x256 | 380 |
| Images | high | 256x256 | 168 |

### 6.5 Command API Backend

#### 6.5.1 Implementation Overview

Every command in the Command API is a subclass of `Command`.

```
/// <summary>
/// Abstract class for a message sent from the controller to the build.
/// </summary>
public abstract class Command
{
    /// <summary>
    /// True if command is done.
    /// </summary>
    protected bool isDone = false;


    /// <summary>
    /// Do the action.
    /// </summary>
    public abstract void Do();


    /// <summary>
    /// Returns true if this command is done.
    /// </summary>
    public bool IsDone()
    {
        return isDone;
    }
}
```

Every command must override `Command.Do()`. Because some commands require multiple frames to finish, they announce that they are “done" via `Command.IsDone()`.

```
///<summary>
/// This is an example command.
/// </summary>
public class ExampleCommand : Command
{
    ///<summary>
    /// This integer will be output to the console.
    /// </summary>
    public int integer;


    public override void Do()
    {
        Debug.Log(integer);
        isDone = true;
    }
}
```

**Commands are *automatically* serialized and deserialized as JSON dictionaries** In a user-made controller script, `ExampleCommand` looks like this:

```
{"$type": "example_command", "integer": 15}
```

If the user sends that JSON object from the controller, the build will deserialize it to an `ExampleCommand`-type object and call `ExampleCommand.Do()`, which will output `15` to the console.

### 6.5.2 Type Inheritance

The Command API relies heavily on type inheritance, which is handled automatically by the JSON converter. Accordingly, new commands can easily be created without affecting the rest of the API, and bugs affecting multiple commands are easy to identify and fix.

```
/// <summary>
/// Manipulate an object that is already in the scene.
/// </summary>
public abstract class ObjectCommand : Command
{
    /// <summary>
    /// The unique object ID.
    /// </summary>
    public int id;


    public override void Do()
    {
        DoObject(GetObject());
        isDone = true;
    }


    /// <summary>
    /// Apply command to the object.
    /// </summary>
    /// <param name="co">The model associated with the ID.</param>
    protected abstract void DoObject(CachedObject co);

    /// <summary>
    /// Returns a cached model, given the ID.
    /// </summary>
    protected CachedObject GetObject()
    {
        // Additional code here.
    }
}


/// <summary>
/// Set the object's rotation such that its forward directional vector points
/// towards another position.
/// </summary>
public class ObjectLookAtPosition : ObjectCommand
{
    /// <summary>
    /// The target position that the object will look at.
    /// </summary>
    public Vector3 position;


    protected override void DoObject(CachedObject co)
    {
        co.go.transform.LookAt(position);
```

```
    }
}
```

The TDW backend includes a suite of auto-documentation scripts that scrape the `<summary>` comments to generate a markdown API page complete with example JSON per command, like this:

`object_look_at_position`

Set the object's rotation such that its forward directional vector points towards another position.

```
{"$type": "object_look_at_position", "id": 1, "position": {"x": 3.5, "y": -45, "z": 0}}
```

| Parameter | Type | Description |
|---|---|---|
| `"id"` | int | The unique object ID. |
| `"position"` | Vector3 | The target position that the object will look at. |

## References


[1] Pulkit Agrawal, Ashvin Nair, Pieter Abbeel, Jitendra Malik, and Sergey Levine. Learning to poke by poking: Experiential learning of intuitive physics. *CoRR*, abs/1606.07419, 2016.

[2] Peter Battaglia, Jessica Hamrick, and Joshua Tenenbaum. Simulation as an engine of physical scene understanding. *Proceedings of the National Academy of Sciences of the United States of America*, 110, 10 2013.

[3] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *CoRR*, abs/1612.00222, 2016.

[4] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[6] Michael B. Chang, Tomer Ullman, Antonio Torralba, and Joshua B. Tenenbaum. A compositional object-based approach to learning physical dynamics. *CoRR*, abs/1612.00341, 2016.

