Tutorial – Physics in Unity

In this tutorial you'll be provided with the framework of a simple platformer game level, which you'll finish off by adding physics and relevant scripts.

Activity 0 - Get and load the tutorial project:

The starting files for this tutorial are available via the student portal. Download and extract the project, then open Scenes/TutorialLevel the Unity Editor.

The scene already has a number of objects in it. Everything is initially set up as static colliders - Collider components are present, but nothing has a Rigidbody. As a result, playing the scene doesn't do anything.

Activity 1 - Basic jumping:

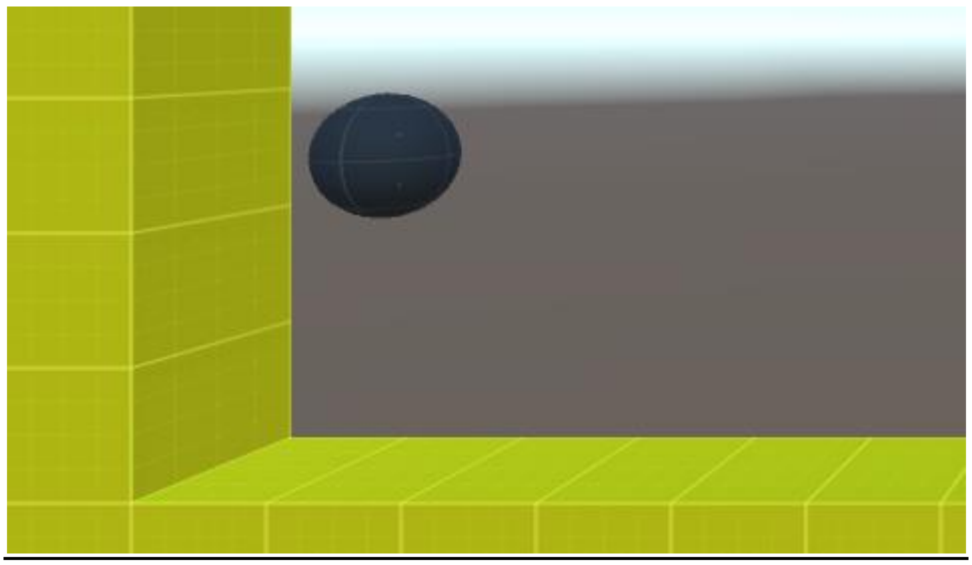


Figure 1: Our player object

Lets start by giving our intended player object, the blue sphere, the ability to jump.

1. Start by adding a Rigidbody component to the "PlayerBall" GameObject.
2. Open Scripts/PlayerBallController.cs. This script is currently empty.
3. We need access to our Rigidbody component. To achieve this, create a private variable m_Rigidbody and populate it in Start() using GetComponent<Rigidbody>().
4. Make a private variable m_JumpInput.
5. In Update(), use Input.GetButtonDown(...) to read the state of the "Jump" input into m_JumpInput.
6. In FixedUpdate(), if m_JumpInput is true then:
    a. apply an upwards force (you can use "Vector3.up") to your Rigidbody using ForceMode.Impulse., then...

   b. ... set m_JumpInput to false. *Note: As discussed in the lecture, input is synchronised with Update() but not with FixedUpdate(). This approach makes sure that our jumping behaves consistently.*

7. Attach your PlayerBallController script to the "PlayerBall" GameObject.
8. Run the game and try jumping. The ball should move, but it probably won't move much.
9. Add a public float m_JumpForce to your script, and multiply this into your AddForce(...).
10. Use this to tune your jump height, running your scene as required to test. For this level the player should be able to jump over two blocks high, but not over three. With the default physics settings, this will be a value of around 7.

Activity 2 - Only jump from the ground:

You might have noticed that we can jump again while we're still in the air. That's not how platformers usually work, so lets check that we're near the ground before we add our jump force.

