Nguyen D. & Burdea G. Virtual Reality Laboratory Manual, 3rd Ed.

# LAB THREE

# Dynamic Scene

**Requirement**

1. **Unity 3D** (using version 2018.4+)
2. **Blender** (using version 2.8+)
3. **Source-code Editor** (of your choice)

**Contents**

Exporting Models . . . . . . . . . . . . . . . . . . . . . 01
Physics . . . . . . . . . . . . . . . . . . . . . . . . . . 02
Rigidbodies . . . . . . . . . . . . . . . . . . . . . . . . 02
Joints . . . . . . . . . . . . . . . . . . . . . . . . . . 04
Scripting (C#) . . . . . . . . . . . . . . . . . . . . . . 05
Simple Scripting Tutorial . . . . . . . . . . . . . . . . 07
(Adv.) More Complex Scripting . . . . . . . . . . 10
(Adv.) Basic Unity Animations . . . . . . . . . . . 11
Scripted Animations . . . . . . . . . . . . . . . . . . . . 13
Post-Lab Homework Assignments . . . . . . . . . 15
References & Additional Resources . . . . . . . . 17

**Scripting** allows you to manipulate the objects within a scene and have a player able to dynamically interact with that scene.

In this lab, you will be combining the two tools we have learned so far (Unity and Blender) and creating a more complex and dynamic scene using custom models and scripting.

First, make sure you have your preferred source-code editor ready to go. Unity comes with MS Visual Studio, which is perfectly fine to use, or you can use another editor

To change the external editor that you want to use, go into the preferences settings by navigating to **Edit>Preferences...>External Tools**. Click on the external script editor drop down menu and select or browse for the editor of your choice. Now when you double click or open a script, it will use the editor that you have specified. (Note: you are expected to know how to code already)

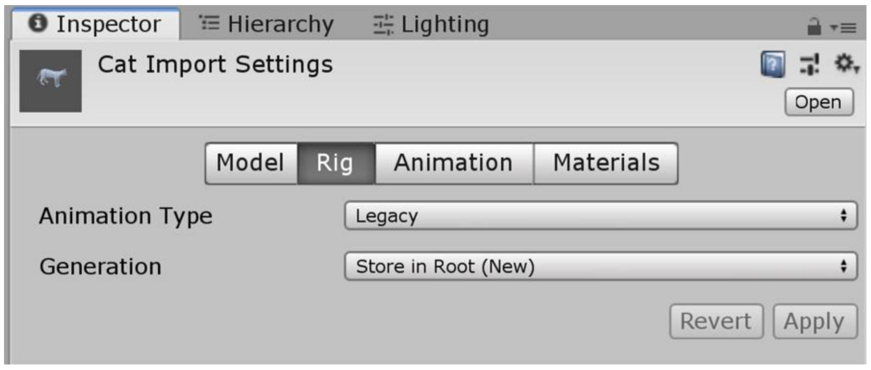
## Exporting Models

You will want to export the models you had created in Blender into Unity. There are several ways to do this. A very fast and efficient way is to save the Blender project file inside the Unity asset folder. This will export everything in the Blender project. If you want certain placeholder objects to not be exported, place them inside a separate collection and deselect that collection's checkbox. When you make changes and save the Blender project, the model instances within Unity will also be changed!

Another way to export models out of Blender is to use the built-in export tool. You can navigate to **File>Export>(select file type).** We recommend either using .fbx or .obj as file extensions for your exports. .fbx will preserve different meshes of your model while .obj will combine everything into a single mesh.

If you want to use the model's exported animations, there are a few settings within Unity you need to set up before it will work. After you export a model, find it in the asset folder and open up its inspector window.

Import



Cat Import Settings

| | |
|---|---|
| Animation Type | Legacy |
| Generation | Store in Root (New) |

Rig tab in import settings. Figure 1

Here you will find import settings that you can change if desired. Under the "Model" tab you can find settings that change how the mesh is imported. For example, if you want Unity to calculate the normal instead of importing them, you can select "Calculate" next to the "Normals" setting. This allows you to smooth out low polygon models visually.

If you want to import animations, go to the "Rig" tab (**Figure 1**) and change "Animation Type" to "Legacy" and make sure "Generation" is set to "Store in Root". After that you can go into the "Animation" tab to preview and change the animation settings. You should be able to see a list of animations where you can select individually. To make an animation loop, select the animation clip and select "Loop" under the "Wrap Mode" setting below.

Now there are also ways to animate GameObjects within Unity as well, which we will go over later. You can even script these animations to occur when something triggers them (e.g. opening a drawer by clicking on it).

## Physics ❷

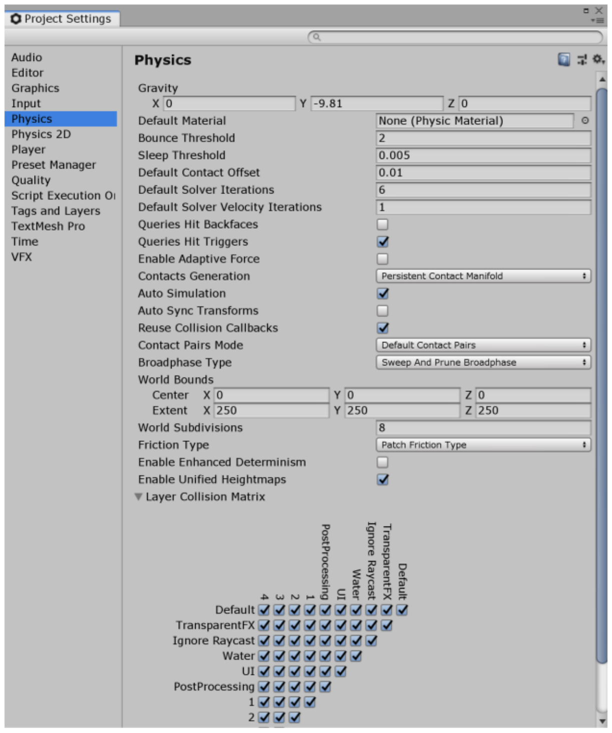
Physics as you already know concerns the nature and properties of matter and energy. For your 3D environment to have realistic physical properties, you must understand how to use Unity's physics engine. There are many parameters that you can change which allow objects to behave as you wish.

