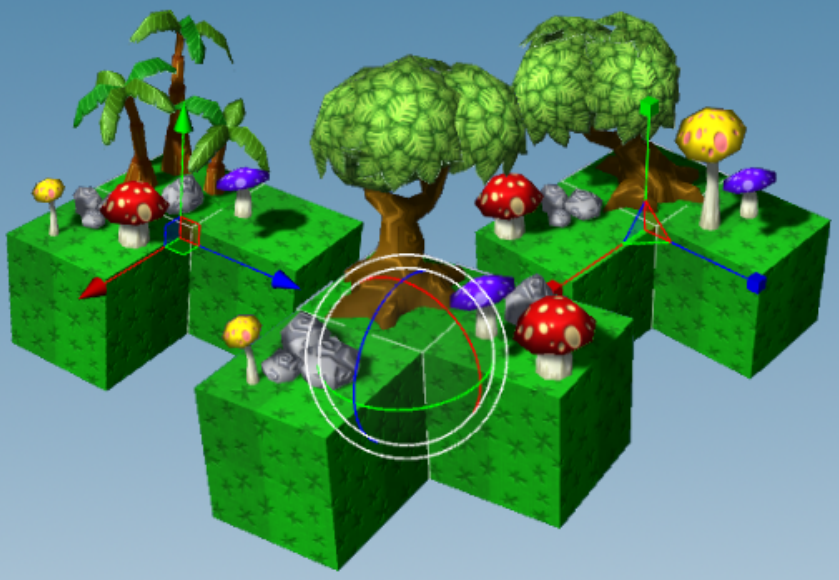


RUNTIME TRANSFORM GIZMOS

Table of Contents

1 Introduction

This document explains how you can start using the **Runtime Transform Gizmos** package. It provides information about all the gizmo properties and system settings and it also offers a couple of explanations related to scripting so that you can extend the system to suit your own needs.

The package is useful in case you are trying to build your own runtime editor application or a game in which you need to move, rotate and scale objects. Although, the main purpose of the package is to allow you to use transform gizmos at runtime, you also have access to object selection functionality camera manipulation and an Undo/Redo system.

2 What the package offers

This package gives you a starting point in case you need to build your own runtime editor application or an application in which the user can select and manipulate game objects. It gives you access to standard gizmo functionality, much like the one that you have access to inside the Unity Editor, but it also provides other types of functionality like object selection and an Undo/Redo system for basic operations.

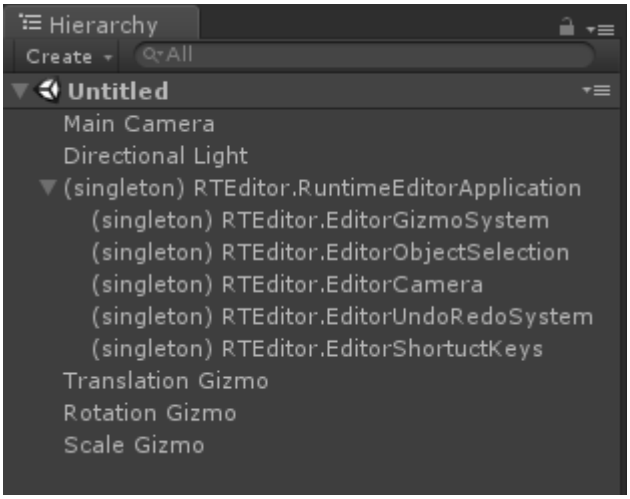
With this system you can:

- translate, rotate and scale objects using runtime gizmos;
- change the gizmo transform space (global or local) and pivot point (center or mesh pivot);
- perform snapping for all gizmo types (including vertex snapping for the translation gizmo);
- select objects in the scene using standard object selection functionality;
- assign/remove objects to/from selection masks (these allow you to specify which objects in the scene can not be selected);
- manipulate the camera:
  - ➔ pan – Standard and Smooth;
  - ➔ zoom – Standard and Smooth;
  - ➔ focus on object selection – Constant Speed, Instant and Smooth;
  - ➔ orbit around focus point.
- undo/redo different operations like object selection (select/deselect), object selection masks (assign/remove), object gizmo manipulation and a few properties like the gizmo transform space and pivot point;
- customize the object selection mechanism using handlers.

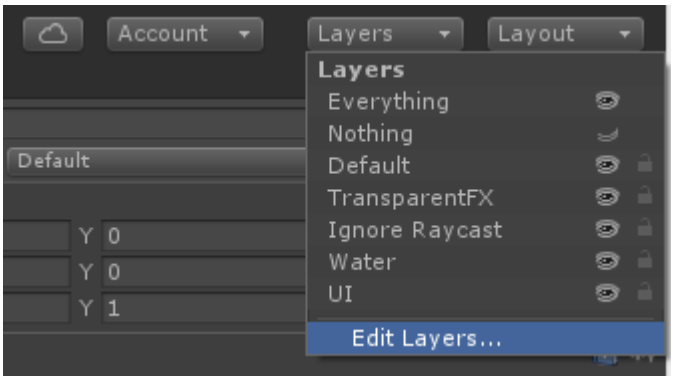
3 System Setup

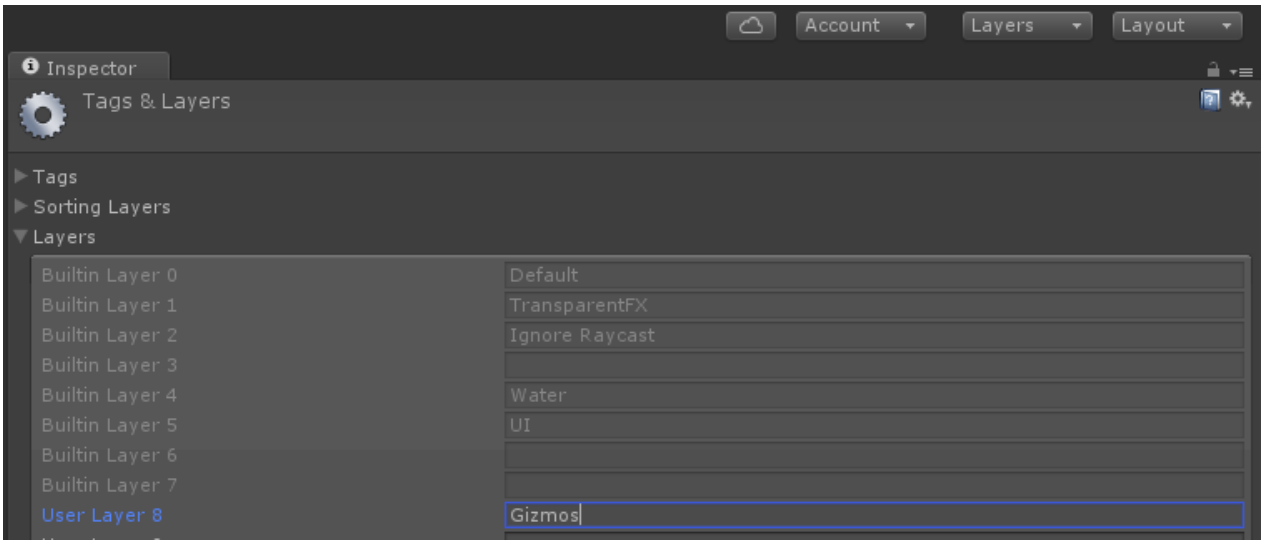
When you import the package, there are a couple of things that you must do in order to setup the system:

- import the **Runtime Transform Gizmos** package;
- in the top menu, click on **Runtime Editor Application → Create Subsystems.** At this point the hierarchy view should look something like in the following image:

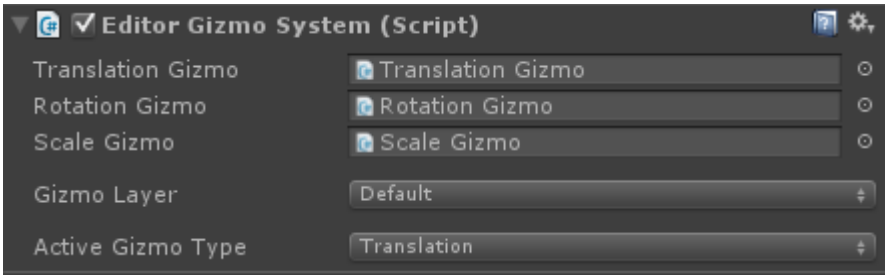


- Create a new layer named **Gizmos** (you can give it any name you want, but as we will see later this layer is used with the gizmo objects) by going to **Layers → Edit Layers:**





- Click the **RTEditor.EditorGizmoSystem** object in the hierarchy view. This will activate the following inspector GUI:



- Set the **Gizmo Layer** property to the layer which you have just created as shown in the image above. Don't worry about this for now. We will cover all the properties later.

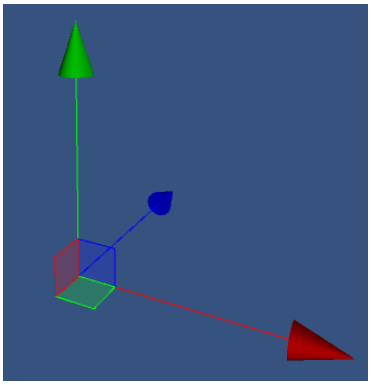
4 The Gizmos

In this chapter we will take a look at how the gizmos work and all the properties which can be modified for each gizmo. We will also talk about transform spaces and transform pivot points.

All gizmos behave like the ones that you have access to inside the Unity Editor with a few minor exceptions. For example, the scale gizmo allows you to scale along 2 axes at once using a set of scale triangles. Also, when you hover one of the gizmo axes with the mouse cursor, the color of the axis will change to the color which was set for the **selected** state. You don't need to press the left mouse button for the hovered axis to change color.

4.1 The Translation Gizmo

The translation gizmo allows you to move objects around in the scene and it behaves in the same way as that of the Unity Editor. The following image shows a screen-shot of the translation gizmo:



4.1.1 Using The Translation Gizmo

If you click and drag one of the gizmo axis you will perform a translation along the corresponding gizmo axis. The gizmo also has a set of squares. Clicking and dragging one of the squares will allow you to perform a translation along 2 axes at once.

If you hold down the **SHIFT** key, a square will be shown centered around the gizmo position. Clicking and dragging the mouse while the square is selected will perform a translation along the camera right and up axes. This is a little bit like performing a translation in screen space.

4.1.2 Snapping

The translation gizmo supports 2 types of snapping: **Step** and **Vertex Snapping**.

4.1.2.1 Step Snapping

Step snapping allows you to perform translations in increments of a specified step value. For example, if the step value is set to 1, this means that a translation is only performed when the accumulated translation amount is >= 1. In order to use step snapping, you have to keep the **CTRL/CMD** button pressed and then manipulate the gizmo as you would normally do (using the axes or the squares or pressing the **SHIFT** key to translate along the camera right and up axes).

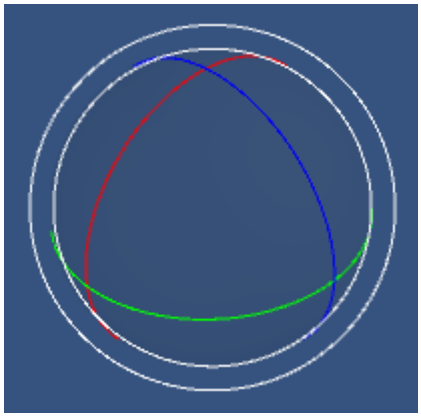
4.1.2.2 Vertex Snaping

Vertex snapping works in the same way as in the Unity Editor. Press the **V** key on the keyboard and then move the mouse around to select one of the vertices of the selected game objects. This is the vertex that will be snapped to the destination position. Once you found the vertex that you are interested in, press the left mouse button and move the mouse around to snap the selected objects. The objects will be snapped to the vertex which is closest to the mouse cursor.

Note: If the none of the objects that you have selected have a mesh attached to them, when you press the **V** key and start moving the mouse around you will be selecting the objects' positions. However, when you press the left mouse button, snapping will only occur if there are any vertices around. Otherwise, nothing will happen.