1. In PlayerBallController.cs, add a new public float m_JumpCheckHeight.
2. In FixedUpdate(), just before using AddForce(...), we want to do a raycast directly downwards, for the distance specified by m_JumpCheckHeight.
3. Then, only call AddForce(...) if that raycast hits something.
4. Tune your m_JumpCheckHeight so that jumping feels reliable. You'll want a value that is slightly larger than your SphereCollider's radius, without being too big. A value of 0.55f seems to work well. Run your scene as required to test.

Activity 3 - Sideways movement:

Jumping is a great start, but it's not much use without also being able to move from side to side. Lets do that next.

1. In PlayerBallController.cs, make a public float called m_MoveForce and a private float called m_MoveInput.
2. In Update(), use Input.GetAxis(...) to read horizontal input into m_MoveInput.
3. In FixedUpdate, use AddForce(...) with ForceMode.Acceleration to accelerate your Rigidbody sideways. Make sure you use both m_MoveInput and m_MoveForce for this.
4. Tune your m_MoveForce to taste, running the scene as required to test.

Activity 4 - Pushing a crate:

Being able to push objects around is common in games. There's already a brown box in our scene, so lets make it useful.

1. Select the "Crate" GameObject in the scene.
2. Attach a Rigidbody component to it.
3. Run the scene, and try to push it by rolling your PlayerBall into it. Results will vary depending on the force behind your ball's movement. We want the crate to be easily pushed, so give it a low mass such as 0.1.

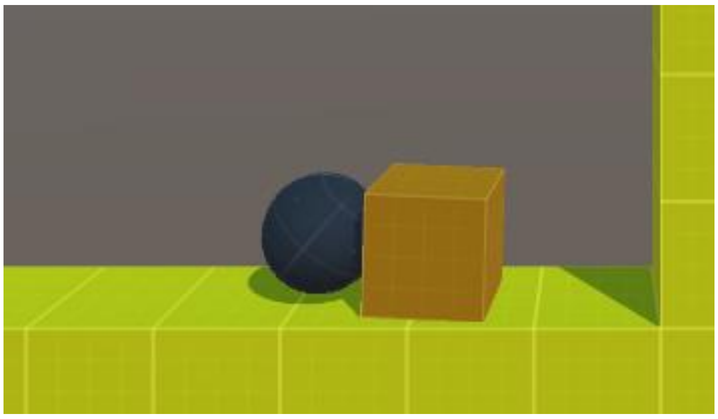


Figure 2: The PlayerBall pushing the Crate

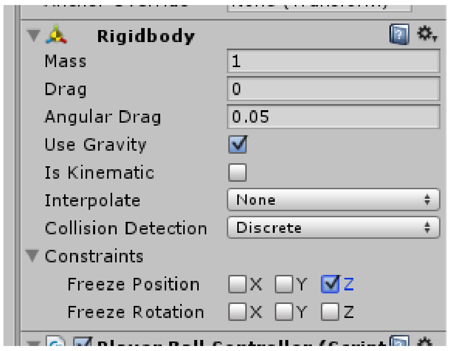
Activity 5 - Stop objects from falling off the level:

If you run into the Crate a few times you'll probably notice that it's fairly easy for either the Crate or the PlayerBall to fall off the front or back of our platforms. There are two ways we can deal with this.

Approach one: Rigidbody constraints. The easiest method is to constrain the movement of the Rigidbody components.

1. Select the "PlayerBall" GameObject.
2. Look at its Rigidbody in the Inspector.
3. Tick the box COnstraints -> Freeze Position -> Z.
4. Repeat for the Crate.

This tells the physics system not to allow movement on the Z axis as a result of forces. It's a great and easy solution, as long as our game only involves movement along one of the world axes.



Rigidbody	
Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	☑
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	☐ X ☐ Y ☑ Z
Freeze Rotation	☐ X ☐ Y ☐ Z

Figure 3: A Rigidbody component constrained on the Z axis

Approach two: "Invisible walls". This takes a little more work, as you'll need to create the colliders and test that they're working as desired. It allows you significantly more flexibility, though.

1. Create a box with GameObject -> 3D Object -> Cube.
2. Position the cube behind the platforms.
3. Scale it so that it covers the whole back of the level.
4. Disable or remove its Mesh Renderer component.
5. Repeat, creating another collider for the front of the platforms.

