Runtime Collider & Light Gizmos

Table of Contents

Introduction

Runtime Collider & Light Gizmos is a plugin for Unity that allows you to integrate collider and light gizmos into your application. This type of functionality is necessary when working on level editor applications or modding tools for your games and you would like to allow users to change collider and light properties in the same way as can be done inside the Unity Editor.

The plugin implements gizmos for the following entities:

- **3D Box Colliders**
- **3D Capsule Colliders**
- **3D Character Controllers**
- **Sphere Colliders**
- **Point Lights**
- **Spot Lights**
- **Directional Lights**

Bonus features include:

- **Free Move Camera**
- **Scene Grid**
- **Undo/Redo**

This document serves as a quick introduction to the plugin API and it will get you up to speed with the most essential tasks such as creating gizmos, assigning target colliders and lights, enabling/disabling gizmos, gizmo settings and a few others.

Don't forget to download the demo from here.

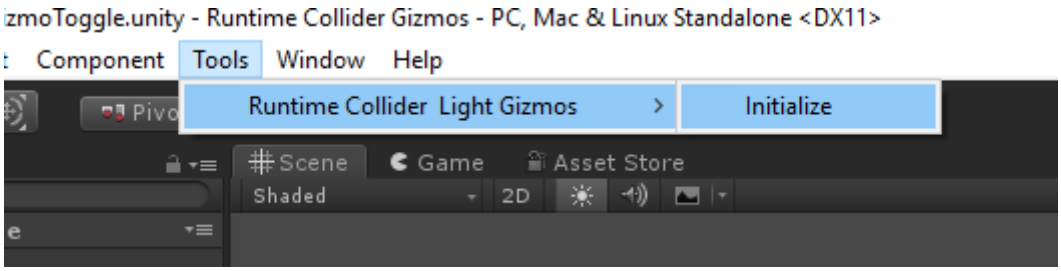
Also, if you'd like to take a quick look at how the gizmos work, please check this video.

For any questions or suggestions, you can contact me at octamodius@yahoo.com

Setup

In order to start using the plugin, you will have to perform the necessary steps:

1. import the plugin into your project;
2. on the top menu, go to **Tools->Runtime Collider & Light Gizmos->Initialize** as shown in the image below:



3. in the hierarchy view, you will notice a hierarchy of objects that were created for you automatically. The root of the hierarchy is called **RTGApp**. This hierarchy with all its children needs to be in the scene. **Note:** All objects in the hierarchy need to be present in the scene, so you should not delete them.
4. in the **RTGApp** hierarchy there is a child object called **RTFocusCamera**. Select it, and make sure that the **Target camera** field points to a camera in the scene. If the scene contains a camera tagged as **Main Camera**, it will automatically be assigned to this field. Otherwise, you will have to assign a camera;
5. you can now use the plugin.

Integration with Runtime Transform Gizmos

RCLG can be integrated with the Runtime Transform Gizmos plugin. **Note:** it is important to follow the integration steps in the exact order as they are listed.

1. make sure you start with a clean slate in your project. That is, make sure that none of the 2 plugins are present in your project. If they are, delete both;
2. import **RCLG** into your project;
3. delete the **Common** folder. The folder can be found at the following path: **Runtime Collider & Light Gizmos/Scripts**;
4. import the **Runtime Transform Gizmos** pack into your project;
5. in the top menu, go to **Tools->Runtime Transform Gizmo->Initialize**;
6. the 2 plugins can now be treated as one.

Creating Gizmos

In this section we are going to cover the necessary steps for creating gizmos. We are going to create a gizmo of each type. **Note:** we are going to assume that this code is part of a **MonoBehaviour** script and that the working scene contains objects with the relevant components attached to them (colliders and lights).

Note: The C# script associated with this tutorial is **Tutorial_0_CreatingGizmos.cs** and can be found in the **Tutorials** folder.