4.2 The Rotation Gizmo

The rotation gizmo allows you to rotate objects in the scene and it behaves in the same way as that of the Unity Editor. The following image shows a screen-shot of the rotation gizmo:



4.2.1 Using The Rotation Gizmo

The rotation gizmo has 3 colored circles that can be used to rotate around a single axis. You can rotate around a single axis by clicking on one of the circles and then start dragging the mouse.

As you can see in the image above, there is also an outer circle (the one which encloses the rotation sphere and which is also slightly bigger). Clicking on this circle and then dragging the mouse will allow you to rotate around the camera view vector.

If you click on a point on the imaginary sphere (not on any of the components that we have discussed so far), you will be able to rotate around the camera right and up axes.

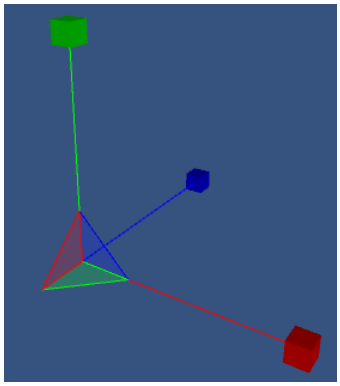
4.2.1.1 Snapping

The rotation gizmo supports **Step** snapping which allows you to rotate in increments of a specified step value (expressed in units of degrees). For example, if this step value is set to 15, a rotation is performed only when the accumulated rotation amount is >= 15. In order to use **Step** snapping, you have to keep the **CTRl/CMD** button pressed and then manipulate the gizmo as you normally do.

Note: Step snapping works only when you are using one of the 3 colored circles.

4.3 The Scale Gizmo

The scale gizmo works in almost the same way as the one which exists in the Unity Editor with a small exception. The following image shows the scale gizmo:



4.3.1 Using The Scale Gizmo

The scale gizmo has 3 colored axes. Clicking on of these axes and dragging the mouse, will perform a scale operation along the specified axis.

As you can see in the image above, there are also 3 multi-axis triangles. Clicking on one of these triangles and than dragging the mouse, will perform a scale operation along 2 axes at once.

If you want to perform a scale operation along all axes at once, you have to keep the **SHIFT** key pressed and then drag the mouse around.

4.3.1.1 Snapping

The scale gizmo supports **Step** snapping which allows you to scale in increments of a specified world unit amount. For example, if the step value is set to 1, a scale operation will be performed only when the accumulated scale is >= 1. In order to use **Step** snapping, you have to keep the **CTRL/CMD** key pressed and then use the scale gizmo as you normally do (drag the axes or the multi-axis triangles or press **SHIFT** to scale along all axes at once).

4.4 Gizmo Properties

In this chapter we will take a look at all the gizmo properties which are available for modification.

These properties can be modified from both the inspector and at runtime. For example, assuming you are working on your own editor application, you may want to provide a settings dialog to the user that allows them to change different gizmo properties.

4.4.1 Common Gizmo Properties

There are a few properties which apply to all gizmos. These properties will always be shown at the top of a gizmo objects' Inspector. We will discuss these properties here and we will ignore them when we discuss the properties for each gizmo individually. The following image shows the common gizmo properties which appear in the Inspector GUI of each type of gizmo:

First of all notice that there is a label at the top which tells you that properties which are prefixed by the **'*'** character can only be verified at runtime. This means that when you change one of these properties and want to see how it affects the appearance or behavior of the gizmo, you will have to switch to play mode. The other properties can be verified in the scene view without having to enter play mode. **If you select one of the gizmo objects, a rough representation of the gizmo will be shown in the scene view.**

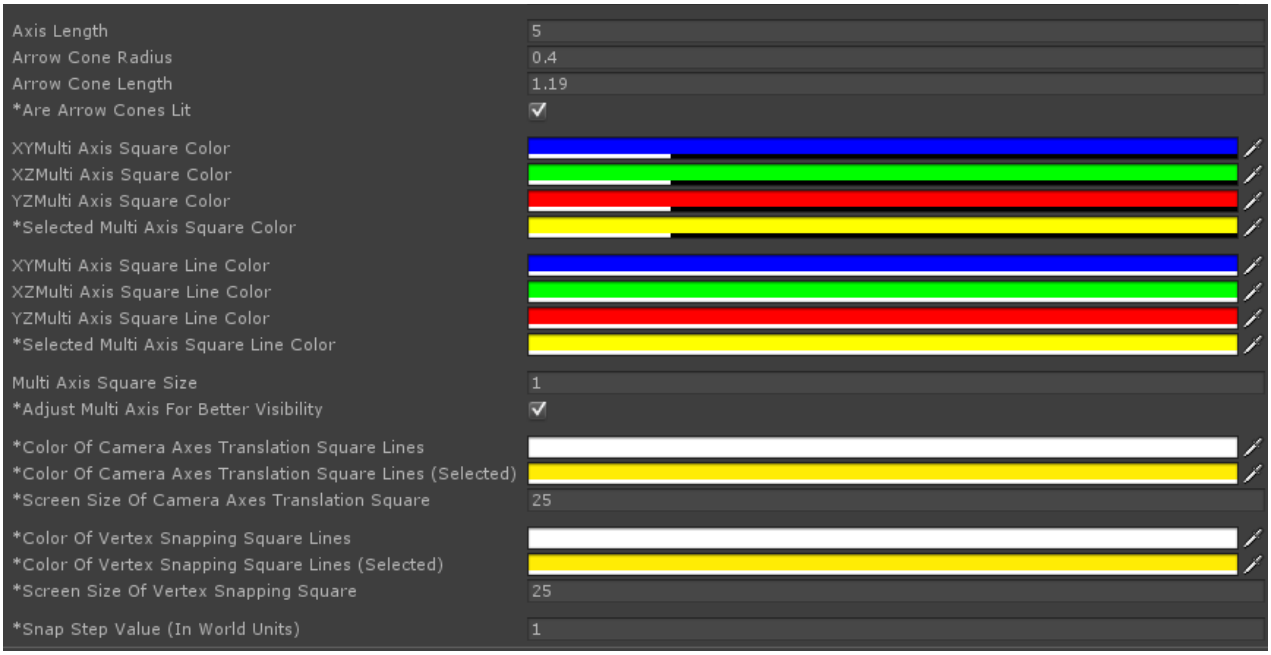
We will now discuss the properties in the same order in which they appear in the Inspector GUI:

- **Gizmo Base Scale** → this property allows you to control the scale of the gizmo to make it bigger or smaller as needed. The scale is applied to all axes, so it is a uniform scale. One important thing to remember here is that the scale of the gizmo may or may not be set to this exact same value. If the the **Preserve Gizmo Screen Size** property (discussed next) is unchecked, then the scale of the gizmo will be set to exactly the same value that you specify for the base scale property. Otherwise, the scale of the gizmo will be set to the base scale multiplied by a scale factor. In any case, adjusting this property will scale the gizmo to make it bigger or smaller.
- **Preserve Gizmo Screen Size** → if this property is checked, the gizmo will maintain roughly the same size no matter how close or far away from the camera it is. If unchecked, the gizmo size will change as the gizmo moves closer or away from the camera. **Note:** This property is especially important when using a perspective camera. When using an orthographic camera, the size of the objects doesn't change with distance. However, an orthographic camera can also be zoomed in or out and in this case the property will affect the size of the gizmo object.
- **X, Y, Z Axis Color** → these are 3 color fields which allow you to control the colors of the gizmo axes. For a translation gizmo, these are the axes that allow you to translate along a single axis at once. For a rotation gizmo, these are the 3 circles that allow you to rotate along a single axis at once. Finally, for a scale gizmo, these are the 3 axes which allow you to scale along a single axis at once.

- **Selected Axis Color** → this is the color which must be used when drawing the selected axis. The selected axis is the one which is currently hovered by the mouse cursor.

4.4.2 Translation Gizmo Properties

In this chapter we will take a look at all the properties which can be modified for the translation gizmo. The following image shows the translation gizmo Inspector GUI (the common gizmo properties were left out):



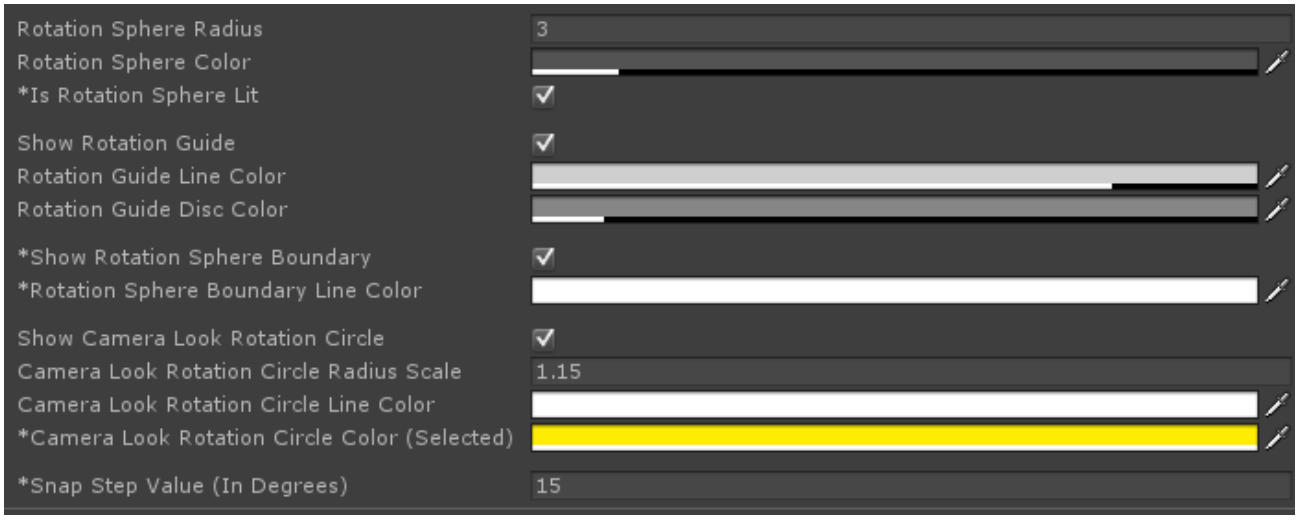
We will now discuss the properties in the same order in which they appear in the Inspector GUI:

- **Axis Length** → this allows you to control the length of the translation axes;
- **Arrow Cone Radius** → this allows you to control the radius of the arrow cones which sit at the tip of the translation axes;
- **Arrow Cone Length** → this allows you to control the length of the arrow cones which sit at the tip of the translation axes;
- **Are Arrow Cones Lit** → allows you to specify whether or not the arrow cones should be affected by lighting. You can uncheck this property if you would like the gizmos to have a flat look;
- **XY, XZ, YZ Multi Axis Square Color** → this allows you to control the color of the multi-axis squares which allow you to translate along the XY, XZ and YZ axes respectively.
- **Selected Multi Axis Square Color** → this allows you to control the color of the selected multi-axis square. The selected multi-axis square is the square which is hovered by the mouse cursor.
- **XY, XZ, YZ Multi Axis Square Line Color** → this allows you to control the color of the border lines which surround the multi-axis squares.
- **Selected Multi Axis Square Line Color** → this allows you to control the color of the border lines for the multi-axis square which is currently selected (i.e. hovered by the mouse cursor).

- **Multi Axis Square Size** → allows you to control the size of the multi-axis squares;
- **Adjust Multi Axis For Better Visibility** → if this property is checked, the positions of the multi-axis squares will always be adjusted based on the camera view vector in such a way that each square can easily be selected no matter how the camera is oriented. If this property is unchecked, the positions of the squares will always be the same and for some camera angles, some squares may become hard to select;
- **Color Of Camera Axes Translation Square Lines** → allows you to control the color of the lines that make up the square which is activated when you press the **SHIFT** key on the keyboard. This is the square which, when selected allows you to translate along the camera right and up axes;
- **Color Of Camera Axes Translation Square Lines (Selected)** → same as the property discussed above, but it applies when the square is hovered by the mouse cursor;
- **Screen Size Of Camera Axes Translation Square** → allows you to control the screen size of the square which allows you to translate along the camera right and up axes;
- **Color Of Vertex Snapping Square Lines** → allows you to control the color of the lines which make up the square that is drawn when vertex snapping is active;
- **Color Of Vertex Snapping Square Lines (Selected)** → same as the property discussed above, but it applies when the square is **hovered by the mouse cursor and the left mouse button is pressed.**
- **Screen Size Of Vertex Snapping Square** → allows you to control the screen size of the vertex snapping square.
- **Snap Step Value (In World Units)** → this allows you to control the step value that is used when step snapping is enabled. The value is expressed in world units.

