

O'REILLY®

Fourth
Edition

Head First

C#

A Learner's Guide to
Real-World Programming
with C# and .NET Core

Andrew Stellman
& Jennifer Greene



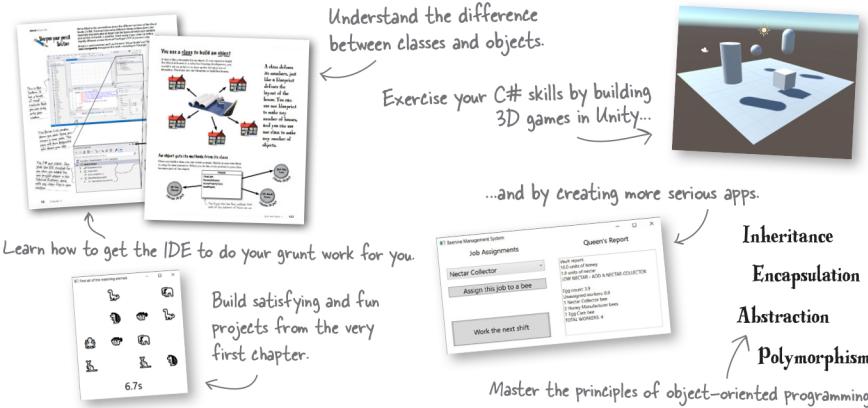
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then you'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

.NET

US \$64.99

CAN \$85.99

ISBN: 978-1-491-97670-8



5 6 4 9 9
9 781491 976708

"Thank you so much!
Your books have
helped me to launch
my career."

—Ryan White
Game Developer

"Andrew and Jennifer
have written a
concise, authoritative,
and most of all, fun
introduction to C#
development."

—Jon Galloway
Senior Program Manager on the
.NET Community Team
at Microsoft

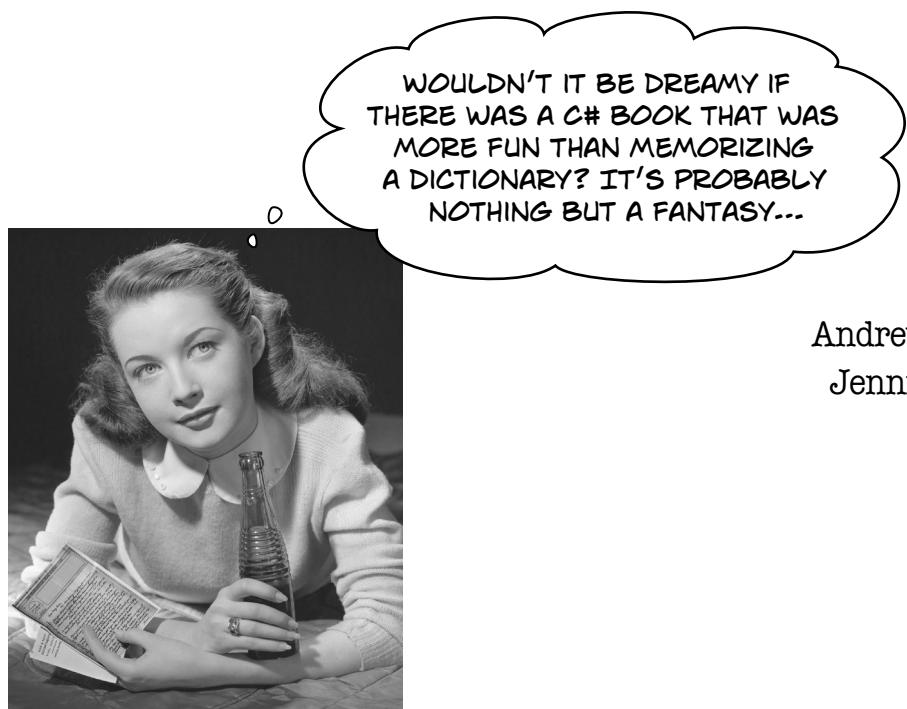
"If you want to learn
C# in depth and have
fun doing it, this is THE
book for you."

—Andy Parker
Fledgling C# programmer

O'REILLY®

Head First C#

Fourth Edition



WOULDN'T IT BE DREAMY IF
THERE WAS A C# BOOK THAT WAS
MORE FUN THAN MEMORIZING
A DICTIONARY? IT'S PROBABLY
NOTHING BUT A FANTASY...

