O'REILLY®

Fourth Edition

Head First C#

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

Andrew Stellman
& Jennifer Greene

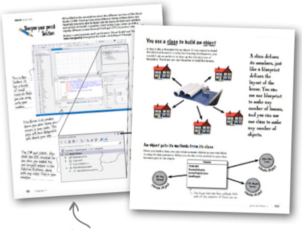
A Brain-Friendly Guide

Head First

C#

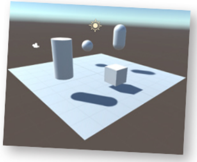
What will you learn from this book?

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!

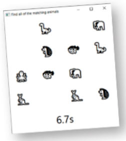


Understand the difference between classes and objects.

Learn how to get the IDE to do your grunt work for you.



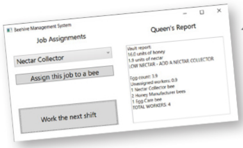
Exercise your C# skills by building 3D games in Unity...



Find all of the matching animals

6.7s

Build satisfying and fun projects from the very first chapter.



Beehive Management System

Job Assignments

Nectar Collector

Assign this job to a bee

Work the next shift

Queen's Report

```
Vault report:
16.0 units of honey
1.9 units of nectar
LOW NECTAR - ADD A NECTAR COLLECTOR

Egg count: 3.9
Unassigned workers: 0.9
1 Nectar Collector bee
2 Honey Manufacturer bees
1 Egg Care bee
TOTAL WORKERS: 4
```

...and by creating more serious apps.

Inheritance

Encapsulation

Abstraction

Polymorphism

Master the principles of object-oriented programming.

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.

> "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

.NET

US $64.99 **CAN $85.99**

ISBN: 978-1-491-97670-8

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

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

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 #6

Scene Navigation

In the last Unity Lab, you created a scene with a floor (a plane) and a player (a sphere nested under a cylinder), and you used a NavMesh, a NavMesh Agent, and raycasting to get your player to follow your mouse clicks around the scene.

Now we'll pick up where the last Unity Lab left off. The goal of these labs 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 you create. In this lab, you'll use Unity's navigation system to make your GameObjects move themselves around a scene.

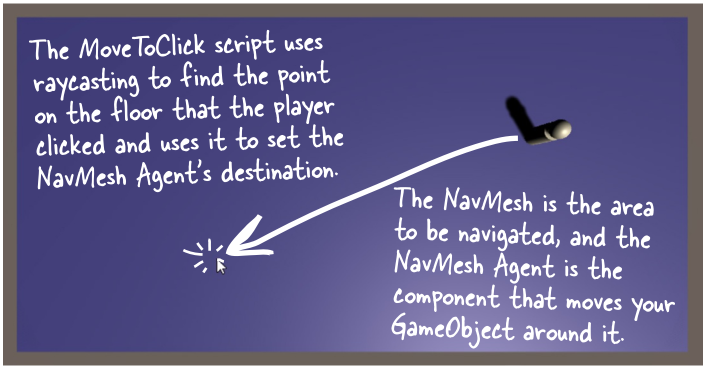
Along the way you'll learn about useful tools: you'll create a more complex scene and bake a NavMesh that lets an agent navigate it, you'll create static and moving obstacles, and most importantly, you'll **get more practice writing C# code**.

Let’s pick up where the last Unity Lab left off

In the last Unity Lab, you created a player out of a sphere head nested under a cylinder body. Then you added a NavMesh Agent component to move the player around the scene, using raycasting to find the point on the floor that the player clicked. In this lab, you’ll pick up where the last one left off. You’ll add GameObjects to the scene, including stairs and obstacles so you can see how Unity’s navigation AI handles them. Then you’ll add a moving obstacle to really put that NavMesh Agent through its paces.

So go ahead and **open the Unity project** that you saved at the end of the last Unity Lab. If you’ve been saving up the Unity Labs to do them back to back, then you’re probably ready to jump right in! But if not, take a few minutes and flip through the last Unity Lab again—and also look through the code that you wrote for it.

If you’re using our book because you’re preparing to be a professional developer, being able to go back and read and refactor the code in your old projects is a really important skill—and not just for game development!



The MoveToClick script uses raycasting to find the point on the floor that the player clicked and uses it to set the NavMesh Agent’s destination.

The NavMesh is the area to be navigated, and the NavMesh Agent is the component that moves your GameObject around it.

there are no Dumb Questions

Q: There were a lot of moving parts in the last Unity Lab. Can you go through them again, just so I’m sure I have everything?

A: Definitely. The Unity scene you created in the last lab has four separate pieces. It’s easy to lose track of how they work together, so let’s go through them one by one:

1. First there’s the **NavMesh**, which defines all of the “walkable” places your player can move around on in the scene. You made this by setting the floor as a walkable surface and then “baking” the mesh.