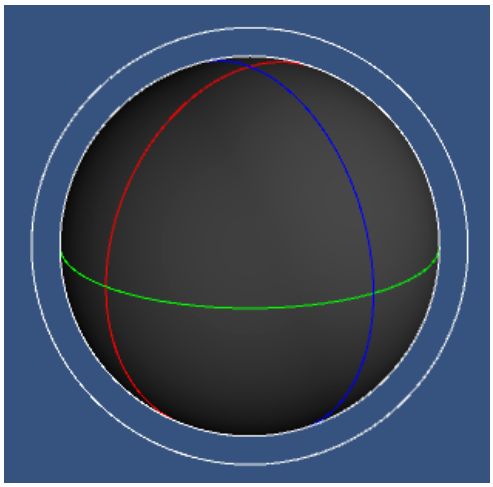
4.4.3 Rotation Gizmo Properties

In this chapter we will take a look at all the properties which can be modified for the rotation gizmo. The following image shows the rotation gizmo Inspector GUI (the common gizmo properties were left out):

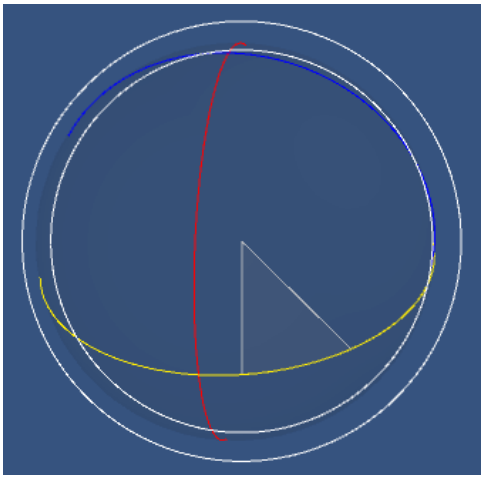


We will now discuss the properties in the same order in which they appear in the Inspector GUI:

- **Rotation Sphere Radius** → this allows you to control the radius of the rotation sphere. Changing this property will make the gizmo bigger or smaller;
- **Rotation Sphere Color** → this allows you to control the color which is used to render the rotation sphere. The following image shows the rotation gizmo when this color is set to an opaque gray:



- **Is Rotation Sphere Lit** → allows you to specify whether or not the rotation sphere must be affected by lighting when rendered. You can uncheck this poperty if you wish to have a rotation gizmo with a flat look.
- **Show Rotation Guide** → when you perform a rotation using one of the rotation circles, a rotation guide will be shown if this property is checked. The rotation guide offers a visual representation of the amount of rotation that has accumulated. The following image shows the rotation guide:

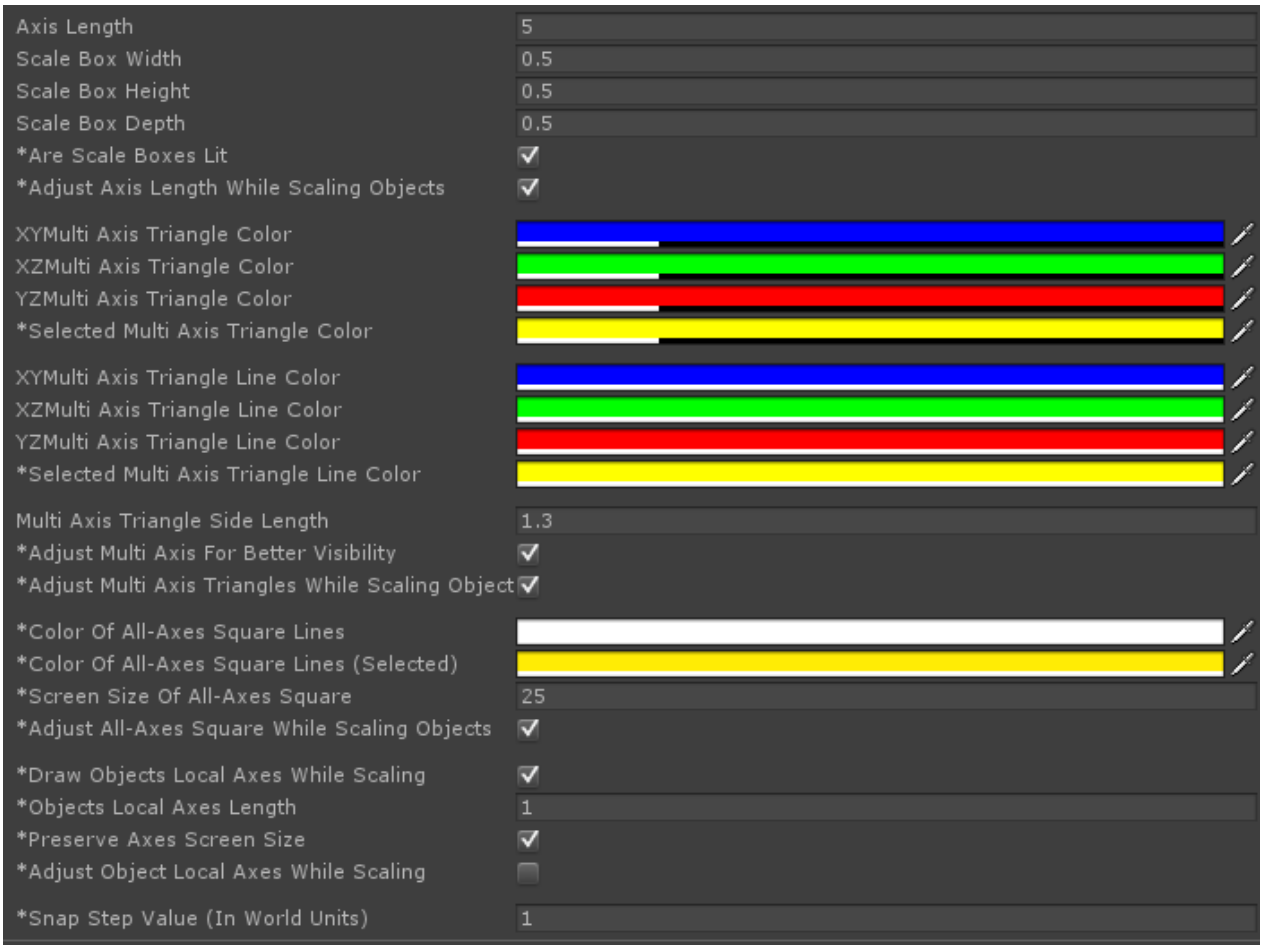


The rotation guide is composed of 2 guide lines which form a rotation arc. The area which is formed by the 2 guide lines is called the rotation guide disc.

- **Rotation Guide Line Color** → allows you to control the color of the rotation guide lines;
- **Rotation Guide Disc Color** → allows you to control the color of the rotation guide disc;
- **Show Rotation Sphere Boundary** → if this property is checked, the boundary of the rotation sphere will be drawn.;
- **Rotation Sphere Boundary Line Color** → allows you to control the color of the rotation sphere boundary;
- **Show Camera Look Rotation Circle** → this allows you to specify whether or not the circle which allows you to rotate along the camera view vector must be drawn. **Note:** If this is not checked, you will not be able to rotate along the camera view vector.
- **Camera Look Rotation Circle Radius Scale** → allows you to scale the radius of the camera look rotation circle in order to make the circle bigger or smaller. **Note:** The scale is applied relative to the radius of the rotation sphere. For example, if the rotation sphere has a radius of 2, and the radius scale is 1.5, the final radius of the circle is 2 * 1.5 = 3.
- **Camera Look Rotation Circle Line Color** → allows you to control the color of the camera look rotation circle;
- **Camera Look Rotation Circle Color (Selected)** → same as the property discussed above, but it applies when the circle is selected (i.e. hovered by the mouse cursor);
- **Snap Step Value (In Degrees)** → this allows you to control the step value that is used when step snapping is enabled. The value is expressed in degrees.

4.4.4 Scale Gizmo Properties

In this chapter we will take a look at all the properties which can be modified for the scale gizmo. The following image shows the scale gizmo Inspector GUI (the common gizmo properties were left out):



We will now discuss the properties in the same order in which they appear in the Inspector GUI:

- **Axis Length** → this allows you to control the length of the gizmo scale axes;
- **Scale Box Width/Height/Depth** → these 3 properties allow you to control the size of the scale boxes which sit at the tip of the gizmo axes;
- **Are Scale Boxes Lit** → if checked, the scale boxes will be affected by lighting when rendered. You can uncheck this property if you desire a gizmo with a flat look;
- **Adjust Axis Length While Scaling Objects** → if this is checked, the length of the gizmo axes will be scaled along with the objects involved in the scale operation. For example, if you are scaling along the X axis and this property is checked, as the objects become bigger or smaller along the X axis, the gizmo's X axis will also become bigger or smaller;
- **XY, XZ, YZ Multi Axis Triangle Color** → these 3 properties allow you to control the color that is used to draw the triangles which allow you to scale objects along 2 axes at once;
- **Selected Multi Axis Triangle Color** → this allows you to control the color of the selected (i.e. hovered) multi-axis triangle;

- **XY, XZ, YZ Multi Axis Triangle Line Color** → these 3 properties allow you to control the color that is used to draw the border lines of the multi-axis triangles;
- **Selected Multi Axis Triangle Line Color** → this property allows you to control the color that is used to draw the border line for a selected (i.e. hovered) multi-axis triangle;
- **Multi Axis Triangle Side Length** → allows you to control the length of the adjacent sides of the multi-axis triangles. This allows you to make the triangles bigger or smaller;
- **Adjust Multi Axis For Better Visibility** → if this property is checked, the multi-axis triangles will always have their positions adjusted based on the camera view vector. The position will always be calculated in such a way that it will always be easy to select any one of the 3 triangles regardless of the current camera angle. If this is unchecked, the triangles will always sit in the same position and for some camera angles, it may become hard to select some of the triangles;
- **Adjust Multi Axis Triangles While Scaling Objects** → if this is checked, the area of the multi-axis triangles will be scaled along with the objects involved in the scale operation. For example, if you are scaling along the X and Y axes and this property is checked, as the objects become bigger or smaller along the X and Y axes, the gizmo's XY triangle will also become bigger or smaller;
- **Color Of All-Axes Square Lines** → this allows you to control the color of the lines that make up the square that appears when you want to scale along all axes at once;
- **Color Of All-Axes Square Lines (Selected)** → this is the same as the property discussed above, but it applies when the square is selected (i.e. hovered by the mouse cursor);
- **Screen Size Of All-Axes Square** → allows you to control the screen size of the square which appears when you want to perform a scale operation along all 3 axes at once;
- **Adjust All-Axes Square While Scaling Objects** → if this is checked, the area of the square which allows you to scale along all 3 axes at once will be scaled along with the objects involved in the scale operation.
- **Draw Object Local Axes While Scaling** → if this property is checked, the local axes of the objects involved in a scale operation will be drawn while the scale operation is performed. **Note:** The color of these axes will be the same as the color used to draw the gizmo axes. These color properties were discussed in the **Common Gizmo Properties** chapter.
- **Objects Local Axes Length** → This property allows you to control the length of the object local axes when they are rendered. This property only applies if **Draw Object Local Axes While Scaling** is checked.
- **Preserve Axes Screen Size** → If this property is checked, the size of the object local axes lines will remain roughly the same no matter how far the objects are from the camera. Otherwise, the lines become bigger or smaller as they are moved closer to or away from the camera. This property is similar to the **Preserve Gizmo Screen Size** property discussed in the **Common Gizmo Properties** chapter.
- **Adjust Object Local Axes While Scaling** → if this is checked, the length of the object local axes lines will be scaled along with the objects involved in the scale operation. For example, if you are scaling along the X and Y axes and this property is checked, as the objects become bigger or smaller along the X and Y axes, the X and Y local axes lines of the scale objects will also become bigger or smaller;

- **Snap Step Value (In World Units)** → this allows you to control the step value that is used when step snapping is enabled. The value is expressed in world units.