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

Copyright © 2021 Jennifer Greene, Andrew Stellman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Series Creators:

Kathy Sierra, Bert Bates

Cover Designer:

Ellie Volckhausen

Brain Image on Spine:

Eric Freeman

Editors:

Nicole Taché, Amanda Quinn

Proofreader:

Rachel Head

Indexer:

Potomac Indexing, LLC

Illustrator:

Jose Marzan

Page Viewers:

Greta the miniature bull terrier and Samosa the Pomeranian

Printing History:

November 2007: First Edition.

May 2010: Second Edition.

August 2013: Third Edition.

December 2020: Fourth Edition



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, Windows, Visual Studio, MSDN, the .NET logo, Visual Basic, and Visual C# are registered trademarks of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI]

[2020-11-13]

Unity Lab #5

Raycasting

When you set up a scene in Unity, you're creating a virtual 3D world for the characters in your game to move around in. But in most games, most things in the game aren't directly controlled by the player. So how do these objects find their way around a scene?

The goal of labs 5 and 6 is to get you familiar with Unity's **pathfinding and navigation system**, a sophisticated AI system that lets you create characters that can find their way around the worlds that you create. In this lab, you'll build a scene out of GameObjects and use navigation to move a character around it.

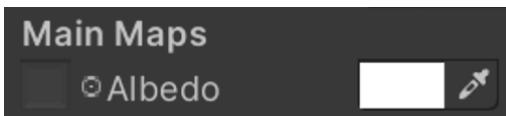
You'll use **raycasting** to write code that's responsive to the geometry of the scene, **capture input**, and use it to move a GameObject to the point where the player clicked. Just as importantly, you'll **get practice writing C# code** with classes, fields, references, and other topics we've discussed.

Create a new Unity project and start to set up the scene

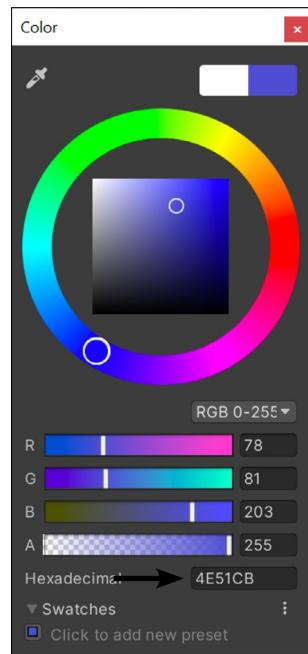
Before you begin, close any Unity project that you have open. Also close Visual Studio—we'll let Unity open it for us. Create a new Unity project using the 3D template, set your layout to Wide so it matches our screenshots, and give it a name like **Unity Labs 5 and 6** so you can come back to it later.

Start by creating a play area that the player will navigate around. Right-click inside the Hierarchy window and **create a Plane** (GameObject >> 3D Object >> Plane). Name your new Plane GameObject *Floor*.

Right-click on the Assets folder in the Project window and **create a folder inside it called Materials**. Then right-click on the new Materials folder you created and choose **Create >> Material**. Call the new material *FloorMaterial*. Let's keep this material simple for now—we'll just make it a color. Select Floor in the Project window, then click on the white box to the right of the word Albedo in the Inspector.

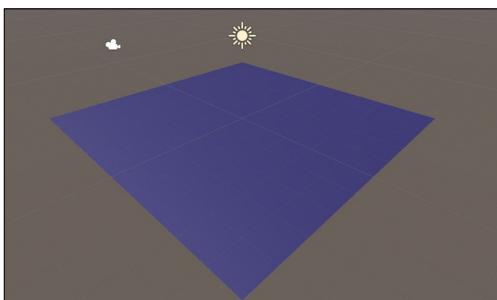


You can use this dropper to grab a color from anywhere on your screen.



In the Color window, use the outer ring to choose a color for the floor. We used a color with number 4E51CB in the screenshot—you can type that into the Hexadecimal box.

Drag the material from the **Project window onto the Plane GameObject in the Hierarchy window**. Your floor plane should now be the color that you selected.



A Plane has no Y dimension. What happens if you give it a large Y scale value? What if the Y scale value is negative? What if it's zero?

Think about it and take a guess. Then use the Inspector window to try various Y scale values and see if the plane acts the way you expected. (Don't forget to set them back!)