To change project level Physics settings, navigate to **Edit>Project Settings>Physics** and you should see a window like the one in **Figure 2**.

## Rigidbodies ❷

In this physics simulation, there are collisions, gravity, acceleration, momentum, and more. The primary component that you will be using will be called a "Rigidbody." A GameObject with a Rigidbody component attached will respond to gravity. Additionally, if it has colliders attached, it will respond to collisions.

Physics



Physics

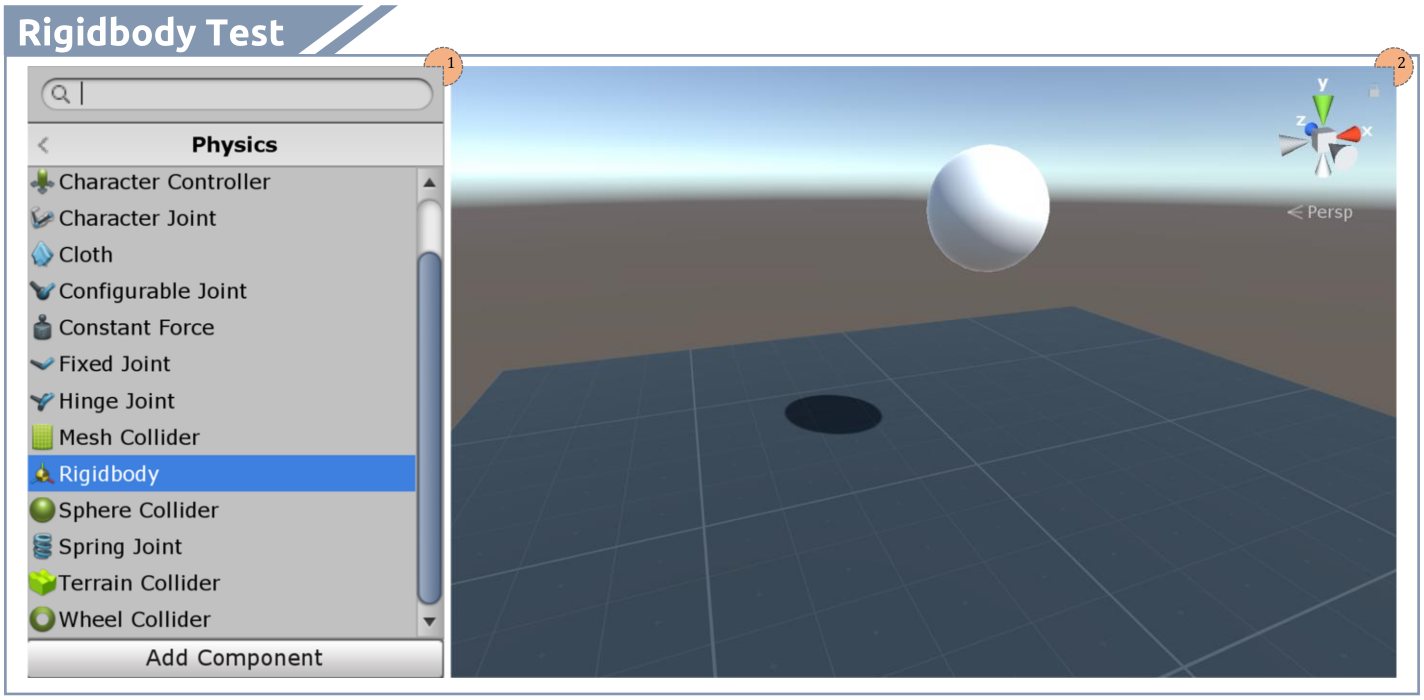
| | | | |
|---|---|---|---|
| Gravity | X 0 | Y -9.81 | Z 0 |
| Default Material | None (Physic Material) | | |
| Bounce Threshold | 2 | | |
| Sleep Threshold | 0.005 | | |
| Default Contact Offset | 0.01 | | |
| Default Solver Iterations | 6 | | |
| Default Solver Velocity Iterations | 1 | | |
| Queries Hit Backfaces | ☐ | | |
| Queries Hit Triggers | ✓ | | |
| Enable Adaptive Force | ☐ | | |
| Contacts Generation | Persistent Contact Manifold | | |
| Auto Simulation | ✓ | | |
| Auto Sync Transforms | ☐ | | |
| Reuse Collision Callbacks | ✓ | | |
| Contact Pairs Mode | Default Contact Pairs | | |
| Broadphase Type | Sweep And Prune Broadphase | | |
| World Bounds | | | |
| Center | X 0 | Y 0 | Z 0 |
| Extent | X 250 | Y 250 | Z 250 |
| World Subdivisions | 8 | | |
| Friction Type | Patch Friction Type | | |
| Enable Enhanced Determinism | ☐ | | |
| Enable Unified Heightmaps | ✓ | | |

Layer Collision Matrix

| | 4 | 3 | 2 | 1 | PostProcessing | UI | Water | Ignore Raycast | TransparentFX | Default |
|---|---|---|---|---|---|---|---|---|---|---|
| Default | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| TransparentFX | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Ignore Raycast | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| Water | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | |
| UI | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | |
| PostProcessing | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | |
| 1 | ✓ | ✓ | ✓ | ✓ | | | | | | |
| 2 | ✓ | ✓ | ✓ | | | | | | | |

Physics settings window for project level settings. Figure 2

Try adding a Rigidbody component to a sphere as seen in **[Figure 3, #1]**. Then, place the sphere above a plane GameObject. Both GameObjects should already have colliders. Position the camera to where you can see the sphere and plane and press play. You should see the sphere falling due to gravity and stopping when it collides with the plane **[Figure 3, #2]**.



Applying Rigidbody component to sphere and testing it's physics and collision. Figure 3

There are also some settings in the Rigidbody component that may be useful to change, depending on the situation. You can turn off gravity or change the type of collision detection. You can even change its mass and drag to influence it's "weight" or resistance behavior within the physics simulation. If the "Is Kinematic" setting is set to **true**, the object will not be driven by the physics engine. This is useful for anchoring joints, moving platforms, and other uses.

