Build a self-driving car without a car: ML problem-solving with a game engine

Paris Buttfield-Addison (@parisba), Tim Nugent (@the_mcjones), Mars Geldard (@themartianlife)

Additional Material by Jon Manning (@desplesda)

Unity Machine Learning Agents Toolkit

The Unity Machine Machine Learning Agents Toolkit (ML-Agents) is an open-source suite of tools, including Unity plugins, Python scripts, and algorithm implementations, that enables the Unity environment to serve for both training and inference of intelligent agents.

This document provides an outline of the material from the tutorial Build a self-driving car without a car: ML problem-solving with a game engine from the O'Reilly Artificial Intelligence Conference 2019, in San Jose.

It's not really intended to stand-alone, without readers having attended the conference, but it could still be useful!

This document is sourced from, and accompanied by, the contents of this GitHub repository: https://github.com/parisba/ORM-AI-SJC-2019-Unity-ML-Agents/

WARNING	This document isn't intended to explain why everything works the way it does. This document is here to help you keep your place with the tutorial content at AI Conference. It's so you don't fall behind, and have something to refer to if you need to catch up. We recommend you keep your own notes about why things work the way they do.

Structure

We've structured this document, and the tutorial, as follows:

1. **Our Approach**

   We outline our approach to teaching Unity and ML-Agents.

2. **Setting Up**

   Getting ready to explore Unity ML-Agents by setting up Unity, the Python environment, and ML-Agents itself.

3. **Activity 1: Introducing Unity**

First, we'll introduce you to Unity, the game development environment that we'll be using to create simulations to perform machine learning with.

We'll spend a little time learning Unity, as a game developer would, so we're comfortable working with Unity for AI later on. There'll be no AI or ML in this section!

4. **Activity 2: Self-driving Car**

   In this activity, we'll look at training a car to drive around a track using both *reinforcement learning* and *offline imitation learning*, and *visual observations* (a camera).

5. **Activity 3: Robot Warehouse**

   Next, we'll build a little robot warehouse, with a cute little robot, and teach it to sort crates into the right corner of the warehouse. We'll use *reinforcement learning* to do this, and *vector observations*.

6. **Activity 4: Bouncer**

   For this activity, we'll use the cute little robot from the robot warehouse, and make him jump to collec treats. We'll look at both *reinforcement learning* and *online imitation learning*, *vector observations*, and *on-demand decisions*.

7. **Activity 5: Treat Collector**

   Finally, we'll train an agent to collect good treats, and try to avoid bad treats. We'll use *reinforcement learning*, and *vector observations*.

8. **Next Steps**

   We'll conclude with some advice on where to take your learning next, some suggested activities, and some challenges for you to complete in your own time.

Approach

This tutorial has the following goals:

- Teach the very basics of the Unity game engine
- Explore a scene setup in Unity for both training and use of a ML model
- Show how to train a model with TensorFlow (and Docker) using the Unity scene
- Discuss the use of the trained model and potential applications
- Show you how to train AI agents in complicated scenarios and make the real world better by leveraging the virtual

NOTE	We consider learning Unity to be as important as learning ML-Agents.

This is exercise in **applied** artificial intelligence and machine learning. The focus of this session is to make you comfortable using Unity and ML-Agents, so you can go forth and use your software

engineering skills and machine learning skills, together with your Unity and ML-Agents knowledge, to build interesting and useful simulations.

If you haven't done much AI or ML in the past, don't worry! We'll explain everything you need, and it should be clear what you need to learn next to explore and understand the ML side more.

Today is about building fun things in Unity, with ML-Agents!

Setting Up

You need three major things to work with Unity ML-Agents:

1. Unity
2. Python and ML-Agents (and the associated Python dependencies)
3. A Unity project that's set up to use ML-Agents

In this section, we'll get those things installed!

WARNING	Everything we're working with will work on Windows or macOS, and most of it will probably work with Linux. We're not Linux experts, but we'll try our best to help you out with any problems you encounter if you're game enough to try this out on Linux.

Installing Unity

Installing Unity is the easiest bit. We recommend downloading and using the official Unity Hub to manage your installs of Unity:

- Download the Unity Hub for Windows or macOS

The Unity Hub allows you to manage multiple installs of different versions of Unity, and lets you select which version of Unity you open and create projects with.

WARNING	We've pinned the version of Unity being used for this tutorial to **Unity 2019.1.8f1**. Everything will probably work with a newer version, but we make no guarantees. It's easier for everyone if you stick to the verison we suggest!

If you don't want to use the Unity Hub, you can download different versions of Unity for your platform manually:

- Download a specific version of Unity for Windows or macOS

We strongly recommend that you use the Unity Hub to manage your Unity installs, as it's the easiest way to stick to a specific version of Windows, and manage your installs. It really makes things easier.

If you like using command line tools, you can also try the U3d tool to download and manage Unity install's from the terminal.

When you're installing Unity, you might be asked which Unity Modules you want to install as well. We recommend that you install the "Build Support" module for the platform you're running Unity on: for example, if you're installed Unity on macOS, then also install the "Mac Build Support (IL2CPP)" module. We also recommend that you install the "Documentation" module (for, hopefully, obvious reasons!)

Once you've got Unity installed, move to to install the Unity Machine Learning Agents Toolkit.

Installing Python and ML-Agents

1. Make a new directory to keep everything in for this tutorial. Ours is called *UnityML_Workshop_Environment*.
2. Create a new Anaconda environment using Python 3.6. You can do this on the terminal with the following command:

   `conda create -n UnityML python=3.6` Note that you can replace the name of the Anaconda Environment with something of your choosing. Ours is called *UnityML*. Anaconda will take a moment to create an environment for you, as shown in Our Anaconda environment being created.

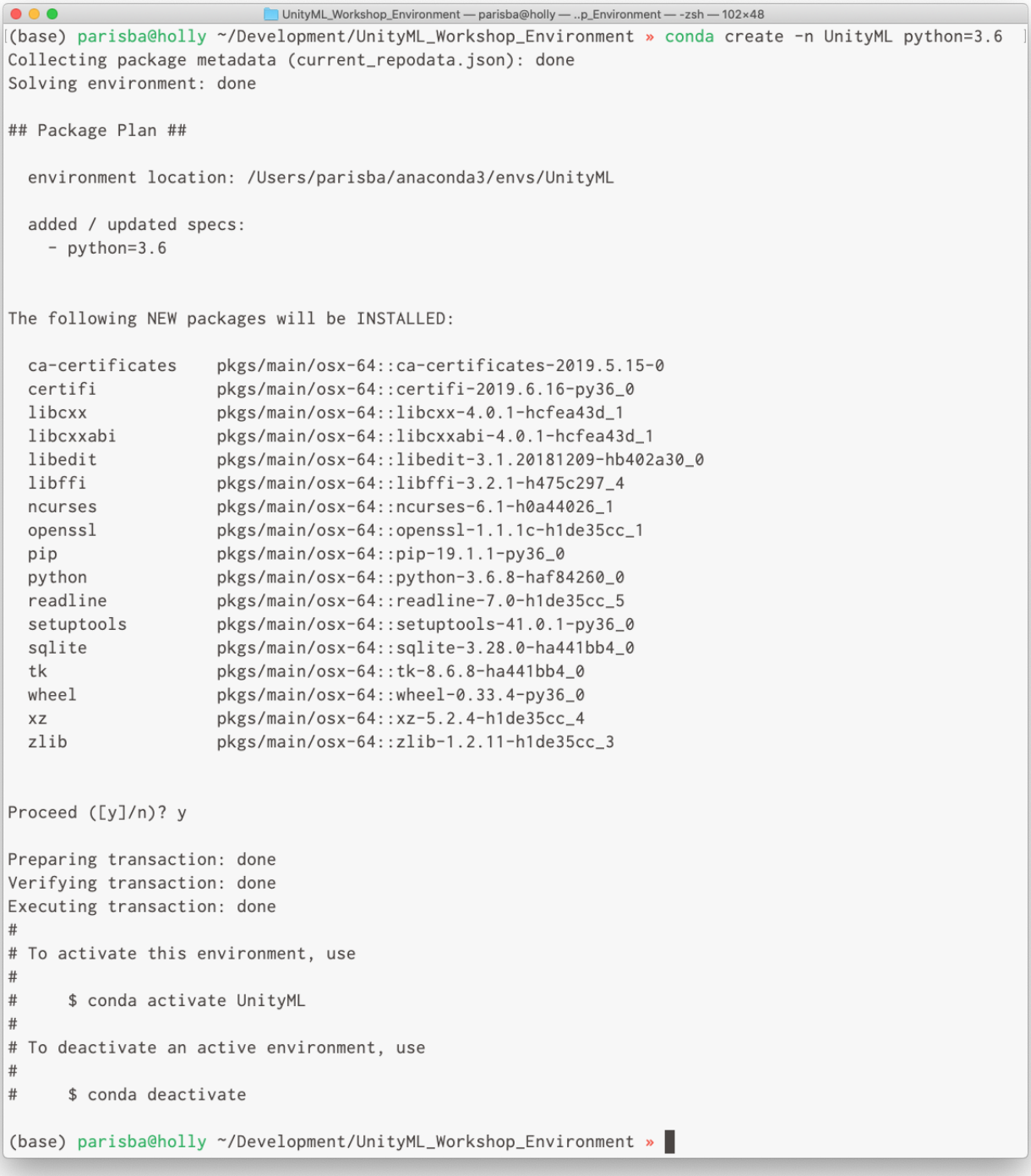

Figure 1. Our Anaconda environment being created

3. Once the Anaconda environment has been created, activate is using the following command:

   ```
   conda activate UnityML
   ```

4. Install TensorFlow 1.7.1 using pip, using the following command:

   ```
   pip install tensorflow==1.7.1
   ```

5. And finally (almost) install ML-Agents, using the following command:

   ```
   pip install mlagents==0.8.2
   ```

6. Once this is done, you can check that ML-Agents is installed successfully using the following command:

   `mlagents-learn --help` You should see an output including an ASCII Unity logo, as shown in Checking the ML-Agents is successfully installed.

