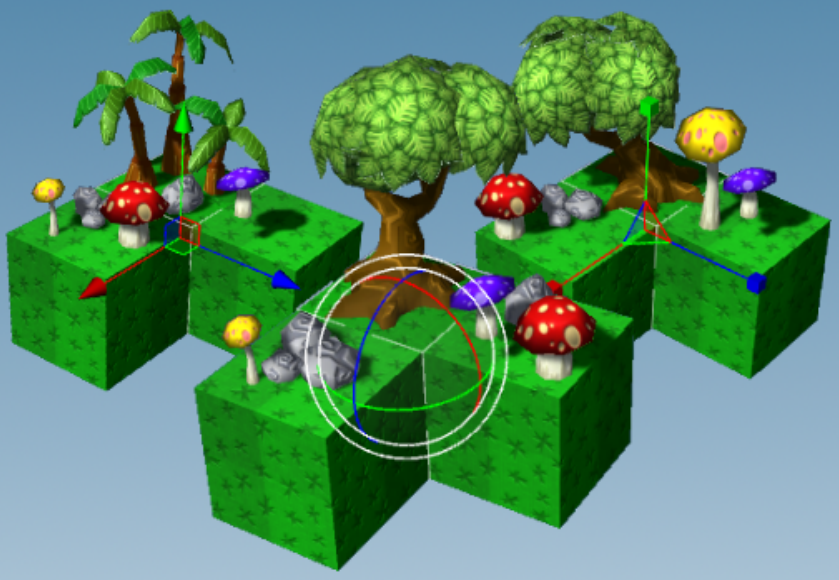


RUNTIME TRANSFORM GIZMOS

Table of Contents

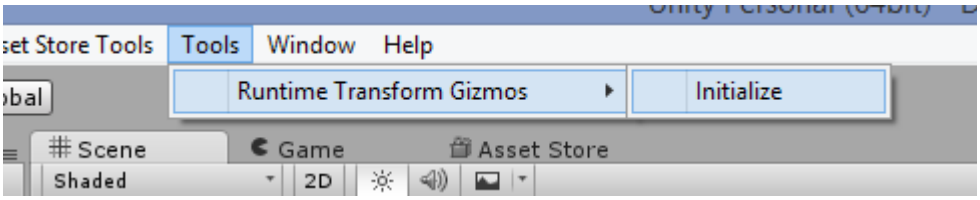
1 Introduction

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 helps you understand how to use the system API for your own app development needs.

2 System Setup

After you have imported the package, you are just a few clicks away from setting up the system. Here are the steps that you need to perform:

- import the **Runtime Transform Gizmos** package;
- go to **Tools->Runtime Transform Gizmos** and click on **Initialize** as shown in the following image:



That's all there is to it.

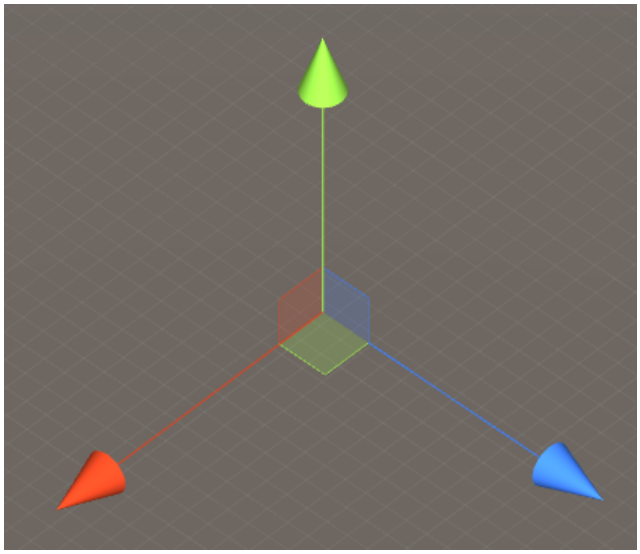
3 The Gizmos

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.

3.1 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:



3.1.1 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.

Note: The move gizmo can be used to perform certain special operations, like the camera axes translation discussed above or vertex snapping etc. When a special operation is being performed, a square will appear in the center of the gizmo. I will be referring to this as the special operation square or special op square for short.

3.1.1.1 Terrain and Mesh Surface Placement

The move gizmo can be used to greatly speed up terrain and mesh surface placement. With it you can place an object (or collection of objects) on the surface of a terrain or mesh and also align the objects' axes to the surface normal.

Here are the shortcut keys that allow you to do this:

- **SPACE –** the special op square will appear and if you left click the left mouse button inside the square and then drag while hovering a terrain/mesh, the object(s) and the gizmo will move along the terrain/mesh surface and the objects' Y axis will be aligned with the surface normal;
- **SPACE + X –** same, but this time the X axis will be aligned with the surface normal;
- **SPACE + Z** – same, but this time the Z axis will be aligned with the surface normal;

- **SPACE + LCTRL –** with this combination the object(s)' positions will be snapped to the surface, but no axis alignment will be performed. **Note:** In this case, because no axis alignment is done for the objects, the objects will most likely become embedded inside the surface because the system no longer knows how to offset the object accordingly since no reference axis is specified.

3.1.1.2 Grid Surface Placement

The move gizmo can also be used to place and align objects on the grid surface using the same combination of keys discussed previously. **Note:** Grid placement will only be performed if the hotkeys are active and if no terrain or mesh object is hovered by the mouse cursor.

3.1.1.3 Snapping

The translation gizmo supports 3 types of snapping: **Step, Vertex Snapping** and **Box Snapping.**

3.1.1.3.1 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).

3.1.1.3.2 Vertex to Vertex Snapping

Vertex to 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.

3.1.1.3.3 Vertex to Grid Snapping

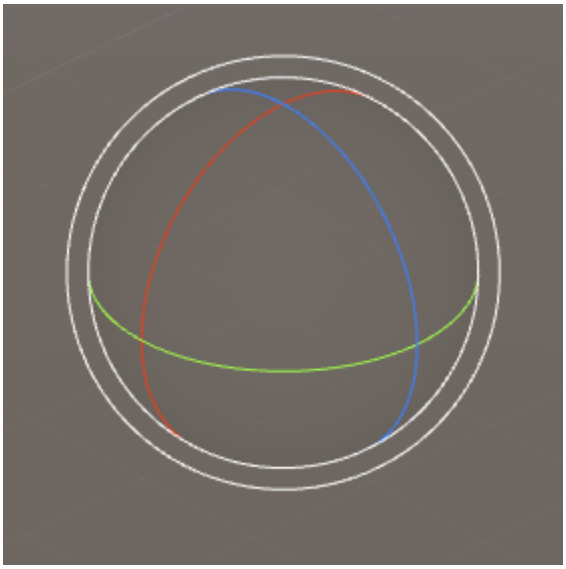
This works in the same way as **vertex-to vertex snapping,** but this time the source vertex will be snapped to one of the grid's cell corner points.

3.1.1.3.4 Box Snapping

Works in the same way as vertex snapping, but it applies to the center and corner points of the objects' bounding box.

3.2 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:



3.2.1 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.