4.5 Gizmo Transform Space

The system allows you to choose the space in which the objects are transformed. This is the same as you can do inside the Unity Editor. There are 2 possible transform spaces: **Global** and **Local.**

Note: The gizmo transform spaces apply only to the translation and rotation gizmos. **The scale gizmo will always scale objects along their local axes.**

4.5.1 The Global Transform Space

When this is active, the gizmos and the objects that they control will be transformed using the global coordinate system axes. For a translation gizmo this means that its axes will always be aligned with the global coordinate system axes and translations will occur along those axes. For a rotation gizmo it means that the rotation circles will always go around the global axes which means that when a rotation is applied to a game object, the game object will be rotated around the global axes.

Note: The scale gizmo is not affected by this transform space. The scale gizmo will always transform objects along their local axes.

4.5.2 The Local Transform Space

When this space is active, the gizmos will inherit the orientation of the last game object which was selected. Assuming we are dealing with only one selected game object, if the local transform space is active, the gizmo will inherit the orientation of that object. That means that its axes will also be oriented in the same manner. For a translation gizmo, this means that its translation axes will coincide with the object's local axes and this allows you to translate an object along its local right, up and look vectors. For a rotation gizmo, it means that the rotation circles will go around the object's local axes and this allows you to rotate a game object along its local right, up and look vectors. For a scale gizmo, it means that the scale axes will coincide with the object's local axes and the scale is applied along the object's local right, up and look vectors.

4.6 Gizmo Transform Pivot Point

The system allows you to choose the transform pivot point This is the same as you can do inside the Unity Editor. The transform pivot point affects both the position of the gizmos and the way in which the objects are transformed. The possible pivot points are: **Center** and **Mesh Pivot**.

Note: As far as the translation gizmo is concerned, the pivot point only affects the gizmo's position, but it doesn't really affect the way in which you translate the game objects.

4.6.1 The Center Transform Pivot Point

When the center pivot point is active the gizmo will be positioned in the center of the object selection. Also, the center pivot point affects the transformations in the following way:

- when rotating a group of objects, the objects will be rotated around the center of the selection;
- when rotating a single object, the object is rotated around its center point;

- when scaling a group of objects, the objects are scaled from their center and their positions are moved closer o further away from the selection center based on how the scale is performed;
- when scaling a single object, the object is scaled from its center.

4.6.2 The Mesh Pivot Transform Pivot Point

This type of pivot point is useful when dealing with models whose pivot points have been adjusted inside a modeling package. It allows you to easily rotate objects like doors, levers, windows etc whose rotation does not naturally happen around the object's center.

When this pivot point is active, the position of the gizmo will be set to the last game object which was selected. Also, the mesh pivot point affects the transformations in the following way:

- when rotating a group of objects, the objects will be rotated around their individual pivot points and **not as a group as is the case with the Center pivot point**;
- when rotating a single object, the object is rotated around its pivot point;
- when scaling a group of objects, the objects are scaled from their pivot points and their positions **are not moved closer o further away from the selection center as is the case with the Center pivot point;**
- when scaling a single object, the object is scaled from its pivot point.

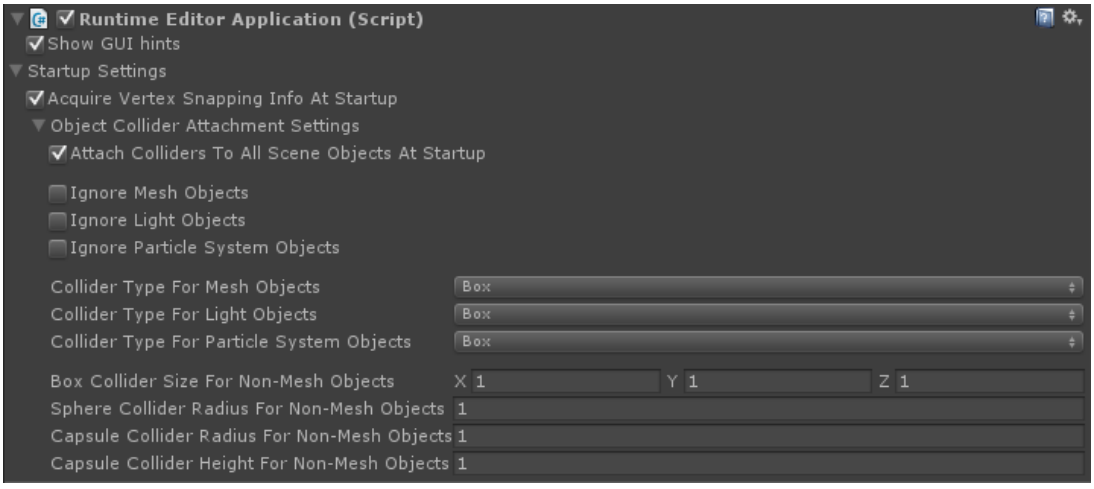
Note: For objects that don't have a mesh attached to them, the pivot point is the same as their center.

5 Runtime Editor Subsystems

This chapter discusses the runtime editor subsystems which make everything work and we will look at each of their properties and see what they mean.

5.1 The Runtime Editor Application

The name of this object is set to **(singleton) RTEditor.RuntimeEditorApplication** when it is first created, but you can change it if you like. This represents the actual runtime editor application itself and its inspector GUI contains a few properties which can be modified. The following image shows the inspector GUI:



We will now discuss all the properties in the order in which they are shown in the inspector:

- **Show GUI hints –** this is only used inside the editor when running in play mode and it allows you to specify if the GUI hints (e.g. shortcuts etc) can be shown;
- **Acquire Vertex Snapping Info At Startup** → if this property is checked, the application will acquire at startup all the necessary information which is needed to perform vertex snapping with the translation gizmo. Most of the times, you will probably set this to true. However, you can uncheck this property if you wish to perform this step manually (discussed later). **Note:** If this property is checked, vertex snapping information will be refreshed at every scene load.
- **Attach Colliders To All Scene Objects At Startup** → if this property is checked, the application will process all scene objects at startup and attach colliders to them based on the settings discussed next. Colliders are necessary because without them, you will not be able to select objects in the scene. The possible collider types are: **Box, Sphere, Capsule** and **Mesh Colliders**. Again, you may leave this unchecked if you wish to perform this step manually.

  Note: If this property is checked, the collider attachement will occur every time a new scene is loaded. The settings which we are about to discuss next apply only if this property is checked.
- **Ignore Mesh Objects –** if this property is checked, mesh objects will be ignored and their colliders will be left intact. Otherwise, any colliders that these objects might have will be removed and replaced with the chosen collider type (see **Collider Type For Mesh Objects**).
- **Ignore Light Objects** - if this property is checked, light objects will be ignored and their colliders will be left intact. Otherwise, any colliders that these objects might have will be removed and replaced with the chosen collider type (see **Collider Type For Light Objects**).
- **Ignore Particle System Objects –** if this property is checked, particle system objects will be ignored and their colliders will be left intact. Otherwise, any colliders that these objects might have will be removed and replaced with the chosen collider type (see **Collider Type For Particle System Objects**).
- **Collider Type For Mesh Objects** → This is the type of collider which must be attached to game objects that have a mesh attached to them. This could be a mesh associated with a mesh filter component or a mesh which is associated with a skinned mesh renderer.
- **Collider Type For Light Objects** → This is the type of collider which must be attached to game objects which have a light component attached to them, **but don't have any meshes attached.**
- **Collider Type For Particle System Objects** → This is the type of collider which must be attached to game objects which have a particle system component attached to them, **but don't have a mesh or light attached.**
- **Box Collider Size For Non-Mesh Objects** → When you specify a box collider for light and particle system objects, this property allows you to control the size of the box collider that is attached to those objects.
- **Sphere Collider Radius For Non-Mesh Objects** → When you specify a sphere collider for light and particle system objects, this property allows you to control the radius of the sphere collider that is attached to those objects.
- **Capsule Collider Radius For Non-Mesh Objects** → When you specify a capsule collider for light and particle system objects, this property allows you to control the radius of the capsule

collider that is attached to those objects.

- **Capsule Collider Height For Non-Mesh Objects** → When you specify a capsule collider for light and particle system objects, this property allows you to control the height of the capsule collider that is attached to those objects.

At the moment it is not possible to attach colliders to objects which don't have a mesh, light or particle system object. If this is required, you will need to perform this step manually. Also, as we will see when we discuss scripting details, you have the possibility to easily attach colliders to object hierarchies that get created at runtime using similar collider attachment settings.

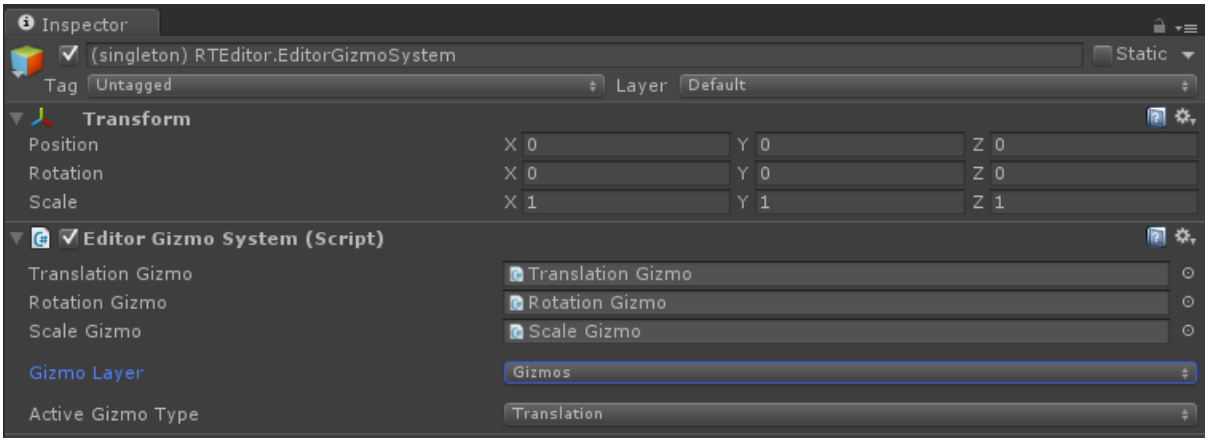
Note: If any of the **Ignore Mesh/Light/Particle System Objects** properties are checked, and if those objects didn't have a collider assigned to them in the Unity Editor, no collider will be attached to them at application startup which means you won't be able to select them. However, you can manually attach colliders to your scene objects as we will see later.

5.2 The Gizmo System

The name of this object is set to **(singleton) RTEditor.EditorGizmoSystem** when it is first created, but you can change it if you wish. This represents the system which manages all the gizmo objects. It allows you to switch between different gizmo types, it calculates the position and orientation of the gizmo objects and it also allows you to change the gizmo transform space and transform pivot point.

Another thing for which the gizmo system is responsible is to make sure that there is always a directional light in the scene which has the same orientation as the camera. This light is created automatically and it is used to light the gizmo objects.

The following image shows the inspector GUI for the gizmo system:



We will now discuss all the properties in the order in which they are shown in the inspector:

- **Translation Gizmo** → this is the translation gizmo which can be used to translate objects in the scene. When you press the **Create Subsystems** menu item in the top menu, the translation gizmo is created automatically and this property will be set to point to it.
- **Rotation Gizmo** → this is the rotation gizmo which can be used to rotate objects in the scene. When you press the **Create Subsystems** menu item in the top menu, the rotation gizmo is created automatically and this property will be set to point to it.
- **Scale Gizmo** → this is the scale gizmo which can be used to scale objects in the scene. When

you press the **Create Subsystems** menu item in the top menu, the scale gizmo is created automatically and this property will be set to point to it.

- **Gizmo Layer** → this is the layer to which the gizmo objects belong. The gizmo system will automatically create a directional light at startup which will always point in the same direction as the camera view vector. This light is meant to light only the gizmo objects and this is why specifying a gizmo layer is needed. It allows the light to know which objects in the scene it must affect. Its culling mask will be adjusted so that it will affect only the objects in the layer that you specify here.
- **Active Gizmo Type** → this represents the gizmo which should initially be active when starting the application. When the first object(s) get selected, this is the gizmo that will be activated. The user can switch between different gizmo types using the **W (translation), E (rotation)** and **R (scale)** keys.