Let's try making the sphere bounce! There is something called a "Physics Material" ❓. Using this allows you to adjust friction and bouncing effects of colliding objects. You can read more about the technical details of the different components of a physics material in the Unity Manual. For now, let's apply a material that is bouncy.

Go to the "**Sphere Collider**" component on the sphere and locate the "**Material**" parameter. If you have the Unity Standard assets downloaded, there are some premade physics materials you can choose from. However, we will create our own physics material.

First, in the project folder window, create a physics material. In the inspector window, you should see a few parameters defining friction and bounciness. Change the "**Bounciness**" parameter to 1. The range for these parameters is [0, 1]. Set the physics material you just created into the "Material" parameter of the Sphere Collider. When you press play now, the sphere should bounce a few times! While still in play mode, go to the Scene window and move the sphere up and drop it again to see the bouncing effect if you missed it the first time.

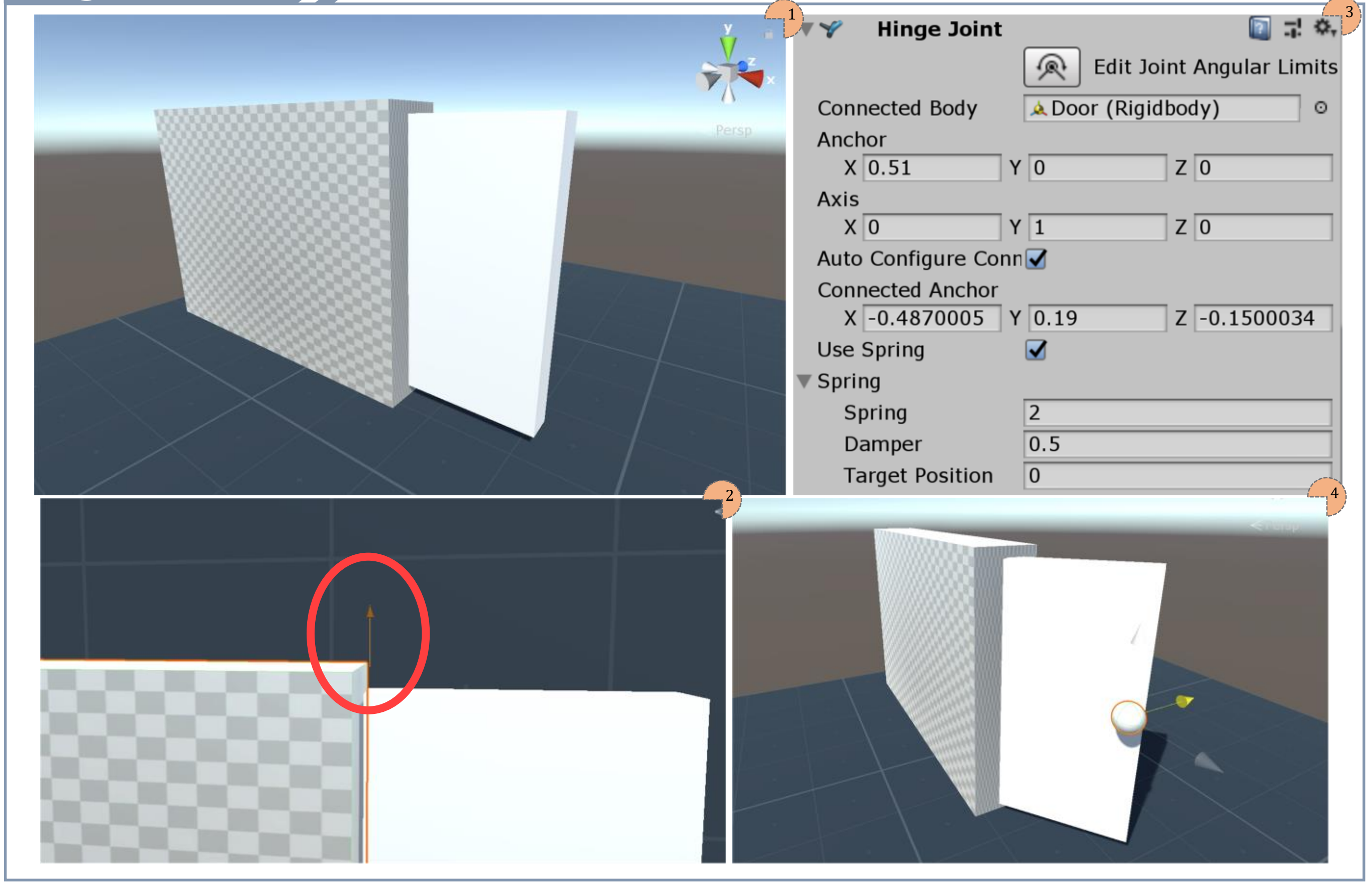
**TIP:** Going into the scene window while in Play mode is a good way to interact with the physics engine to test certain things out. You can pause the simulation, move the objects to where you want them, and un-pause for quick testing. You can even do this to debug running scripts!

## Joints ❓

Joints are a type of component that connect a Rigidbody to *another* Rigidbody or a fixed point in space. Joints are used to constrain Rigidbodies, so they move only in certain axes and they can also apply forces such as spring and dampening forces. Unity has 5 types of joints (not including 2D joints). These are: Character, Configurable, Fixed, Hinge, & Spring Joints. These joints have a lot of settings, parameters, and information which you can find in the Unity Manual.

We will go over the basics of the "Hinge Joint" ❓ so you can get a better understanding on how to use it. Let's use this type of joint to create a door that opens when it is collided with. First make two cubes and scale them to size. We want a thin door and a wall that the door can hinge onto as seen in **[Figure 4, #1]**.

