BUTTON SMASH GAMES

Ground Checking Kit

Table of Contents

Getting Started

What is this kit?

This is our comprehensive ground checking kit inside of Unity. The aim of this kit is to save developers time without having to make sacrifices to limit yourself to non-modular tools. In other words, this kit is extremely flexible and can save you and others so much time when creating games. Have you ever started writing your own character controller and realised that you have to detect whether or not your player is grounded? Or are you sick of being restricted to using Unity's CharacterController component? Whatever the case may be, writing your own ground checking solution every time you make a game can take up unnecessary time. Our ground checking kit takes care of all that for you, just add one component to your player GameObject, set it up and you're ready to start finding some ground!

It supports any kind of game as it uses an interface implementation. This means you can use the same component for 2D and 3D games! The package currently provides 4 unique implementations for ground checking, which we call ground checkers. There are two for 3D games and two for 2D games. It's really as simple as plug-and-play! And don't forget you can make your own *ground checkers* too! For more info check out the Creating Custom Ground Checkers section.

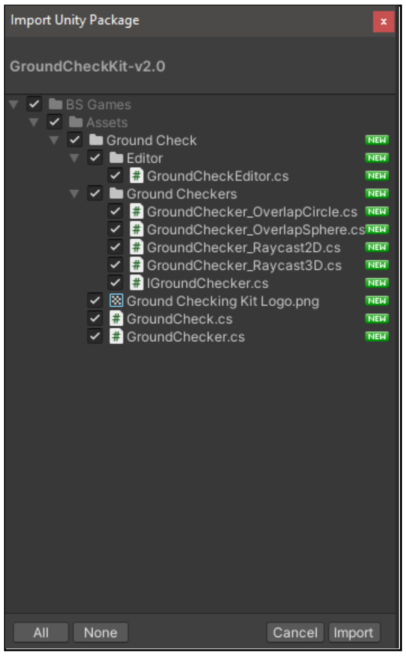
What's Included in the kit?

This kit includes the **GroundCheck** component itself, alongside this documentation. It also includes **4** *ground checkers* which you can use in your projects or as an example to write your own implementations. These are the four implementations included in the package:

- 2D Raycast
  - Uses a simple raycast in 2D space with the specified direction and range to detect a surface
- 2D Overlap Circle
  - Uses a slightly more complex approach of checking all colliders hitting an invisible circle, optionally you can choose a list of GameObject tags to ignore e.g. "Player"
- 3D Raycast
  - Uses a simple raycast in 3D space with the specified direction and range to detect a surface
- 3D Overlap Sphere
  - Uses a more complex approach of checking all hits within an invisible sphere, optionally ignoring any GameObjects with specified tags e.g. "Player"

Importing to your Unity project

The package is very easy to import. All you have to do is drag the .unitypackage file into your Unity assets folder and open Unity (you can do this while Unity is open too!). Then you will get a dialog box which looks something like this:


Import Unity Package

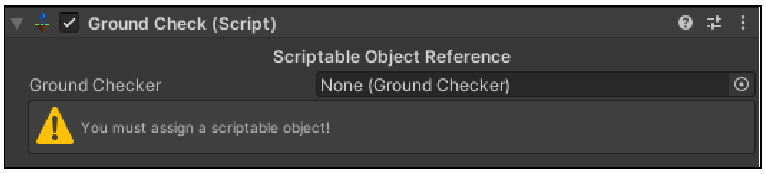
GroundCheckKit-v2.0

```
BS Games
  Assets
    Ground Check                            NEW
      Editor                                NEW
        GroundCheckEditor.cs                NEW
      Ground Checkers                       NEW
        GroundChecker_OverlapCircle.cs      NEW
        GroundChecker_OverlapSphere.cs      NEW
        GroundChecker_Raycast2D.cs          NEW
        GroundChecker_Raycast3D.cs          NEW
        IGroundChecker.cs                   NEW
      Ground Checking Kit Logo.png          NEW
      GroundCheck.cs                        NEW
      GroundChecker.cs                      NEW
```

Simply click **Import** and let Unity take care of the rest!