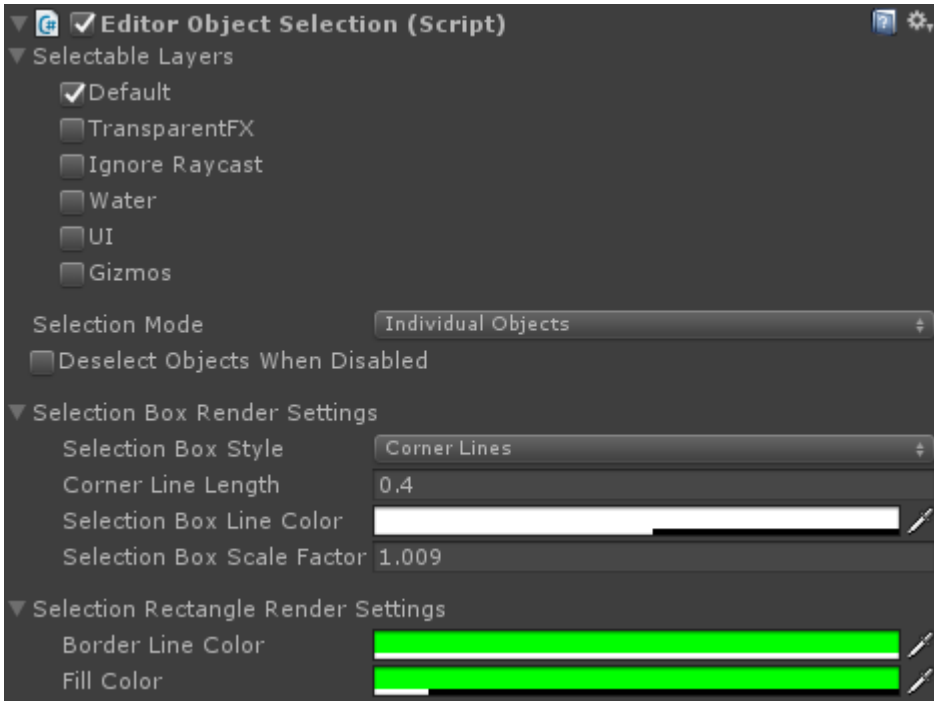
5.3 The Object Selection System

The name of this object is set to **(singleton) RTEditor.EditorObjectSelection** when it is first created, but you can change it if you wish. This system is responsible for allowing you to perform object selections using standard object selection mechanisms. The system allows you to:

- click on objects to select them or add them to the current selection;
- move the mouse while the left mouse button is pressed to select objects using a selection shape;
- use a series of shortcut keys which allow you to control whether or not objects get added or removed from the current selection.

Note: Only objects that have a collider attached to them can be selected.

The following image shows the inspector GUI for the object selection system:



We will now discuss all the properties in the order in which they are shown in the inspector:

- **Selectable Layers –** this is a section which allows you to specify which layers can be selected. When a layer is checked, the objects which belong to that layer can be selected;
- **Selection Mode –** there are 3 possible selection modes available:
  - ➔ **Individual Objects –** only those objects that are clicked by the mouse cursor or those which enter the selection shape are selected/deselected;
  - ➔ **Entire Hierarchy –** when an object is clicked, or if when it enters the selection shape area, the entire hierarchy to which the object belongs gets selected.
  - ➔ **Custom –** custom selection mode. For this to work, you have to implement a series of handlers. Please see chapter 7.5.4.
- **Deselect Objects When Disabled** - the object selection system exposes a method called **SetEnabled** which receives a boolean that specifies whether or not the object selection system should be enabled or disabled. If this property is set to true, when you disable the object selection system, all selected objects will be deselected.

The second category of settings apply to the object selection boxes:

- **Selection Box Style** → this is the style of the object selection boxes that are drawn for the selected objects in the scene. The possible values are: **Corner Lines** and **Wire Box**.

  The following image shows an object which was selected when **Corner Lines** is selected:

The next image shows the same selected object when the **Wire Box** style is used:

- **Corner Line Length** → this porperty is used only when the object selection box style is set to **Corner Lines**. It controls the length of the lines which meet in the corner. **Note:** If the length of the lines is big enough that the lines from different corners overlap or meet, the tool will clamp the length and leave a little bit of space between the lines.
- **Selection Box Line Color** → this is the color used to draw the selection box lines.
- **Selection Box Scale Factor** → this is a scale factor which can be used to scale the selection boxes in order to make them bigger or smaller as needed.

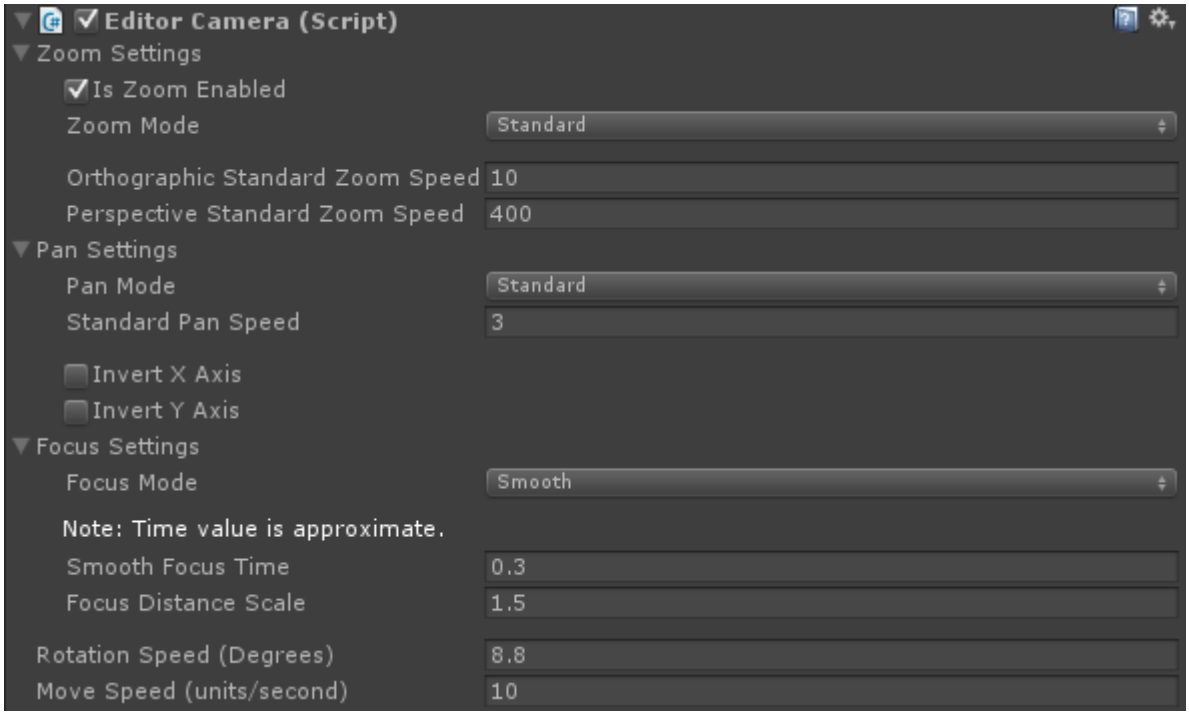
The next couple of settings apply to the object selection rectangle which allows you to select multiple objects in the scene:

- **Border Line Color** → this is the color that is used to draw the border of the object selection rectangle.
- **Fill Color** → this is the color which is used to fill the selection rectangle.

5.4 The Editor Camera

The name of this object is set to **(singleton) RTEditor.EditorCamera** when it is first created, but you can change it if you wish. This is the camera that allows you to navigate the scene at runtime. It provides basic rotation, panning and zooming functionality.

The inspector GUI is shown in the following image:



The GUI is different based on the specified zoom and pan modes, but the differences are quite small and they will be pointed out accordingly.

The first section allows you to modify the camera zoom settings:

- **Is Zoom Enabled –** allows you to toggle camera zoom on/off as needed;
- **Zoom Mode –** there are 2 possible zoom modes to choose from: **Standard** and **Smooth**. Standard zooming allows you to zoom the camera based on the speed by which you are rotating the mouse scroll wheel. Smooth scrolling is essentially the same, but the zoom speed slowly decreases over time.
- **Orthographic Smooth Value –** when the zoom mode is set to **Smooth,** this allows you to control how fast the zoom speed reaches 0. Smaller values produce a longer zoom effect. Bigger values produce a shorter zoom effect. This value applies only when the editor camera is set to orthographic.
- **Perspective Smooth Value –** same as the orthographic smooth value, but it applies when the editor camera is set to perspective.
- **Orthographic Smooth Zoom Speed –** this allows you to control the zoom speed when the zoom mode is set to **Smooth.** This value applies only when the editor camera is set to orthographic.
- **Perspective Smooth Zoom Speed –** same as the orthographic smooth value, but it applies when the editor camera is set to perspective.

Note: When the zoom mode is set to **Standard,** the smooth values disappear from the Inspector and the zoom speed properties are replaced with the zoom speeds for the **Standard** zoom mode. The fact that you have separate zoom speeds based on the camera type and zoom mode is useful because the same zoom speed works differently with different camera types and zoom modes.

The second section allows you to control pan settings.The same idea applies here as in the zooming

case. You have access to 2 possible pan modes: **Standard** and **Smooth** and the settings you can modify are similar to the zoom related ones with the exception that in this case there is no more differentiation between an orthographic and perspective camera.

For the pan settings, there are also 2 properties which allow you to invert the pan X and Y axes. These are called **InvertXAxis** and **InvertYAxis**.

The next section allows you to control camera focus settings. These are the settings which control the way in which the camera is focused on the object selection:

- **Focus Mode –** allows you to choose the camera focus mode. There are 3 options available here:
  - ➔ **Smooth –** the speed by which the camera position is adjusted slowly decreases over time. This mode is similar to the way in which Unity performs camera focus in the editor;
  - ➔ **Constant Speed –** the camera is focused by having its position adjusted at a constant speed;
  - ➔ **Instant –** the camera position is instantly snapped to the correct position in order to achieve the focus effect.
- **Focus Distance Scale** – when the camera is focused, a position will be calculated for the camera such that it will sit right in front of the object selection. This property allows you to scale the distance between the camera and the object selection. Bigger values will cause the camera to move further away from the focus point. The minimum possible scale is 1.0f.

The rest of the focus settings differ based on the chosen focus mode. Let's start with the **Smooth** focus mode. For this mode, we have **Smooth Focus Time.** This is the time it takes to complete the focus effect. As is written in the Inspector GUI, this value is approximate;

For the **ConstantSpeed** focus mode, we have **Constant Focus Speed,** which represents the constant speed which is used to adjust the camera position.

The **Instant** focus mode does not have any additional properties.

The last section allows you to control rotation and move settings:

- **Rotation Speed (Degrees)** → allows you to specify the camera rotation speed in degrees. **Note:** This value applies to all types of rotations: normal (look around) and orbit.
- **Move Speed (units/second)** → allows you to control the camera move speed when the camera is moved using the WASD and QE keys.

Currently, these are the camera controls:

- WASD, QE + right mouse button → move the camera around (the same as you can do inside the Unity Editor);
- mouse move + middle mouse button pressed → pans the camera;
- mouse move + right mouse button pressed → rotates the camera;
- **F** – focus the camera on the current object selection (only works if there is at least one object selected);
- mouse move + ALT + right mouse button pressed → orbits the camera (only works if the camera has been focused);

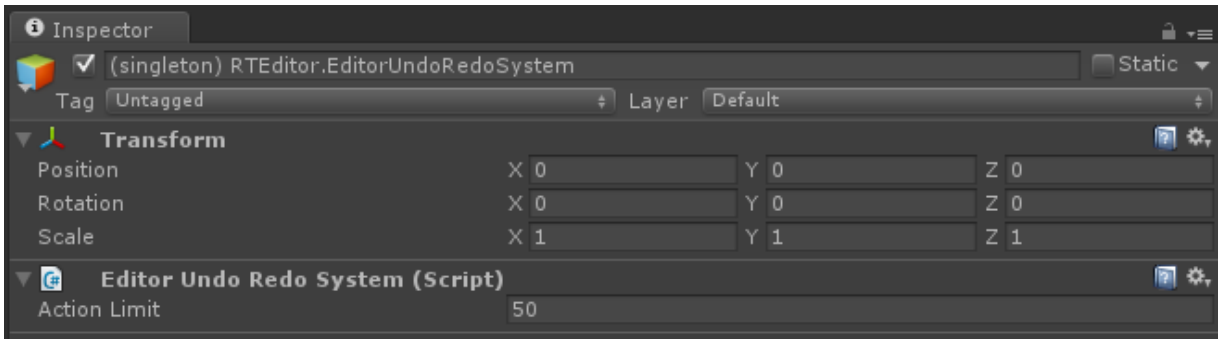
- scroll wheel → zooms the camera.