Let's first create a 3D box collider gizmo:

```
// Box collider
Gizmo gizmo = RTGizmosEngine.Get.CreateGizmo();
BoxColliderGizmo3D boxColliderGizmo3D = gizmo.AddBehaviour<BoxColliderGizmo3D>
();
boxColliderGizmo3D.SetTargetCollider(cubeObject.GetComponent<BoxCollider>());
```

That's it. Assuming the scene contains a game object called **cubeObject** that has a box collider attached to it, when you run the app, a box collider gizmo will appear at runtime and you can use it to change the box collider size.

Before moving on, let's take a look at the code above and talk about some of the things that might have caught your eye. First of all, notice that we first call the **CreateGizmo** function of the **RTGizmosEngine** singleton. The return type is **Gizmo**. You can think of a gizmo as the equivalent of Unity's **GameObject** class. The actual box collider gizmo is implemented as a gizmo behaviour. So the next thing we do, is call the **AddBehaviour** function of the **Gizmo** class to create a box collider gizmo.

Finally, once we have a reference to the box collider gizmo, we call **SetTargetCollider** to tell the gizmo about the collider that it will be operating on.

Let's go ahead and create gizmos for the remaining collider types. We'll also create a gizmo for a character controller:

```
// Sphere collider
gizmo = RTGizmosEngine.Get.CreateGizmo();
SphereColliderGizmo sphereColliderGizmo =
gizmo.AddBehaviour<SphereColliderGizmo>();
sphereColliderGizmo.SetTargetCollider(sphereObject.GetComponent<SphereCollider>
());

// Capsule collider
gizmo = RTGizmosEngine.Get.CreateGizmo();
CapsuleColliderGizmo3D capsColliderGizmo =
gizmo.AddBehaviour<CapsuleColliderGizmo3D>();
capsColliderGizmo.SetTargetCollider(capsuleObject.GetComponent<CapsuleCollider>
());

// Character controller
gizmo = RTGizmosEngine.Get.CreateGizmo();
CharacterControllerGizmo3D cCtrlGizmo =
gizmo.AddBehaviour<CharacterControllerGizmo3D>();
cCtrlGizmo.SetTargetController(characterObject.GetComponent<CharacterController>
());
```

Finally, let's create gizmos for directional, point and spot lights:

```
// Directional light
gizmo = RTGizmosEngine.Get.CreateGizmo();
DirectionalLightGizmo3D dirLightGizmo =
gizmo.AddBehaviour<DirectionalLightGizmo3D>();
dirLightGizmo.SetTargetLight(dirLightObject.GetComponent<Light>());

// Point light
gizmo = RTGizmosEngine.Get.CreateGizmo();
PointLightGizmo3D ptLightGizmo = gizmo.Addbehaviour<PointLightGizmo3D>();
ptLightGizmo.SetTargetLight(ptLightObject.GetComponent<Light>());
```

```
// Spot light
gizmo = RTGizmosEngine.Get.CreateGizmo();
SpotLightGizmo3D spotLightGizmo = gizmo.Addbehaviour<SpotLightGizmo3D>();
spotLightGizmo.SetTargetLight(spotLightObject.GetComponent<Light>());
```

That's it. As you can see, the code is pretty much the same for all gizmo types. There are minor differences, but the idea remains the same.

Note: Regarding light gizmos, they will only show up if the light type matches the gizmo type. For example, a point light gizmo will not show up on the screen if the target light is directional.

The RTGApp.Initialized Event

In the previous section we covered gizmo creation. Before moving on, there is one thing left to mention. All gizmo creation should be performed after the **RTGApp** object has performed all the necessary initializations. This is done by registering an event handler to the **RTGApp.Initialized** event.

Let's see a simple example of this:

```
public class MyGizmoApp : MonoBehaviour
{
    private void Awake()
    {
        // Register event handler for the Initialized event
        RTGApp.Get.Initialized += OnAppInitialized;
    }

    private void OnAppInitialized()
    {
        // Create gizmos
        // ...
    }
}
```

Enabling/Disabling Gizmos

In order to enable or disable gizmos, you need to use the **SetEnabled** function of the **Gizmo** class.