2. Next there’s the **NavMesh Agent**, a component that can “take over” your GameObject and move it around the NavMesh just by calling its SetDestination method. You added this to your *Player* GameObject.

3. The camera’s **ScreenPointToRay method** creates a ray that goes through a point on the screen. You added code to the Update method that checks if the player is currently pressing the mouse button, If so, it uses the current mouse position to compute the ray.

4. **Raycasting** is a tool that lets you cast (basically “shoot”) a ray. Unity has a useful Physics.Raycast method that takes a ray, casts it up to a certain distance, and if it hits something tells you what it hit.

Q: So how do those parts work together?

A: Whenever you’re trying to figure out how the different parts of a system work together, ***understanding the overall goal*** is a great place to start. In this case, the goal is to let the player click anywhere on the floor and have a GameObject move there automatically. Let’s break that down into a set of individual steps. The code needs to:

- **Detect that the player clicked the mouse.**
Your code uses Input.GetMouseButtonDown to detect a mouse click.

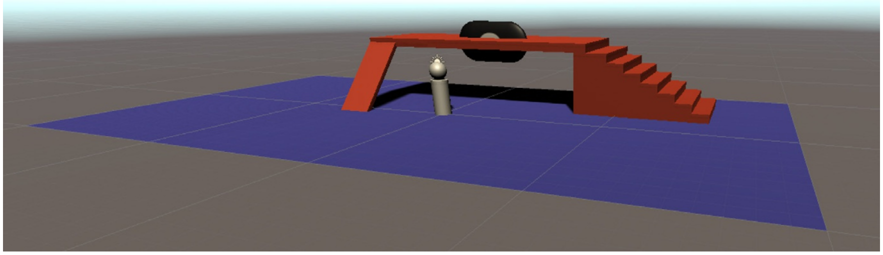
- **Figure out what point in the scene that click corresponds to.**
It uses Camera.ScreenPointToRay and Physics.Raycast to do raycasting and figure out which point in the scene the player clicked.

- **Tell the NavMesh Agent to set that point as a destination.**
The NavMeshAgent.SetDestination method triggers the agent to calculate a new path and start moving towards the new destination.

The MoveToClick method ***was adapted from code on the Unity Manual page*** *for the NavMeshAgent.SetDestination method. Take a minute and read it right now—****choose Help >> Scripting Reference*** *from the main menu, then search for NavMeshAgent.SetDestination.*

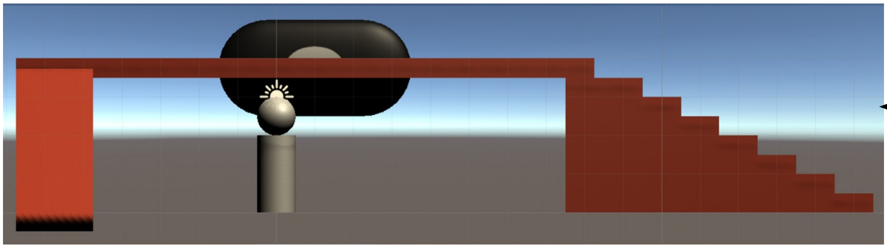
Add a platform to your scene

Let's do a little experimentation with Unity's navigation system. To help us do that, we'll add more GameObjects to build a platform with stairs, a ramp, and an obstacle. Here's what it will look like:



Sometimes it's easier to see what's going on in your scene if you switch to an isometric view. You can always reset the layout if you lose track of the view.

It's a little easier to see what's going on if we switch to an **isometric** view, or a view that doesn't show perspective. In a **perspective** view, objects that are further away look small, while closer objects look large. In an isometric view, objects are always the same size no matter how far away they are from the camera.



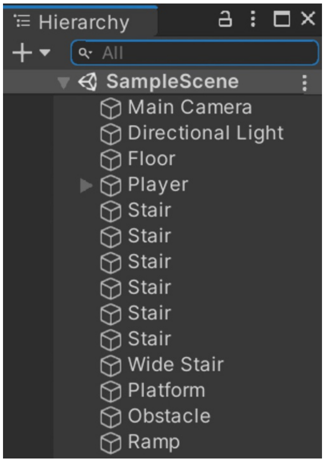
When you start Unity, the label (◁Persp) under the Scene Gizmo shows the view name. The three lines (◁) indicate that the Gizmo is in perspective mode. Click the cones to change the view to "Back" (◁Back). If you click the lines they'll change to three parallel lines (≡), which toggles the view to isometric mode.