Figure 2. Checking the ML-Agents is successfully installed

Acquiring a Unity Project

At this point, you could manually create a project, set it up to use Unity ML-Agents, and then go get the bits of ML-Agents you need from GitHub, put them in the project, and start making ML environments.

However, that's a bit of a chore, and we have a better solution! We've build a repository that contains everything you need for this session, and you can clone that instead:

1. Clone our GitHub repository to your machine:

   `git clone https://github.com/parisba/ORM-AI-SJC-2019-Unity-ML-Agents.git`

   Inside the cloned repository, you'll find a copy of this running sheet (hello!) and a folder called "Projects". This is the folder we want to spend the majority of our time in.

2. Use your command line to change directory into this folder, and then activate your UnityML Anaconda Environment.

This *ml-agents* directory contains the source code for ML-Agents, a whole of lot useful configuration files, as well starting point Unity projects for you to use. It's based on the default Unity project provided by Unity, but we've also added our examples for this session to it.

You can find Unity's version of an ML-Agents repository on GitHub:

- https://github.com/Unity-Technologies/ml-agents

WARNING	We've pinned the version of ML-Agents being used for this tutorial to **ML-Agents Beta 0.8.2**. Everything will probably work with a newer version, but we make no guarantees. Using the same version of ML-Agents as us is probably more important than using the same version of Unity.

To download the version of ML-Agents we're using, but without our additions to the Unity project, grab the following (we don't recommend doing this if you want to follow along, **use our repository instead**):

- https://github.com/Unity-Technologies/ml-agents/releases/tag/0.8.2

NOTE	You can also clone the git repository, but we're focusing on **ML-Agents Beta 0.8.2**, and things might be a little different if you track the repository.

Everything is ready!

Activity 1: Introducing Unity

We're not here to learn game development with Unity! We're here to explore machine learning! But… to do that, we need to understand how to use Unity. We cannot emphasise this enough! **Being comfortable with Unity is as important as being comfortable with ML-Agents!**

NOTE	**In this activity we're going to:** • build a little Unity scene • show how a Unity scene includes game objects, components, and how this makes different behaviours possible • show how scripts interact with the Unity scene

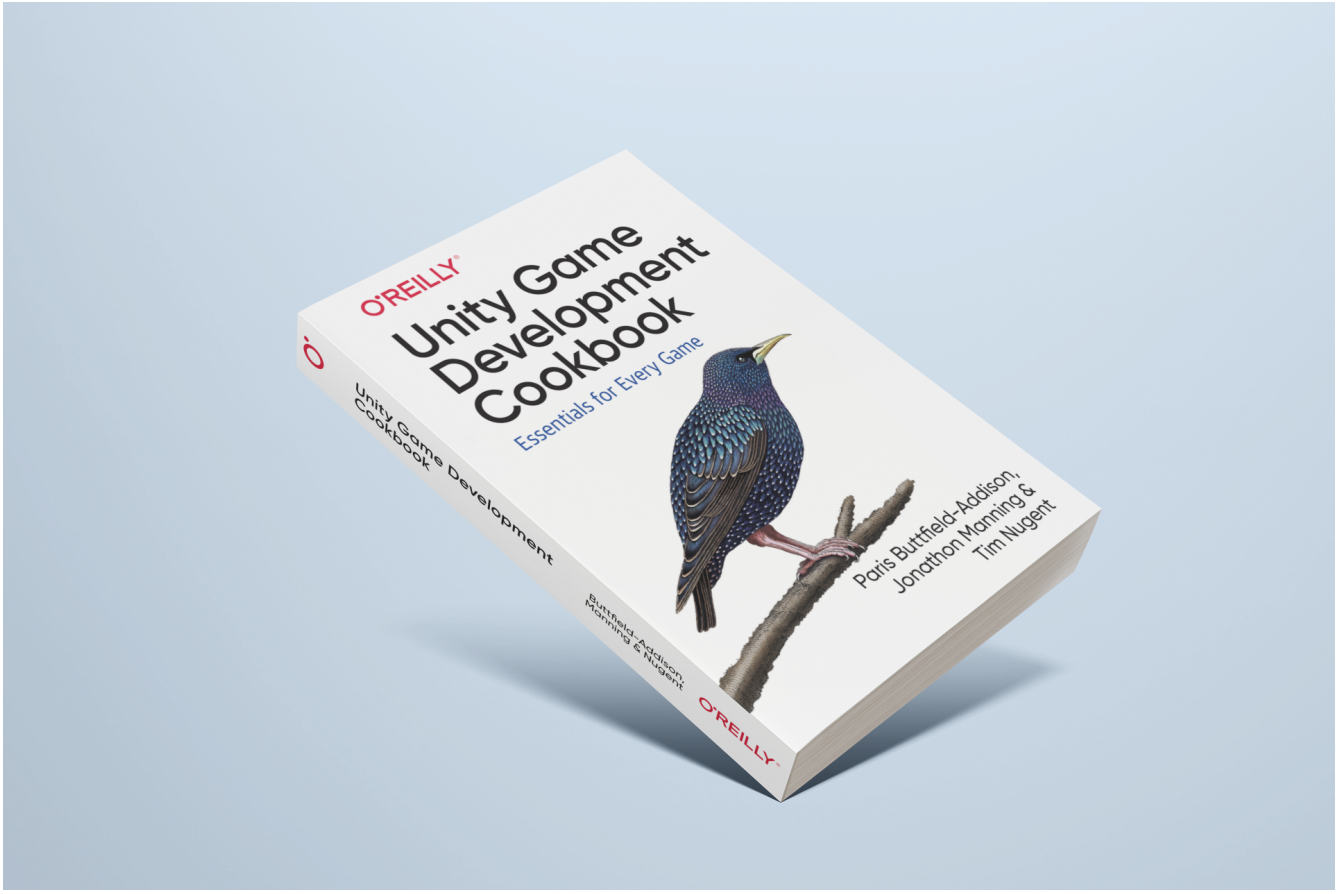

Figure 3. Our Unity Game Development Cookbook

TIP	If you would like to learn Unity, check out our current books on Unity! *Mobile Game Development with Unity* and *Unity Game Development Cookbook* (shown in the image below)! We're very proud of our books. Here ends the shameless plug.

Before we start, make sure you have **Unity 2019.2.4f1** installed.

TIP	It's not the end of the world if you're running a slightly different version of Unity, just try to be as close to our version as possible.

Creating a bouncing ball

Let's learn to find our way around Unity by building a simple 3D environment in Unity. This environment won't have any machine learning, or even be connected with the ML-Agents Toolkit. Let's get started:

1. Open the *Unity Hub* application, and use the *Add* button on the *Projects* screen to open our provided Unity project. The folder you want to open is *ORM-AI-SJC-2019-Unity-ML-Agents/ML-Agents/ml-agents/UnitySDK*. This folder is our Unity project.
2. Create a new Scene in the "Activity1-UnityBasics" folder. Open the scene.

2. Your new Unity scene will open, as shown in Your empty Unity project. Unity's default view is made up of some standard components:
    - The *Scene* and *Game* views in the middle. The *Scene* is editable, and the *Game* shows what environment looks like when running.

- The *Hierarchy* on the left, which shows the contents of the current *Scene*.
- The *Console* on the bottom left, which shows console output.
- The *Project* view in the center bottom, which shows the contents of the project (this maps to the) contents of the *Assets* directory in the project's overall directory.
- The *Inspector* on the right, which shows the parameters and components of the currently selected object (selected in any of the *Hierarchy*, *Scene*, or *Project* views).

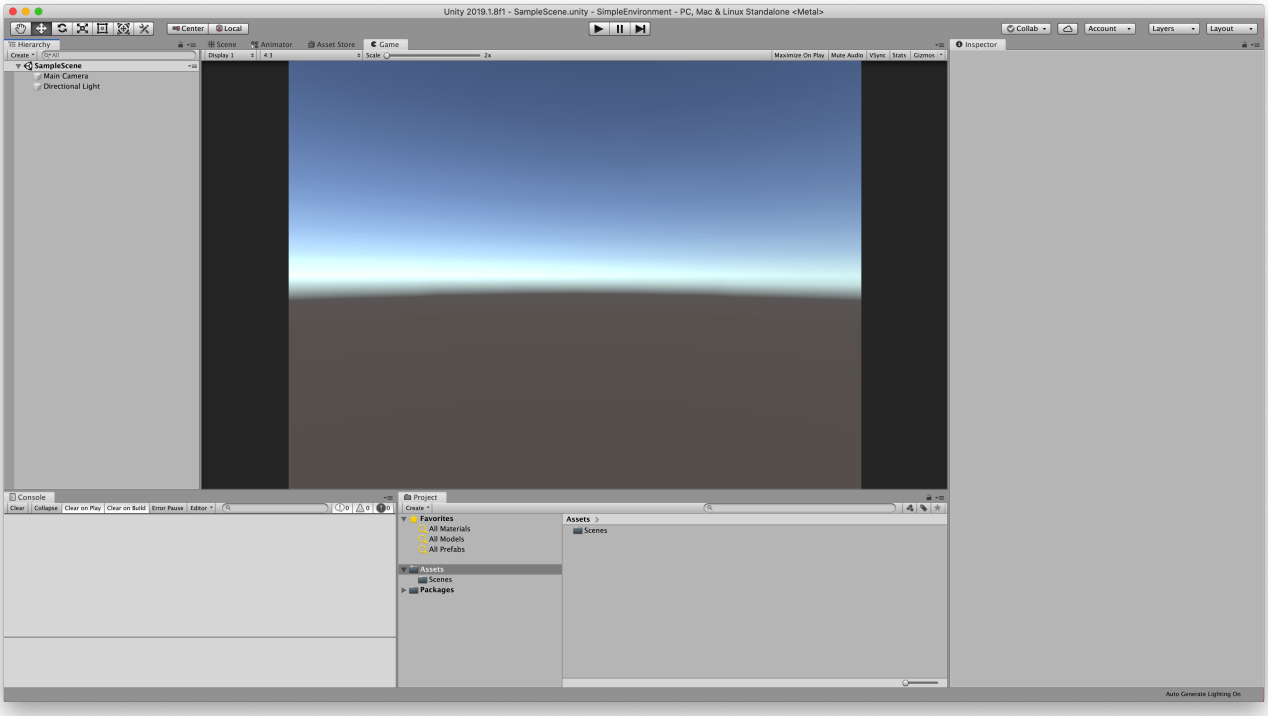

Figure 4. Your empty Unity project

3. Add a sphere to the scene using the GameObject → 3D Object → Sphere menu entry (you can also right-click on the *Hierarchy*). Make sure the new sphere is selected in the *Hierarchy*, then use the *Inspector* to rename it to "Bouncy Ball", as shown in Renaming the sphere.