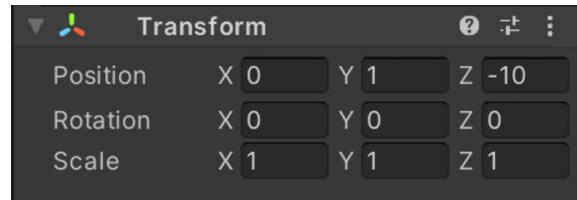
A **Plane** is a flat square object that's 10 units long by 10 units wide (in the X-Z plane), and 0 units tall (in the Y plane). Unity creates it so that the center of the plane is at point (0,0,0). This center point of the plane determines its position in the scene. Just like our other objects, you can move a plane around the scene by using the Inspector or the tools to change its position and rotation. You can also change its scale, but since it has no height, you can only change the X and Z scale—any positive number you put into the Y scale will be ignored.

The objects that you can create using the 3D Object menu (planes, spheres, cubes, cylinders, and a few other basic shapes) are called **primitive objects**. You can learn more about them by opening the Unity Manual from the Help menu and searching for the “**Primitive and placeholder objects**” help page. Take a minute and open up that help page right now. Read what it says about planes, spheres, cubes, and cylinders.

Set up the camera

In the last two Unity Labs you learned that a GameObject is essentially a “container” for components, and that the Main Camera has just three components: a Transform, a Camera, and an Audio Listener. That makes sense, because all a camera really needs to do is be at a location and record what it sees and hears. Have a look at the camera’s Transform component in the Inspector window.

Notice how the position is $(0, 1, -10)$. Click on the Z label in the Position line and drag up and down. You’ll see the camera fly back and forth in the scene window. Take a close look at the box and four lines in front of the camera. They represent the camera’s **viewport**, or the visible area on the player’s screen.



Move the camera around the scene and rotate it using

the Move tool (W) and Rotate tool (E), just like you did with other GameObjects in your scene. The Camera Preview window will update in real time, showing you what the camera sees. Keep an eye on the Camera Preview while you move the camera around. The floor will appear to move as it flies in and out of the camera’s view.

Use the context menu in the Inspector window to reset the Main Camera’s Transform component. Notice how it **doesn’t reset the camera to its original position**—it resets both the camera’s position and its rotation to $(0, 0, 0)$. You’ll see the camera intersecting the plane in the Scene window.



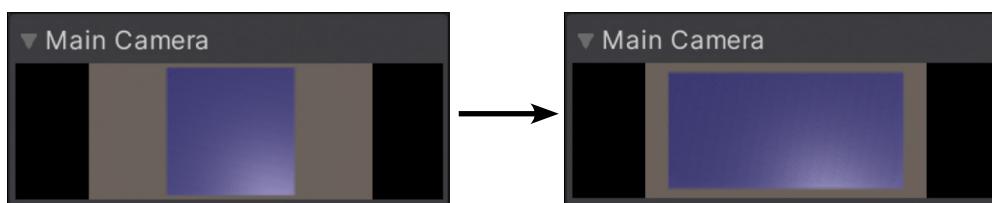
Now let’s point the camera straight down. Start by clicking on the X label next to Rotation and dragging up and down. You’ll see the viewport in the camera preview move. Now **set the camera’s X rotation to 90** in the Inspector window to point it straight down.

You’ll notice that there’s nothing in the Camera Preview anymore, which makes sense because the camera is looking straight down below the infinitely thin plane. **Click on the Y position label in the Transform component and drag up** until you see the entire plane in the Camera Preview.

Now **select Floor in the Hierarchy window**. Notice that the Camera Preview disappears—it only appears when the camera is selected. You can also switch between the Scene and Game windows to see what the camera sees.

Use the Plane’s Transform component in the Inspector window to **set the Floor GameObject’s scale to (4, 1, 2)** so that it’s twice as long as it is wide. Since a Plane is 10 units wide and 10 units long, this scale will make it 40 units long and 20 units wide. The plane will completely fill up the viewport again, so move the Camera further up along the Y axis until the entire plane is in view.

You can
switch
between
the Scene
and Game
windows
to see
what the
camera
sees.