Add 10 GameObjects to your scene. **Create a new material called *Platform*** in your Materials folder with albedo color CC472F, and add it to all of the GameObjects except for Obstacle, which uses a **new material called *8 Ball*** with the 8 Ball Texture map from the first Unity Lab. This table shows their names, types, and positions:

Name	Type	Position	Rotation	Scale
Stair	Cube	(15, 0.25, 5)	(0, 0, 0)	(1, 0.5, 5)
Stair	Cube	(14, 0.5, 5)	(0, 0, 0)	(1, 1, 5)
Stair	Cube	(13, 0.75, 5)	(0, 0, 0)	(1, 1.5, 5)
Stair	Cube	(12, 1, 5)	(0, 0, 0)	(1, 2, 5)
Stair	Cube	(11, 1.25, 5)	(0, 0, 0)	(1, 2.5, 5)
Stair	Cube	(10, 1.5, 5)	(0, 0, 0)	(1, 3, 5)
Wide stair	Cube	(8.5, 1.75, 5)	(0, 0, 0)	(2, 3.5, 5)
Platform	Cube	(0.75, 3.75, 5)	(0, 0, 0)	(15, 0.5, 5)
Obstacle	Capsule	(1, 3.75, 5)	(0, 0, 90)	(2.5, 2.5, 0.75)
Ramp	Cube	(–5.75, 1.75, 0.75)	(–46, 0, 0)	(2, 0.25, 6)

Try creating the first stair, then duplicating it five times and modifying its values.

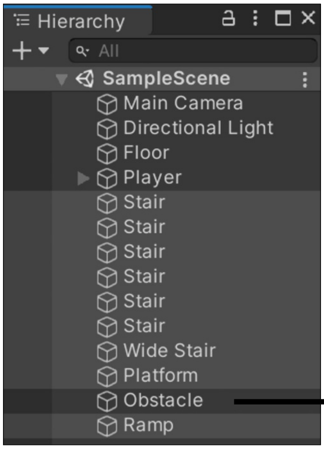
A Capsule is like a cylinder that has spheres on its ends.



Hierarchy
All
SampleScene
Main Camera
Directional Light
Floor
Player
Stair
Stair
Stair
Stair
Stair
Stair
Wide Stair
Platform
Obstacle
Ramp

Use bake options to make the platform walkable

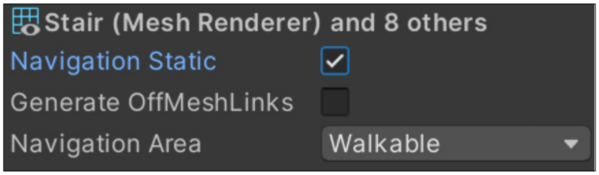
Use Shift-click to select all of the new GameObjects that you added to the scene, then use Control-click (or Command-click on a Mac) to deselect Obstacle. Go to the Navigation window and click the Object button, then **make them all walkable by** checking Navigation Static and setting the Navigation Area to Walkable. **Make the Obstacle GameObject not walkable** by selecting it, clicking Navigation Static, and setting Navigation Area to Not Walkable.



Hierarchy

All

SampleScene
- Main Camera
- Directional Light
- Floor
- Player
- Stair
- Stair
- Stair
- Stair
- Stair
- Stair
- Wide Stair
- Platform
- Obstacle
- Ramp



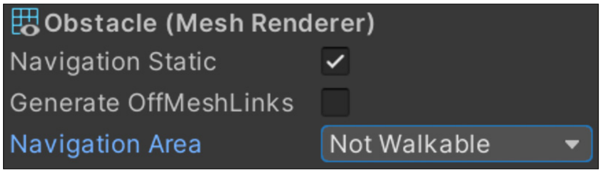
Stair (Mesh Renderer) and 8 others

Navigation Static ☑

Generate OffMeshLinks □

Navigation Area: Walkable

You can multiselect several different GameObjects in the Hierarchy and set their navigation options at the same time.



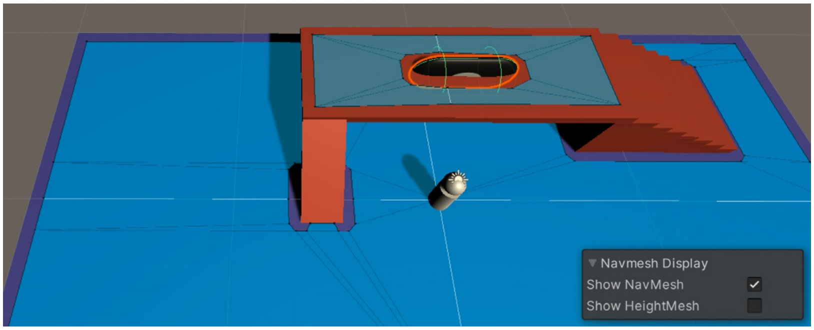
Obstacle (Mesh Renderer)