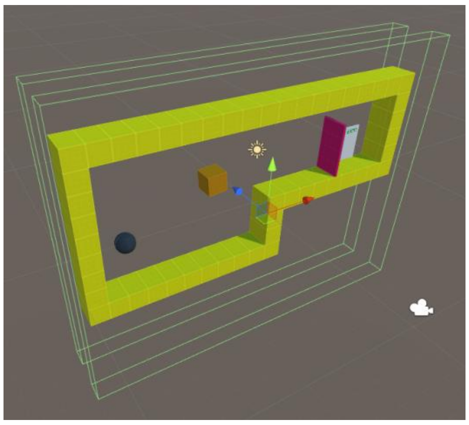


Figure 4: Invisible wall colliders in front of and behind the platforms

Activity 6 - Pulling objects:

Once our Crate gets in a corner it's stuck there, because we can't get on the other side to push it out. To help with this lets give our ball a "pull" ability.

1. In PlayerBallController.cs make two public floats called m_PullForce and m_PullRange, and a private bool called m_PullInput.
2. In Update(), use Input.GetButton(...) to read the state of "Fire1" into m_PullInput.
3. In FixedUpdate(), check if m_PullInput is true. If it is, then...
    - *a.* Use Physics.OverlapSphere(...) and m_PullRange to get an array of all nearby Colliders. *Hint: Look up Physics.OverlapSphere(...) in the Scripting Reference.*
    - b. Use a loop to look at each Collider in that array. For each one that has a Rigidbody...
        - i. Calculate the direction towards the player by subtracting the player's Rigidbody's position from the collider's Rigidbody's position.
        - ii. Use AddForce(...) with ForceMode.Acceleration and m_PullForce to accelerate the Rigidbody towards the PlayerBall.
4. Tune the m_PullForce and m_PullRange variables to taste, running the scene as needed to test. Good starting values are 10 and 3, respectively.

Activity 7 - Opening a door:

With our abilities so far we're now able to get up to the higher platform on the right side of the level. We can't get to the exit, though, because the red door is blocking our path. So lets add the ability to open doors.

This has two parts. First we need to add the ability to open and close to the door. Next, we need to set up a trigger that causes it to open.

1. Find Scripts/BlockerDoor in the Project panel, and attach it to the "BlockerDoor" GameObject.
2. Open the BlockerDoor.cs script.
3. In Update(), write some code that moves the door towards m_OpenPosition when m_Open is true, and towards m_ClosedPosition if m_Open is false.
4. The door has the following requirements:
    - It needs to be collidable
    - It needs to be able to move
    - The player should not be able to push it
    - We intend to move it via its Transform (that this is not strictly necessary in this example)

    Referencing the lecture notes or the Unity Manual, we see that we should have a Rigidbody with isKinematic set to true. (Note: This will sometimes work with no Rigidbody at all, but can be unreliable.)

Now we've got a door that we can open. Finally, we set up a trigger to do so.

5. Next, we need to make a trigger to open the door. GameObject -> 3D Object -> Cube. Give it a useful name. Put it on the left side of the "BlockerDoor" GameObject, and make it reasonably large. Set "isTrigger" on its Collider to true. Disable or remove its Mesh Renderer.
6. Create a new script, "BlockerDoorTrigger.cs".
7. Remove the Start() and Update() methods from this script, as we do not need them.
8. Add a public variable of type BlockerDoor called m_BlockerDoor.
9. Create a method OnTriggerEnter(Collider other) {...}. This will be called by Unity any time a Rigidbody enters the trigger collider.
10. In your OnTriggerEnter(...) method, call the BlockerDoor's Open() method. *Note: As we have a reference to the Collider which entered the trigger we could be more selective about what is allowed to open the door. For now we're happy to let any Rigidbody do it.*
11. Attach this script to your trigger's GameObject and assign its public fields in the Inspector.
12. Run the scene and enter the trigger to open the door.