Usage

Setup



Ground Check (Script)

Scriptable Object Reference

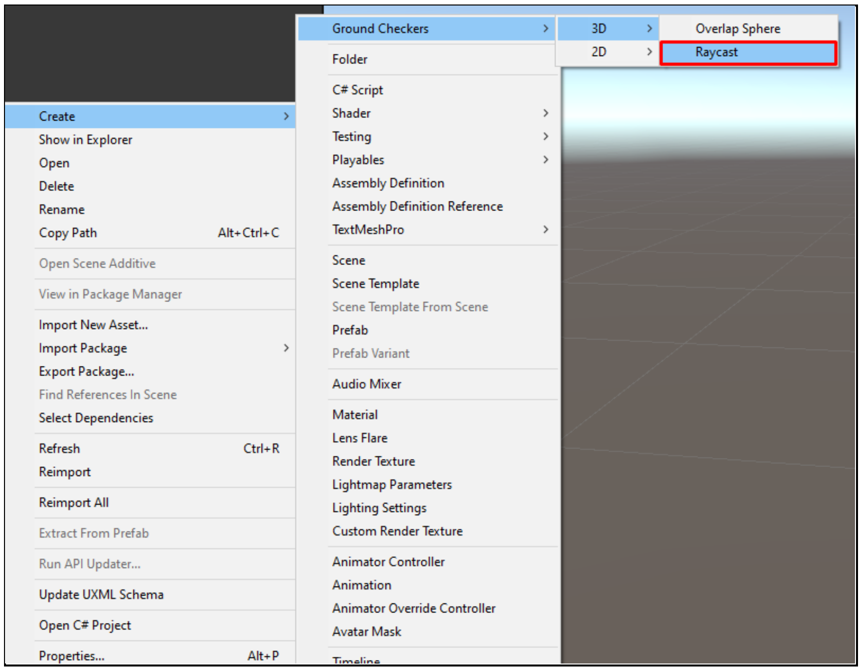
Ground Checker None (Ground Checker)

You must assign a scriptable object!

Ground Check component before setting up

Setting up the component isn't too complicated either! Start by adding the **Ground Check** component to your player GameObject (or other object). The component works by executing code from a special type of ScriptableObject called a **GroundChecker**. We will look more into this later when we learn how to create our own ground checkers! As previously mentioned, this package comes with 4 ground checkers already included. So how do we use them?

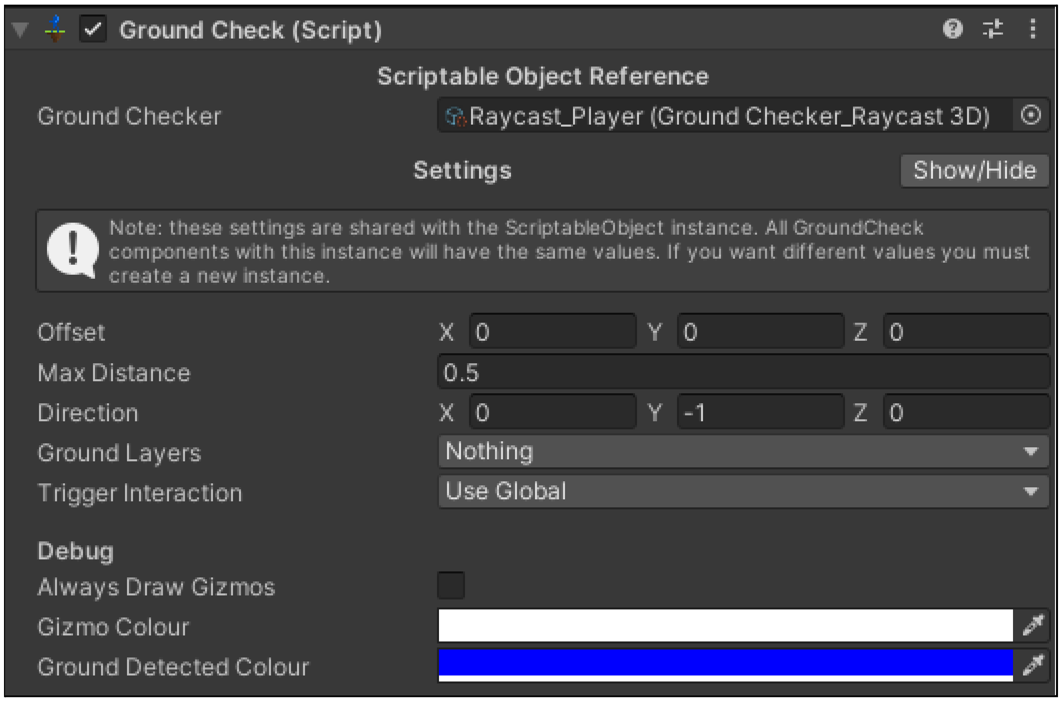
Firstly, you must create an instance of the **GroundChecker** object. We should be organised when we do this, so let's start by creating a folder called "Ground Checkers" in your assets folder. Next, create an instance of the **GroundChecker** object like so:



Creating an instance of the Ground Checker

Note: You will have to create a new instance of the **GroundChecker** for each variant. This is because all of the settings are stored within the ScriptableObject. This allows you to reuse the same settings across multiple different GameObjects.

I suggest calling this something like “[GC_Type]_Player”, in this case “Raycast_Player”. Next head back to your GameObject e.g. your player, and assign the ScriptableObject variable. You need to click **Show/Hide** to show the settings for the **GroundChecker**. It should look something like this:



Ground Check (Script)

Scriptable Object Reference

Ground Checker: Raycast_Player (Ground Checker_Raycast 3D)

Settings

Show/Hide

Note: these settings are shared with the ScriptableObject instance. All GroundCheck components with this instance will have the same values. If you want different values you must create a new instance.

Offset: X 0 Y 0 Z 0

Max Distance: 0.5

Direction: X 0 Y -1 Z 0

Ground Layers: Nothing

Trigger Interaction: Use Global

Debug

Always Draw Gizmos: ☐

Gizmo Colour: ⬜

Ground Detected Colour: 🟦

Ground Checker (3D Raycast) Settings

Feel free to tweak the settings now and make it suit your game. In this case, the **GroundCheck** component would detect the ground using a 3D raycast. Read on to learn how to use this in your scripts!

Code

The coding part is super easy - all you have to do is create a reference to your **GroundCheck** component. There are many ways to do this, here is one example:

```
public class DemoCharacterController : MonoBehaviour
{
    // Let’s create a reference to a Rigidbody component so we can move!
    public Rigidbody rbody = null;
    // This is our reference to the GroundCheck component - remember to
    // assign the value in the inspector
    public GroundCheck groundCheck = null;

    private void Update()
    {

    }

}
```

Now let’s see how to check if we’re on the ground.

```
public class DemoCharacterController : MonoBehaviour
{

    // Let’s create a reference to a Rigidbody component so we can move!
    public Rigidbody rbody = null;
    // This is our reference to the GroundCheck component - remember to
    // assign the value in the inspector
    public GroundCheck groundCheck = null;

    private void Update()
    {
        if (groundCheck.IsGrounded())
            Debug.Log(“We’re on the ground!”);
    }

}
```

It's that easy! Now let's make our player jump! Of course, you can add whatever you want from here, but we're going to show just a simple example to get you started.

```
public class DemoCharacterController : MonoBehaviour
{

    // Let's create a reference to a Rigidbody component so we can move!
    public Rigidbody rbody = null;
    // This is our reference to the GroundCheck component - remember to
    // assign the value in the inspector
    public GroundCheck groundCheck = null;

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space) && groundCheck.IsGrounded())
            Jump();
    }

    private void Jump()
    {
        Vector3 newVelocity = Vector3.up * 5.0f;
        rbody.velocity = newVelocity;
    }

}
```