Hinged Door



| | |
|---|---|
| Connected Body | Door (Rigidbody) |
| Anchor | X 0.51 Y 0 Z 0 |
| Axis | X 0 Y 1 Z 0 |
| Auto Configure Conn | ✓ |
| Connected Anchor | X -0.4870005 Y 0.19 Z -0.1500034 |
| Use Spring | ✓ |
| Spring | |
| Spring | 2 |
| Damper | 0.5 |
| Target Position | 0 |

Setting up a hinged door using the "Hinge Joint" component. Figure 4

Both objects must be Rigidbodies, so apply the joint component to both door and wall. We don't want the wall moving and driven by the physics engine, so need to enable "**Is Kinematic**" on the wall GameObject. Then, apply the "Hinge Joint" component to the wall. In the "Connected Body" parameter, find and select the door (or drag and drop from the hierarchy). We want the anchor to be on the left side of the door and we want it to rotate around the Y Axis. To do so, in the "Axis" parameters, change the Y axis to 1 and the others to 0. Then, we can move the anchor point to where we want it by changing the "Anchor" transform until we see the arrow is where our "hinge" should be (think about a real door) **[Figure 4, #2]**.

Then, we want the door to spring back into place, so enable "Use Spring" and set the parameters as seen in **[Figure 4, #3]**. To test, create a small sphere that has a Rigidbody that is set to "Is Kinematic". Now you can test the door by going into play mode and manually moving the sphere into the door in the scene window as seen in **[Figure 4, #4]**. Alternatively, you can use a standard asset character controller and walk into the door.

## Scripting (C#) ❷

Unity's supports C# and in the past, it also supported it's own version of JavaScript called UnityScript. However, it is now deprecated (Why?), so we will only be using C#.

Create a new script in the project window and name it something relevant. The file name and class name must be the same, so if you change the file name later, remember to also change the class name. Double clicking will open the script in the source-code editor of your choice. By default, it should look something like in **Snippet 1**.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TestScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

**SNIPPET 1** *TestScript.cs*

Unity offers a detailed scripting API Reference Manual that you can use to help with functions, callbacks, data types, and more! You can find the reference here. Remember to specify the version of Unity you are using at the top left corner of the page since there are some functions that may be deprecated or changed between various Unity versions.

Let's analyze the code snippet further. **MonoBehaviour** ❶ is the base class from which every Unity script derives. For the default template, it also has two Event Functions ❷ *Start()* and *Update()*. As you can see from the comments, *Start* is called before the first frame. This means you can initialize code in this function. *Update* gets called at every frame, which means the code inside will be executed every frame.

There are a few more event/callback functions that you may need to use in the future. Click on the names in **Table 1** for short descriptions of these functions.

**Event Functions**

| | |
|---|---|
| **Start** | Called when a script is enabled, just before any of the Update methods are called the first time. |
| **Awake** | Called when the script instance is being loaded. Used to initialize any variables or game state before the game starts |
| **Update** | Called every frame. Most commonly used function to implement any kind of game script. |
| **FixedUpdate** | Has the frequency of the physics system. It is called every fixed frame-rate frame |
| **LateUpdate** | LateUpdate is called after all Update functions have been called. |
| **OnGUI** | Called for rendering and handling GUI events. |
| **OnCollisionEnter** | Called when this collider/rigidbody has begun touching another rigidbody/collider. |
| **OnCollisionStay** | Called once per frame for every collider/rigidbody that is touching rigidbody/collider. |
| **OnCollisionExit** | Called when this collider/rigidbody has stopped touching another rigidbody/collider. |
| **OnTriggerEnter** | Called when the GameObject collides with another GameObject |
| **OnTriggerStay** | Called once per physics update for every Collider '*other*' that is touching the trigger GameObject. |
| **OnTriggerExit** | Called when the Collider '*other*' has stopped touching the trigger. |

Table 1

As you can see, there are quite a few functions that you can use. We have not listed all the available event functions, but these are the most important ones for the labs. We will not go over every Event Function in this Lab. However, we recommend reading what they do in the Unity scripting manual when you have the time between this Lab and next Lab.

There are many other classes, properties, and methods the Unity API offers as well. Again, we cannot go over every single one, so we will go over some of the most useful ones at this time in Table 2, 3, & 4. (Click on the names to learn more)

## Classes

| | |
|---|---|
| **Transform** | Position, rotation and scale of an object. |
| **GameObject** | Base class for all entities in Unity Scenes. |
| **Rigidbody** | Control of an object's position through physics simulation. |
| **Vector3** | Representation of 3D vectors and points. |

Table 2

## Properties

| | |
|---|---|
| **Transform.position** | The position of the Transform in word space/coordinates. |
| **Transform.rotation** | A Quaternion that stores the rotation of the Transform in world space. |
| **GameObject.transform** | The Transform class attached to a GameObject. |
| **Rigidbody.position** | The position of a rigidbody. |
| **Rigidbody.rotation** | The rotation of a rigidbody. |
| **Vector3.x** | X component of a vector. |
| **Vector3.y** | Y component of a vector. |
| **Vector3.z** | Z component of a vector. |

Table 3

## Methods

| | |
|---|---|
| **Transform.Translate** | Moves a transform in the direction and distance of translation. |
| **GameObject.GetComponent** | Returns the component of Type <type> if the game object has one attached, null if it doesn't. |
| **Rigidbody.AddForce** | Adds a force to a Rigidbody. |

Table 4

Something useful to note is that you can use `print()` (which shows up in the console window) to help debug code. Now that you have been introduced to some basic classes, properties, and methods, let's try to use them in a script code.

## Simple Scripting Tutorial

Let's create a few scripts. These scripts will do basic things such as moving around a GameObject, rotating it, scaling it. We will also add a force to Rigidbodies which will push them in a direction that we define.

First, let's make a sphere move around. Create a new script and name it something related to what it does. We will name our script *"Translate"*. We will show you the code snippet first and then explain what the code does.

```
public class Translate : MonoBehaviour
{
    // Public variables will be exposed in the inspector window.
    public float speed = 1.0f;

    void Update()
    {
        transform.Translate(Vector3.forward * Time.deltaTime * speed);
    }
}
```

SNIPPET 2 *Translate.cs*

Let's look at **Snippet 2**. Here we expose a public variable to the inspector window. We will be able to see and change the variable value when we are in the editor. This is very useful for quick changes or to see the current variable value for debugging. By default, private variables are not exposed (not seen in the Inspector window), but they can be serialized to do so. We will show you how to do this in the next code snippet.