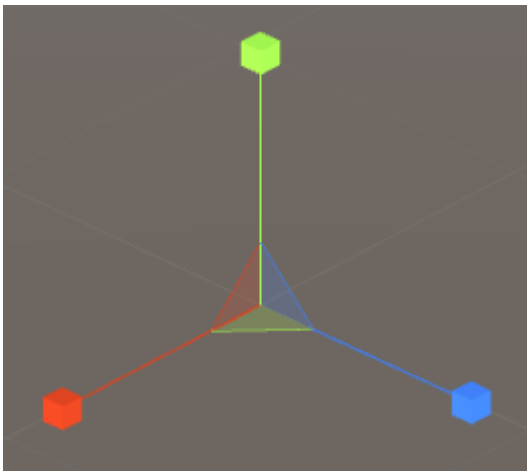
3.2.1.1 Snapping

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.

3.3 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:



3.3.1 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.

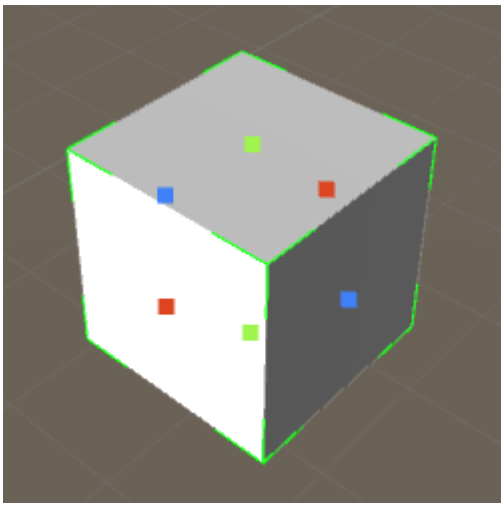
If you want to perform a scale operation along all axes at once, you have to keep the **SHIFT** key pressed and then drag the mouse around.

3.3.1.1 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).

3.4 The Volume Scale Gizmo

The volume scale gizmo is another type of gizmo which allows you to perform object scaling, but it works a bit differently than the standard scale gizmo. Instead of using axes that can be dragged, the gizmo works in almost the same way as the box collider widget that Unity exposes to allow you to modify box colliders. This in combination with the step snapping functionality can provide a much more intuitive scaling interface in some scenarios.



3.4.1 Using the Volume Scale Gizmo

The gizmo is composed of 6 drag handles which can be dragged to scale the object. There are 2 drag handles for each axis. Dragging one of them will scale along the corresponding object local axis.

By default, when you drag, the gizmo will behave like the Unity Box Collider widget meaning that the size and position of the object will be affected.

Holding down **SHIFT before** dragging will cause the scale to happen from the center of the object. This key can be modified from the gizmo's inspector.

Note: The volume scale gizmo only works when a single object is selected and that object has to have a mesh attached to it. When more than one object is selected, the gizmo will be hidden.

3.4.1.1 Snapping

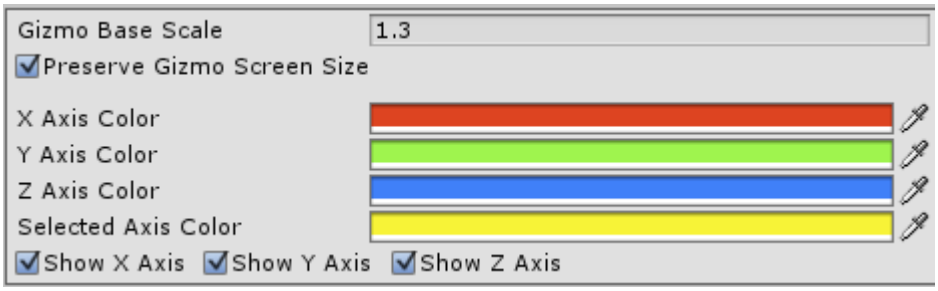
By default, if you hold down **LCTRL** while dragging the handles, the gizmo will scale the object in increments of a specified step size (can be modified in the Inspector). The shortcut key which enabled snapping can also be changed from the Inspector.

3.5 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.

3.5.1 Common Gizmo Properties

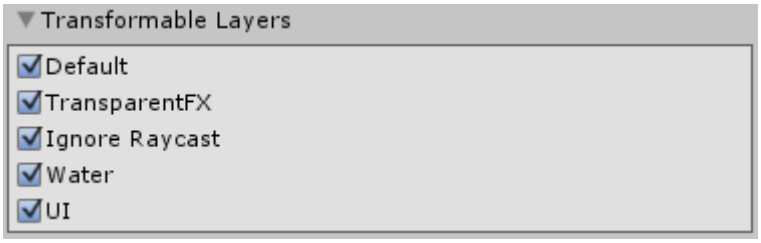
There are a few properties which apply to all gizmos. These properties will always be shown at the top of a gizmo objects' Inspector. We will discuss these properties here and we will ignore them when we discuss the properties for each gizmo individually. The following image shows the first category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



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;
- **Show X/Y/Z Axis –** allows you to show/hide different gizmo axes.

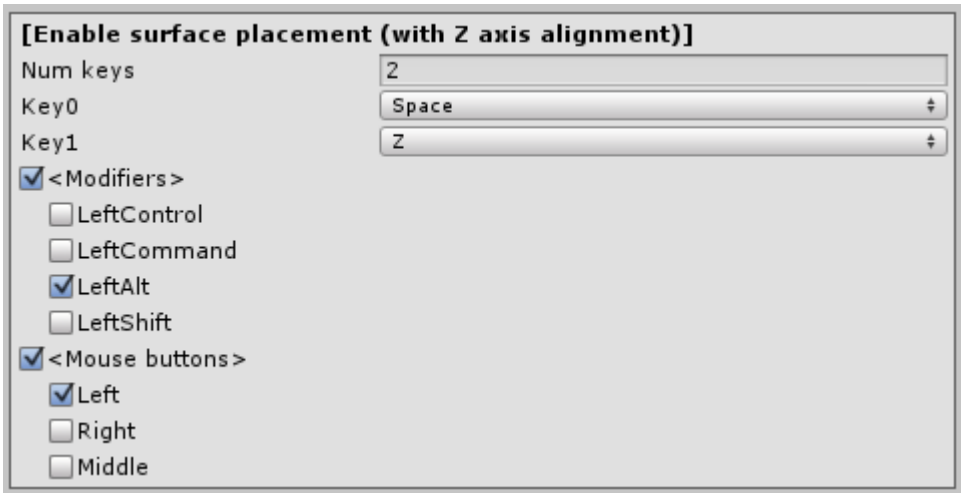
The second category of common properties is shown next:



It simply allows you to specify which layers can be transformed/affected by a gizmo. For example, if the **Default** layer would be unchecked for the translation gizmo, the translation gizmo would not be able to transform objects which belong to the **Default** layer.

3.5.1.1 Key Mappings

All gizmos have key mappings associated with them which allows you to change the hotkeys which activate a specific gizmo behaviour (e.g. vertex snapping, step snapping, surface placement, special op etc). The format in which key mappings are presented in the Inspector is shown in the following image:



This is an example of key mappings for activating surface placement for the move gizmo with Z axis alignment. Note that the default values of the properties are different than what is shown here. They were only modified to prove a point.