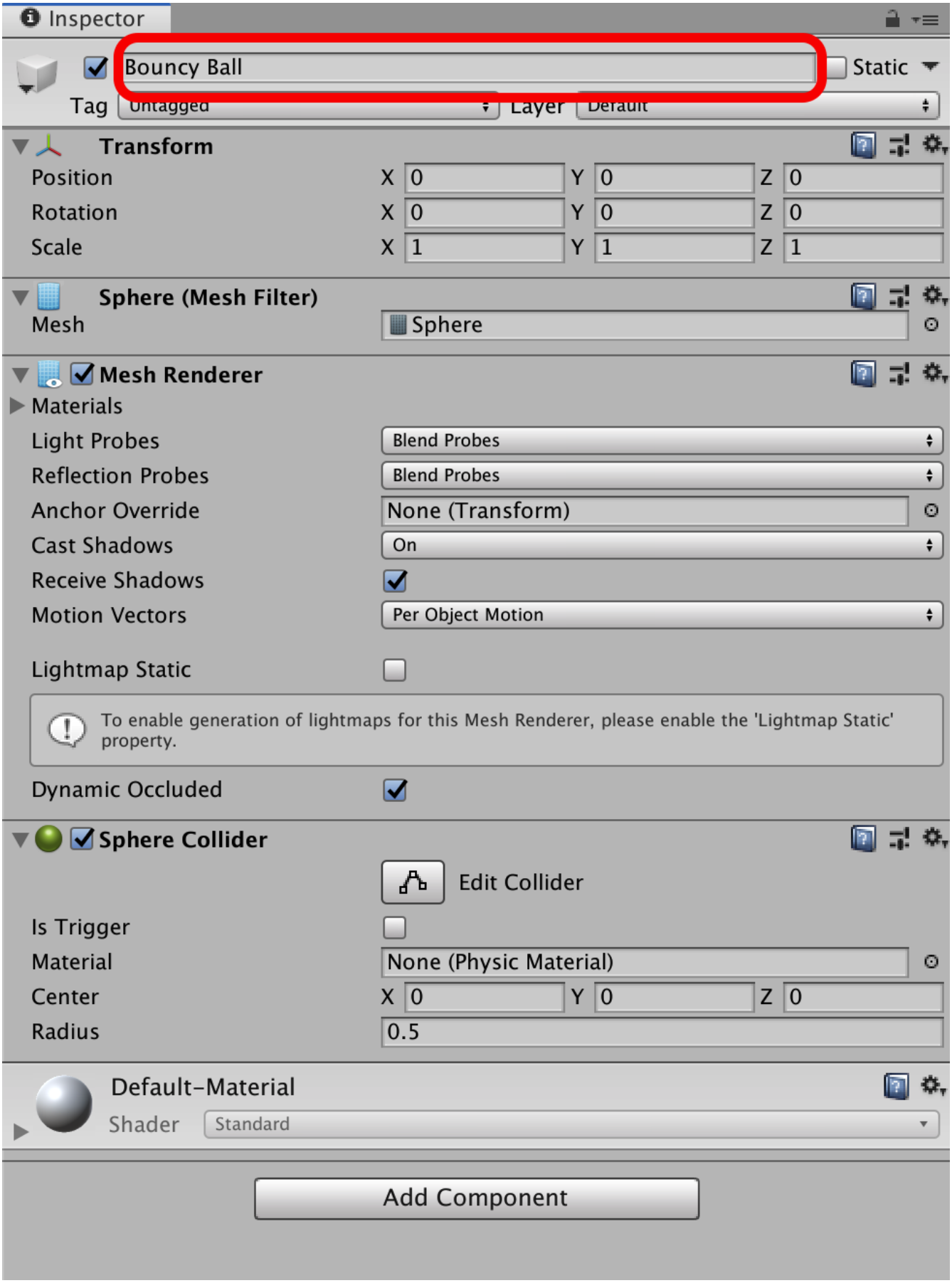

Figure 5. Renaming the sphere

4. Save the scene (it's already saved as SampleScene, so just make sure it's saved), and then play it by clicking the *Play Button*. Notice how absolutely nothing happens (other than Unity switching from the *Scene* view to the *Game* view). Click the *Play Button* again to stop playing.

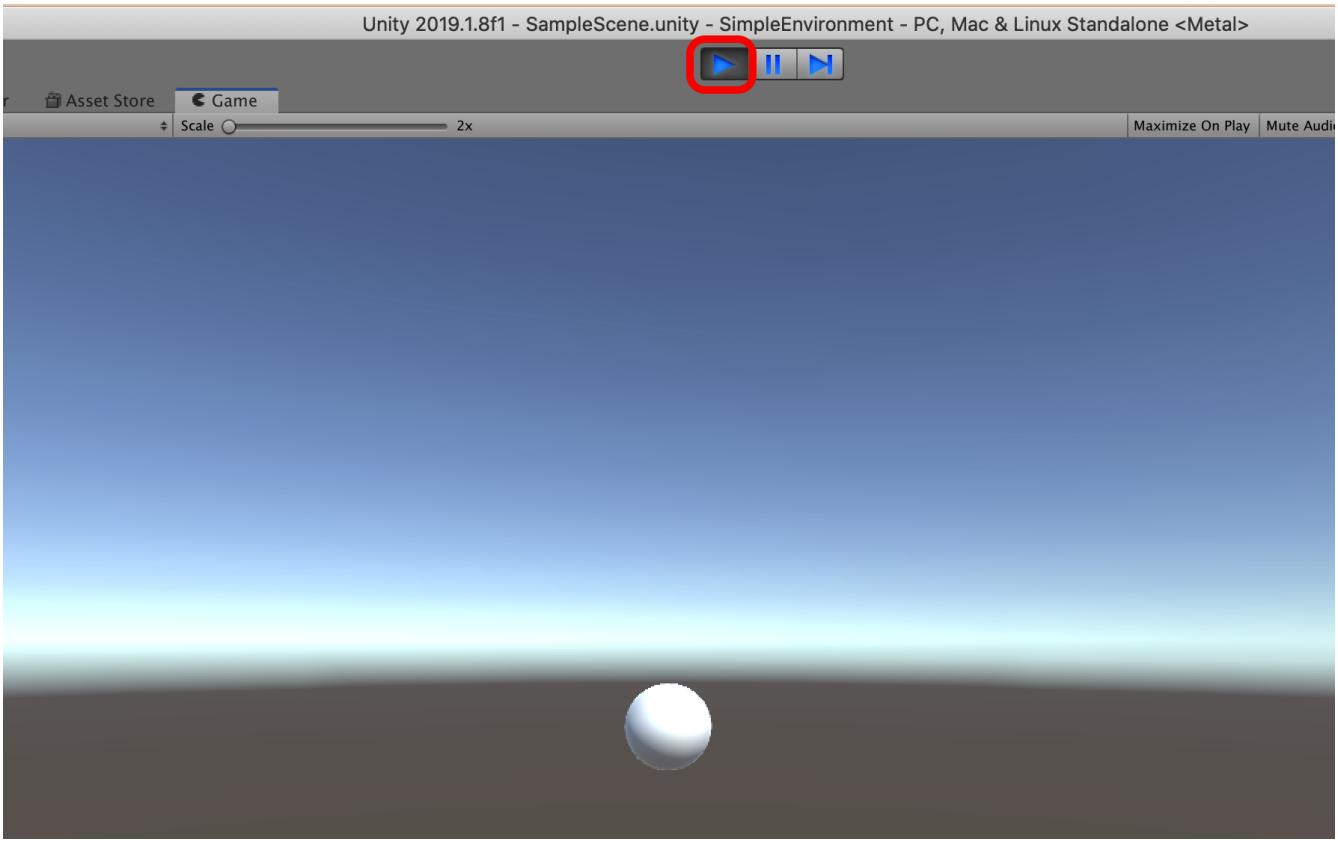

Figure 6. Playing the scene

5. To make things more interesting, we're going to make the sphere, which we've named bouncy ball, live up to its name. To bounce, we need something to bounce off of! We need a floor: add a cube using the GameObject → 3D Object → Cube menu.

Figure 7. The Unity tools

TIP You can also switch between the tools using your keyboard: Q for the *Hand Tool*, W for the *Move Tool*, E for the *Rotate Tool*, R for the *Scale Tool*, as so on.

6. Select the newly created cube, rename it to "Floor", then from the tools selector (shown in The Unity tools) use the *Scale Tool* (4th from the left) to stretch and flatten it, and the *Move Tool* to move it below the sphere.