In the Update function, we have a single line of code that uses the Translate method from the Transform class. Please note that the variable 'transform' is the transform of the GameObject the script is attached to. We can take advantage of this to reduce the amount of code we need to write!

Translate normally takes three parameters: `Transform.Translate(float x, float y, float z)`. However, we can use a Vector3 shorthand property instead. Here, we use `Vector3.forward` which is shorthand for `Vector3(0, 0, 1)`. So, it will only translate in the positive z axis direction. We can multiply that with `Time.deltaTime` ❓ to increment the movement every frame and multiply it with a speed value to change how fast, or slow, our object will move.

After you finish coding, add the script to the GameObject of your choice either by using *Add Component* or dragging the script onto the object in the scene window. You should be able to see the name of the script and any exposed variables in the inspector window, as shown in Figure 5.

**Translate**

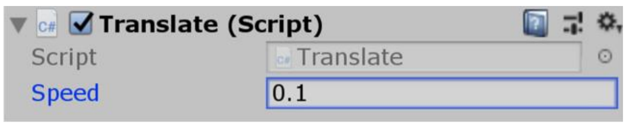


Figure 5

Try pressing play now. The script should start running and the object will move in the positive Z direction in the speed you specified. Try changing the exposed variable while it is running, so to see the change in object speed!

You can increase the complexity of the script by trying to add more code into it. Try making the object rotate as it moves (try not to use an untextured sphere, since you won't be able to see its rotation). You can also make the object teleport to a position in the world space by setting it's `Transform.position` property to a new Vector3 value. Remember to use the Scripting Manual if you don't know how to use a method or property.

In the next code snippet, we will apply a force to a Rigidbody. This can be used for shooting a projectile out of a cannon, giving your character a force push ability, and more!

```
public class Force : MonoBehaviour
{
    // Using the attribute SerializeField, we can expose a private
    // variable in the inspector window!
    [SerializeField]
    private float thrust = 10f;

    // This does not have the attribute, so it will not be exposed.
    private Rigidbody mRigid;

    void Awake()
    {
        mRigid = this.gameObject.getComponent<Rigidbody>();

        // Alternatively:
        // mRigid = GetComponent<Rigidbody>();
    }

    void FixedUpdate()
    {
        mRigid.AddForce(Vector3.forward * thrust);
    }
}
```

**SNIPPET 3** *Force.cs*

Let's look at code **Snippet 3**. Here we expose a private variable using the attribute **`[SerializeField]`** ❷. It is good practice to use private variables unless public variables are absolutely need. However, for the labs, it will not matter whether you choose to use public or private variables.

In this snippet, we initialize the variable 'mRigid' in the Awake function. We assign it to the Rigidbody component of the GameObject it is attached to. When using the **`GameObject.GetComponent<>()`** method, you need to put the component class/type in the brackets to specify what component you want. If the component does not exist on the GameObject, it will return null.

Next, we used FixedUpdate for adding the force here since we want to make sure it synchronizes with the physics engine. In this function, we use the **`Rigidbody.AddForce(`*`float x, float y, float z, ForceMode mode = ForceMode.Force`*`)`** method. Again, we use Vector3.forward to move Rigidbody in the positive Z direction and we multiply that with the amount of thrust we want to give it. Note that if you continue to add force over time, the velocity will increase over time too!

Add it to a Rigidbody GameObject and make sure to add a wall in front of it so it doesn't fall off your platform. You may also notice that if the object is going too fast, it will phase through colliders because the collision is not detected. You should have learned this from the course lecture or from the textbook! There are multiple solutions to this problem which we encourage you to research online!

# Advanced Section

## More Complex Scripting

There are more ways to move Rigidbodies than to use **`Rigidbody.AddForce`**. You can also use **`Rigidbody.MovePosition`** ❓. In this advanced section, we will show you how to use **`Physics.Raycast`** ❓ to make a basic obstacle avoidance algorithm while moving a kinematic rigidbody!

A Raycast casts an invisible ray from a point origin in a coded direction of given maximum length against all colliders in the Scene. There are a lot of applications you can use Raycasts for. For our purposes, we can use the information it gives us to create an algorithm that detects when a ray hits a wall and have our object turn in a different direction or turn completely around.

Look at **Snippet 4** for an example from the documentation on how to use the Raycast to detect a collision with a non-rigidbody object.

```
    void FixedUpdate()
    {
        Vector3 fwd = transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, fwd, 10))
            print("There is something in front of the object!");
    }
```

**SNIPPET 4** *Raycast.cs*

Looking at the code, you can see that it uses **`Transform.TransformDirection`** ❓ which is a method that changes the parameter direction from local space to world space. By now, you should have learned about the differences between local space/coordinates and world space/coordinates in the lecture or from the textbook. Basically, local space uses the coordinate system relative to its own origin, while world space uses the coordinate system relative to the world origin.

The script creates a Raycast and if there is a collision, it will return “true” which then triggers something (in this case a print statement). Now if you place three rays on the left, right, and front of the GameObject, you can use the collision information to determine where you can turn (or go backwards if the object is at a dead end).