Let's see 2 examples of this in action;

```
boxColliderGizmo3D.Gizmo.SetEnabled(false); // Disable box collider gizmo
sphereColliderGizmo.Gizmo.SetEnabled(true); // Enable sphere collider gizmo
```

Note: The C# script associated with this tutorial is **Tutorial_2_GizmoToggle.cs** and can be found in the **Tutorials** folder.

Gizmo Settings

Look & Feel, Settings and Hotkeys

There are 3 categories of settings associated with the gizmos:

- **Look & Feel** - controls the gizmo appearance;
- **Settings** - controls functional settings (e.g. snapping);
- **Hotkeys** - allows you to change the hotkeys that are associated with different types of states that can be activated for different gizmos (e.g. enable/disable snapping).

Reading settings values is done via properties. Settings values is done via **Set###** functions.

The property and function names are pretty much self-explanatory and are not listed here, but let's look at a simple example which shows how to change settings for a box collider gizmo.

```
// Change look and feel
boxColliderGizmo.LookAndFeel.SetWireColor(Color.red);
boxColliderGizmo.LookAndFeel.SetAllTicksColor(Color.blue);
boxColliderGizmo.LookAndFeel.SetMidCapVisible(false);

// Change the snap steps that apply when snapping is enabled. Note that this
// has to be done per axis.
boxColliderGizmo.Settings.SetXSizeSnapStep(1.0f);
boxColliderGizmo.Settings.SetYSizeSnapStep(2.0f);
boxColliderGizmo.Settings.SetZSizeSnapStep(0.5f);
boxColliderGizmo.Settings.SetUniformSizeSnapStep(1.5f);

// By default, snapping is enabled via the LCTRL key for all gizmos.
// In this example, let's disable the LCTRL key and use the 'V' key instead.
boxColliderGizmo.Hotkeys.EnableSnapping.LCtrl = false;
boxColliderGizmo.Hotkeys.EnableSnapping.Key = KeyCode.V;
```

Shared Gizmo Settings

It may be possible that you would like to have more than one gizmo of each type in the scene. In that case, when you want to change for example the look and feel of those gizmos, you would have to loop through all of them and set the correct property to the desired value. However, there is a better way to do this.

All gizmos have **Shared###** properties for look & feel, settings and hotkeys. Here is how you would go about supporting shared settings. In this example, we will assume that you have a list of sphere collider gizmos and all of them are visible in the scene. You want to allow the user to change the wire color for all of them.

Note: The C# script associated with this tutorial is **Tutorial_1_SharedSettings.cs** and can be found in the **Tutorials** folder.

First, you will have to create the settings objects:

```
private SphereColliderGizmoLookAndFeel _sharedLookAndFeel = new
SphereColliderGizmoLookAndFeel();
private SphereColliderGizmoSettings _sharedSettings = new
SphereColliderGizmoSettings();
private SphereColliderGizmoHotkeys _sharedHotkeys = new
SphereColliderGizmoHotkeys();
```

Next you would have to create the gizmos and link the shared settings:

```
private void InitGizmos()
{
    for (int gizmoIndex = 0; gizmoIndex < 10; ++gizmoIndex)
```

```
    {
        // Create the gizmo and store it in the gizmo list
        var gizmo = RTGizmosEngine.Get.CreateGizmo();
        var sphereColGizmo = gizmo.AddBehaviour<SphereColliderGizmo>();
        _sphereColGizmos.Add(sphereColGizmo);

        // Link the shared settings to the gizmo
        sphereColGizmo.SharedLookAndFeel = _sharedLookAndFeel;
        sphereColGizmo.SharedSettings = _sharedSettings;
        sphereColGizmo.SharedHotkeys = _sharedHotkeys;
    }
}
```

Now, let's suppose that the user wants to change the wire color of the gizmos from your app's interface. The function which handles the color change, could be written like this:

```
private void OnSphereColliderGizmoWireColorChanged(Color newColor)
{
    _sharedLookAndFeel.SetWireColor(newColor);
}
```

Note that instead of looping through each gizmo, we just set the property of the shared instance. Because this instance is shared among all sphere collider gizmos, all of them will use the new color.