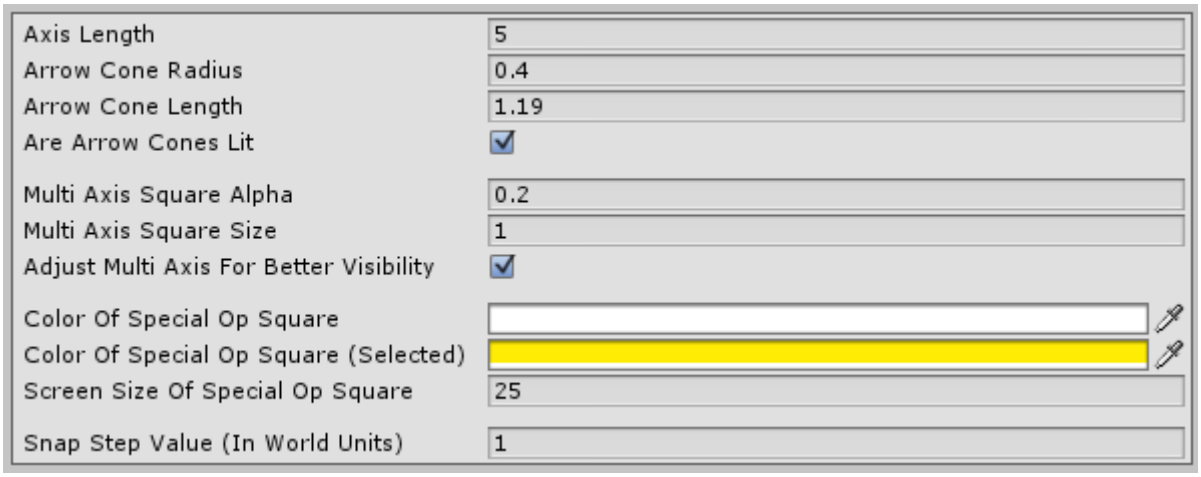
A mapping allows you to specify the hotkeys, modifiers and mouse buttons which must be pressed in order to perform a special action.

At the top, we have the name of the mapping written in bold and which also describes what the mapping is for. The rest of the properties are discussed next:

- **Num keys –** a mapping can have a maximum of 2 hotkeys associated with it and in this case these are the **Space** and **Z** keys. You can set this number to 0 if no keys are required;
- **Key0, Key1 –** the 2 keys which are associated with the mapping. In this case both keys are shown because the number of keys was set to 2. But if it were set to 1, only **Key0** would be visibile. Settings **NumKeys** to 0, would remove both entries from the Inspector. Settings one of the key entries to **None** will deactivate that key. So for example, if you wanted to deactivate **Key1**, you could set it to **None** or set the number of keys 1. If you wanted to deactivate both keys, you could set both keys to **None** or set the number of keys to 0;
- <**Modifiers**> – check this if the mapping should contain modifier keys. If not checked, the modifier selection toggles will be hidden. When checked, you can toggle one or more of the options below to specify which modifier key should be associated with the mapping. Deactivating all modifiers can be done either by unchecking all modifier key options or by unchecking **<Modifiers>**;
- **<Mouse buttons> -** same as **<Modifiers>,** but it applies to mouse buttons.

3.5.2 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):



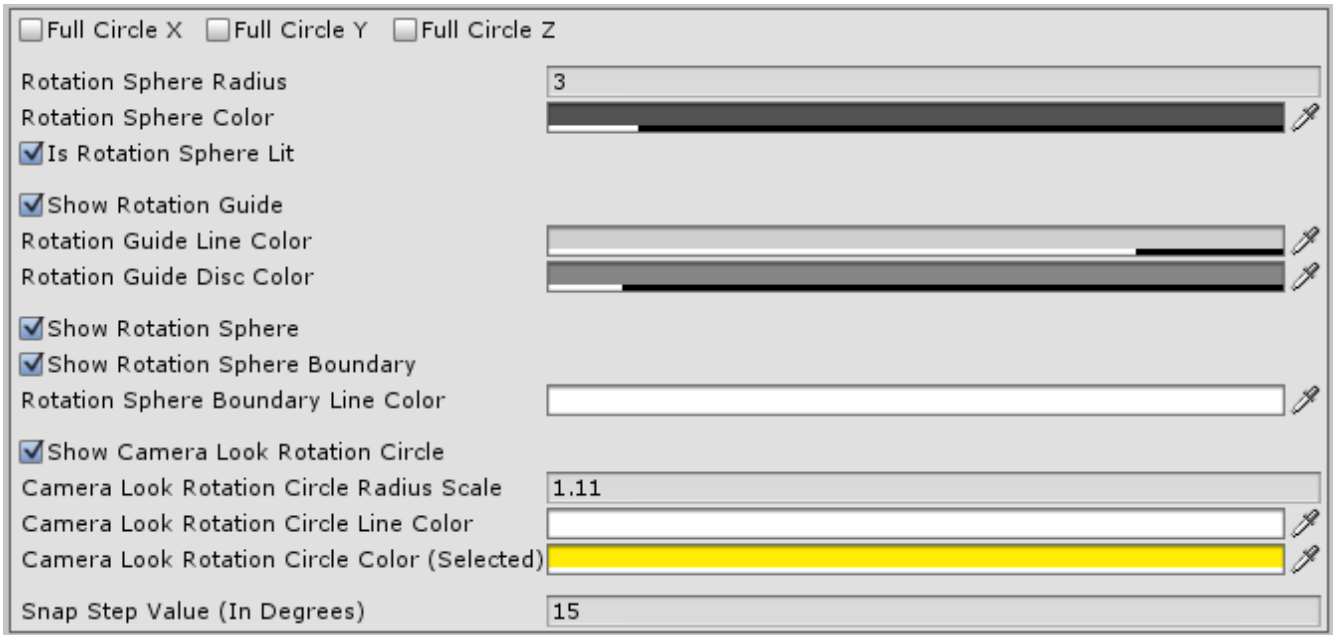
We will now discuss the properties in the same order in which they appear in the Inspector GUI:

- **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;
- **Multi Axis Square Alpha –** allows you to control the alpha value of the multi-axis translation squares;
- **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 Special Op Square** → allows you to control the color of the lines that make up the special op square which is activated when a special operation is about to be performed (camera axes translation, vertex snapping or terrain/grid surface placement);
- **Color Of Special Op Square (Selected)** → same as the property discussed above, but it applies when the square is hovered by the mouse cursor;
- **Screen Size Of Special Op Square** → allows you to control the screen size of the special op 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.

3.5.3 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:

- **Full Circle X/Y/Z –** by default, when the gizmo is rendered, the rotation circles are drawn only in half. These 3 toggles allow you to change that behavior for each rotation circle;
- **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;
- **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 property 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.