Then, just have the object move forward at a constant speed and make it turn when your algorithm detects a collision through one or more Raycasts. You can make your object move at constant speed by using **`Rigidbody.MovePosition`** and have it turn by using Rigidbody.rotation. Looking at code **Snippet 5,** you can see that the rotation uses something called a Quaternion ❓ which is used to represent rotations. Quaternions can be quite complicated to work with in more advanced applications, but for this application, it is very simple.

```
. . .
    Object.MovePosition (transform.position + transform.forward *
          Time.deltaTime * objectSpeed); //Object always moves forward @ given speed.
. . .
    //Object turns right.
    Object.rotation = Quaternion.LookRotation (transform.right);
```

SNIPPET 5 *Pathfinder.cs*

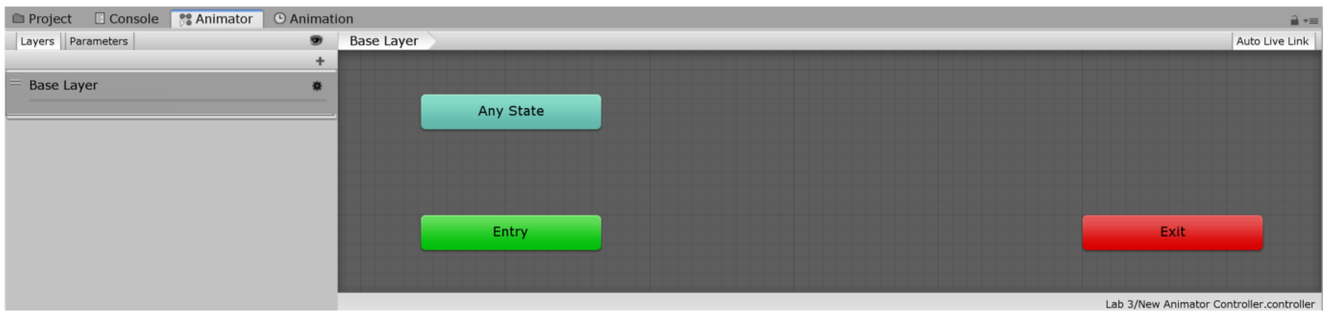
## Basic Unity Animations ❷

In Lab 2 you learned the basics of animation in Blender. Now we will cover animation in Unity. You can also script these animations to trigger when interacted with.

We will go over animating something simple, such as opening and closing a drawer. There are only two motions to work with.

First, open the Animation and Animator windows from the menu bar. Place these windows somewhere so you can see the Scene window and the Animation window at the same time. Then, create an Animator Controller in your project window and open it. You should be able to see the default Animator layout in **Figure 6**.

Animator



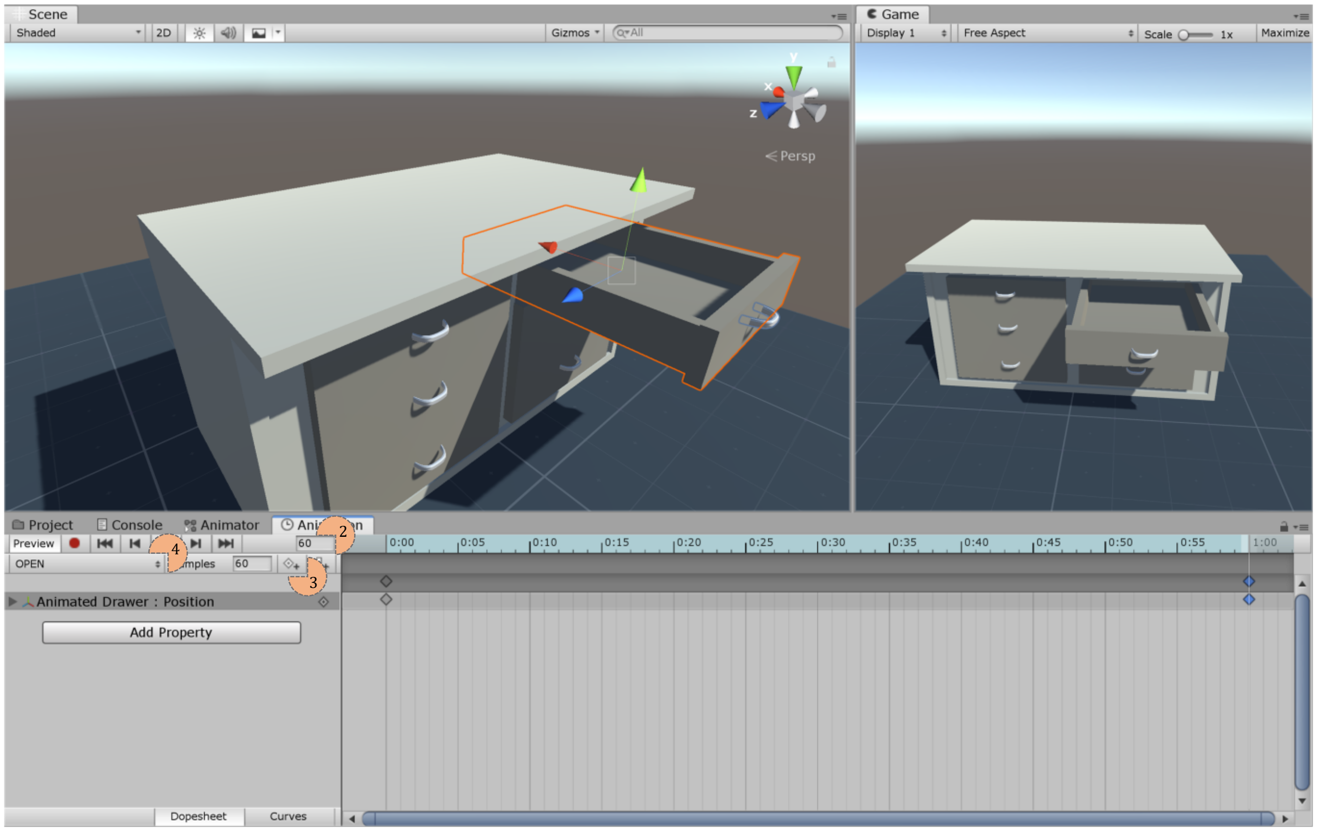
Default Animator layout. Figure 6

Now drag the animator onto the GameObject you want to animate. Alternatively, you can add the “Animator” component in the inspector window and set the Controller parameter to the controller you just made.

With the object still selected, go to the Animation window. You should see a blank animation timeline/dopesheet with a button telling you to create a new Animation Clip.

Click on “Create” and it will prompt you to name and save the animation file somewhere. Name it something relevant (e.g. “Open_Animation”) and save it in your asset folder where you can find it later.

## Animation



Scene, Game, and Animation windows showing a drawer. Figure 7

We are starting with the animation of the drawer opening. First, we need to click on Add Property and add the **Transform>Position** property. This will give us keyframes to work with.