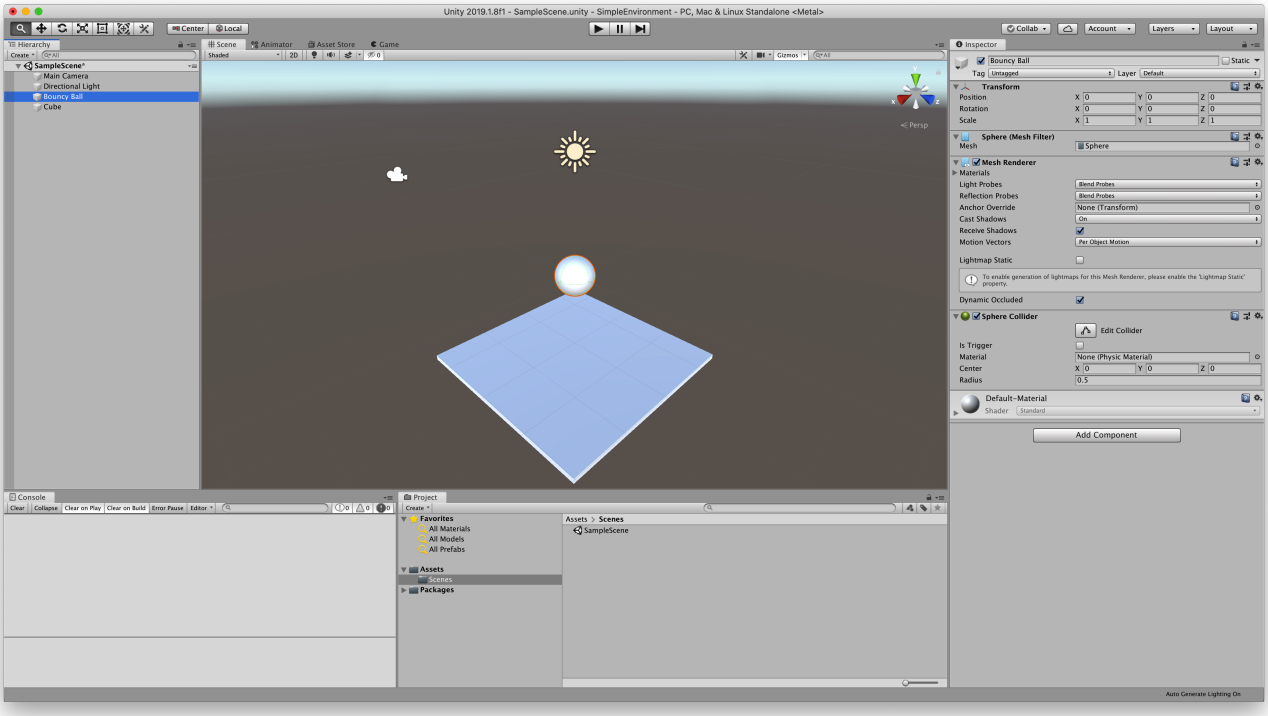

Figure 8. The scene coming together

7. Your scene should look something like The scene coming together. We need to add a *Rigidbody Component* to the ball. Select the ball, and in the *Inspector* click *Add Component* and start typing "Rigidbody", as shown in Adding a Rigidbody Component.

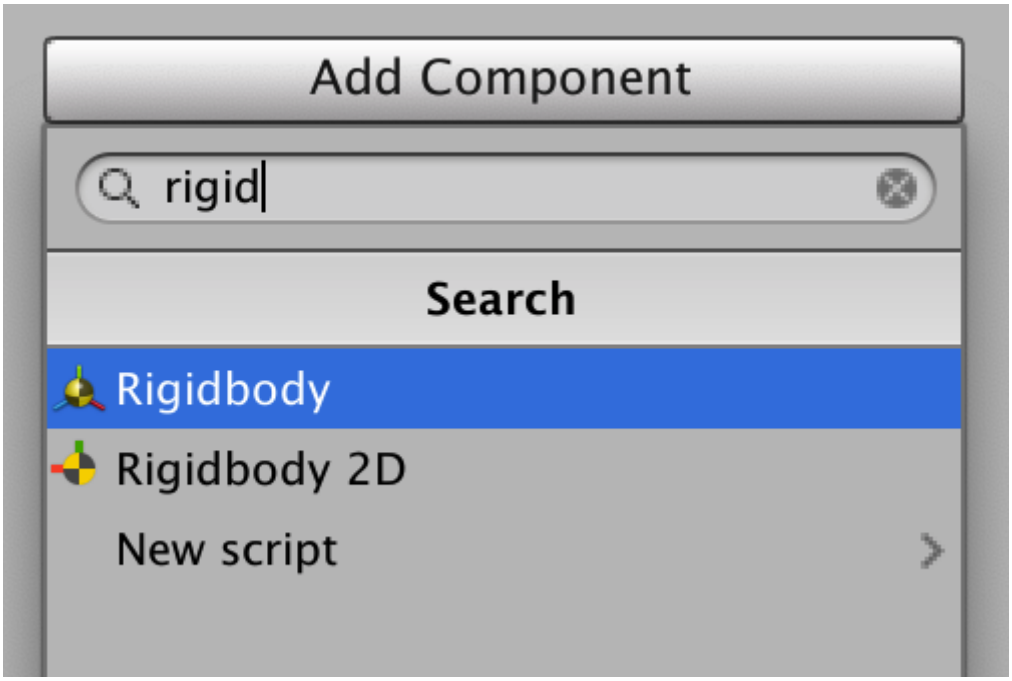

Figure 9. Adding a Rigidbody Component

8. Make sure the *Use Gravity* checkbox is checked in the newly added *Rigidbody Component* on the ball, as shown in The new Rigidbody Component.

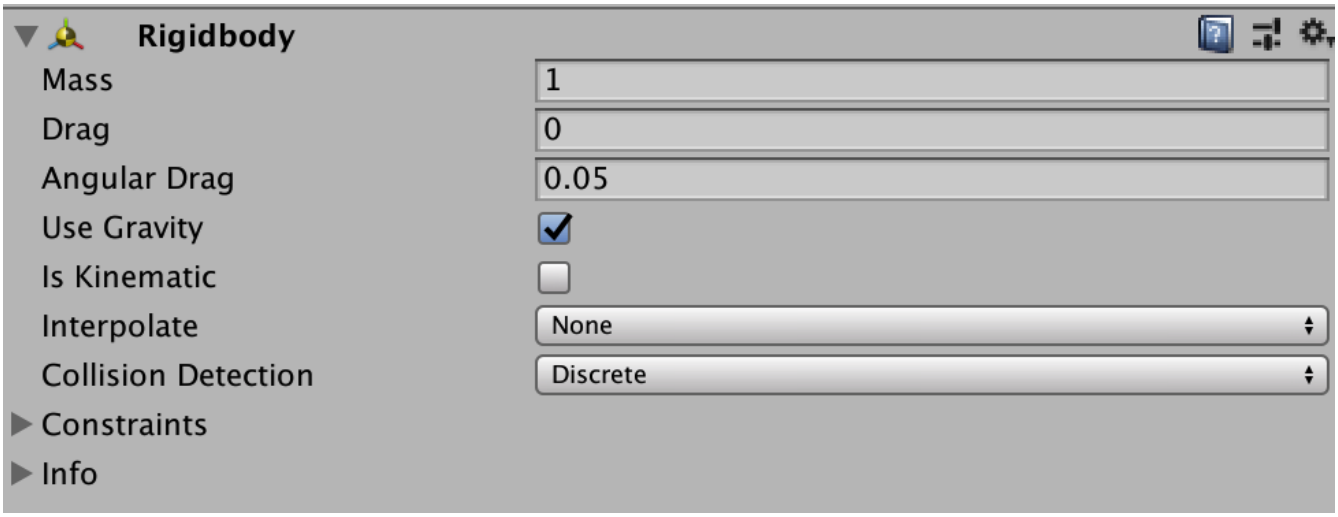

Figure 10. The new Rigidbody Component

1. Play the scene! The ball will fall to the floor and... stop. To make it bounce we need to give it some physical properties that lead to bouncing. In the *Project* view (center bottom), select the root "Assets" folder, and then right-click and select Create → Physic Material, as shown in <<fig:creatingphysicmaterial>. Name the new material "Bouncy Material".

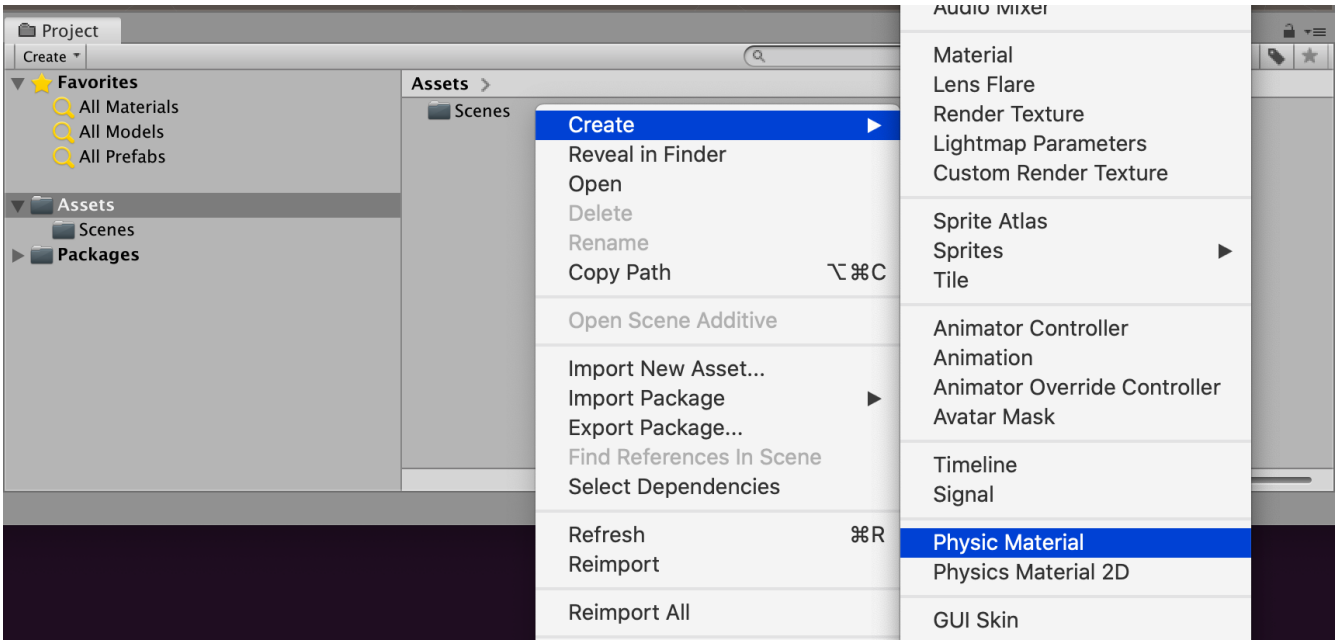

Figure 11. Creating a new Physic Material

10. Select the "Bouncy Material" and use the *Inspector* to set the Bounciness to 1, and Bounce Combine to Maximum.

11. To make the ball bounce, we need to apply the new material to it: select the ball and then either drag the "Bouncy Material" onto it in the *Hierarchy*, or onto the "Material" slot in its "Sphere Collider" component in the *Inspector*, as shown in Setting the material.