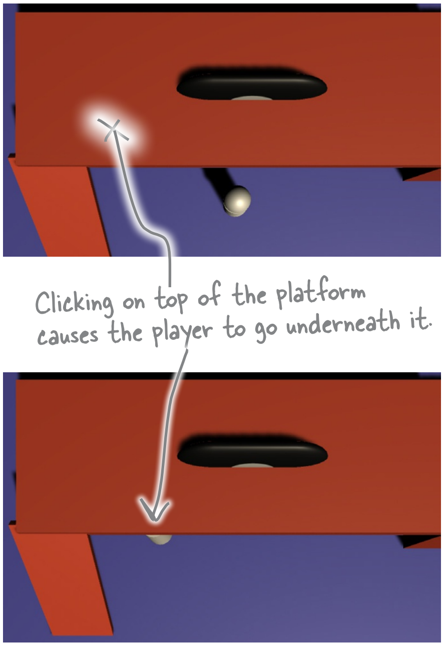
Navigation Static ☑

Generate OffMeshLinks □

Navigation Area: Not Walkable

Now follow the same steps that you used before to **bake the NavMesh**: click the Bake button at the top of the Navigation window to switch to the Bake view, then click the Bake button at the bottom.





Clicking on top of the platform causes the player to go underneath it.

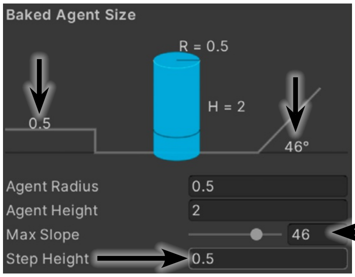
It looks like it worked! The NavMesh now appears on top of the platform, and there's space around the obstacle. Try running the game. Click on top of the platform and see what happens.

Hmm, hold on. Things aren't working the way we expected them to. When you click on top of the platform, the player goes ***under*** it. If you look closely at the NavMesh that's displayed when you're viewing the Navigation window, you'll see that it also has space around the stairs and ramp, but doesn't actually include either of them in the NavMesh. The player has no way to get to the point you clicked on, so the AI gets it as close as it can.

Include the stairs and ramp in your NavMesh

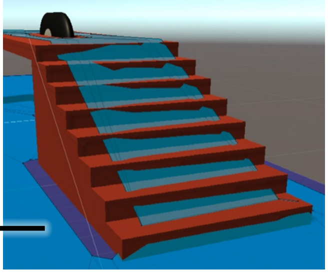
An AI that couldn't get your player up or down a ramp or stairs wouldn't be very intelligent. Luckily, Unity's pathfinding system can handle both of those cases. We just need to make a few small adjustments to the options when we bake the NavMesh. Let's start with the stairs. Go back to the Bake window and notice that the default value of Step Height is 0.4. Take a careful look at the measurements for your steps—they're all 0.5 units tall. So to tell the navigation system to include steps that are 0.5 units, **change the Step Height to 0.5**. You'll see the picture of the step in the diagram get taller, and the number above it change from the default 0.4 value to 0.5.

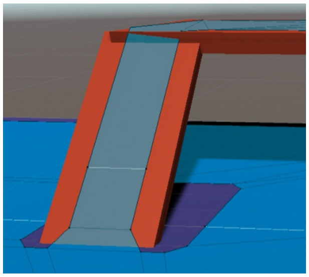
We still need to include the ramp in the NavMesh. When you created the GameObjects for the platform, you gave the ramp an X rotation of –46, which means that it's a 46-degree incline. The Max Slope setting defaults to 45, which means it will only include ramps, hills, or other slopes with at most a 45-degree incline. So **change Max Slope to 46**, then **bake the NavMesh again**. Now it will include the ramp and stairs.



Baked Agent Size

Agent Radius 0.5
Agent Height 2
Max Slope 46
Step Height 0.5





Start your game and test out your new NavMesh changes.