Next, you can see the current frame number of the Dopesheet where **[Figure 7, #2]** is. You can either change this number by clicking on the timeline to change where the scrubber is. Now, move the scrubber to the last frame of your animation and open the drawer in the Scene window. Then insert a keyframe here so click the "Add Keyframe" button at **[Figure 7, #3]**.

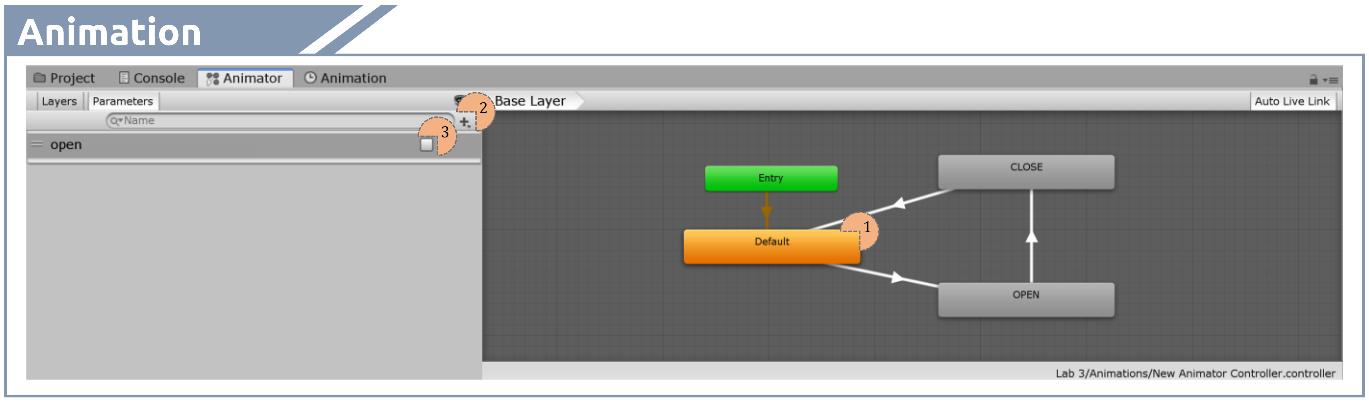
Now that your keyframes are in, you can preview the animation by pressing the Play button in the animator or scrubbing through the dopesheet timeline.

After you are happy with the opening animation, we need to do the drawer closing animation. You should be able to do this on your own, as it is the same process as before, but moving in the opposite direction. To add the closing animation, use the drop-down menu where **[Figure 7, #4]** is and click on "Create New Clip". This will make the animation be attached to the same Animator as the open animation.

Once you have completed both animations, go back to the Animator window. You should see two new rectangles. These are called states and you can create a state machine to transition between the different states. This is where we can use scripts to trigger certain transitions to other animation states.

## Scripted Animations ❷

Before we can make a script to trigger a transition, we need to set up the state transition machine to define the animation logic.



Animator window. Figure 8

We will not be using Any State or Exit State. First, you need to create a Default State. This state is empty and has no animation. However, it is important to be in our state machine since we need it to start in the closed resting position and not animate until it is triggered.

To create this, right click inside the window and create an empty state. Set the state to be the default state by selecting it, then right clicking it, and finally setting it to default state. It should turn orange and the Entry state will automatically transition to it **[Figure 8, #1]**.

Now create the transition connections (by selecting and then right clicking and making a transition) to our animation states as shown in **Figure 8**. You could press Play and see what happens now, but we need to make some changes.

First, we need to turn off animation looping (animations loop by default). Next, we need to set transition parameters to make sure the states only transition when we trigger that with a script. And lastly, we need to change a few settings to make the transitions smooth.

Go to your project window and find your two animation files. Click on them and in the inspector window, deselect Loop Time.

To add a parameter, click go to the parameters tab and click the plus button at **[Figure 8, #2]**. There are many different types, but since we only have two animation states, we can use a Boolean variable. We want the logic to be if the Boolean is TRUE, animate opening the drawer; if the Boolean is FALSE, animate closing of the drawer.

Then, select the transition line and set the Conditions for the transition from Default -> Open to use the Boolean variable and true. Set the Conditions for the transition from Open -> Close to use the Boolean variable and false. And for both conditions, deselect Has Exit Time. This will make it so we do not have to wait for the animation to finish to transition to the next state. (Leave the transition from Close -> Default alone). That allows the drawer to be closed when only partially open.

Now you can try it out! Go into Play mode and click on the checkbox next to the Boolean parameter in the Animator Controller **[Figure 8, #3]**. If the state machine was properly set up, it will transition the states from closed to open depending on the value of the Boolean parameter.

Now to script it, you can use the Animator class and it's properties and methods. Refer to **Snippet 6** for a basic script that toggles the Boolean variable when the drawer is clicked on using the callback function **OnMouseUp()** ❓.

```
public class ToggleOpen : MonoBehaviour {

    private Animator animator;

    void Awake()
    {
        animator = GetComponent<Animator>();

        // Updates the animator during the physic loop in order to
        // have the animation system synchronized with the physics engine.
        animator.updateMode = AnimatorUpdateMode.AnimatePhysics;
    }

    void OnMouseUp()
    {
        animator.SetBool("open", !animator.GetBool("open"));
    }
}
```

**SNIPPET 6** *ToggleOpen.cs*

The script uses the Animator class and uses the `Animator.SetBool(string name, bool value)` to set the Boolean value to the inverse of `Animator.GetBool(string name)` which returns the current Boolean value (thus toggling it).

The other line of code that incorporates `Animator.UpdateMode` is used to have the animation system synchronized with the physics engine. This way, friction and other forces can be calculated which allows objects to be placed inside the drawer and slide with it.

**TIP:** Sometimes you can reuse animations! Because we only animated the position of one axis and the other drawers also move in the same axis direction, we can reuse the same animation controller for all the drawers! Just apply both the Animation Controller and script and they should all open and close after you click them.

## Post Lab Homework Assignment – Option A

### [BASIC] (Undergraduate Students)

Now that you have learned how to export your custom models into Unity, create a full scene where a first-person character can walk around in. This can be an outdoor scene or an indoor scene, the choice is yours!