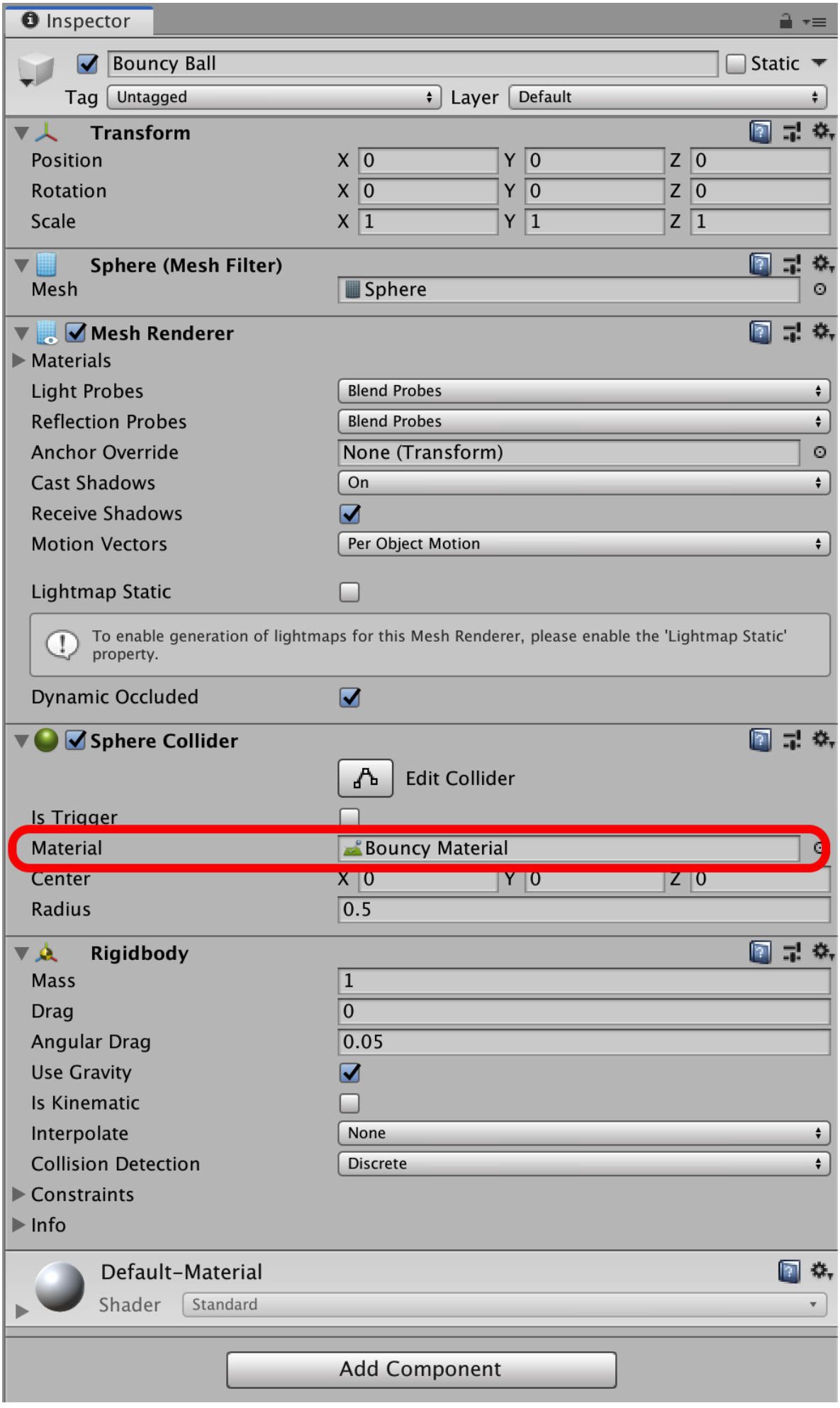
Inspector
Bouncy Ball
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Sphere (Mesh Filter)
Mesh
Sphere
Mesh Renderer
Materials
Light Probes
Blend Probes
Reflection Probes
Blend Probes
Anchor Override
None (Transform)
Cast Shadows
On
Receive Shadows
Motion Vectors
Per Object Motion
Lightmap Static
To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.
Dynamic Occluded
Sphere Collider
Edit Collider
Is Trigger
Material
Bouncy Material
Center
Radius
0.5
Rigidbody
Mass
1
Drag
0
Angular Drag
0.05
Use Gravity
Is Kinematic
Interpolate
None
Collision Detection
Discrete
Constraints
Info
Default-Material
Shader
Standard
Add Component

Figure 12. Setting the material

12. Play the scene! The ball will now bounce. Isn't that exciting? Don't forget to stop playing when you're done watching the ball bounce. And don't forget to save the scene.

Scripting the bouncing ball

Let's look at basic Unity scripting now. Remember the console? We want it to print something everytime something hits the floor.

1. In the *Project* view (center bottom), select the root "Assets" folder, and then right-click and select Create → C# Script. Name the new script "CollisionDetection". Open the script and replace its contents with the following (leave the imports where they are):

```
public class CollisionDetection : MonoBehaviour
{
    public bool printDebug = false;

    void OnCollisionEnter(Collision c) {
        if(printDebug) {
            Debug.Log(c.gameObject.name + " hit me!");
        }
    }

}
```

2. Drag the script from the *Project* view onto the *Floor* object in the *Hierarchy*, as shown in The CollisionDetection script attached to our floor object.

WARNING The file name of the script must match the class name.

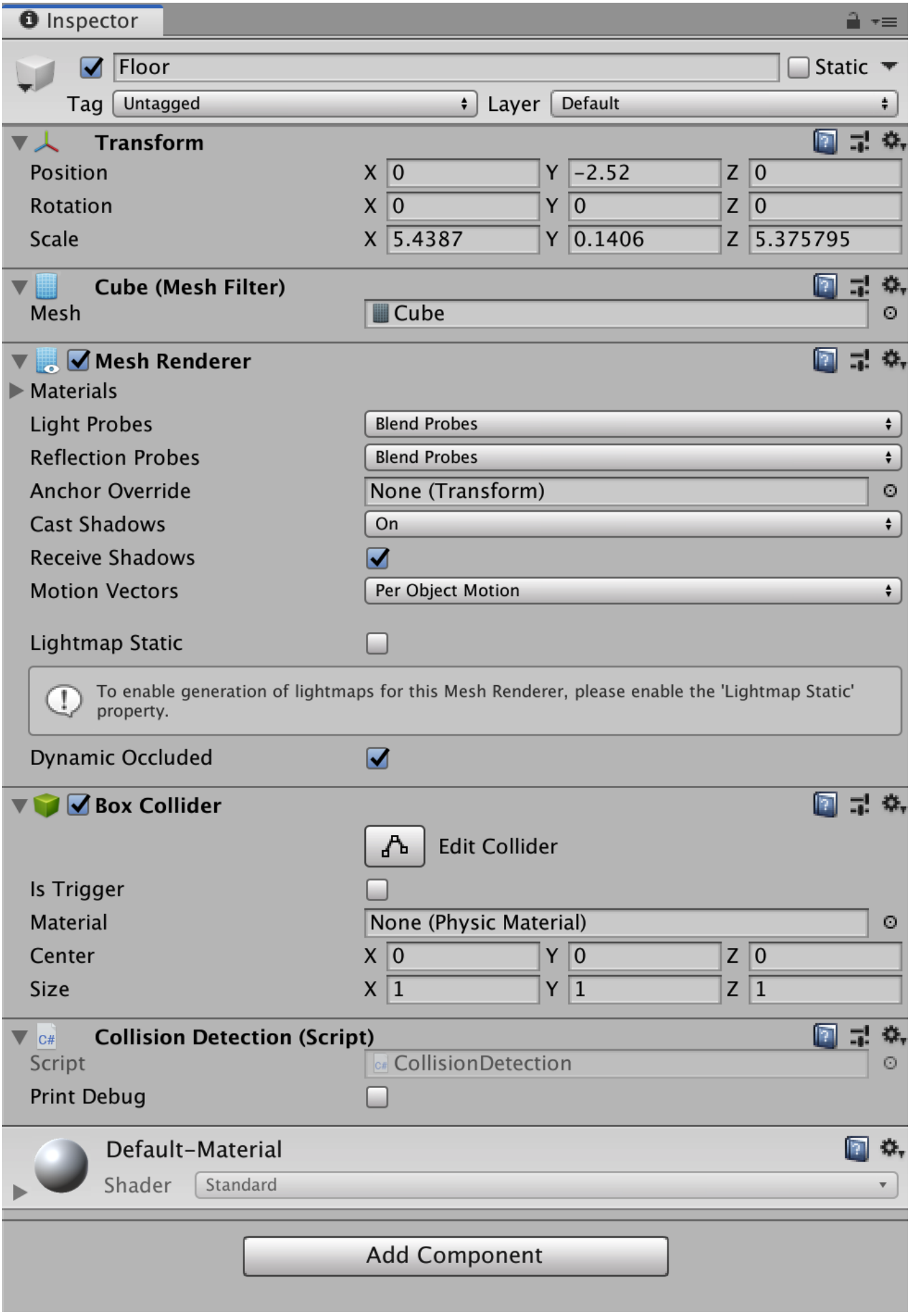

Figure 13. The CollisionDetection script attached to our floor object

1. Play the game. While the game is playing, select the floor in the *Hierarchy* and check the "Print Debug" checkbox in the new script's entry in the floor's *Inspector*. Now, every time the something (in this case, the ball) collides with the floor it will print out a message, as shown in Console output.