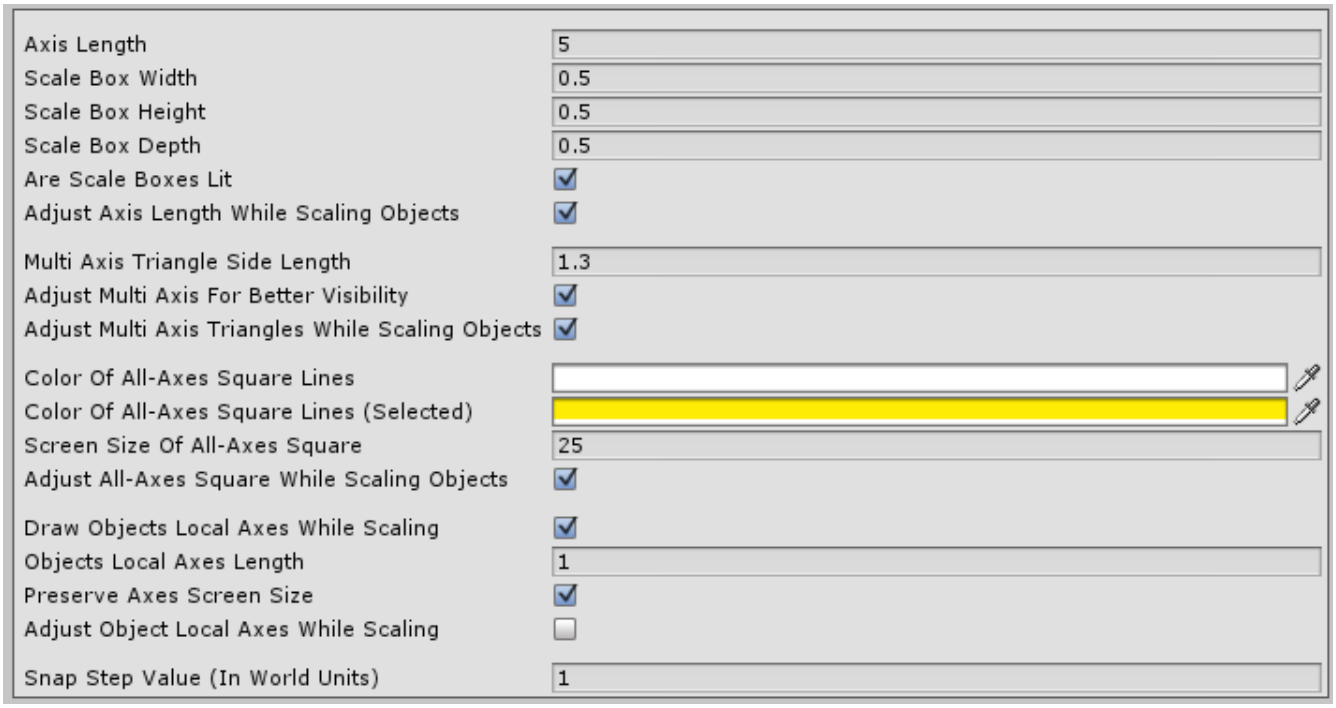
The rotation guide is composed of 2 guide lines which form a rotation arc. The area which is formed by the 2 guide lines is called the rotation guide disc.

- **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.

3.5.4 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;

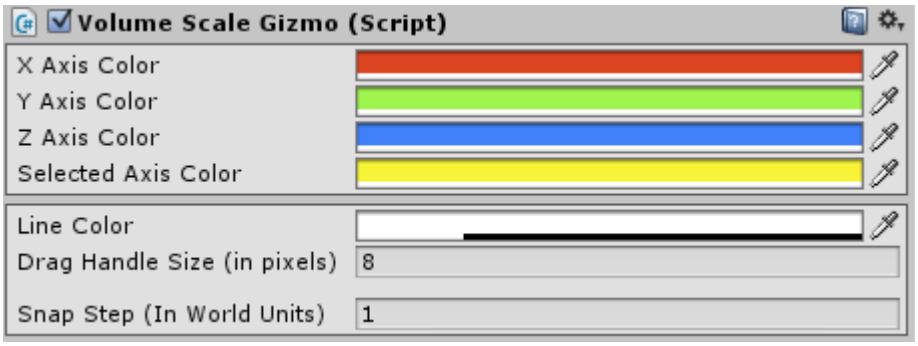
- **Scale Box Width/Height/Depth** → these 3 properties allow you to control the size of the scale boxes which sit at the tip of the gizmo axes;
- **Are Scale Boxes Lit** → if checked, the scale boxes will be affected by lighting when rendered. You can uncheck this property if you desire a gizmo with 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;
- **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.

3.5.5 Volume Scale Gizmo Properties

The following image shows all the properties which can be modified in the volume scale gizmo's inspector:

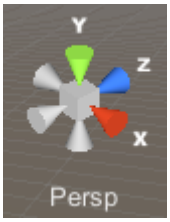


- **X, Y, Z, Selected Axis Color –** allows you to control the color of the handles which correspond to different axes;
- **Line Color –** the color of the lines which make up the gizmo's box. By default a low alpha value is used so that the lines don't totally overwrite the object selection box lines. Modifying this color could mean that you also need to modify the color of the object selection box lines so that the 2 can play well together;
- **Drag Handle Size (in pixels) –** the pixel size of the drag handles;
- **Snap Step (In World Units) –** when snapping is active, this controls the snap step increments. **Note:** The snap increments are specified in world units. For example, if this value is set to 2, the gizmo will scale objects such that their size will increase/decrease in increments of 2 world units.

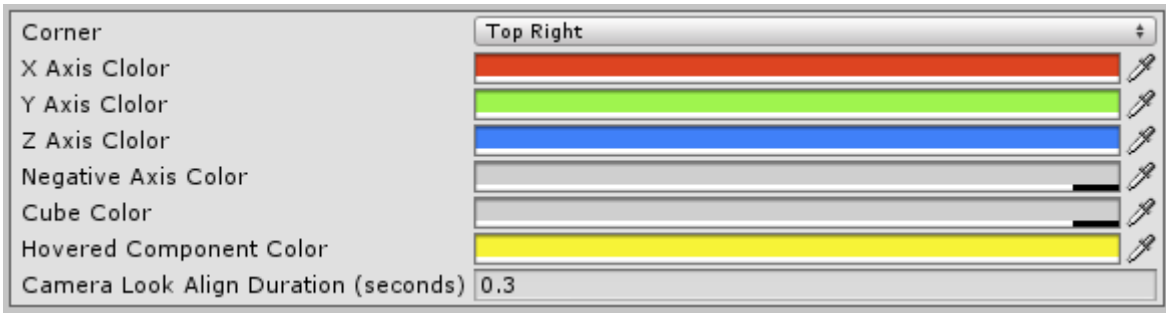
3.5.6 Scene Gizmo

The package contains a scene gizmo very similar to the one which is implemented in Unity with a few minor exceptions:

- it does not support a smooth perspective transition. The perspective switch is done instantly;
- when clicking on one of the cones, the camera position will not be affected. Only its rotation will be adjusted so that the camera look vector becomes aligned with the corresponding axis.



The next image shows the scene gizmo Inspector GUI:



- **Corner –** allows you to choose the corner in which the scene gizmo will be drawn. Possible values are **TopLeft, TopRight, BottomRight** and **BottomLeft**;
- **X,Y,Z Axis Color –** allows you to specify the color of the gizmo axes (positive axes);
- **Negative X Axis Color –** the color of the negative axes;
- **Cube Color –** the color of the cube which connects the gizmo axes cones;
- **Hovered Component Color –** the color of a gizmo component (cube, cone) when it is hovered by the cursor;
- **Camera Look Align Duration (seconds) –** the duration in seconds for the camera look alignment. This alignment happens when clicking on of the gizmo axis cones.

3.6 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.**

3.6.1 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.

3.6.2 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.

3.7 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.

3.7.1 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.

3.7.2 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.