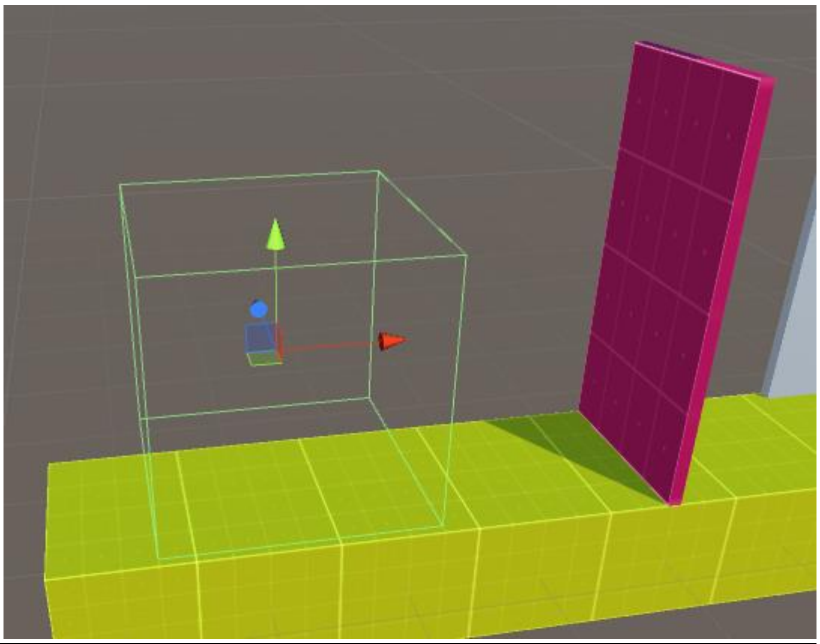


Figure 5: Potential positioning and size of a BlockerDoorTrigger GameObject

Activity 8 - Win your level! :-)

Now you can get to the exit and win the game!

As you have probably noticed, creating physics based gameplay can involve *a lot* of tuning of values. This process often requires careful design consideration and quite a number of iterations. Thankfully it also gets easier and faster with practice, and physics systems give us a lot of power and flexibility in our games.

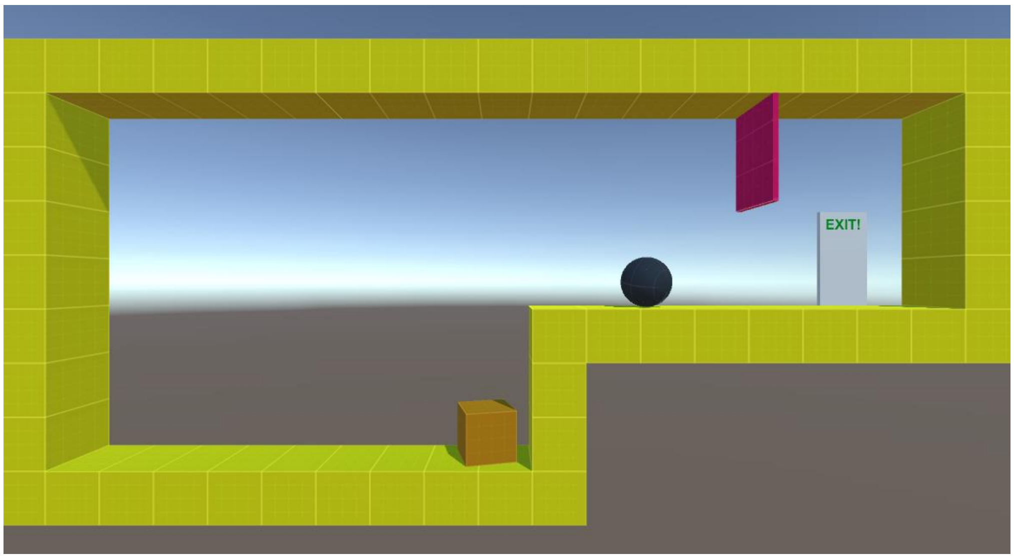


Figure 6: A PlayerBall on its way to the exit. :-)