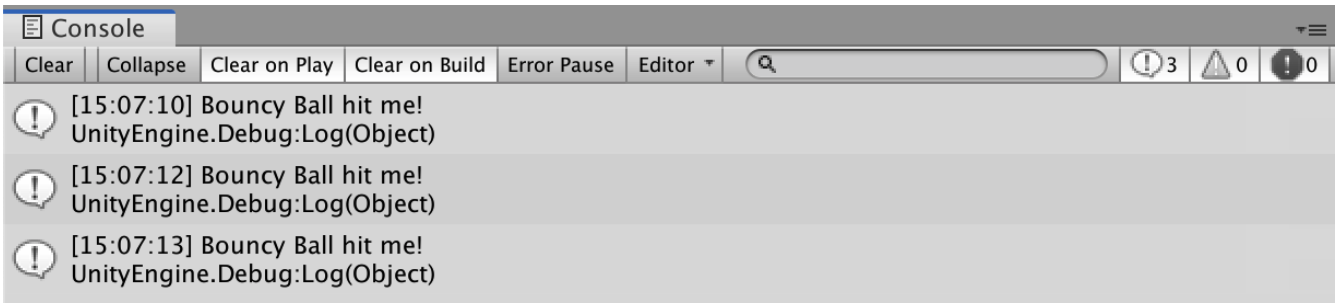

Figure 14. Console output

There's a lot more (a whole lot more) than you could learn about Unity, but that's everything we think you need to get into Unity for ML. We'll cover the rest as we go, or you can follow up and learn more about general Unity development in your own time!

Extra Credit

For fun, and if you have time, you might want to consider how you'd do the following:

- add a camera to the ball, pointed at the floor, so we can see its perspective as it bounces. Make this camera the primary camera.
- add more balls, set them at different heights, and name them differently, so we can watch them bounce
- make a cube, and see if you can make it bounce

Activity 2: Self-driving car

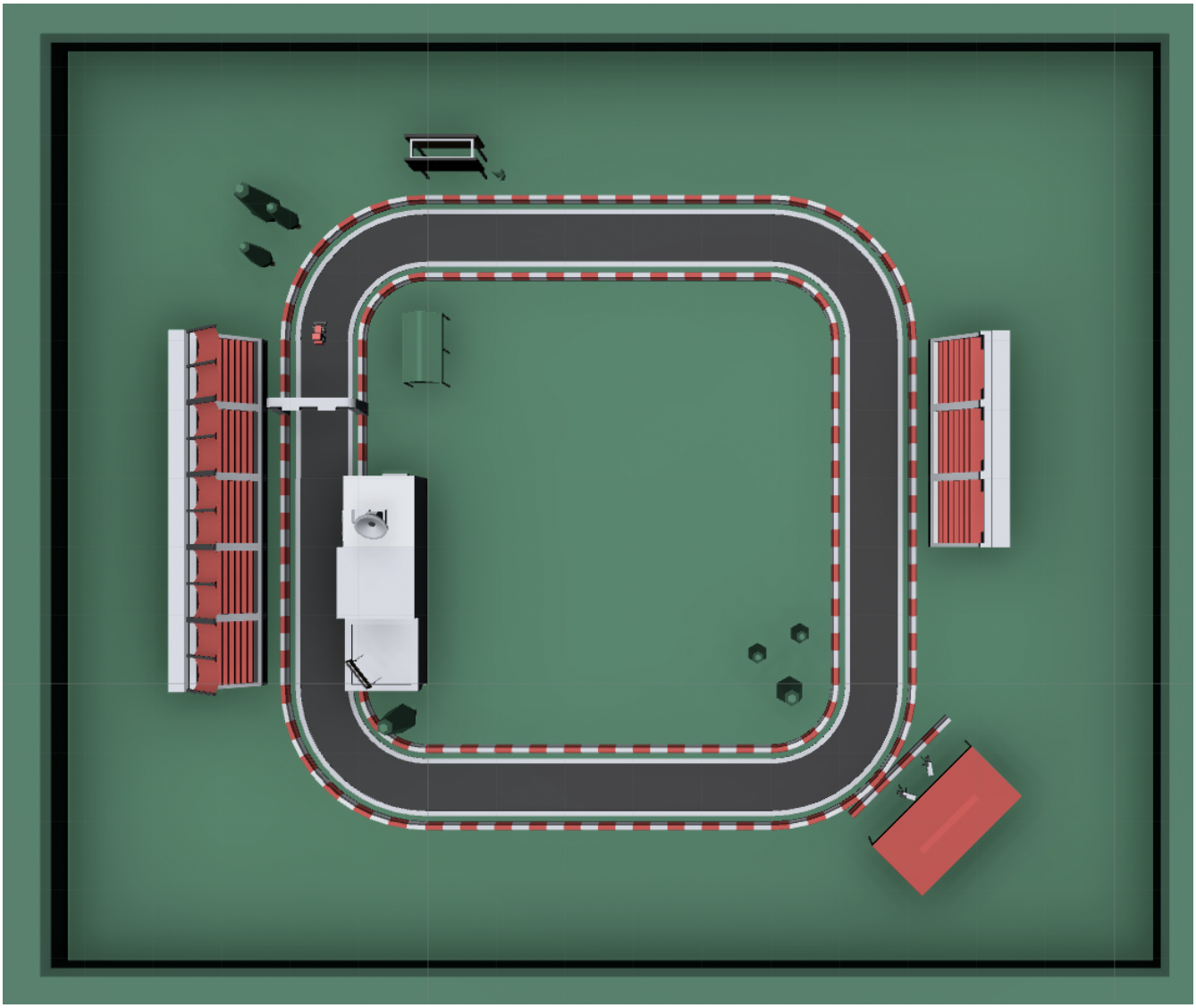

Figure 15. The track for our car

- **Environment** ---- The Track
- **Agent** ---- The Car
- **Policy** ---- Convolutional Neural Network (as we're dealing with Images)

We're going to take a brand new, empty brain and let it start learning from scratch.

TIP	We could also use some form of supervised learning, like imitation learning, and train that, then use reinforcement learning to improve it.

We're going to start with something that's conceptually pretty straightforward: we want to build a simulated car that can autonomously drive around a track.

- The **Environment** will be a race track.
- The **Agent** will be a car.
- The **Goal** will be the car autonomously driving around the track.
- The **Actions** available will be steering left and right. The car's throttle will happen automatically.

To make this happen, we need to answer some questions. Those questions are:

- **Question 1**: What sort of learning to do we want to use?
- **Question 2**: What Observations will the Agent have about the Environment?

To answer **Question 1**, we'll take a look at two specific approaches: Reinforcement Learning, and Imitation Learning. We'll look at Reinforcement Learning in passing, showing off how it works, because it can take quite a long time to train. We'll look at Imitation Learning in more detail, because we can get things working quicker.

To answer **Question 2**, we need to think about the knowledge the Agent needs in order to be able to drive the track. At the simplest level, it needs to know the following things:

- whether it has left the road
- where it is on the road, in relation to the sides of the road

We can give it this knowledge in a variety of ways. The first, perhaps most obvious way if you approach this simulation from the perspective of a game developer, is to give it a whole bunch of raycasts ---- essentially perfect laser measuring tools ---- to see how far away it is from things, and send those raycasts out from a variety of directions on the car.

The second, and perhaps most obvious way if you approach this from the perspective of a computer person or generally observant person, is to use cameras.

We're going to use visual observations (which means cameras); we'll be using vector observations, which is the term for the other kind of observations, in the other activities.

Setting up the Car to Drive

Some notes on the layout of the car:

`carController` and `rigidBody` store references to bits of the car. `lapTime` will be used to store the current lap time, `bestLapTime` will store the best lap time of the current run (it's not persisting anything anywhere or anything).

We will use `isCollided` by setting it to true when the car collides with something that it shouldn't (as far as what we want it to learn goes). `startLinePassed` will be used as a flag to figure out if we've lapped the course.

NOTE

`resetPoint` and `trackWaypoints` are `public`, which as you may remember means they get exposed in the *Inspector*. We'll use `resetPoint` to store a `Transform` representing the reset point for the car, and we'll use `trackWaypoints` to store an array of `Transform`s, representing a path around the track. We'll use those to reset the car back to nearby where it crashed (which, in this context, is colliding with something) by picking the closest one when a crash happens.

`agentIsTraining` will be used (and exposed in the *Inspector*) to change the car's behaviour a little bit when we're training, vs when we're not. We could this by asking the ML-Agents system what its brain settings are, but we're doing it this way to make it clearer what's going on.

1. Expand "Activity2-SelfDrivingCar" in the *Project* pane of Unity.
2. Open the "Scripts" folder in the project, and find CarAgent.cs.
3. Inside CarAgent.cs, find the `AgentAction()` function.
4. Add the following code:

```
public override void AgentAction(float[] vectorAction, string textAction) {
            float h = vectorAction[0];
            carController.Move(h, 1, 0, 0);
}
```

5. Drive the car! What problems do we see here?
6. We need to give the car some awareness that it's collided with something. Add the following code below the code we added earlier, inside `AgentAction()`:

```
// Once the actions are done, we need to check:
if(isCollided) {
    // we hit something
    Done();
} else {
    // we did not hit something
}
```

7. Drive the car! Now, if we hit the barriers, we'll get reset. Neat, right?

Adding Rewards

1. Inside our `AgentAction()` function, we need to add some rewards. Add the following penalty "reward" before we call `Done()` inside the collision check:

```
AddReward(-1.0f);
```

2. And add the following reward for driving properly if we did not hit something:

```
AddReward(0.05f);
```

3. The check should now look like this:

```
// Once the actions are done, we need to check:
if(isCollided) {
    // we hit something
    AddReward(-1.0f); // you get a punishment, you get a punishment, we all get
punishments!
    Done();
} else {
    // we did not hit something
    AddReward(0.05f); // what a good car you are!
}
```

4. Drive the car again! Now we can collide.

The next step is training the car to drive itself.