The same rules apply for the other categories of settings.

Note: One important thing to keep in mind is that once you share settings like this, the **LookAndFeel**, **Settings** and **Hotkeys** properties will return the shared settings. For example:

```
// Link shared settings
sphereColGizmo.SharedLookAndFeel = _sharedLookAndFeel;
sphereColGizmo.SharedSettings = _sharedSettings;
sphereColGizmo.SharedHotkeys = _sharedHotkeys;

// NOTE: The LookAndFeel property will now return the shared instance.
// Changing its properties will actually affect all gizmos that use the same
shared
// settings.
sphereColGizmo.LookAndFeel.SetWireColor(Color.green);
sphereColGizmo.Settings.SetRadiusSnapStep(0.5f);        // Same ...
sphereColGizmo.Hotkeys.EnableSnapping.LShift = false;   // Same ...
```

If at some point you wish to make a gizmo unique, you can unplug the shared instance:

```
sphereColGizmo.SharedLookAndFeel = null;
```

From this point on, changing the look and fell for this gizmo will not affect the other gizmos.

Enabling/Disabling Snapping

Most gizmos support snapping. For example, by default, when you hold down the **Left Control** key, snapping will be activated and will allow you to adjust the collider or light properties in increments of a specified snap value that is stored in the gizmo **Settings**. Snapping is enabled or disabled automatically based on the state of the hotkeys.

However, there may be times when you want to enable/disable snapping based on certain conditions. In that case, you can use the **SetSnapEnabled** function:

```
boxColliderGizmo.SetSnapEnabled(isEnabled);
sphereColliderGizmo.SetSnapEnabled(isEnabled);
capsuleColliderGizmo.SetSnapEnabled(isEnabled);
charControllerGizmo.SetSnapEnabled(isEnabled);
ptLightGizmo.SetSnapEnabled(isEnabled);
spotLightGzmo.SetSnapEnabled(isEnabled);
```

Note: Even if you call this function passing **false** as parameter, snapping will still be activated when the snap enable hotkey is held down. If you wish to disable the hotkeys, you can do it like this:

```
boxColliderGizmo.Hotkeys.EnableSnapping.IsEnabled = false;
```

Just keep in mind that if the gizmo uses shared hotkeys, this will affect all other gizmos.

Enabling/Disabling Scale from Center

Collider gizmos support a mode called **scaling from center**. By default, when you click and drag one of the gizmo ticks, the size of the collider changes, but the opposite tick is used as a pivot, so the center of the collider changes. This is the behaviour of the gizmos used by the Unity Editor. These gizmos use a hotkey that allows you to change this behaviour so that when a tick is dragged, the scaling happens from the center of the collider.

As is the case with snapping, this scale from center behaviour can be enabled/disabled via a function call:

```
boxColliderGizmo.SetScaleFromCenterEnabled(isEnabled);
sphereColliderGizmo.SetScaleFromCenterEnabled(isEnabled);
capsuleColliderGizmo.SetScaleFromCenterEnabled(isEnabled);
charControllerGizmo.SetScaleFromCenterEnabled(isEnabled);
```

Note: Even if you call this function passing **false** as parameter, scaling from center will still be activated when the hotkey is held down. If you wish to disable the hotkeys, you can do it like this:

```
boxColliderGizmo.Hotkeys.ScaleFromCenter.IsEnabled = false;
```

Just keep in mind that if the gizmo uses shared hotkeys, this will affect all other gizmos.