[7] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. *GitHub repository*, 2016.

[8] Amy Fire and Song-Chun Zhu. Learning perceptual causality from video. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 7(2):23, 2016.

[9] Katerina Fragkiadaki, Pulkit Agrawal, Sergey Levine, and Jitendra Malik. Learning visual predictive models of physics for playing billiards. 11 2015.

[10] Doug L James, Jernej Barbič, and Dinesh K Pai. Precomputed acoustic transfer: output-sensitive, accurate sound generation for geometrically complex vibration sources. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 987–995. ACM, 2006.

[11] Eric Kolve, Roozbeh Mottaghi, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*, 2017.

[12] Markus Kuhlo and Enrico Eggert. Architectural rendering with 3ds max and v-ray, 2010.

[13] Roozbeh Mottaghi, Mohammad Rastegari, Abhinav Gupta, and Ali Farhadi. "what happens if..." learning to predict the effect of forces in images. *CoRR*, abs/1603.05600, 2016.

[14] Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li F Fei-Fei, Josh Tenenbaum, and Daniel L Yamins. Flexible neural representation for physics prediction. In *Advances in Neural Information Processing Systems*, pages 8799–8810, 2018.

[15] Judea Pearl. *Causality*. Cambridge University Press, 2009.

[16] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *CVPR*, pages 8494–8502, 2018.

[17] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A platform for embodied ai research. *ICCV*, 2019.

[18] Tianjia Shao*, Aron Monszpart*, Youyi Zheng, Bongjin Koo, Weiwei Xu, Kun Zhou, and Niloy Mitra. Imagining the unseen: Stability-based cuboid arrangements for scene understanding. *ACM SIGGRAPH Asia 2014*, 2014. * Joint first authors.

[19] Shuran Song, Fisher Yu, Andy Zeng, Angel X Chang, Manolis Savva, and Thomas Funkhouser. Semantic scene completion from a single depth image. *CVPR*, 2017.

[20] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[21] James Traer, Maddie Cusimano, and Josh H. McDermott. A perceptually inspired generative model of rigid-body contact sounds. *Digital Audio Effects (DAFx)*, pages 136–143, 2019.

[22] Yunyun Wang, Chuang Gan, Max H Siegel, Zhoutong Zhang, Jiajun Wu, and Joshua B Tenenbaum. A computational model for combinatorial generalization in physical auditory perception. *CCN*, 2017.

[23] Yi Wu, Yuxin Wu, Georgia Gkioxari, and Yuandong Tian. Building generalizable agents with a realistic and rich 3d environment. *arXiv preprint arXiv:1801.02209*, 2018.

[24] Fei Xia, William B Shen, Chengshu Li, Priya Kasimbeg, Micael Edmond Tchapmi, Alexander Toshev, Roberto Martín-Martín, and Silvio Savarese. Interactive gibson benchmark: A benchmark for interactive navigation in cluttered environments. *IEEE Robotics and Automation Letters*, 5(2):713–720, 2020.

[25] Fei Xia, Amir R Zamir, Zhiyang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson env: Real-world perception for embodied agents. In *CVPR*, pages 9068–9079, 2018.

[26] Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, et al. SAPIEN: A simulated part-based interactive environment. *CVPR*, 2020.

[27] Tian Ye, Xiaolong Wang, James Davidson, and Abhinav Gupta. Interpretable intuitive physics model. In *The European Conference on Computer Vision (ECCV)*, September 2018.

[28] Kexin Yi, Chuang Gan, Yunzhu Li, Pushmeet Kohli, Jiajun Wu, Antonio Torralba, and Joshua B Tenenbaum. CLEVRER: Collision events for video representation and reasoning. *ICLR*, 2020.

[29] Zhoutong Zhang, Qiujia Li, Zhengjia Huang, Jiajun Wu, Josh Tenenbaum, and Bill Freeman. Shape and material from sound. In *NIPS*, pages 1278–1288, 2017.