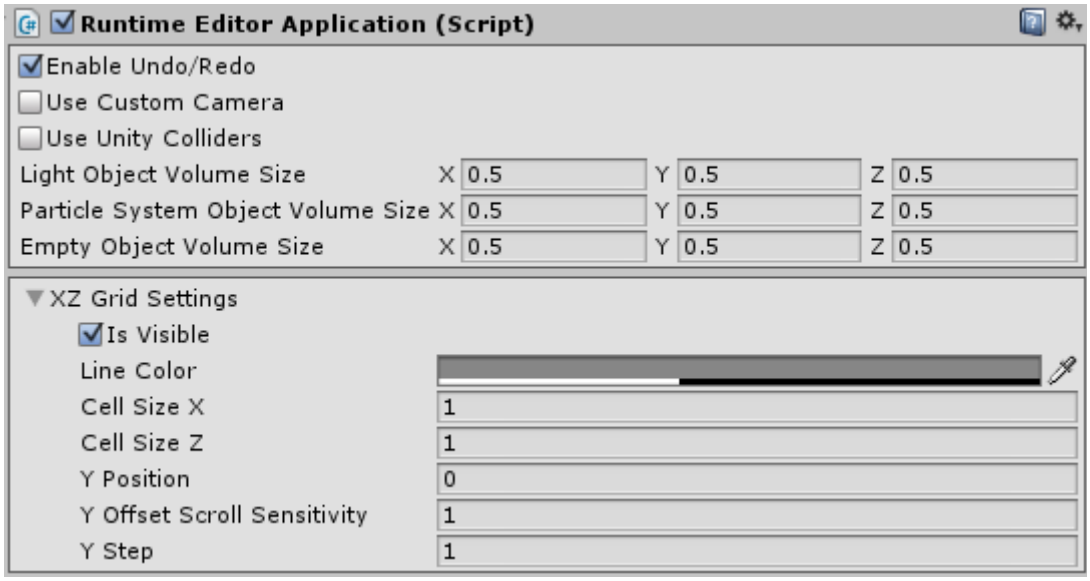
Note: For objects that don't have a mesh attached to them, the pivot point is the same as their center.

4 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.

4.1 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. All other subsystems (including the gizmos themselves) are child objects of the Runtime Editor Application object. The following image shows the inspector GUI:



- **Enable Undo/Redo –** if this is checked, the Undo/Redo system will be enabled. Uncheck this if you don't want to have access to Undo/Redo functionality;
- **Use Custom Camera –** Check this if you wish to use your own custom camera. By default the system uses a camera which allows you to navigate the scene and perform some other useful operations such as focus on selected objects. If however you would like to use another camera, you can uncheck this toggle which will cause a new field to appear. You can then drag and drop the desired camera object onto that field. **Note:** The **EditorCamera** object will still appear in the scene regardless of whether or not this toggle is checked. Calling **EditorCamera.Instance.Camera** from script will return the camera you have chosen to use.
- **Use Unity Colliders –** if this is checked, you will need to attach colliders to the game objects in order to be able to interact with them (e.g. select, surface placement, vertex snapping etc). If not checked, the system will use its own API to handle object interaction. The advantage of this is that you won't need to attach any colliders to your game objects. **Note:** I recommend that you **check** this option if you are working with meshes which are very high-res. In that case, you might experience slow frame rates at application startup. On the other hand, if you are working with sprite objects, you will need to **uncheck** this because otherwise you will not be able to interact with them.
- **Light/Particle System/Empty Object Volume Size** → only visible if **Use Unity Colliders** is

unchecked. In that case, the tool now uses its own internal object representation which makes interacting with the scene objects possible. These 3 values allow you to define the volume size for light, particle systems and empty objects. Empty objects are objects that do not have a mesh, light, terrain, sprite or particle system component attached to them. If selection is enabled for any of these object types, the system will use this volume size to define the objects' bounding box to determine if the object was selected or not (via click or selection rectangle);

The next section discusses the settings which relate to the scene XZ grid. This grid is useful in a number of ways:

- it provides a sense of orientation;
- it can be used for vertex snapping with the move gizmo;
- it can be used as a surface to place objects on with the move gizmo;

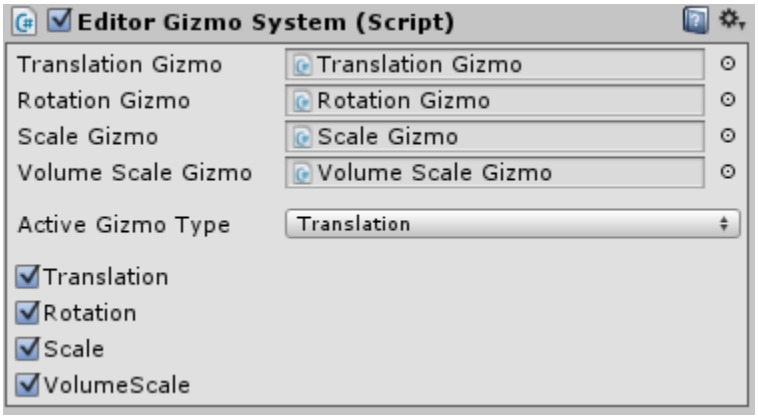
Here are the settings which relate to the grid:

- **Is Visible –** allows you to specify if the grid should be rendered in the scene;
- **Line Color –** the grid line color;
- **Cell Size X –** the grid cell size on the X axis;
- **Cell Size Z –** the grid cell size on the Z axis;
- **Y Position –** the position of the grid along the Y axis;
- **Y Offset Scroll Sensitivity –** when adjusting the height of the grid using the mouse scroll wheel, this property is used to specify how sensitive the grid is to the mouse scroll wheel; **Note:** This property only applies when NO snapping/stepping is used.
- **Y Step –** the step value which is used when the Y position of the grid is modified using stepping/snapping enabled.

4.2 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.

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. This is automatically set when you initialize the system.
- **Rotation Gizmo** → this is the rotation gizmo which can be used to rotate objects in the scene. This is automatically set when you initialize the system.
- **Scale Gizmo** → this is the scale gizmo which can be used to scale objects in the scene. This is automatically set when you initialize the system.
- **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.
- **Translation/Rotation/Scale/VolumeScale –** a series of toggle buttons which allow you to turn different types of gizmos on/off. For example, if you don't need the rotation gizmo, then you can uncheck the **Rotation** toggle and the rotation gizmo will not be available.

The remaining controls in the GUI allow you to specify different key mappings and they are not shown here because they should be self-explainable. Please see chapter 3.4.1.1 for more details on how key mappings work.