Training the Car

We'll now look at training the car with **reinforcement learning** and **imitation learning**!

To train the car with **reinforcement learning**, you'll need a yaml file in the config directory (PROJECT/Projects/ML-Agents/ml-agents/config), named something like aiconf_config.yaml, with the following in it:

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    use_curiosity: false
    curiosity_strength: 0.01
    curiosity_enc_size: 128

Car_LearningBrain:
    max_steps: 1.0e6
    batch_size: 100
    beta: 0.001
    buffer_size: 12000
    gamma: 0.995
    lambd: 0.99
    learning_rate: 0.0003
    normalize: true
    time_horizon: 1000
```

Your learning brain will need to be named the same as the second set of parameters (in this case, "Car_LearningBrain").

TIP

Don’t forget to set the parameters of the brain and academy in Unity for training! You’ll want the control checkbox checked next to the learning brain, any existing models detached from the brain, and you probably want the speed and quality of the simulation turned down.

To train the reinforcement learning brain, the following command will be used:

```
mlagents-learn config/aiconf_config.yaml --run-id=AIConfCar1 --train
```

We recommend incrementing the run-id parameter if you change something significant. You can also resume training on a run that was used before (adding more information to the neural net), by adding `--load` to the end of the above command. That will resume the named run-id.

To train the car with **imitation learning**, you'll need a yaml file in the config directory (PROJECT/Projects/ML-Agents/ml-agents/config), named something like aiconf_imitation_config.yaml, with the following in it:

```
default:
    trainer: offline_bc
    batch_size: 64
    summary_freq: 1000
    max_steps: 5.0e4
    batches_per_epoch: 10
    use_recurrent: false
    hidden_units: 128
    learning_rate: 3.0e-4
    num_layers: 2
    sequence_length: 32
    memory_size: 256
    demo_path: ./UnitySDK/Assets/Demonstrations/PATH_TO_DEMO.demo
```

You'll need to relace the .demo file in the parameters with one you want to use, as recorded in the Unity environment. To record a demo:

- Add the "BC Recording Helper" and "Demonstration Recorder" components to your Agent and assign a name.
- Play the game with a Player Brain attached to the Agent (and the Academy).
- Drive the car!
- We recommend driving for about 100 seconds. Once you're done driving, remove the components we added a moment ago.
- You can now point the config yaml file to the .demo file you just made.

To train the imitation learning brain, the following command will be used:

```
mlagents-learn config/aiconf_imitation_config.yaml --run-id=AIConfCarIL1 --train
```

We recommend incrementing the run-id parameter if you change something significant. You can also resume training on a run that was used before (adding more information to the neural net), by adding `--load` to the end of the above command. That will resume the named run-id.

Activity 3: Building a robot warehouse

For this activity we're going to build a robot warehouse. It'll look something like Our robot warehouse, and it's going to use reinforcement learning, without any imitation of a human involved at all.

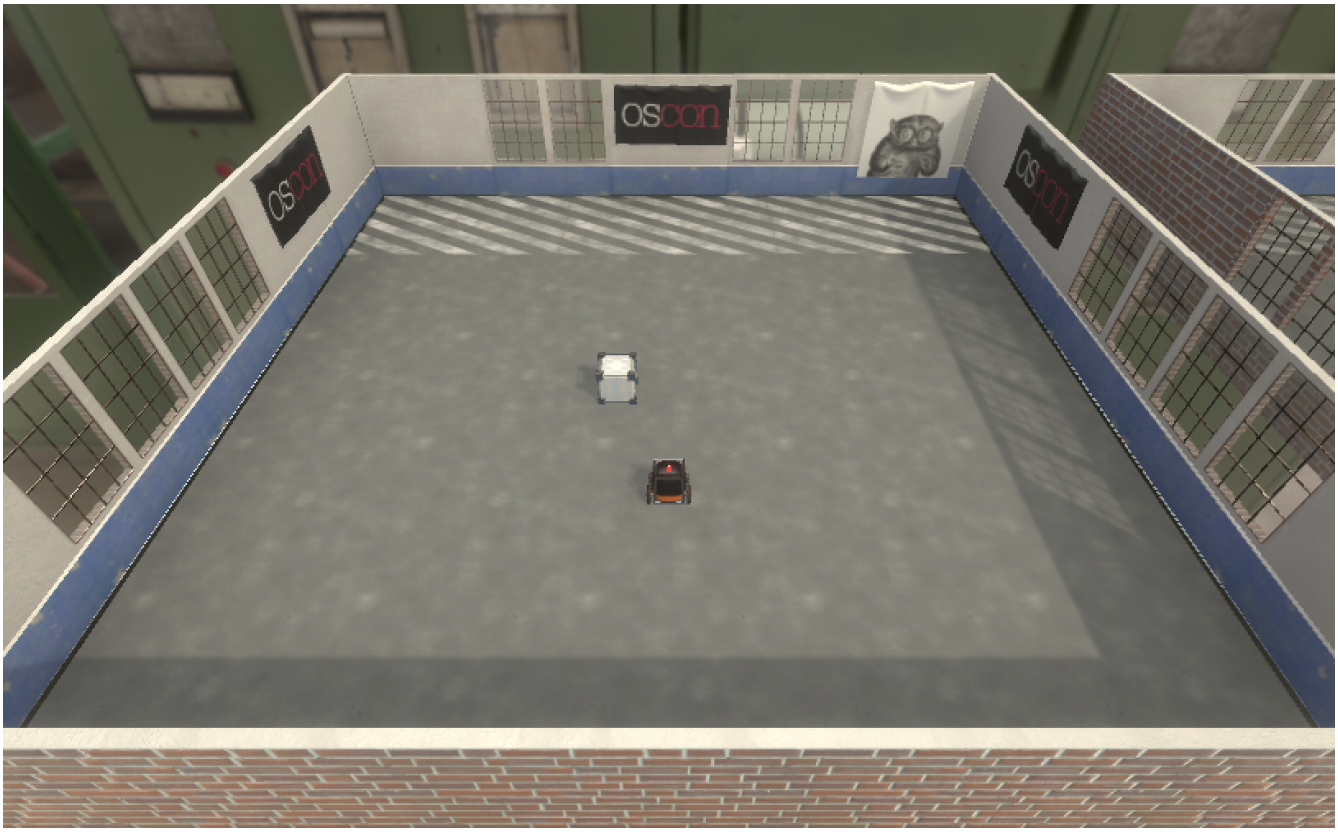

Figure 16. Our robot warehouse

The steps we'll cover in this activity are:

- Exploring the Robot Warehouse
- Playing the Robot Warehouse
- Adding Machine Learning to the Robot Warehouse
- Training the Robot

The "Robot Warehouse" Environment

The **Agent** in this environment is the little robot.

The **Goal** of the Agent is to push the cubes to the right corner of the warehouse.

The **Brain** (there is only one, linked to the Agent) has one **Vector Observation**, corresponding to its position on the spectrum of possible positions, and can take two **Discrete Vector Actions** (move left, or move right).

The **Rewards** are *+0.1* for arriving in any state that isn't optimal, and *+1.0* for arriving in an optimal state.

1. Expand the "Activity3-RobotWarehouse" folder in the *Project* pane. Open the first scene (from the "Scenes" folder).
2. Open the BeepoAgent.cs script.
3. Now we need to do some work in `CollectObservations()`:

```
   public override void CollectObservations()
    {
        var rayDistance = 12f;

        float[] rayAngles = { 0f, 45f, 90f, 135f, 180f, 110f, 70f };

        var detectableObjects = new[] { "crate", "goal", "wall" };

        AddVectorObs(rayPer.Perceive(rayDistance, rayAngles, detectableObjects, 0f,
0f));

        AddVectorObs(rayPer.Perceive(rayDistance, rayAngles, detectableObjects,
1.5f, 0f));
    }
```

4. And `AgentAction()`:

```
public override void AgentAction(float[] vectorAction, string textAction)
    {
        // Move the agent using the action.
        MoveAgent(vectorAction);