Exercise

Here's another Unity coding challenge! Earlier, we pointed the camera straight down by setting its X rotation to 90. Let's see if we can get a better look at the player by making the arrow keys and mouse scroll wheel control the camera. You already know almost everything you need to get it to work—we just need to add a little code. *It may seem complicated, but **you <u>can</u> do this!***

- **Create a new script called MoveCamera and drag it onto the camera.** It should have a Transform field called Player. Drag the Player GameObject out of the Hierarchy and **onto the Player field in the Inspector**. Since the field's type is Transform, it copies a reference to the Player GameObject's Transform component.
- **Make the arrow keys rotate the camera around the player.** Input.GetKey(KeyCode.LeftArrow) will return true if the player is currently pressing the left arrow key, and you can use RightArrow, UpArrow, and DownArrow to check for the other arrow keys. Use this method just like you used Input.GetMouseButtonDown in your MoveToClick script to check for mouse clicks. When the player presses a key, call transform.RotateAround to rotate around the player's position. The player's position is the first argument; use Vector3.left, Vector3.right, Vector3.up, or Vector3.down as the second argument, and a field called Angle (set to 3F) as the third argument.
- **Make the scroll wheel zoom the camera.** Input.GetAxis("Mouse ScrollWheel") returns a number (usually between –0.4 and 0.4) that represents how much the scroll wheel moved (or 0 if it didn't move). Add a float field called ZoomSpeed set to 0.25F. Check if the scroll wheel moved. If it did, do a little vector arithmetic to zoom the camera by multiplying transform.position by (1F + scrollWheelValue * ZoomSpeed).
- **Point the camera at the player.** The transform.LookAt method makes a GameObject look at a position. Then <u>reset the Main Camera's Transform</u> to position (0, 1, –10) and rotation (0, 0, 0).

It's <u>not</u> cheating to peek at the solution!

It's OK if your code looks a little different than ours as long as it works. There are a LOT of ways to solve any coding problem! Just make sure to take some time and understand how this code works.

Here's another Unity coding challenge! We pointed the camera straight down by setting its X rotation to 90. Let's see if we can get a better look at the player by making the arrow keys and mouse scroll wheel control the camera. You already know almost everything you need to get it to work—we just need to give you a little code to include. *It seems like a lot, but* ***you can do this!***

```
public class MoveCamera : MonoBehaviour
{
  public Transform Player;
  public float Angle = 3F;
  public float ZoomSpeed = 0.25F;

  void Update()
  {
    var scrollWheelValue = Input.GetAxis("Mouse ScrollWheel");
    if (scrollWheelValue != 0)
    {
        transform.position *= (1F + scrollWheelValue * ZoomSpeed);
    }

    if (Input.GetKey(KeyCode.RightArrow))
    {
        transform.RotateAround(Player.position, Vector3.up, Angle);
    }

    if (Input.GetKey(KeyCode.LeftArrow))
    {
        transform.RotateAround(Player.position, Vector3.down, Angle);
    }

    if (Input.GetKey(KeyCode.UpArrow))
    {
        transform.RotateAround(Player.position, Vector3.right, Angle);
    }

    if (Input.GetKey(KeyCode.DownArrow))
    {
        transform.RotateAround(Player.position, Vector3.left, Angle);
    }

    transform.LookAt(Player.position);
  }
}
```

Did you remember to reset the Main Camera's position and rotation? If not, it may look a little jumpy when the player starts moving (it's because of the way the angle is computed in the Camera.LookAt method).

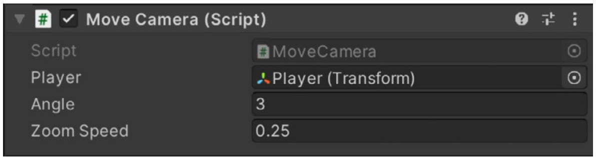
This is an example of how simple vector arithmetic can simplify a task. A GameObject's position is a vector, so multiplying it by a 1.02 moves it a little further away from the zero point, and multiplying it by .98 moves it a little closer.

This is just like how you used transform.RotateAround in the last two Unity Labs, except instead of rotating around Vector3.zero (0,0,0) you're rotating around the player.

We asked you to create the Player field with type Transform. That gives you a reference to the Player GameObject's Transform component, so Player.position is the player's position.

Use the arrow keys to move the camera so it's looking up at the player. You can see right through the floor plane!

Don't forget to drag Player onto the field in the script component in the Main Camera.



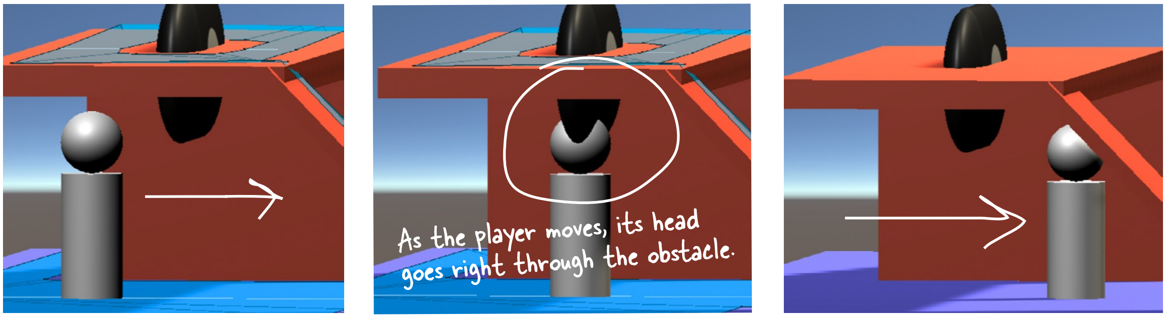
Move Camera (Script)

Script: MoveCamera
Player: Player (Transform)
Angle: 3
Zoom Speed: 0.25

Try experimenting with different angles and zoom speeds to see what feels best to you.