5.5 The Editor Undo/Redo System

The name of this object is set to **(singleton) RTEditor.EditorUndoRedoSystem** when it is first created, but you can change it if you wish. It is responsible for allowing you to perform undo and redo operations. Currently, the following operations can be undo/redone:

- object manipulation with gizmos (translation, rotation and scale);
- object selection (selecting/deselecting objects and assigning/removing objects to/from the selection mask, changing the object selection mode);
- gizmo transform space change;
- gizmo transform pivot point change;
- gizmo type change.

The following image shows the inspector GUI for the undo/redo system:



As you can see, there is currently only one property here and this is an integer field labeled **Action Limit**. This property allows you to specify the maximum number of actions which can be undone or redone.

5.6 The Editor Shortcut Keys

The name of this object is set to **(singleton) RTEditor.EditorShortuctKeys** when it is first created, but you can change it if you wish. There are currently no properties which can be modified in the inspector, so we will only take a look at the shortcut keys which you can use at runtime:

- **W (right mouse button must not be pressed)** → activate the translation gizmo;
- **E (right mouse button must not be pressed)** → activate the rotation gizmo;
- **R** → activate the scale gizmo;
- **Q right mouse button must not be pressed)** → turn off gizmos. This allows you to perform object selections without having a gizmo active in the scene;
- **G** → activates the global transform space;
- **L** → activates the local transform space;
- **P** → toggles the transform pivot point between **Center** and **MeshPivot.**

- **F** → focus the camera on the object selection (only works when there is at least one selected object);
- Gizmo specific keys:
  - Translation gizmo:
    - **V** → while held down, allows you to perform vertex snapping. When released, vertex snapping is disabled;
    - **CTRL/CMD** → while held down, allows you to perform step snapping;
    - **SHIFT** → while held down, allows you to translate along the camera right and up axes;
  - Rotation gizmo:
    - **CTRL/CMD** → while held down, allows you to perform step snapping;
  - Scale gizmo:
    - **CTRL/CMD** → while held down, allows you to perform step snapping;
    - **SHIFT** → while held down, allows you to perform a scale operation along all axes at once.
- Object selection specific keys:
  - **CTRL/CMD** → while held down, allows you to add objects to the current selection. For example, you can hold down this key and click on individual game objects to add them to the selection or drag the mouse while the left mouse button is pressed in order to add objects to the selection using the object selection shape. If this key is not held down, when you select new objects, the previous selection is cleared. **Note:** If you click on a game object while this key is pressed and the game object is already selected, it will be removed from the selection.
  - **SHIFT** → while held down, allows you to deselect multiple objects using the object selection shape.
- Undo/Redo specific keys:
  - When in play mode in the Unity Editor:
    - **CTRL/CMD + SHIFT + Z** → Undo
    - **CTRL/CMD + SHIFT + Y** → Redo;
  - When in build mode:
    - **CTRL/CMD + Z** → Undo;
    - **CTRL/CMD + Y** → Redo;

6 Scripting

This chapter provides scripting information which you may find useful if you decide to extend the system. This is an inevitable step if you would like to build your own editor application. The package gives you a good starting point, but there are a few things that you need to be aware of if you would like to go further.

We will start by examining some of the scripts which you may need to modify or add new content to and then we will look at a few examples of how you might use the existing architecture to add new functionality.

6.1 Gizmos

6.1.1 Gizmo events

The gizmo objects expose some events which you may need to listen to sometimes. These are:

- **OnGizmoDragStart –** sent when one of the gizmo components is hovered and the left mouse button is clicked;
- **OnGizmoDragUpdate –** sent when the gizmo is used to transform objects;
- **OnGizmoDragEnd –** sent when the left mouse button is released after the gizmo was dragged. This event can be sent after **OnGizmoDragStart** or **OnGizmoDragUpdate**;

The following image shows an example of how you can register for these events:

```
public class EventsDemo : MonoBehaviour
{
    private void Start()
    {
        // Register event handlers with the translation gizmo
        EditorGizmoSystem.Instance.TranslationGizmo.OnGizmoDragStart += OnDragStart;
        EditorGizmoSystem.Instance.TranslationGizmo.OnGizmoDragEnd += OnDragEnd;
        EditorGizmoSystem.Instance.TranslationGizmo.OnGizmoDragUpdate += OnDragUpdate;

        // Register event handlers with the rotation gizmo
        EditorGizmoSystem.Instance.RotationGizmo.OnGizmoDragStart += OnDragStart;
        EditorGizmoSystem.Instance.RotationGizmo.OnGizmoDragEnd += OnDragEnd;
        EditorGizmoSystem.Instance.RotationGizmo.OnGizmoDragUpdate += OnDragUpdate;

        // Register event handlers with the scale gizmo
        EditorGizmoSystem.Instance.ScaleGizmo.OnGizmoDragStart += OnDragStart;
        EditorGizmoSystem.Instance.ScaleGizmo.OnGizmoDragEnd += OnDragEnd;
        EditorGizmoSystem.Instance.ScaleGizmo.OnGizmoDragUpdate += OnDragUpdate;
    }

    private void OnDragStart(Gizmo gizmo)
    {
        Debug.Log("Drag start");
    }

    private void OnDragEnd(Gizmo gizmo)
    {
        Debug.Log("Drag end");
    }

    private void OnDragUpdate(Gizmo gizmo)
    {
        Debug.Log("Drag update");
    }
}
```

So listening to these events is a simple matter of creating the event handlers and then registering them with the gizmos. In this example, there is an event handler for each type of event, but you might want to create event handlers only for the events that you are interested in.

6.1.2 Gizmo Object Masks

It may sometimes be useful to instruct the gizmos to ignore certain objects or object layers. For example, you may want the move gizmo to ignore some trees objects in the scene. In that case you can assign the tree objects to the move gizmo's mask. Or, if the tree objects have been assigned to a dedicated layer, you can mask the entire object layer.

Here are the methods that you will need in order to mask/unmask objects and object layers. These methods belong to the '**Gizmo**' base class so they can be used with any gizmo type:

- **public void MaskObject(GameObject gameObject);**
- **public void UnmaskObject(GameObject gameObject);**
- **public void MaskObjectCollection(IEnumerable<GameObject> collection);**
- **public void UnmaskObjectCollection(Ienumerable<GameObject> collection);**
- **public bool IsGameObjectMask(GameObject gameObject);**
- **public void MaskObjectLayer(int objectLayer);**
- **public void UnmaskObjectLayer(int objectLayer);**
- **public void IsObjectLayerMasked(int objectLayer);**
- **public bool CanObjectBeManipulated(GameObject gameObject) –** returns true if the game object can be manipulated by the gizmo. Currently this is equivalent to checking if the game object is not masked and it doesn't belong to a masked layer;

Some small examples with the translation gizmo:

```
TranslationGizmo trGizmo = EditorGizmoSystem.Instance.TranslationGizmo;
trGizmo.MaskObject(myObject);
trGizmo.MaskObjectLayer(objectLayer);
trGizmo.MaskObjectCollection(myObjectCollection);
trGizmo.UnmaskObjectCollection(myObjectCollection);
```

… and so on for other methods and gizmos.

6.1.2.1 Gizmo Masks and Object Hierarchies

There is an important detail to be remembered when using masks with object hierarchies. For example, let's assume that object B is a child of object A and somewhere in code you called **MaskObject(B).** However, because object A hasn't been masked, when object A is moved, B is going to be affected also.

6.2 The Messaging System

A messaging system is used to communicate important events that happen at runtime, such as when the object selection changes or when the user has finished manipulating objects with a gizmo.

6.2.1 Message Types

The current message types that are sent are defined in the **Assets\Runtime Transform Gizmos\Scripts\Messaging System\MessageType.cs**. This file defines an enum called **MessageType** which holds all message types. Here is a complete list:

- **GizmoTransformedObjects** → this message is sent when the user has finished transforming objects with a gizmo. This process begins when the user selects one of the gizmo components, presses the left mouse button, moves the mouse around and it ends when the left mouse button is released.
- **GizmoTransformOperationWasUndone** → this message is sent when the user has undone a gizmo transform operation (e.g. move objects with translation gizmo and then undo).
- **GizmoTransformOperationWasRedone** → this message is sent when the user has redone a gizmo transform operation (e.g. move object with translation gizmo, undo and then redo).
- **ObjectSelectionChanged** → this message is sent whenever the object selection is changed in any way (i.e. when objects are added or removed from the selection).
- **ObjectSelectionModeChanged** → this message is sent whenever the object selection mode is changed.
- **ObjectsAddedToSelectionMask** → this message is sent when one or more objects were added to the selection mask.
- **ObjectsRemovedFromSelectionMask** → this message is sent when one or more objects were removed from the selection mask.
- **TransformSpaceChanged** → this message is sent whenever the gizmo transform space is changed.
- **GizmosTurnedOff** → this message is sent when the gizmos are turned off.
- **TransformPivotPointChanged** → this message is sent when the transform pivot point has changed.
- **ActiveGizmoTypeChanged** → this message is sent when the user activates a new type of gizmo (e.g. switches from a translation gizmo to a scale gizmo).
- **VertexSnappingEnabled** → this message is sent when vertex snapping is enabled while using a translation gizmo;
- **VertexSnappingDisabled** → this message is sent when vertex snapping is disabled while using a translation gizmo.

Currently these are all the messages that are sent by different modules. More will most likely be added in future updates.

6.2.2 The 'Message' class

There is a base class called **Message** which is derived by all types of messages which can be sent. This class is implement in the **Message.cs** script which can be found at the following location: **Assets\Runtime Transform Gizmos\Scripts\Messaging System.** The following image shows a screenshot of this class:

```
/// <summary>
/// This class represents a message that can be sent to listeners. It is a base
/// abstract class which must be derived by each type of message that can be sent.
/// </summary>
public abstract class Message
{
    #region Private Variables
    /// <summary>
    /// The message type.
    /// </summary>
    private MessageType _type;
    #endregion

    #region Public Properties
    /// <summary>
    /// Returns the message type.
    /// </summary>
    public MessageType Type { get { return _type; } }
    #endregion

    #region Constructors
    /// <summary>
    /// Constructor.
    /// </summary>
    public Message(MessageType type)
    {
        _type = type;
    }
    #endregion
}
```

As you can see, the **Message** class is an abstract class that has only one member variable of type **MessageType**. This represents the type of message that is associated with an instance of a derived message type. The type can only be specified via the constructor.

6.2.3 Derived Message Types

The **DeirvedMessageTypes.cs** scripts contains a series of message classes which derive from **Message.** This script can be found at the following location: **Assets\Runtime Transform Gizmos\Scripts\Messaging System.**

This file contains classes for all message types which are defined in the **MessageType** enum. The image below shows an example of a derived message class for the object selection changed message:

```
/// <summary>
/// This message is sent when the object selection changes.
/// </summary>
public class ObjectSelectionChangedMessage : Message
{
    #region Constructors
    /// <summary>
    /// Constructor.
    /// </summary>
    public ObjectSelectionChangedMessage()
        : base(MessageType.ObjectSelectionChanged)
    {
    }
    #endregion

    #region Public Static Functions
    /// <summary>
    /// Convenience function for sending an object selection changed message to
    /// all interested listeners.
    /// </summary>
    public static void SendToInterestedListeners()
    {
        var message = new ObjectSelectionChangedMessage();
        MessageListenerDatabase.Instance.SendMessageToInterestedListeners(message);
    }
    #endregion
}
```

All derived message classes follow the same format (some of them might have additional data associated with them). The constructor of the derived class calls the base class constructor specifying the correct type of message.

All message classes also have a static function called **SendToInterestedListeners**. This is just a shortcut function which allows the client code to send a message of that type to all interested listeners. In the image above no parameter is required, but for other message classes, this function may receive one or more parameters depending on what data is needed to construct the message instance.