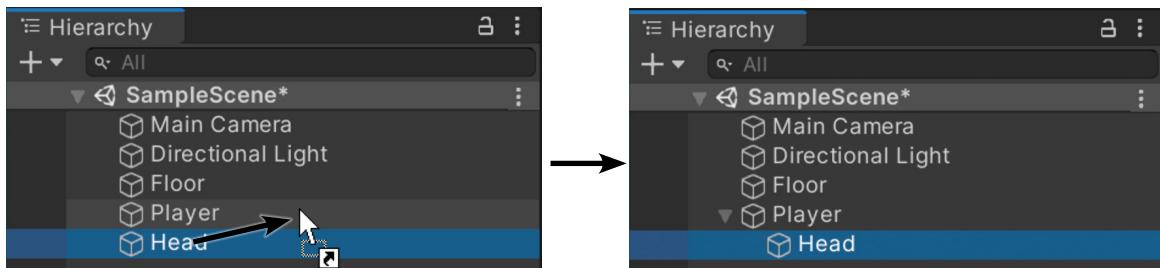


Create a GameObject for the player

Your game will need a player to control. We'll create a simple humanoid-ish player that has a cylinder for a body and a sphere for a head. Make sure you don't have any objects selected by clicking the scene (or the empty space) in the Hierarchy window.

Create a Cylinder GameObject (3D Object >> Cylinder)—you'll see a cylinder appear in the middle of the scene. Change its name to *Player*, then **choose Reset from the context menu** for the Transform component to make sure it has all of its default values. Next, **create a Sphere GameObject** (3D Object >> Sphere). Change its name to *Head*, and reset its Transform component as well. They'll each have a separate line in the Hierarchy window.

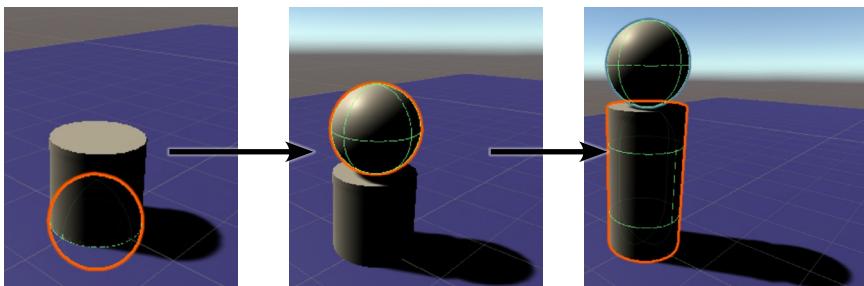
But we don't want separate GameObjects—we want a single GameObject that's controlled by a single C# script. This is why Unity has the concept of **parenting**. Click on Head in the Hierarchy window and **drag it onto Player**. This makes Player the parent of Head. Now the Head GameObject is **nested** under Player.



Select Head in the Hierarchy window. It was created at (0, 0, 0) like all of the other spheres you created. You can see the outline of the sphere, but you can't see the sphere itself because it's hidden by the plane and the cylinder. Use the Transform component in the Inspector window to **change the Y position of the sphere to 1.5**. Now the sphere appears above the cylinder, just the right place for the player's head.

Now select Player in the Hierarchy window. Since its Y position is 0, half of the cylinder is hidden by the plane. **Set its Y position to 1**. The cylinder pops up above the plane. Notice how it took the Head sphere along with it. Moving Player causes Head to move along with it because moving a parent GameObject moves its children too—in fact, *any* change that you make to its Transform component will automatically get applied to the children. If you scale it down, its children will scale, too.

Switch to the Game window—your player is in the middle of the play area.



When you modify the Transform component for a GameObject that has nested children, the children will move, rotate, and scale along with it.