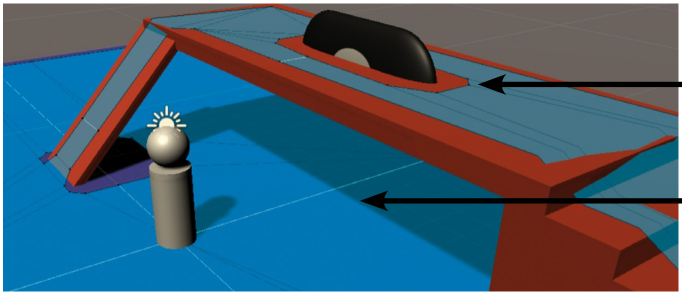
Fix height problems in the NavMesh

Now that we've got control of the camera, we can get a good look at what's going on under the platform—and something doesn't look quite right. Start your game, then rotate the camera and zoom in so you can get a clear view of the obstacle sticking out under the platform. Click the floor on one side of the obstacle, then the other. It looks like the player is going right through the obstacle! And it goes right through the end of the ramp, too.


As the player moves, its head goes right through the obstacle.

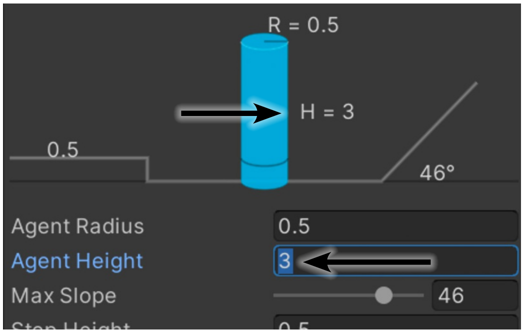
But if you move the player back to the top of the platform, it avoids the obstacle just fine. What's going on?

Look closely at the parts of the NavMesh above and below the obstacle. Notice any differences between them?

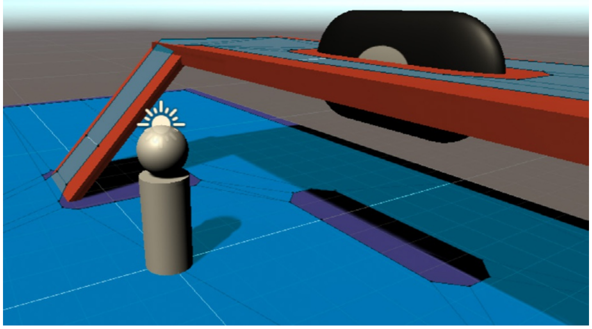


There's clearance in the NavMesh around the obstacle on top of the platform, but there are no gaps in the NavMesh below the platform, so the player will just walk straight through the obstacle.

Go back to the part of the last lab where you set up the NavMesh Agent component—specifically, the part where you set the Height to 3. Now you just need to do the same for the NavMesh. Go back to the Bake options in the Navigation window and **set the Agent Height to 3, then bake your mesh again**.



Agent Radius 0.5
Agent Height 3
Max Slope 46

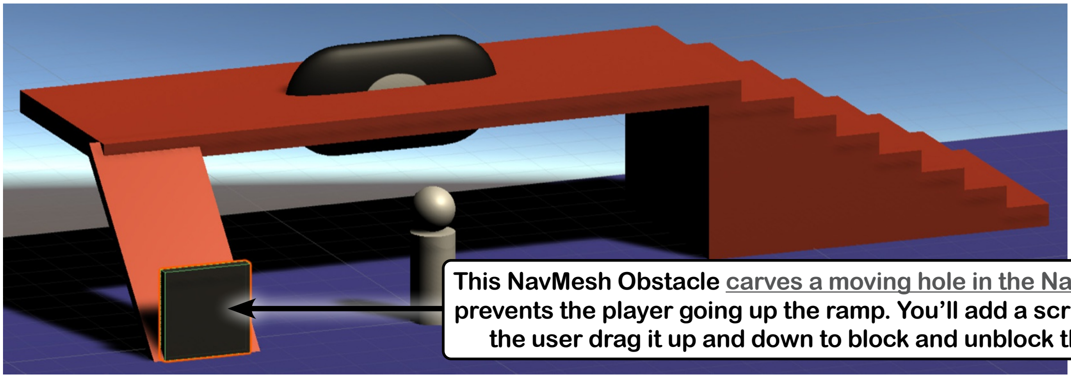


This created a gap in the NavMesh under the obstacle and expanded the gap under the ramp. Now the player doesn't hit either the obstacle or the ramp when moving around under the platform.