6.2.4 The 'IMessageListener' Interface

Classes that want to listen and respond to messages must first implement the **ImessageListener** interface. This interface is defined inside the **ImessageListener.cs** script which can be found at the following location: **Assets\Runtime Transform Gizmos\Scripts\Messaging System**.

Currently, the interface provides only one method: void **RespondToMessage(Message message).** This method must be implemented by all classes that want to listen to different kinds of messages.

6.2.5 The MessageListenerDatabase Class

Making a class implement the **IMessageListener** interface is not enough in order to allow it to receive messages from other modules. You must also register the instance of that class as a listener with the

message listener database.

The message listener database is implemented inside the **MessageListenerDatabase.cs** script at the following location: **Assets\Runtime Transform Gizmos\Scripts\Messaging System**.

The **MessageListenerDatabase** class is a singleton class which provides 2 public methods that are important:

```
#region Public Methods
/// <summary>
/// Sends the specified message to all listeners which are interested in that message.
/// </summary>
public void SendMessageToInterestedListeners(Message message)
{
    // Get the list of listeners which listen to the message and send the message to them
    HashSet<IMessageListener> interestedListeners = null;
    if (TryGetListenersForMessage(message, out interestedListeners)) SendMessageToListeners(message, interestedListeners);
}

/// <summary>
/// Registers the specified listener for the specified message type. If the
/// specified listener already listens to the specified message type, the
/// method has no effect.
/// </summary>
public void RegisterListenerForMessage(MessageType messageType, IMessageListener messageListener)
{
    // Already registered?
    if (DoesListenerListenToMessage(messageType, messageListener)) return;

    // Register the listener
    RegisterNewMessageTypeIfNecessary(messageType);
    _messageTypeToMessageListeners[messageType].Add(messageListener);
}
#endregion
```

SendMessageToInterestedListeners allows you to send a message to all listeners which listen to that type of message.

RegisterListenerForMessage allows you to specify that a listener wants to listen to a certain message type. The first parameter to this method is the message type to which the listener wants to listen and the second parameter is an instance of a class which implements the **IMessageListener** interface. **Note:** A listener can listen to more than one type of message.

Let's assume that you built a new module for your editor application called **MyNewModule.** This could be a Monobehaviour or a standard C# class. Let's also assume that you want this class to listen to 2 message types: the object selection change message and the active gizmo type changed message. Then, the class will have to look something like this:

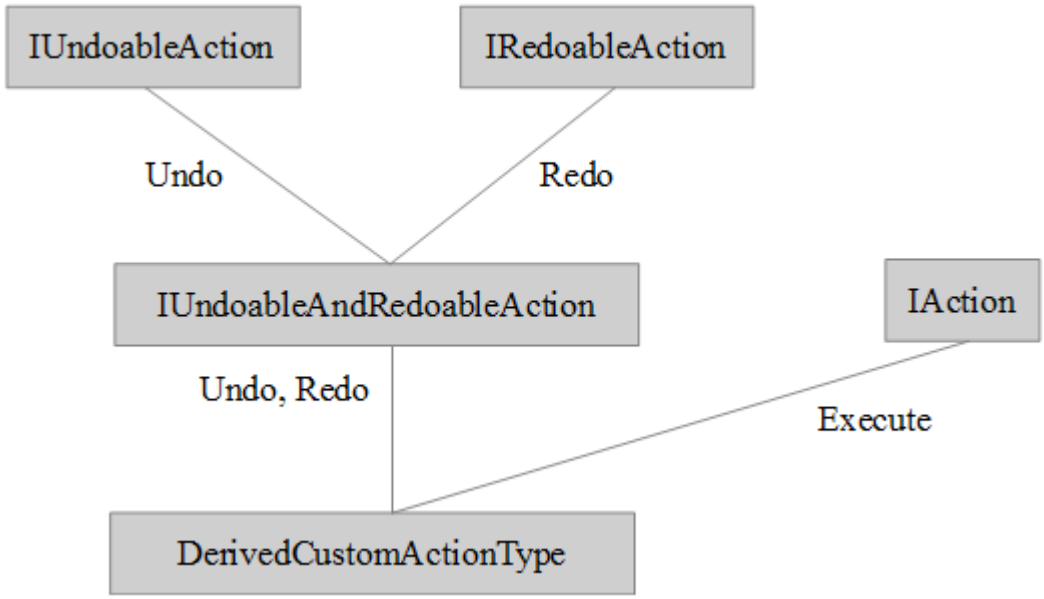
```
public class MyNewModule : MonoBehaviour, IMessageListener
{
    private void Start()
    {
        MessageListenerDatabase listenerDatabase = MessageListenerDatabase.Instance;
        listenerDatabase.RegisterListenerForMessage(MessageType.ObjectSelectionChanged, this);
        listenerDatabase.RegisterListenerForMessage(MessageType.ActiveGizmoTypeChanged, this);
    }

    #region Message Handlers
    public void RespondToMessage(Message message)
    {
        switch(message.Type)
        {
            case MessageType.ObjectSelectionChanged:

                RespondToMessage(message as ObjectSelectionChangedMessage);
                break;

            case MessageType.ActiveGizmoTypeChanged:

                RespondToMessage(message as ActiveGizmoTypeChangedMessage);
                break;
        }
    }

    private void RespondToMessage(ObjectSelectionChangedMessage message)
    {
        // Write code to handle the object selection changed message
    }

    private void RespondToMessage(ActiveGizmoTypeChangedMessage message)
    {
        // Write code to handle the active gizmo type changed message
    }
    #endregion
}
```

As you can see in the image above, in order to have a class listen to different kinds of messages, you have to derive from the **IMessageListener** interface and then implement the **RespondToMessage(Message message)** method. Here, we are using a switch statement to detect the type of message that we are dealing with and then call the corresponding overloaded **RespondToMessage** method.

In the **Start** method, we also register the class instance as a listener with the message listener database. In this case we are dealing with a class which derives from Monobehaviour and this is why we are using the **Start** method for the registration. If the class was a standard C# class, the same registration code could reside inside the class's constructor.

6.3 Actions

Every action that can be executed and then undone or redone resides in its own class which implements 2 interfaces: **IUndoableAndRedoableAction** and **IAction.** The following diagram shows the relationship between the derived action types and the interfaces they implement:



Every action that can be executed and undone or redone, implements the **IUndoableAndRedoableAction** and **IAction** interfaces. These interfaces reside at the following location: **Assets\Runtime Transform Gizmos\Scripts\Editor Undo Redo System\Actions.**

For example, when the active gizmo type is changed to the translation gizmo, the following code is executed (the code is taken from the **EditorShortcutKeys** class which can be found at the following location: **Assets\Runtime Transform Gizmos\Scripts\Editor Shortcut Keys)**:

```
if (Input.GetKeyDown(KeyCode.W))
{
    var action = new ActiveGizmoTypeChangeAction(gizmoSystem.ActiveGizmoType, GizmoType.Translation);
    action.Execute();
}
```

In this case the action is executed when the **W** key is pressed. The action's constructor takes 2 parameters: the first one represents the current active gizmo type and the second one represents the new gizmo type (the one which must be activated).

Note: If you decide to let the user change the active gizmo type from another place (most likely a GUI button), you have to write the same code in order to ensure that the action can be undone and redone properly.

For example, let's look at how you might implement an event handler for a button which is supposed to change the active gizmo type to the translation gizmo:

```
private void OnTranslationGizmoActivateButtonClick()
{
    var action = new ActiveGizmoTypeChangeAction(EditorGizmoSystem.Instance.ActiveGizmoType, GizmoType.Translation);
    action.Execute();
}
```

Here, we use the singleton instance of the **EditorGizmoSystem** to access the active gizmo type which is needed for the first parameter.

6.3.1 Anatomy Of An Action

Every action implements the **IUndoableAndRedoableAction** and **IAction** interfaces, so each action must implement 3 methods: **Execute** (from **IAction**), **Undo** and **Redo** (from **IundoableAndRedoableAction).**

Let's see an example of this. The next images show the constructor and the 3 methods as they are implemented for the **TransformSpaceChangeAction**. This is an action that is executed when the transform space is changed.

```
#region Constructors
/// <summary>
/// Constructor.
/// </summary>
/// <param name="oldTransformSpace">
/// This is the transform space which is active before the action is executed.
/// </param>
/// <param name="newTransformSpace">
/// The new transform space.
/// </param>
public TransformSpaceChangeAction(TransformSpace oldTransformSpace, TransformSpace newTransformSpace)
{
    _oldTransformSpace = oldTransformSpace;
    _newTransformSpace = newTransformSpace;
}
#endregion
```

```
/// <summary>
/// Executes the action.
/// </summary>
public void Execute()
{
    // Execute the action only if the transform spaces differ
    if (_oldTransformSpace != _newTransformSpace)
    {
        EditorGizmoSystem.Instance.TransformSpace = _newTransformSpace;
        TransformSpaceChangedMessage.SendToInterestedListeners(_oldTransformSpace, _newTransformSpace);
        EditorUndoRedoSystem.Instance.RegisterAction(this);
    }
}
```

```
/// <summary>
/// This method can be called to undo the action.
/// </summary>
public void Undo()
{
    EditorGizmoSystem gizmoSystem = EditorGizmoSystem.Instance;
    gizmoSystem.TransformSpace = _oldTransformSpace;
}

/// <summary>
/// This method can be called to redo the action.
/// </summary>
public void Redo()
{
    EditorGizmoSystem gizmoSystem = EditorGizmoSystem.Instance;
    gizmoSystem.TransformSpace = _newTransformSpace;
}
```

The constructor receives 2 parameters: the old transform space (i.e. the one that is currently active) and the new one which must be activated. The old transform space is needed so that the action can be undone.

The **Execute** method first checks to see if the 2 transform spaces differ. This is not strictly required, but it is better to do it this way because otherwise the action will be registered with the undo system for no good reason. Then, it performs the following steps:

- It activates the new transform space using the editor gizmo system singleton instances;
- It sends a **TransformSpaceChangedMessage** to all interested listsners;
- It registers the action with the undo system so that it can be undone/redone if needed.

The **Undo** method activates the transform space which was active before the action was executed and the **Redo** method activates the transform space which was activated when the action was executed.

Depending on the type of action that you have to implement, these 3 methods can be more simple or complex depending on what needs to be done. Also, it is not strictly necessary to send a message inside the **Execute** method, but I recommend that you always do this. Usually, when an action is performed, other modules will need to know about it.

6.3.2 Defining An Action

In this chapter we will see an example of how you might go about implementing a new action that you might need for your own editor application. Let's assume that you have implemented a graphical user interface that allows the user to change different gizmo properties. Let's also assume that you have decided to let the user undo and redo the action of changing a gizmo's axis color.

The **Gizmo** base class contains 2 methods that will help you do this:

- **public Color GetAxisColor(GizmoAxis axis);**
- **public void SetAxisColor(GizmoAxis axis, Color color);**

We will also assume that you would like other modules to know when this action happens. **Note:** This might not be needed in reality. It really depends on what you are trying to accomplish, but it serves as a good example of how you might define your own message type.

Here are the steps that you need to perform to accomplish this:

- In the **MessageType** enum, define a new message called **GizmoAxisColorChanged;**
- In the **DerivedMessageTypes.cs** script, define a new message which corresponds to the message type that you have just added. The implementation of this class might look somehting like in the following image:

```
public class GizmoAxisColorChangedMessage : Message
{
    private Color _newColor;        // The new color of the gizmo axis
    private GizmoAxis _gizmoAxis;   // The gizmo axis whose color was changed
    private Gizmo _gizmo;           // The gizmo whose axis color was changed

    public Color NewColor { get { return _newColor; } }
    public GizmoAxis GizmoAxis { get { return _gizmoAxis; } }
    public Gizmo Gizmo { get { return _gizmo; } }

    public GizmoAxisColorChangedMessage(Color newColor, GizmoAxis gizmoAxis, Gizmo gizmo)
        : base(MessageType.GizmoAxisColorChanged)
    {
        _newColor = newColor;
        _gizmoAxis = gizmoAxis;
        _gizmo = gizmo;
    }

    public static void SendToInterestedListeners(Color newColor, GizmoAxis gizmoAxis, Gizmo gizmo)
    {
        var message = new GizmoAxisColorChangedMessage(newColor, gizmoAxis, gizmo);
        MessageListenerDatabase.Instance.SendMessageToInterestedListeners(message);
    }
}
```

Note: Implementing the static function **SendToInterestedListeners** is not mandatory, but it relieves the client code from having to access the message listener database instance and also from creating a message instance. It is just a convenience function.