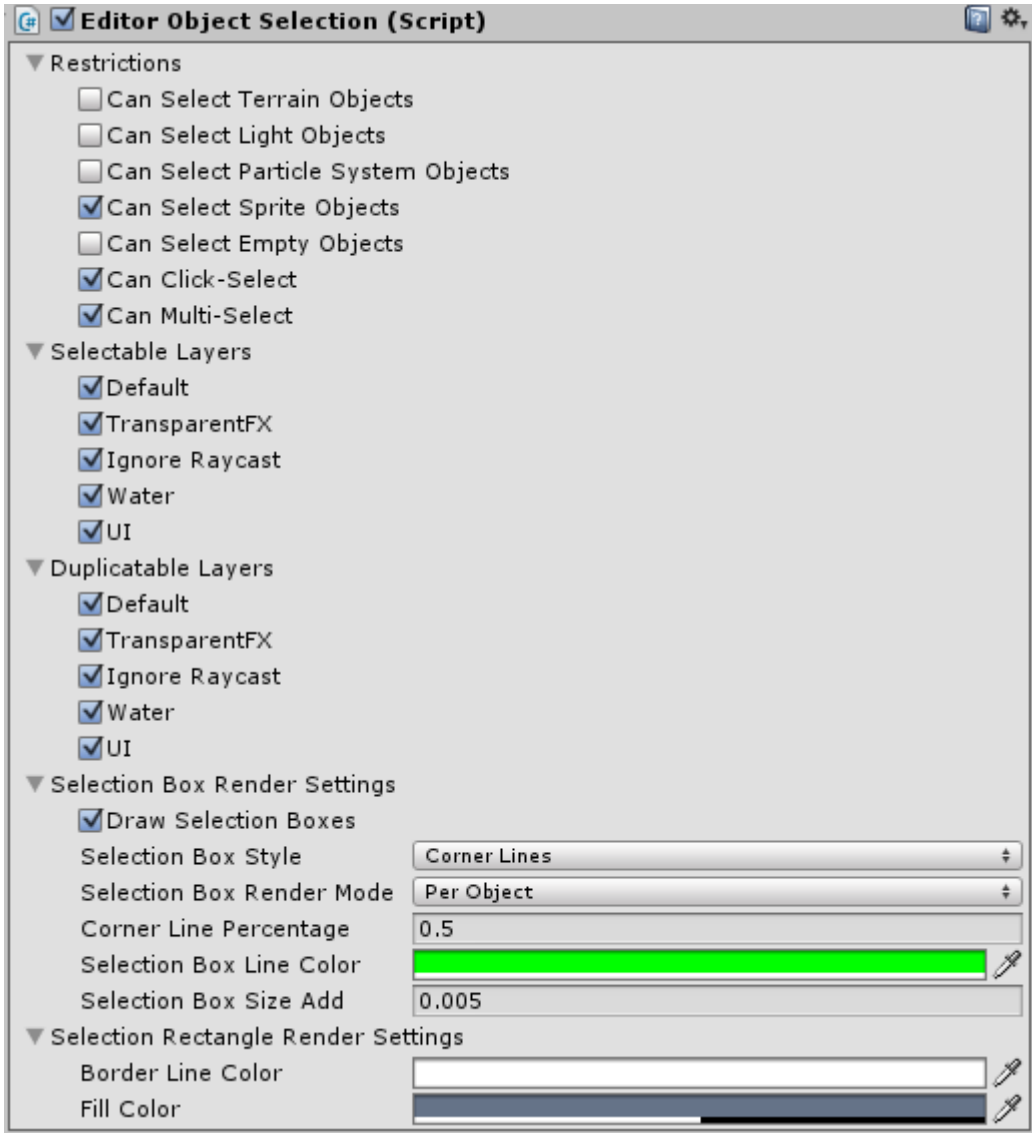
4.3 The Object Selection System

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: If **Use Unity Colliders** is checked in the **RuntimeEditorApplication** Inspector, you will only be able to select objects that have a collider attached to them.

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:

- **Can Select Terrain Objects –** if this is checked, terrain objects can be selected;
- **Can Select Light Objects –** if this is checked, light objects can be selected;
- **Can Select Particle System Objects –** if this is checked, particle system objects can be selected;
- **Can Select Sprite Objects –** if this is checked, sprite objects can be selected;
- **Can Select Empty Objects –** if this is checked, empty objects can be selected. Empty objects are objects that do not have a mesh, light, particle system, sprite or terrain component attached to them;
- **Can Multi-Select –** allows you to specify if objects can be selected using the selection rectangle;
- **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;

- **Duplicatable Layers –** this is a section which allows you to specify which layers can be duplicated. When a layer is checked, objects which belong to that layer can be duplicated.

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

- **Draw Selection Boxes** → you can uncheck this if you don't want to draw 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:

- **Selection Box Render Mode –** specifies the way in which the selection boxes are rendered. The possible values are:
  - **PerObject –** a selection box is rendered for each selected object;
  - **FromParentToBottom –** the system will separate the parents from children in the selected

object collection and will only render a selection box which encapsulates the parent and all its children;

- **Corner Line Percentage** → this property is used only when the object selection box style is set to **Corner Lines**. It specifies a percentage of half the box width/height/depth which can is used to render the corner lines. For example, a value of 0.5 indicates that the corner line will be half of the box half dimensions. A value of 1 will make the corner lines equal to the box half dimensions;
- **Selection Box Line Color** → this is the color used to draw the selection box lines.
- **Selection Box Size Add** → a small offset that is added to the selection box size when it is rendered. Necessary in order to avoid Z fighting. For example, when cubes are selected and assuming no additional offset was used, the selection box line pixels would be fighting with the cube pixels and gaps would appear in the selection box lines;

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

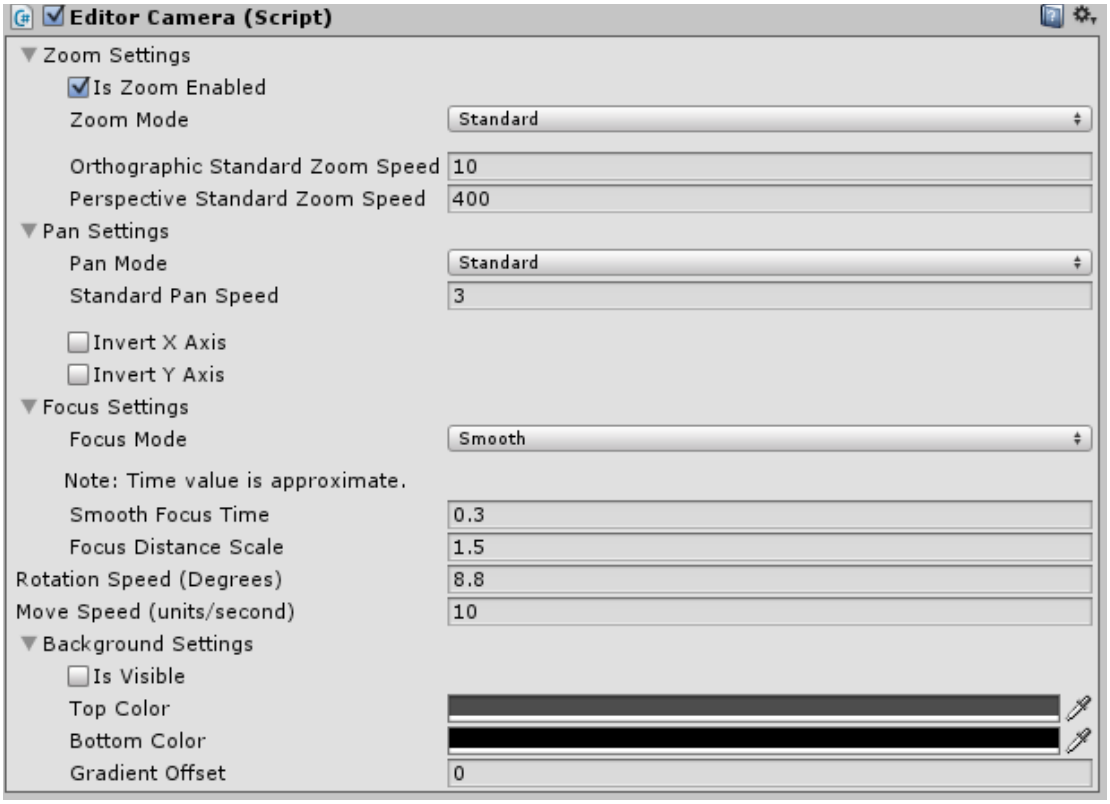
- **Border Line Color** → this is the color that is used to draw the border of the object selection rectangle.
- **Fill Color** → this is the color which is used to fill the selection rectangle.

The remaining controls in the GUI allow you to specify different key mappings and they are not shown here because they should be self-explainable. Please see chapter 3.4.1.1 for more details on how key mappings work.