Add a NavMesh Obstacle

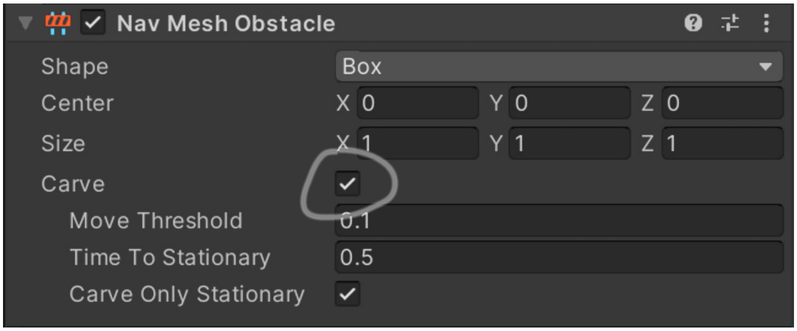
You already added a static obstacle in the middle of your platform: you created a stretched-out capsule and marked it non-walkable, and when you baked your NavMesh it had a hole around the obstacle so the player has to walk around it. What if you want an obstacle that moves? Try moving the obstacle—the NavMesh doesn't change! It still has a hole *where the obstacle was*, not where it currently is. If you bake it again, it just creates a hole around the obstacle's new location. To add an obstacle that moves, add a **NavMesh Obstacle component** to a GameObject.

Let's do that right now. **Add a Cube to your scene** with position (−5.75, 1, −1) and scale (2, 2, 0.25). Create a new material for it with a dark gray color (333333) and name your new GameObject *Moving Obstacle*. This will act as a kind of gate at the bottom of the ramp that can move up out of the way of the player or down to block it.



This NavMesh Obstacle carves a moving hole in the NavMesh that prevents the player going up the ramp. You'll add a script that lets the user drag it up and down to block and unblock the ramp.

We just need one more thing. Click the Add Component button at the bottom of the Inspector window and choose Navigation >> Nav Mesh Obstacle to **add a NavMesh Obstacle component** to your Cube GameObject.



Nav Mesh Obstacle

Shape	Box
Center	X 0 Y 0 Z 0
Size	X 1 Y 1 Z 1
Carve	✓
Move Threshold	0.1
Time To Stationary	0.5
Carve Only Stationary	✓

The Shape, Center, and Size properties let you create an obstacle that only partially blocks NavMesh Agents. If you have an oddly shaped GameObject, you can add several NavMesh Obstacle components to create a few different holes in the NavMesh.

If you leave all of the options at their default settings, you get an obstacle that the NavMesh Agent can't get through. Instead, the Agent hits it and stops. **Check the Carve box**—this causes the obstacle to ***create a moving hole in the NavMesh*** that follows the GameObject. Now your Moving Obstacle GameObject can block the player from navigating up and down the ramp. Since the NavMesh height is set to 3, if the obstacle is less than 3 units above the floor it will create a hole in the NavMesh underneath it. If it goes above that height, the hole disappears.

The Unity Manual has thorough—and readable!—explanations for the various components. Click the Open Reference button (?) at the top of the Nav Mesh Obstacle panel in the Inspector to open up the manual page. Take a minute to read it—it does a great job of explaining the options.

Add a script to move the obstacle up and down

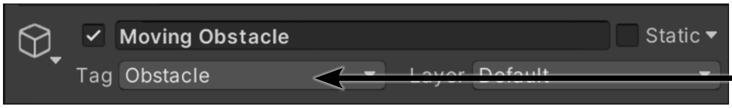
This script uses the **OnMouseDrag** method. It works just like the OnMouseDown method you used in the last lab, except that it's called when the GameObject is dragged.

You used Input.GetAxis earlier to use the scroll wheel. Now you're using the mouse's up-down movement—along the Y axis—to move the obstacle by modifying its Y position.

```
public class MoveObstacle : MonoBehaviour
{
   void OnMouseDrag()
   {
      transform.position += new Vector3(0, Input.GetAxis("Mouse Y"), 0);
      if (transform.position.y < 1) {
         transform.position = new Vector3(transform.position.x, 1, transform.position.z);
      }
      if (transform.position.y > 5) {
         transform.position = new Vector3(transform.position.x, 5, transform.position.z);
      }
   }
}
```

The first `if` statement keeps the block from moving below the floor, and the second keeps it from moving too high. Can you figure out how they work?