- Inside the **EditorActions.cs** script, define a new action class that represents the gizmo axis color change action. This is script is located at the following location: **Assets\Runtime Transform Gizmos\Scripts\Editor Undo Redo System\Actions\Editor Actions**. The implementation of the class might look something like in the following image:

```
public class GizmoAxisColorChangeAction : IUndoableAndRedoableAction, IAction
{
    private Color _oldColor;        // The color of the gizmo axis before the change action is executed
    private Color _newColor;        // The new color of the gizmo axis after the change action is executed
    private GizmoAxis _gizmoAxis;   // The axis whose color is changed when the action is executed
    private Gizmo _gizmo;           // The gizmo object whose axis color is changed when the action is executed

    public GizmoAxisColorChangeAction(Color oldColor, Color newColor,
                                      GizmoAxis gizmoAxis, Gizmo gizmo)
    {
        _oldColor = oldColor;
        _newColor = newColor;
        _gizmoAxis = gizmoAxis;
        _gizmo = gizmo;
    }
    public void Execute()
    {
        if(_newColor != _oldColor)
        {
            _gizmo.SetAxisColor(_gizmoAxis, _newColor);
            GizmoAxisColorChangedMessage.SendToInterestedListeners(_newColor, _gizmoAxis, _gizmo);
            EditorUndoRedoSystem.Instance.RegisterAction(this);
        }
    }
    public void Undo()
    {
        _gizmo.SetAxisColor(_gizmoAxis, _oldColor);
        GizmoAxisColorChangedMessage.SendToInterestedListeners(_newColor, _gizmoAxis, _gizmo);
    }
    public void Redo()
    {
        _gizmo.SetAxisColor(_gizmoAxis, _newColor);
        GizmoAxisColorChangedMessage.SendToInterestedListeners(_newColor, _gizmoAxis, _gizmo);
    }
}
```

Now, in order to change the color of the active gizmo's X axis for example, you would have to write the following code:

```
EditorGizmoSystem gizmoSystem = EditorGizmoSystem.Instance;
Gizmo activeGizmo = gizmoSystem.ActiveGizmo;

var action = new GizmoAxisColorChangeAction(activeGizmo.GetAxisColor(GizmoAxis.X), newColor, GizmoAxis.X, activeGizmo);
action.Execute();
```

6.3.3 The 'EditorActions.cs' Script

This script contains all the actions that can be executed and undone or redone and it is located at the following location: **Assets\Runtime Transform Gizmos\Scripts\Editor Undo Redo System\Actions\Editor Actions.** You can define your own actions inside this file.

Note: The actions currently defined in this file are all the actions which can be undone and redone at the moment.

6.4 The 'EditorShortcutKeys.cs' Script

This script contains a Monobehaviour class that handles all the shortcut keys. The code here is pretty much self-explainable and you can easily modify it if needed. One thing to note is that the shortcut key handling is done in separate methods based on the system to which they are related. This can be seen in the class's **Update** method shown in the next image:

```
#region Private Methods
/// <summary>
/// Called every frame to perform any necessary updates.
/// </summary>
private void Update()
{
    HandleEditorGizmoSystemKeys();
    HandleEditorUndoRedoSystemKeys();
    HandleEditorObjectSelectionKeys();
}
#endregion
```

As you can see there are 3 methods that handle the shortcut keys for the gizmo system, the undo/redo system and the object selection system. I chose to do it this way because it feels like it helps keep things more organized but you can certainly change this if you wish.

6.5 Object Colliders

In order to take advantage of the object selection functionality, objects in the scene must have colliders attached to them. Here are a few things to keep in mind:

- The system supports **box, sphere, capsule and mesh colliders**;
- The system currently aids you in the process of attaching colliders **only to game objects which have a mesh, light or particle system component attached to them**.

6.5.1 Attaching Colliders At Runtime

Your editor application might allow the user to create objects at runtime. In order to make sure those objects can be selected, you need to attach colliders to them. You can do this easily using the **ObjectColliderAttachement** singleton class. This class has a method called **AttachCollidersToObjectHierarchy** which attaches colliders to all game objects that reside in a specified hierarchy. The method has 2 parameters. The first parameter represents the root of the object hierarchy that must be processed and the second parameter is an instance of the **ObjectColliderAttachmentSettings** class which controls the collider attachment process.

6.5.1.1 The 'ObjectColliderAttachmentSettings' Class

This class can be found inside the **ObjectColliderAttacmentSettings.cs** script which resides at the following location: **Assets\Runtime Transform Gizmos\Scripts\Object Colliders.**

We will now discuss the properties which this class exposes. These are the properties which allow you to control the way in which colliders are attached to the game objects:

- **ColliderTypeForMeshObjects** → this is a member of the **ObjectColliderType** enum and it allows you to specify what type of collider must be attached for objects that have a mesh attached to them. This can be a mesh which belongs to a mesh filter or to a skinned mesh renderer;
- **ColliderTypeForLightObjects** → this is the type of collider that must be attached to objects that have a light component attached to them;
- **ColliderTypeForParticleSystemObjects** → this is the type of collider that must be attached to objects that have a particle system component to them;
- **BoxColliderSizeForNonMeshObjects** → if a box collider was specified as a collider type for light or particle system objects, this allows you to control the size of the box collider;
- **SphereColliderRadiusForNonMeshObjects** → if a sphere collider was specified as a collider type for light or particle system objects, this allows you to control the radius of the sphere collider;
- **CapsuleColliderRadiusForNonMeshObjects** → if a capsule collider was specified as a collider type for light or particle system objects, this allows you to control the radius of the capsule collider;
- **CapsuleColliderHeightForNonMeshObjects** → if a capsule collider was specified as a collider type for light or particle system objects, this allows you to control the height of the capsule collider.
- **Ignore Mesh Objects** → if this is set to true, mesh objects will be ignored during the collider attachment process. That means that their colliders will be left intact. If they have no colliders, none will be attached.
- **Ignore Light Objects** → if this is set to true, light objects will be ignored during the collider attachment process. That means that their colliders will be left intact. If they have no colliders, none will be attached.
- **Ignore Particle System Objects** → if this is set to true, particle system objects will be ignored during the collider attachment process. That means that their colliders will be left intact. If they have no colliders, none will be attached.

It is up to you to create an instance of this class and set the correct property values when you need to attach colliders to game objects.

Note: This functionality exists to relieve you of the chore of adding colliders to game objects, but you can certainly create your own methods and rules for handling collider attachment.

6.5.1.2 Attaching Colliders At Startup

In chapter 5.1, we discussed that you can control certain settings that define the way in which colliders are attached to game objects at startup. The code that is responsible for performing this step can be found in the **RuntimeEditorApplication.cs** script located at **Assets\Runtime Transform Gizmos\Scripts\Runtime Editor**. The following image shows the line of code which performs the collider attachment step:

```
#region Private Methods
/// <summary>
/// Performs any necessary initializations.
/// </summary>
private void Start()
{
    // If necessary, create the mesh vertex group mappings for vertex snpping
    if (_editorApplicationStartupSettings.AcquireVertexSnappingInfoOnStartup) MeshVertexGroupMappings.Instance.CreateMappingsForAllSceneMeshObjects();

    // Attach any colliders if necessary
    if ( editorApplicationStartupSettings.AttachObjectCollidersToAllSceneObjectsAtStartup)
        ObjectColliderAttachment.Instance.AttachCollidersToAllSceneObjects(_editorApplicationStartupSettings.ObjectColliderAttachmentSettings);
}
#endregion
```

You can modify this if you wish in order to suit your own application needs.

6.6 Object Selection

6.6.1 Object Selection Masks

It may be possible that you want some of the objects in the scene to be ignored by the object selection module. If this is the case, you can assign objects to the object selection mask at runtime. Objects assigned to the selection mask will never be selected.

There are 2 ways to assign objects to a selection mask:

- with **no** Undo/Redo support:

```
var objectCollection = new List<GameObject>();  // Populate this as needed.
EditorObjectSelection.Instance.AddGameObjectCollectionToSelectionMask(objectCollection);
```

- with Undo/Redo support:

```
var objectCollection = new List<GameObject>();  // Populate this as needed.
var action = new AssignObjectsToSelectionMaskAction(objectCollection);
action.Execute();
```

Note: When you add objects to a selection mask, if any of those objects are selected, they will be automatically deselected.

Similarly, you can also remove objects from the selection mask:

- with **no** Undo/Redo support:

```
var objectCollection = new List<GameObject>();  // Populate this as needed.
EditorObjectSelection.Instance.RemoveGameObjectCollectionFromSelectionMask(objectCollection);
```

- with Undo/Redo support:

```
var objectCollection = new List<GameObject>();  // Populate this as needed.
var action = new RemoveObjectsFromSelectionMaskAction(objectCollection);
action.Execute();
```

6.6.2 Changing The Object Selection Mode

You can change the object selection mode at runtime and you can do it in 2 ways, much like the way in which you do with object selection masks:

- with **no** Undo/Redo support:

```
// Assume we want to activate the 'EntireHierarchy' selection mode
EditorObjectSelection.Instance.ObjectSelectionSettings.ObjectSelectionMode = ObjectSelectionMode.EntireHierarchy;
```

- with Undo/Redo support:

```
// Assume we want to activate the 'EntireHierarchy' selection mode
var action = new ObjectSelectionModeChangeAction(ObjectSelectionMode.EntireHierarchy);
action.Execute();
```

6.6.3 Enabling and Disabling Object Selection

There may be times when you wish to disable the object selection mechanism to stop it from selecting objects in the scene. For this purpose, you can use the **SetEnabled** method and pass true if you want to enable and false to disable.

Note: When you disable the object selection mechanism, the system will check if the **DeselectObjectsWhenDisabled** property is true. If it is, any selected objects will be deselected when the selection mechanism is disabled.

6.6.4 Customizing the Object Selection Mechanism

You can customize the way in which object selection is performed. In order to do this, you need to perform the following steps:

- create a class which derives from **ObjectSelectionGameObjectClickedHandler.** This class has an abstract method called **Handle** and the parameter that it receives represents the game objects which was clicked in the scene. The method must return true when the selection changes and false otherwise.
- create a class which derives from

ObjectSelectionGameObjectsEnteredSelectionShapeHandler. This class has an abstract method called **Handle** and the parameter it receives represents the game objects which have entered the object selection shape (currently, this can only be a selection rectangle). The method must return true when the selection changes and false otherwise.

- create a class which derives from **ObjectSelectionBoxCalculator.** This class has an abstract method which is called **CalculateForObjectSelection.** This method receives a has set which contains all the currently selected objects and it must return a list of selection boxes. How you calculate the selection boxes and return them to the caller is totally up to you. For example, when the **IndividualObjects** selection mode is active, the implementation of this method calculates a selection box for each object and returns the selection boxes to the caller.
- use the **EditorObjectSelection.Instance.CustomObjectClickedHandler, EditorObjectSelection.Instance.CustomObjectsEnteredSelectionShapeHandler, EditorObjectSelection.Instance.CustomObjectSelectionBoxCalculator** properties to set your own custom handlers;
- activate the **Custom** object selection mode using one of the methods described in chapter 7.5.2.

When customizing object selection, you might want to take a look at how the handlers for the other object selection modes were implemented. These are:

- **EntireHierarchyObjectSelectionGameObjectClickedHandler**;
- **EntireHierarchyObjectSelectionGameObjectsEnteredSelectionShapeHandler**;
- **IndividualObjectsSelectionGameObjectClickedHandler**;
- **IndividualObjectsSelectionGameObjectsEnteredSelectionShapeHandler**;
- **EntireHierarchySelectionBoxCalculator**;
- **IndividualObjectsSelectionBoxCalculator**.

Note: If you activate the **Custom** object selection mode, but forget to activate the custom handlers, object selection will not work correctly. All handlers must be specified in order for custom selection to work correctly. For example, if you specified the game object clicked handler, but forgot to specify the other handlers, you will be able to select by clicking objects in the scene, but you will not be able to use a selection shape for object selection. Also, because the selection box calculator was not specified, you won't see any selection boxes drawn in the scene when objects are selected.