4.4 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 next 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.

The last section allows you to control camera background settings:

- **Is Visible –** specifies if the background is visible. If it is, a gradient will be rendered in the background;
- **Top Color –** specified the background gradient top color;
- **Bottom Color –** specifies the background gradient bottom color;
- **Gradient Offset –** takes values in the [-1, 1] interval and it allows you to specify that one color should be rendered more than the other. For example, smaller values will render more of the top color. Bigger values will render more of the bottom color. A value of -1 will render only the top color. A value of 1 will render only the bottom color;

4.5 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.

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 Default hotkeys

Although, you can customize the key mappings for different actions that you can perform, this chapter lists the default shortcut keys. **Note:** Hotkeys that contains **LCTRL** will not work properly inside the Game View and you will need to add the **SHIFT** key to the mix in order to make them work.

- **W** → activate the translation gizmo;
- **E** → activate the rotation gizmo;
- **R** → activate the scale gizmo;
- **U** → activate volume scale gizmo;
- **Q** → 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);
- **SPACE + MouseScrollWheel –** modify grid Y position;
- **SPACE + LCTRL + MouseScrollWheel –** modify grid Y position with STEP;
- Object duplication:
  - When in play mode in the Unity Editor:
    - **LCTRL + SHIFT + D;**
  - When in build mode:
    - **LCTRL + D;**
- Gizmo specific keys:
  - Translation gizmo:

        - **V** → while held down, allows you to perform vertex snapping. When released, vertex snapping is disabled;
        - **B –** while held down, allows you to perform box snapping. When released, box snapping is disabled;
        - **LCTRL** → while held down, allows you to perform step snapping;
        - **SHIFT** → while held down, allows you to translate along the camera right and up axes;
        - **SPACE –** terrain (or grid if no terrain is hovered) surface placement with Y axis alignment;
        - **SPACE + X –** terrain (or grid if no terrain is hovered) surface placement with X axis alignment;
        - **SPACE +Z -** terrain (or grid if no terrain is hovered) surface placement with Z axis alignment;
        - **LCTRL + SPACE** -terrain (or grid if no terrain is hovered) surface placement with **no axis alignment;**
        - **LALT –** activate move scale (scale value can be specified in the gizmo inspector).
    - Rotation gizmo:
        - **LCTRL** → while held down, allows you to perform step snapping;
    - Scale gizmo:
        - **LCTRL** → while held down, allows you to perform step snapping;
        - **LSHIFT** → while held down, allows you to perform a scale operation along all axes at once.
    - Volume scale gizmo:
        - **LCTRL** → while held down, allows you to perform step snapping;
        - **LSHIFT** → holding this down before a drag starts, will cause the gizmo to scale from the center of the object;
- Object selection specific keys:
    - **LCTRL** → 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.
    - **LSHIFT** → 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 will most likely need when building your application. The most important thing is to learn how to register to different types of events that happen in your application (object selection, gizmo manipulation etc) so that your applicatino can act accordingly.

6.1 The IRTEditorEventListener interface

There may be times when you wish to listen to different types of events which can be sent to each object individually. For this reason, the system exposes the **IRTEditorEventListener** which can be implemented by all **Monobheaviors** which must listen to those events.

The interface is shown in the following image:

```
public interface IRTEditorEventListener
{
    bool OnCanBeSelected(ObjectSelectEventArgs selectEventArgs);
    void OnSelected(ObjectSelectEventArgs selectEventArgs);
    void OnDeselected(ObjectDeselectEventArgs deselectEventArgs);
    void OnAlteredByTransformGizmo(Gizmo gizmo);
}
```

Now let's talk about each method exposed by the interface. **Note:** We will discuss the **ObjectSelectEventArgs** and **ObjectDeselectEventArgs** in the end, after all methods have been covered.

- **OnCanBeSelected –** called by the object selection module when the object (Monobehaviour) which implements the interface is about to be selected. This gives you even more control over which objects can be selected or not. The method must return true if the object can be selected and false otherwise;
- **OnSelected –** called by the selection module **after** the object was selected;
- **OnDeselected –** called by the selection module **after** the object was deselected;
- **OnAlteredByTransformGizmo –** called by a gizmo (move, rotate or scale) after it was used to transform an object (change position, rotation or scale).

Now let's talk about the **ObjectSelectEventArgs** and **ObjectDeselectEventArgs** parameters.

The next image shows the properties exposed by the **ObjectSelectEventArgs** class:

```
public ObjectSelectActionType SelectActionType { get { return _selectActionType; } }
public GizmoType GizmoType { get { return _gizmoType; } }
public bool IsGizmoActive { get { return _isGizmoActive; } }
```

- **SelectActionType –** specifies the way in which the object was selected (please see chapter 6.5.3 for more details);
- **GizmoType –** the type of the gizmo which was used when the object was selected;
- **IsGizmoActive –** specifies whether or not the gizmos is active. It may have been deactivated by the user via the corresponding hotkey.

The next image shows the properties exposed by the **ObjectDeselectEventArgs** class:

```
public ObjectDeselectActionType DeselectActionType { get { return _deselectActionType; } }
public GizmoType GizmoType { get { return _gizmoType; } }
public bool IsGizmoActive { get { return _isGizmoActive; } }
```

Only the first property differs in this case. It is called **DeselectActionType** and it is used to specify how the object was deselected. Again, please see chapter 6.5.3 for more details.

Here is an example of you might go about implementing this interface:

```
class MyRTEditorEventListener : MonoBehaviour, IRTEditorEventListener
{
    bool OnCanBeSelected(ObjectSelectEventArgs selectEventArgs)
    {
        // Do stuff
        return true;    // Assume we return true in this example
    }
    void OnSelected(ObjectSelectEventArgs selectEventArgs)
    {
        // Do stuff
    }
    void OnDeselected(ObjectDeselectEventArgs deselectEventArgs)
    {
        // Do stuff
    }
    void OnAlteredByTransformGizmo(Gizmo gizmo)
    {
        // Do stuff
    }
}
```

So simply derive from **Monobehaviour** and **IRTEditorEventListener** and implement the interface methods. Of course, you will need to attach this script to all objects which must listen to these events. The most efficient way to do this would be to attach this script to an object, make that object a prefab and use that prefab to instantiate objects in the scene (either at runtime or in the Editor).

6.1.1 Useful GameObject Extensions

Using the **IRTEditorEventListener** interface you can customize the way in which object selection occurs. For example, you might have a group of mesh objects which are all attached to an empty parent

object. Assuming, empty game object selection is allowed (**Can Select Empty Objects** is checked inside the EditorObjectSelection Inspector), you might want to delegate the selection to this object so that you can manipulate all mesh objects at once. This is just one example, but there are many ways in which the selection can be customized.

This task will usually require some kind of processing of the object hierarchies involved. The following listing shows some useful GameObject extension methods and utility functions which can help you in the process:

- **GameObject GetFirstEmptyParent(this GameObject gameObject) –** the method will start from the specified game object and move up the hierarchy until it finds a parent which is empty. This could be a game object which is an immediate parent or an object further up the hierarchy.

Note: An object is considered empty when all the following conditions are met:

- it has no mesh;
- it has no terrain;
- it has no light component;
- it has no particle system component;
- no sprite renderer with valid sprite;

Example usage: **var firstEmptyParent = childObject.GetFirstEmptyParent();**

- **GameObject GetFirstParentOfType<T>(this GameObject gameObject) –** the method will start from the specified game object and move up the hierarchy until it finds a parent which has a component attached to it of type **T**. This could be a game object which is an immediate parent or an object further up the hierarchy.