Drag your script onto the Moving Obstacle GameObject and run the game—uh-oh, something's wrong. You can click and drag the obstacle up and down, but it also moves the player. Fix this by **adding a tag** to the GameObject.



☑ Moving Obstacle □ Static
Tag Obstacle Layer Default

Set the tag for the obstacle just like you did in the last lab, but this time choose "Add tag..." from the dropdown, then use the + button to add a new tag called Obstacle. Now you can use the dropdown to assign the tag to the GameObject.

Then **modify your MoveToClick script** to check for the tag:

```
      if (Physics.Raycast(ray, out hit, 100))
      {
         if (hit.collider.gameObject.tag != "Obstacle")
         {
            agent.SetDestination(hit.point);
         }
      }
```

hit.collider contains a reference to the object that the ray hit.

Run your game again. If you click on the obstacle you can drag it up and down, and it stops when it hits the floor or gets too high. Click anywhere else, and the player moves just like before. Now you can **experiment with the NavMesh Obstacle options** (this is easier if you reduce the Speed in the Player's NavMesh Agent):

- Start your game. Click on *Moving Obstacle* in the Hierarchy window and **uncheck the Carve option**. Move your player to the top of the ramp, then click at the bottom of the ramp—the player will bump into the obstacle and stop. Drag the obstacle up, and the player will continue moving.
- Now **check Carve** and try the same thing. As you move the obstacle up and down, the player will recalculate its route, taking the long way around to avoid the obstacle if it's down, and changing course in real time as you move the obstacle.

there are no Dumb Questions

Q: How does that MoveObstacle script work? It's using += to update transform.position—does that mean it's using vector arithmetic?

A: Yes, and this is a great opportunity to understand vector arithmetic better. Input.GetAxis returns a number that's positive if the mouse moves up and negative if the mouse moves down (try adding a Debug.Log statement so you can see its value). The obstacle starts at position (–5.75, 1, –1). If the player moves the mouse up and GetAxis returns 0.372, the += operation adds (0, 0.372, 0) to the position. That means it **adds both of the X values** to get a new X value, then does the same for the Y and Z values. So the new Y position is 1 + 0.372 = 1.372, and since we're adding 0 to the X and Z values, only the Y value changes and it moves up.

Get creative!

Can you find ways to improve your game and get practice writing code? Here are some ideas to help you get creative:

- Build out the scene—add more ramps, stairs, platforms, and obstacles. Find creative ways to use materials. Search the web for new texture maps. Make it look interesting!
- Make the NavMesh Agent move faster when the player holds down the Shift key. Search for "KeyCode" in the Scripting Reference to find the left/right Shift key codes.
- You used OnMouseDown, Rotate, RotateAround, and Destroy in the last lab. See if you can use them to create obstacles that rotate or disappear when you click them.
- We don't actually have a game just yet, just a player navigating around a scene. Can you find a way to **turn your program into a timed obstacle course**?

This is your chance to experiment. Using your creativity is a really effective way to quickly build up your coding skills.

You already know enough about Unity to start building interesting games—and that's a great way to get practice so you can keep getting better as a developer.

BULLET POINTS

- When you bake the NavMesh, you can specify a **maximum slope and step height** to let NavMesh Agents navigate ramps and stairs in the scene.
- You can also **specify the agent height** to create holes in the mesh around obstacles that are too low for the agent to get around.
- When a NavMesh Agent moves a GameObject around a scene, it will **avoid obstacles** (and, optionally, other NavMesh Agents).
- The label under the Scene Gizmo shows an icon to indicate if it is in **perspective** mode (distant objects look smaller than near objects) or **isometric** mode (all objects appear the same size no matter how far away they are). You can use this icon to toggle between the two views.
- The **transform.LookAt method** makes a GameObject look at a position. You can use it to make the camera point at a GameObject in the scene.
- Calling **Input.GetAxis("Mouse ScrollWheel")** returns a number (usually between –0.4 and 0.4) that represents how much the scroll wheel moved (or 0 if it didn't move).
- Calling **Input.GetAxis("Mouse Y")** lets you capture mouse movements up and down. You can combine it with OnMouseDrag to move a GameObject with the mouse.
- Add a **NavMesh Obstacle** component to create obstacles that can carve moving holes in the NavMesh.
- The Input class has methods to capture input during the Update method, like **Input.GetAxis** for mouse movement and **Input.GetKey** for keyboard input.