You will need to use Rigidbodies and use the physics engine. Perhaps create a building with a door you can push your way into using the hinges component, or a sliding door that detects when first-person is close by.

You also learned the basics of scripting so you will be required to have some scripted actions in your scene. Some ideas would be moving cars on a street, changing the lighting, changing texture of an object.

**Your project must have all the following deliverables:**

- At least 5 Unique models created in Blender;
- First-person controlled character;
- Lighting (e.g. direction lights, lamps);
- Textures and normal mapping;
- At least 2 Unique Scripts;
- Rigidbodies & use of physics engine (e.g. physics materials, kicking a ball around);
- 3D Audio (e.g. sounds of cars passing by).

**What to turn in:**

- Unity Project File
- Lab Report (Explanation & Images)

### [ADVANCED] (Graduate Students)

In *addition* to the basic assignment above, the Advanced Assignment will have you have interactable animated objects as well. Some ideas include clicking on something to open it or a toy that animates when a button is pressed.

**Additional deliverables:**

- At least 3 unique objects that have unique animations and can be interacted with to trigger the animation.

## Labyrinth

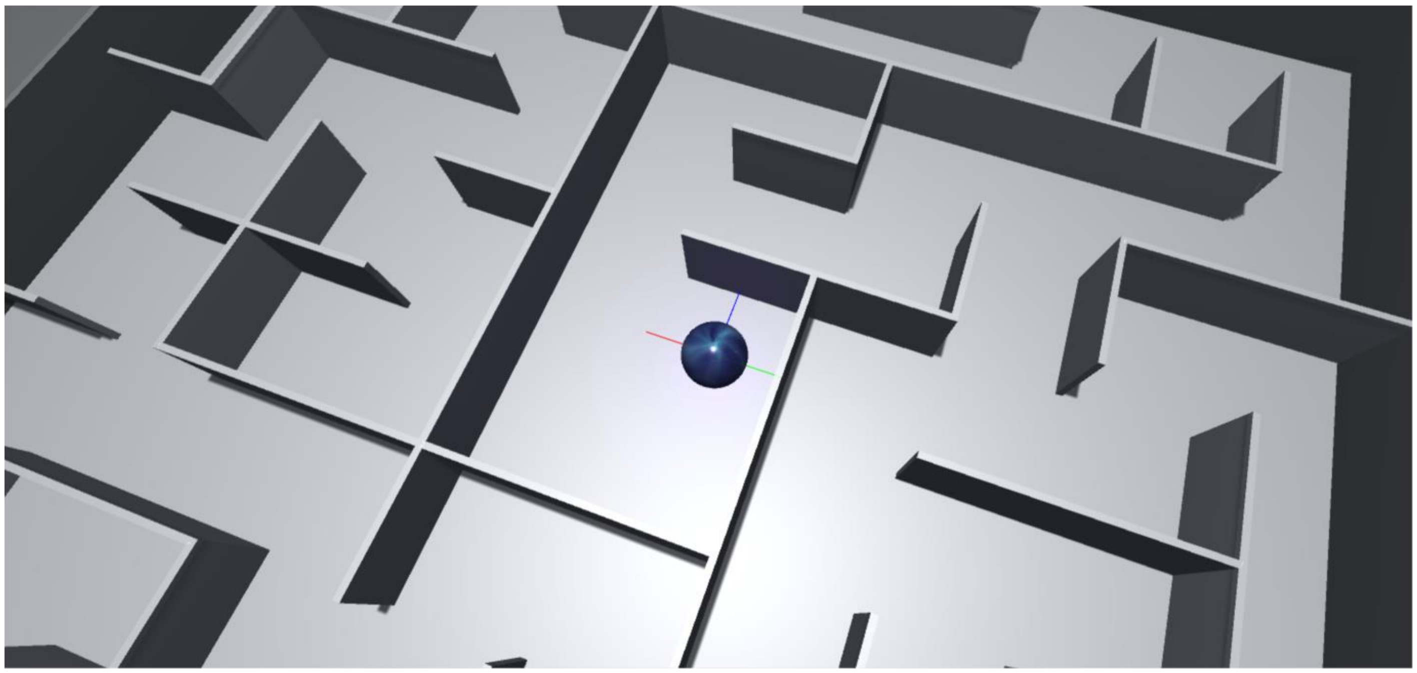


Figure 9

## Post Lab Homework Assignment – Option B

### [BASIC] - Same as Option A

### [ADVANCED] (Graduate Students)

You are not required to do the basic assignment for Option B. Instead, you must create a scene and algorithm that detects and avoids obstacles using the Raycast technique taught in the lab. You can either create a maze-like structure, such as the one seen in **Figure 9** or create an area with blocks and obstacles scattered around. Make sure to keep the object contained inside by creating walls!

Also seen in **Figure 9** are debug rays that you can only see in the Scene window. This is useful to utilize when testing and debugging!

**Your project must have all the following deliverables:**

- A functional algorithm that avoids obstacles using rays.
- It must not stop at any point in time. You will lose points if it gets stuck or hits an obstacle!

**What to turn in:**

- Unity Project File
- Lab Report (Explanation & Images)

## References & Additional Resources

1. Blender Export
    a. https://docs.blender.org/manual/en/latest/files/import_export.html

2. Unity Physics
    a. https://docs.unity3d.com/Manual/PhysicsSection.html
    b. https://docs.unity3d.com/Manual/class-Rigidbody.html
    c. https://docs.unity3d.com/Manual/Joints.html

3. Scripting with C# in Unity
    a. https://docs.unity3d.com/Manual/ScriptingSection.html
    b. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html
    c. https://docs.unity3d.com/Manual/EventFunctions.html
    d. https://docs.unity3d.com/ScriptReference/Time-deltaTime.html
    e. https://docs.unity3d.com/ScriptReference/SerializeField.html
    f. https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
    g. https://docs.unity3d.com/ScriptReference/Quaternion.html

4. Animating in Unity
    a. https://docs.unity3d.com/Manual/AnimationSection.html
    b. https://docs.unity3d.com/ScriptReference/Animator.html