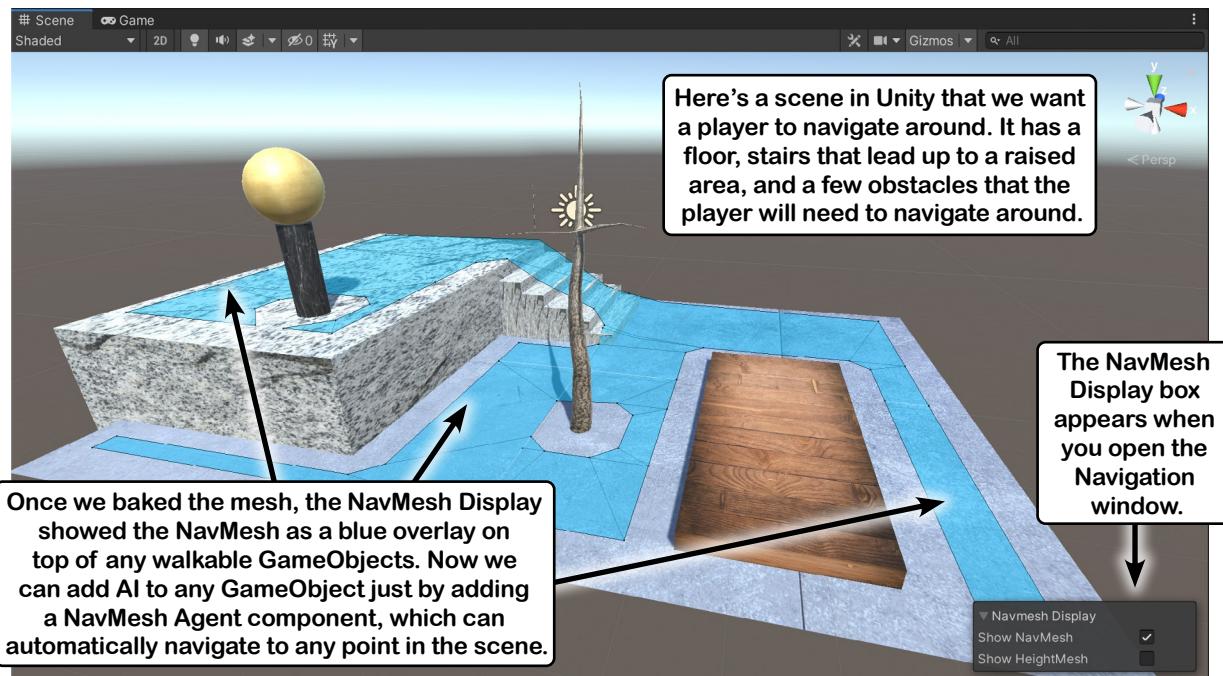
Introducing Unity's navigation system

One of the most basic things that video games do is move things around. Players, enemies, characters, items, obstacles...all of these things can move. That's why Unity is equipped with a sophisticated artificial intelligence-based navigation and pathfinding system to help GameObjects move around your scenes. We'll take advantage of the navigation system to make our player move toward a target.

Unity's navigation and pathfinding system lets your characters intelligently find their way around a game world. To use it, you need to set up basic pieces to tell Unity where the player can go:

- ★ First, you need to tell Unity exactly where your characters are allowed to go. You do this by **setting up a NavMesh**, which contains all of the information about the walkable areas in the scene: slopes, stairs, obstacles, and even points called off-mesh links that let you set up specific player actions like opening a door.
- ★ Second, you **add a NavMesh Agent component** to any GameObject that needs to navigate. This component automatically moves the GameObject around the scene, using its AI to find the most efficient path to a target and avoiding obstacles and, optionally, other NavMesh Agents.
- ★ It can sometimes take a lot of computation for Unity to navigate complex NavMeshes. That's why Unity has a Bake feature, which lets you set up a NavMesh in advance **and precompute (or bake)** the geometric details to make the agents work more efficiently.

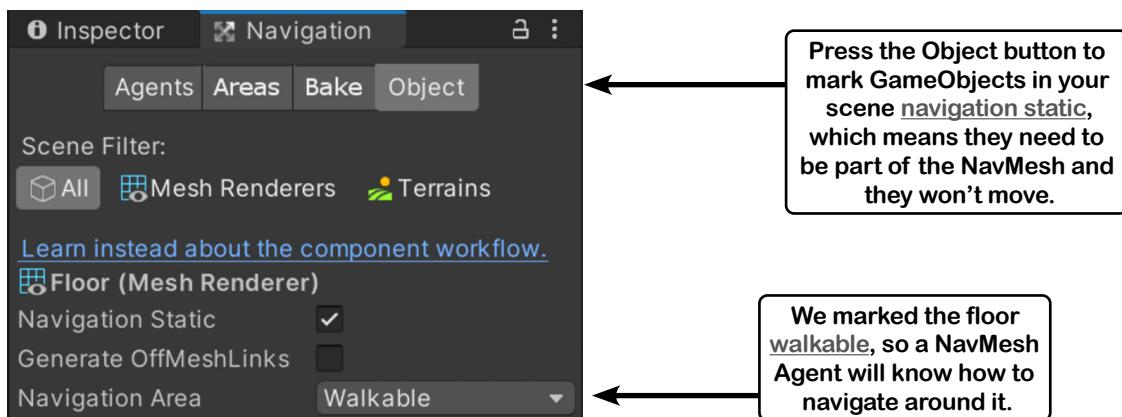
Unity provides a sophisticated AI navigation and pathfinding system that can move your GameObjects around a scene in real time by finding an efficient path that avoids obstacles.