Creating Custom Ground Checkers

We've tried to make this process as intuitive as possible. If you get stuck, or have any ideas on how we could improve please let us know. You can contact us here: https://www.buttonsmashgames.org/contact

Now that's out of the way, let's begin. First you need to create a new C# class which inherits from **GroundChecker**. You need to make sure you implement the abstract class. It should look something like this:

Note: You must add the [CreateAssetMenu()] attribute preceding your class so that the ScriptableObject can be instantiated.

```
using BSGames.Modules.GroundCheck;
using UnityEngine;

[CreateAssetMenu(menuName = "Ground Checkers/My Custom Ground Checker")]
public class GroundChecker_MyGroundChecker : GroundChecker
{

    public override bool IsGrounded()
    {
        return true;
    }

}
```

Currently the **IsGrounded()** method always returns true. This would mean that we are always considered to be on the ground. That isn't great, so now you need to write your own implementation to figure out when you should be considered on the ground or not.

We have provided various methods which you can override to help you create your own **GroundChecker**. Additionally, you can use the **transform** and **gameObject** variables to access the Transform component (or GameObject) that the **GroundCheck** component is attached to.

We recommend taking a look at how we created the default **GroundCheckers** to help you with your learning. Now good luck with your creations and we hope our Ground Checking Kit has helped you!

See below for a full list of variables and methods accessible from within your **GroundChecker**.

```
// The transform component attached to the Ground Check component.
protected Transform transform = null;
// The GameObject that the Ground Check component is attached to.
protected GameObject gameObject = null;

// Helpful Methods
// Called in OnEnable on the GameObject with the GroundCheck component.
public virtual void OnGroundCheckerEnabled() {  }
// Called in OnDisable() on the GameObject with the GroundCheck component.
public virtual void OnGroundCheckerDisabled() {  }
// Called in IsGrounded() on the GameObject with the GroundCheck component.
public abstract bool IsGrounded();
// Called in OnDrawGizmos() on the GameObject with the GroundCheck component.
public virtual void DrawGizmos() {  }
// Called in OnDrawGizmosSelected() on the GameObject with the GroundCheck
component.
public virtual void DrawGizmosSelected() {  }

// Collider 3D Methods
// Called in OnCollisionEnter() on the GameObject with the GroundCheck component.
public virtual void OnCollisionEnter(Collision other) {  }
// Called in OnCollisionStay() on the GameObject with the GroundCheck component.
public virtual void OnCollisionStay(Collision other) {  }
// Called in OnCollisionExit() on the GameObject with the GroundCheck component.
public virtual void OnCollisionExit(Collision other) {  }
// Called in OnTriggerEnter() on the GameObject with the GroundCheck component.
public virtual void OnTriggerEnter(Collider other) {  }
// Called in OnTriggerStay() on the GameObject with the GroundCheck component.
public virtual void OnTriggerStay(Collider other) {  }
// Called in OnTriggerExit() on the GameObject with the GroundCheck component.
public virtual void OnTriggerExit(Collider other) {  }


// Collider 2D Methods
// Called in OnCollisionEnter2D() on the GameObject with the GroundCheck
component.
public virtual void OnCollisionEnter2D(Collision2D other) {  }
```

```
// Called in OnCollisionStay2D() on the GameObject with the GroundCheck
component.
public virtual void OnCollisionStay2D(Collision2D other) {  }
// Called in OnCollisionExit2D() on the GameObject with the GroundCheck
component.
public virtual void OnCollisionExit2D(Collision2D other) {  }
// Called in OnTriggerEnter2D() on the GameObject with the GroundCheck component.
public virtual void OnTriggerEnter2D(Collider2D other) {  }
// Called in OnTriggerStay2D() on the GameObject with the GroundCheck component.
public virtual void OnTriggerStay2D(Collider2D other) {  }
// Called in OnTriggerExit2D() on the GameObject with the GroundCheck component.
public virtual void OnTriggerExit2D(Collider2D other) {  }
```