The Free Move Camera

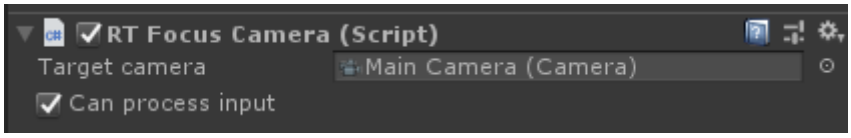
The free move camera included with the plugin behaves in a similar manner to the camera associated with the Unity Editor. The camera behaviour is implemented in the **RTFocusCamera** script. When you initialize the plugin, a **RTFocusCamera** object is automatically created for you and is part of the **RTGApp** hierarchy as shown in the image below:

The camera provides the following operations:

- move around;
- rotate/look around;
- orbit;
- pan;
- zoom;
- focus;
- projection switch;
- rotation switch;

When you select the camera object in the hierarchy view, the Inspector will allow you to change different settings that control the way in which the camera behaves.

There are 2 important fields here that are worth discussing:



- **Target camera** - this is filled out automatically when the plugin is initialized if there is a camera tagged **Main Camera** in the scene. If not, you will have to assign a camera here. This is a mandatory field. a camera has to be assigned here.
- **Can process input** - If you do not wish to use the camera functionality (e.g. rotate, move, orbit etc), then you can uncheck this toggle. You may want to do this if you have your own camera script that you would like to use instead. **Note:** Even if you decide to use your own camera script, the **RTFocusCamera** object needs to stay in the scene as part of the **RTGApp** hierarchy. It is needed by the gizmo engine.

Camera Focus

You can instruct the camera to focus on a particular object or group of objects in the same way as you can do inside the Unity Editor. The following code shows you how you can do this:

```
// Assume this is the list that contains the object on which the camera should
focus
List<GameObject> targetObjects = new List<GameObject>();

// Populate the list
// ...

// Focus
RTFocusCamera.Get.Focus(targetObjects);
```

Camera Rotation Switch

The camera supports an operation called **Rotation Switch**. A rotation switch aligns the camera to a specified rotation. It is what happens when you click on one of the scene gizmo cone axes in the top right corner of the scene view in the Unity Editor.

```
RTFocusCamera.Get.PerformRotationSwitch(targetRotation);
```

Camera Projection Switch

A projection switch happens when the camera switches from perspective to orthographic projection. This is the same behaviour as the one that can be observed when clicking on the scene gizmo cube in the top right corner of the scene view in the Unity Editor.

```
RTFocusCamera.Get.PerformProjectionSwitch();
```

Hotkeys

Gizmo Hotkeys

- **BoxColliderGizmo3D**
  - Enable Snapping - LCTRL
  - Enable Scale from Center - LSHIFT;
- **SphereColliderGizmo**
  - Enable Snapping - LCTRL
  - Enable Scale from Center - LSHIFT
- **CapsuleColliderGizmo3D**
  - Enable Snapping - LCTRL
  - Enable Scale from Center - LSHIFT
- **CharacterControllerGizmo3D**
  - Enable Snapping - LCTRL
  - Enable Scale from Center - LSHIFT
- **PointLightGizmo3D**
  - Enable Snapping - LCTRL
- **SpotLightGizmo3D**
  - Enable Snapping - LCTRL

Undo/Redo

- **Undo** - LCTRL + Z
- **Redo** - LCTRL + Y

Scene Grid

- **Move Up** -]
- **Move Down** - [
- **Snap to Cursor Pick Point** - LALT + Left Click

Camera

- **LMB** + **WASDQE** - move camera forward, left, backwards, right, down and up respectively;
- **LMB** + **MOUSE MOVE** - rotate to look around;
- **LMB** + **LALT** + **MOUSE MOVE** - orbit around focus point;
- **MMB** + **MOUSE MOVE** - pan;
- **MOUSE SCROLL WHEEL** - zoom in/zoom out;