Set up the NavMesh

Let's set up a NavMesh that just consists of the Floor plane. We'll do this using the Navigation window. Choose **AI >> Navigation from the Window menu** to add the Navigation window to your Unity workspace. It should show up as a tab in the same panel as the Inspector window. Then use the Navigation window to **mark the Floor GameObject navigation static and walkable**:

- ★ Press the **Object button** at the top of the Navigation window.
- ★ **Select the Floor plane** in the Hierarchy window.
- ★ Check the **Navigation Static box**. This tells Unity to include the Floor when baking the NavMesh.
- ★ **Select Walkable** from the Navigation Area dropdown. This tells Unity that the Floor plane is a surface that any GameObject with a NavMesh Agent can navigate.



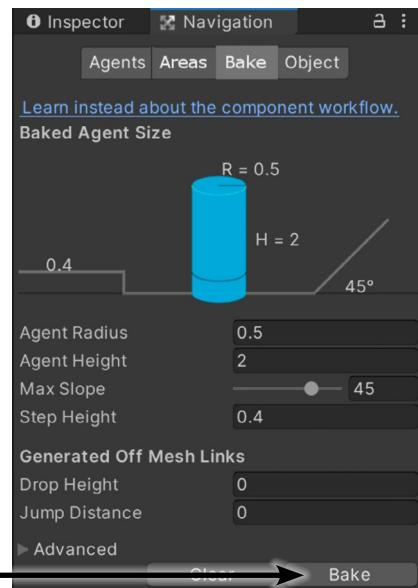
Since the only walkable area in this game will be the floor, we're done in the Object section. For a more complex scene with many walkable surfaces or nonwalkable obstacles, each individual GameObject needs to be marked appropriately.

Click the Bake button at the top of the Navigation window to see the bake options.

Now **click the other Bake button** at the bottom of the Navigation window. It will briefly change to Cancel and then switch back to Bake. Did you notice that something changed in the Scene window? Switch back and forth between the Inspector and Navigation windows. When the Navigation window is active, the Scene window shows the NavMesh Display and highlights the NavMesh as a blue overlay on top of the GameObjects that are part of the baked NavMesh. In this case, it highlights the plane that you marked as navigation static and walkable.

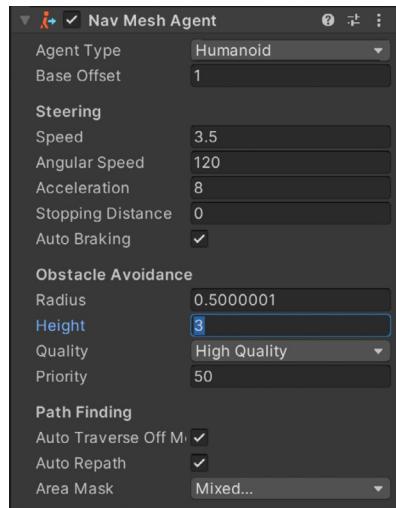
Your NavMesh is now set up.

Click the Bake button to bake the NavMesh.



Make your player automatically navigate the play area

Let's add a NavMesh Agent to your Player GameObject. Select **Player** in the Hierarchy window, then go back to the Inspector window, click the **Add Component** button, and choose **Navigation > NavMesh Agent** to add the NavMesh Agent component. The cylinder body is 2 units tall and the sphere head is 1 unit tall, so you want your agent to be 3 units tall—so set the Height to 3. Now the NavMesh Agent is ready to move the Player GameObject around the NavMesh.