        // Penalty given each step to encourage agent to finish task quickly.
        AddReward(-1f / agentParameters.maxStep);
    }
```

Training the robot

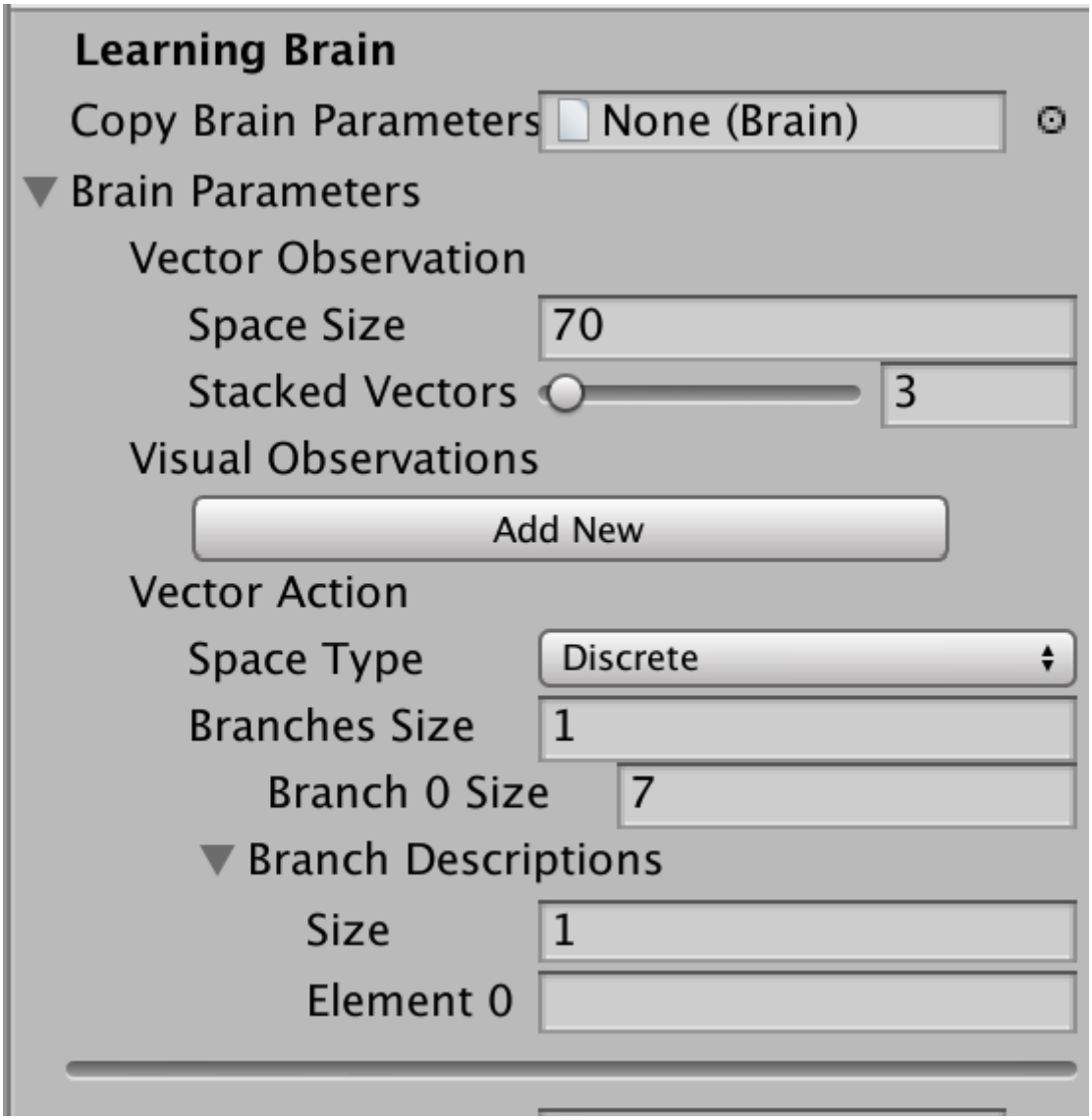

Figure 17. The warehouse brain

1. Create a new ML-Agents Learning Brain.
2. Name it "Warehouse_Learning_OneCrate", and give it a Vector Observation Space Size of 70, with 3 Stacked Vectors, no Visual Observations, Discrete Vector Actions, with 1 Vector Action Branch, with that branch being 7 large, and no Branch Descriptions, as shown in The warehouse brain.
3. Create a Conda environment for the ML-Agents system to be installed in, as per the instructions earlier.
4. Once that's done, activate the environment, and change directories into the copy of Unity's ML-Agents that you downloaded. You should now be at a stage resembling The ML-Agents directory.

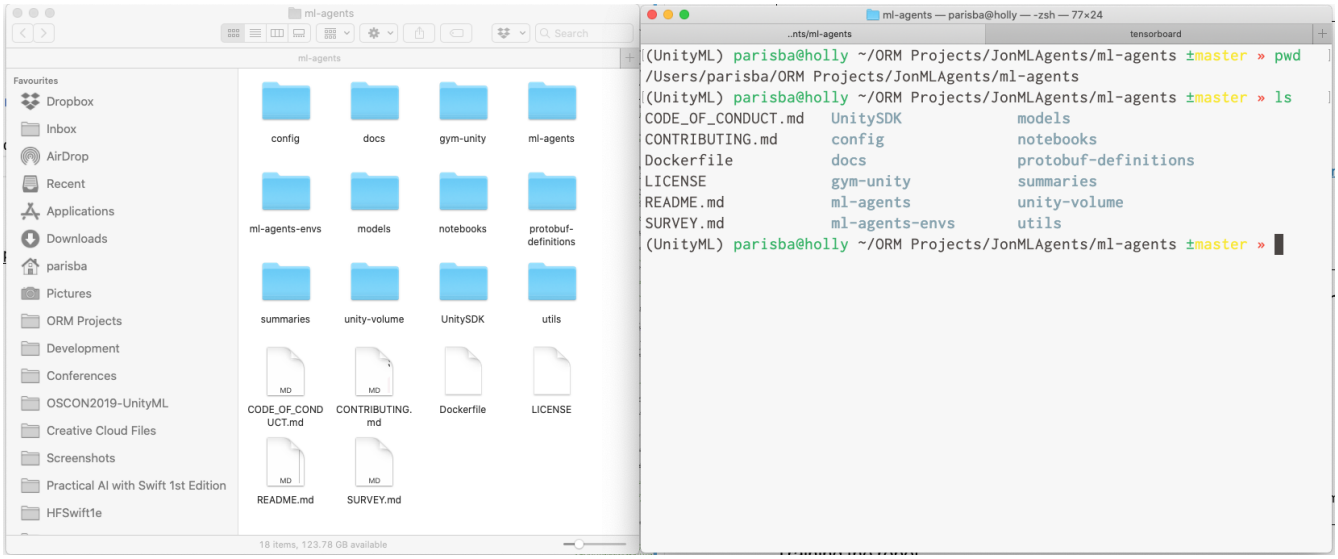

Figure 18. The ML-Agents directory

5. Create a new config file, ours is called ai_robot_trainer.yaml, and add the following:

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    use_curiosity: false
    curiosity_strength: 0.01
    curiosity_enc_size: 128
```

6. Next, below this, for our Robot Warehouse specifically, add:

```
Warehouse_Learning_OneCrate:
    max_steps: 5.0e4
    batch_size: 128
    buffer_size: 2048
    beta: 1.0e-2
    hidden_units: 256
    summary_freq: 2000
    time_horizon: 64
    num_layers: 2
```

Make sure you replace the "Warehouse_Learning_OneCrate" with the name of your Brain, if you named it differently.

7. Point the Academy to the brain you made, and tick the control box. Set the Training Configuration to make it speedy!
8. To start training, issue the following command:

   ```
   mlagents-learn config/ai_robot_trainer.yaml --run-id=Warehouse1 --train
   ```

   Make sure you increment the number of the run-ID, so we can keep track of what we're doing. When you execute this, you'll be asked to press play in Unity.

9. Run the training:

   ```
   mlagents-learn config/ai_robot_trainer.yaml --run-id=Warehouse1 --train
   ```

10. Move the trained .nn file into the project, turn off control in the Academy, and put the .nn file into the brain. Play!

Extra Credit

- Look at the four crate warehouse we supplied. Run it with the brain we made. Think about how you might improve it.
- Implement visual observations instead of vector observations on either the one crate or four crate warehouse.
- Implement imitation learning.

Activity 4: Bouncer

In this activity, we're going to take the warehouse buggy, "Beepo", and give him some treats. The only problem is the treats are up high in the air, and Beepo will need to bounce and jump to get them!

To do this, we're going to use reinforcement learning, and some vector observations.

Next, train the agent!

1. Add the following to a config yaml file:

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    use_curiosity: false
    curiosity_strength: 0.01
    curiosity_enc_size: 128

BeepoBounceLearning:
    normalize: true
    max_steps: 5.0e5
    num_layers: 2
    hidden_units: 64
```

2. And add a learning brain named BeepoBounceLearning with a space size of 12, 3 stacked vectors, continous Vecor Actions of 3 space size.
3. Turn on control on the Academy.
4. Run training:

   ```
   mlagents-learn config/bounce_trainer_config.yaml --run-id=AIBouncer1 --train
   ```

5. Copy the trained model in! Attach it to the brain, and see how you go!

Activity 5: Treat Collector

This one comes pre-made! We're just going to discuss it!

Next Steps

Go further! Here's what we recommend trying next:

- investigate Unity's curriculum learning, and try and build a curriculum

- build a chameleon (it can be a cube) that can learn to change colour based on the environment it's sitting on
- build a car that drives using ray perception, instead of a camera

Problem Solving Notes

Common Problems:

- Not connecting the brains right for training and/or inference:
  - they need an Academy game object, with an script inhering from Academy on it (it's often otherwise empty)
  - the Academy needs to know about the brain they want to work with at the time (e.g. if playing or showing/teaching, a Player Brain, or if Learning or Inferring, a Learning Brain)
  - "Control" checkbox next to Learning Brain needs to be checked if training with TensorFlow (Control checkbox activates external communicator to TensorFlow)
  - Any brain in use also needs to be in the Brain slot of the AGENT(s).
  - If they're using a Learning Brain for Inference, the Brain file (which sits in a slot on the Academy AND on the Agent(s))) needs to point to a TFModel in its model slot.
  - If using a Learning Brain for Training, the Brain file MUST have its Model slot EMPTY.
- When training, a configuration yaml file MUST have the name of the brain you want to train in it. We provide yaml parameters for all brains we'll be using. Imitation Learning uses "offline_bc" config file, everything else uses the default config file. Parameters for training start with the default set and then spill into any specific ones provided (named by the brain).
  - Example default set:

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
    use_curiosity: false
    curiosity_strength: 0.01
    curiosity_enc_size: 128
```

- Example set (put below the default set):

```
WarehouseOneCrate_Learning_IL:
    max_steps: 5.0e4
    batch_size: 128
    buffer_size: 2048
    beta: 1.0e-2
    hidden_units: 256
    summary_freq: 2000
    time_horizon: 64
    num_layers: 2
```

- If a brain called "WarehouseOneCrate_Learning_IL" was training, it would get its parameters from both of the above sets.