Create a Scripts folder and add a script called *MoveToClick.cs*. This script will let you click on the play area and tells the NavMesh Agent to move the GameObject to that spot. You learned about private fields in Chapter 5. This script will use one to store a reference to the NavMeshAgent. Your GameObject's code will need a reference to its agent so it can tell the agent where to go, so you'll call the GetComponent method to get that reference and save it in a **private NavMeshAgent field** called **agent**:

```
agent = GetComponent<NavMeshAgent>();
```

The navigation system uses classes in the `UnityEngine.AI` namespace, so you'll need to **add this using line** to the top of your *MoveToClick.cs* file:

```
using UnityEngine.AI;
```

Here's the **code for your MoveToClick script**:

```
public class MoveToClick : MonoBehaviour
{
    private NavMeshAgent agent;

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();
    }

    void Update()
    {
        if (Input.GetMouseButtonUp(0))
        {
            Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
            Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit, 100))
            {
                agent.SetDestination(hit.point);
            }
        }
    }
}
```

In the last Unity Lab, you used the `Start` method to set a GameObject's position when it first appears. There's actually a method that gets called before your script's `Start` method. The `Awake` method is called when the object is created, while `Start` is called when the script is enabled. The *MoveToClick* script uses the `Awake` method to initialize the field, not the `Start` method.

Here's where the script handles mouse clicks. The `Input.GetMouseButtonUp` method checks if the user is currently pressing a mouse button, and the `0` argument tells it to check for the left button. Since `Update` is called every frame, it's always checking to see if the mouse button is clicked.

Experiment with the Speed, Angular Speed, Acceleration, and Stopping Distance fields in the NavMesh agent. You can change them while the game is running (but remember it won't save any values that you change while running the game). What happens when you make some of them really big?

Drag the script onto Player and run the game. While the game is running, **click anywhere on the floor**. When you click on the plane, the NavMesh Agent will move your player to the point that you clicked.

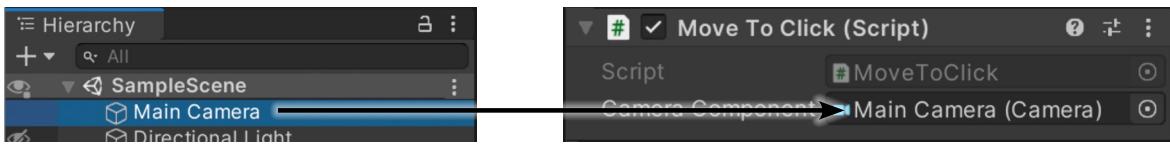


We've talked a lot about object references and reference variables over the last few chapters. Let's do a little pencil-and-paper work to get some of the ideas and concepts behind object references into your brain.

Add this public field to the MoveToClick class:

```
public Camera cameraComponent;
```

Go back to the Hierarchy window, click on Player, and find the new Main Camera field in the Move To Click (Script) component. Then **drag the Main Camera** out of the Hierarchy window and **onto the Camera Component field** in the Player GameObject's Move To Click (Script) component in the inspector:



Now **comment out** this line:

```
Camera cameraComponent = GameObject.Find("Main Camera").GetComponent<Camera>();
```

Run your game again. It still works! Why? Think about it, and see if you can figure it out. Write down the answer:

.....
.....
.....
.....



MY SCRIPT CALLED METHODS THAT
HAD THE WORD **RAY** IN THE NAME. I USED RAYS
BACK IN THE FIRST UNITY LAB. ARE WE USING RAYS TO HELP
MOVE THE PLAYER?

Yes! We're using a really useful tool called raycasting.

In the second Unity Lab, you used Debug.DrawRay to explore how 3D vectors work by drawing a ray that starts at (0, 0, 0). Your MoveToClick script's Update method actually does something similar to that. It uses the **Physics.Raycast** method to “cast” a ray—just like the one you used to explore vectors—that starts at the camera and goes through the point where the user clicked, and **checks if the ray hit the floor**. If it did, then the Physics.Raycast method provides the location on the floor where it hit. Then the script sets the **NavMesh Agent's destination** field, which causes the NavMesh Agent to **automatically move the player** toward that location.

Unity Lab #5

Raycasting

Raycasting Up Close

Your MoveToClick script calls the **Physics.Raycast** method, a really useful tool Unity provides to help your game respond to changes in the scene. It shoots (or “casts”) a virtual ray across your scene and tells you if it hit anything. The Physics.Raycast method’s parameters tell it where to shoot the ray and the maximum distance to shoot it:

Physics.Raycast(where to shoot the ray, out hit, maximum distance)

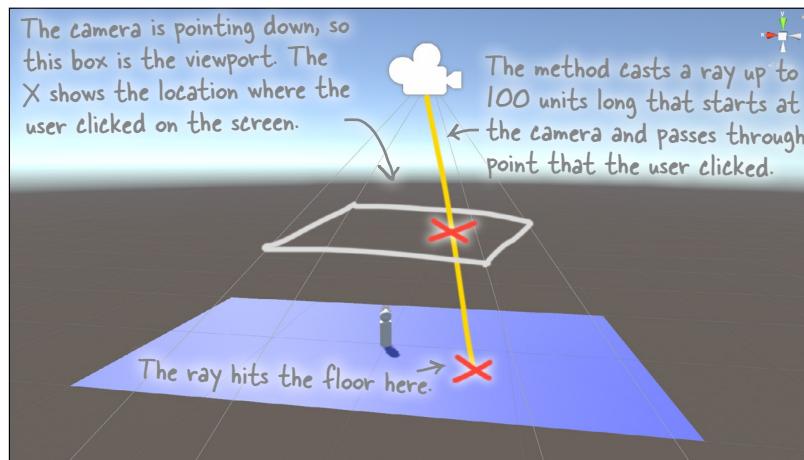
This method returns true if the ray hit something, or false if it didn’t. It uses the out keyword to save the results in a variable, exactly like you saw with int.TryParse in the last few chapters. Let’s take a closer look at how this works.

We need to tell Physics.Raycast where to shoot the ray. So the first thing we need to do is find the camera—specifically, the Camera component of the Main Camera GameObject. Your code gets it just like you got the GameController in the last Unity Lab:

```
GameObject.Find("Main Camera").GetComponent<Camera>();
```

The Camera class has a method called **ScreenPointToRay** that creates a ray that shoots from the camera’s position through an (X, Y) position on the screen. The **Input.mousePosition** method provides the (X, Y) position on the screen where the user clicked. This ray provides you with the location to feed into Physics.Raycast:

```
Ray ray = cameraComponent.ScreenPointToRay(Input.mousePosition);
```



Now that the method has a ray to cast, it can call the Physics.Raycast method to see where it hits:

```
RaycastHit hit;
if (Physics.Raycast(ray, out hit, 100))
{
    agent.SetDestination(hit.point);
}
```

It returns a bool and uses the out keyword—in fact, it works exactly like int.TryParse. If it returns true, then the hit variable contains the location on the floor that the ray hit. Setting agent.destination tells the NavMesh Agent to start moving the player toward the point where the ray hit.

Sharpen your pencil Solution

Run your game again. It still works! Why? Think about it, and see if you can figure it out. Write down the answer:

When my code called `mainCamera.GetComponent<Camera>` it returned a reference to a `GameObject`. I replaced

it with a field and dragged the Main Camera `GameObject` from the Hierarchy window into the Inspector window,

which caused the field to be set to a reference to the same `GameObject`. Those were two different ways to set

the `cameraComponent` variable to reference the same object, which is why it behaved the same way.

You'll be reusing the `MoveToClick` script in later Unity Labs, so after you're done writing down the answers, change the script back to the way it was by removing the `MainCamera` field and restoring the line that sets the `cameraComponent` variable.

BULLET POINTS

- A **Plane** is a flat square object that's 10 units by 10 units wide (in the X-Z plane), and 0 units tall (in the Y plane).
- You can **move the Main Camera** to change the part of the scene that it captures by modifying its Transform component, just like you move any other `GameObject`.
- When you modify the Transform component of a `GameObject` that has **nested children**, the children will move, rotate, and scale along with it.
- Unity's **AI navigation and pathfinding system** can move your `GameObjects` around a scene in real time by finding an efficient path that avoids obstacles.
- A **NavMesh** contains all of the information about the walkable areas in the scene. You can set up a NavMesh in advance and pre-compute—or bake—the geometric details to make the agents work more efficiently.
- A **NavMesh Agent** component automatically moves a `GameObject` around the scene, using its AI to find the most efficient path to a target.
- The **NavMeshAgent.SetDestination** method triggers the agent to calculate a path to a new position and start moving toward the new destination.
- Unity calls your script's **Awake** method when it first loads the `GameObject`, well before it calls the script's Start method but after it instantiates other `GameObjects`. It's a great place to initialize references to other `GameObjects`.
- The **Input.GetMouseButtonUp** method returns true if a mouse button is currently being clicked.
- The **Physics.Raycast** method does *raycasting* by shooting (or “casting”) a virtual ray across the scene and returns true if it hit anything. It uses the `out` keyword to return information about what it hit.
- The camera's **ScreenPointToRay** method creates a ray that goes through a point on the screen. Combine it with `Physics.Raycast` to determine where to move the player.