Example usage: **var firstLightObject = childObject.GetFirstParentOfType <Light>();**

- **List<GameObject> GetRootObjectsFromObjectCollection(List<GameObject> gameObjects) –** given a list of game objects, the function will return all objects which are the roots of the hierarchies to which the input objects belong. For example, imagine that you had 2 hierarchies: A (with B and C as children) and then D (with E and F as children). If the input list would contain B, and F, the output list will contain A and D because A is the root of the hierarchy in which B resides and D is the root of the hierarchy in which F resides;

There is an overload of this function which accepts a **HashSet** as input, but the function works in the same way.

In order to call this function you would need to write code similar to this:

GameObjectExtensions.GetRootObjectsFromObjectCollection(collection);

- **List<GameObject> GetAllRootsAndChildren(List<GameObject> gameObjects)** – same as the function discussed above, with the exception that this function also returns all the children of the identified root objects. Following the same example, with the 2 hierarchy roots A and B, the function would return a list that contains **A**, B, C, **D**, E and F. A and D are the roots and B, C, E and F are their children. **Note:** No assumption should be made about the order in which the objects are stored inside the output list.

In order to call this function you would need to write code similar to this:

GameObjectExtensions.GetAllRootsAndChildren(collection);

- **List<GameObject> GetAllChildren(this GameObject gameObject) –** this is an extension method which returns all children (direct and indirect) of the 'gameObject'.

Example usage: **var objects = gameObject.GetAllChildren ();**

- **List<GameObject> GetAllChildrenIncludingSelf(this GameObject gameObject) –** this is an extension method which returns a list of all children (direct and indirect) + the object itself.

Example usage: **var objects = gameObject.GetAllChildrenIncludingSelf();**

How you plan on using these methods really depends on what you are trying to achieve, but here is a simple example of a common situation. The next image shows how you might implement the **OnSelected** handler of the **IRTEditorEventListener** interface to delegate the selection up to the first object which has a light component attached to it:

```
public void OnSelected(ObjectSelectEventArgs selectEventArgs)
{
    if(!gameObject.GetComponent<Light>())
    {
        EditorObjectSelection objectSelection = EditorObjectSelection.Instance;
        GameObject firstLightObject = gameObject.GetFirstParentOfType<Light>();
        objectSelection.RemoveObjectFromSelection(gameObject, false);
        objectSelection.AddObjectToSelection(firstLightObject, false);
    }
}
```

VERY IMPORTANT: Always pass false here when calling from inside a handler. Otherwise Undo/Redo will not work correctly.

So when an object is selected, the **GetFirstParentOfType<Light>()** method is called to retrieve the object's first parent that has a light component (**no null check is performed here, but in a real world scenario you might want to check for null in case there is no such object present in the hierarchy).** After this object is retrieved, the object which was just selected is removed from the selection and the light object is added instead.

Note: Care must be taken when implementing such functionality because if you are not careful you could enter an infinite recursive loop. This can happen when calling AddObjectToSelection inside the OnSelected handler because AddObjectToSelection will cause the same handler to be called again. So in some cases you might accidentally keep calling the same function for the same object over and over and you would be stuck in an infinite loop. In the example above there is no such danger because we keep retrieving objects from further up the hierarchy. The 'if' statement is used to detect if we are already dealing with a light object.

Here is another example which selects all children of the selected object:

```
public void OnSelected(ObjectSelectEventArgs selectEventArgs)
{
    EditorObjectSelection objectSelection = EditorObjectSelection.Instance;
    List<GameObject> allChildren = gameObject.GetAllChildren();
    foreach (var child in allChildren) objectSelection.AddObjectToSelection(child, false);
}
```

So when an object is selected, the **GetAllChildren** method is called and the children are also added to the selection. So in the end both the object whose **OnSelected** handler was called and its children will be selected.

These were just some simple examples, but you could implement more sophisticated selection

mechanisms using this approach. Just keep in mind to always have a stop condition to exit the recursive chain and pass **false** as the second parameter to the selection modification functions **(AddObjectToSelection, RemoveObjectFromSelection etc)** to allow the Undo/Redo system to work properly.

6.2 Gizmos

6.2.1 Gizmo events

The gizmo objects expose some events which you may need to listen to sometimes. These are **GizmoDragStart, GizmoDragUpdate and GizmoDragEnd.** All event handlers accept a reference to the gizmo which triggered the event:

```
#region Events
public delegate void GizmoDragStartHandler(Gizmo gizmo);
public delegate void GizmoDragUpdateHandler(Gizmo gizmo);
public delegate void GizmoDragEndHandler(Gizmo gizmo);

public event GizmoDragStartHandler GizmoDragStart;
public event GizmoDragUpdateHandler GizmoDragUpdate;
public event GizmoDragEndHandler GizmoDragEnd;
#endregion
```

So listening to these events is a simple matter of creating the event handlers and then registering them with the gizmos.

In order to register to gizmo events, you will need to access the gizmo objects which you can do as shown below:

```
EditorGizmoSystem.Instance.TranslationGizmo.GizmoDragStart += MyHandler;
EditorGizmoSystem.Instance.RotationGizmo.GizmoDragStart += MyHandler;
EditorGizmoSystem.Instance.ScaleGizmo.GizmoDragStart += MyHandler;
```

In the image above, the same handler is used for the drag start event of all gizmos, but the important thing to remember here is the way in which the gizmos are accessed.

There is another type of event to which you can subscribe, but it belongs to the **EditorGizmoSystem** class. It allows you to detect when the type of gizmo has changed.

```
public delegate void ActiveGizmoTypeChangedHandler(GizmoType newGizmoType);
public event ActiveGizmoTypeChangedHandler ActiveGizmoTypeChanged;
```

Subscribing to this event can be done with the following code:

EditorGizmoSystem.Instance.ActiveGizmoTypeChanged += MyHandler;

MyHandler has to have a prototype which matches the delegate in the above image. The only parameter which the handler must accept is the new type of gizmo.

6.2.2 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.

Note: Object Masks are not supported when using the Volume Scale Gizmo.

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 IsGameObjectMasked(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.

6.2.2.1 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.

6.2.3 Axes Masks

All gizmos (**except for the Volume Scale Gizmo)** support axes masks and these masks apply per object. So for example, you could tell the move gizmo that it can not move a certain object along the Z axis.

The axes masks function calls belong to the **Gizmo** base class so they can be used with all 3 gizmo types. **Note:** The masks apply only when transforming the objects using one of the gizmo axes or multi-axes. Here is a short summary of the gizmo components that are effected by masks:

- move gizmo → axes lines, cones and multi-axis squares;

- rotation gizmo → rotation circles;
- scale gizmo → axes lines, boxes and multi-axis triangles;

Here are the 2 functions which let you register axes masks for different objects:

- **void SetObjectAxisMask(GameObject gameObj, bool[] axisMask)** – registers a mask for the specified game object. The mask is stored in the second array element. A value of **true** indicates that the axis is **not masked.** A value of **false** indicates that the axis is masked;
- **void SetObjectAxisMask(GameObject gameObj, int axisIndex, bool isMasked)** – sets the mask for the specified object for the specified axis (0 – X, 1 – Y, 2 – Z). The last parameter specifies the mask state. If true, the axis will be masked. Otherwise it will be unmasked.

Here are a couple of example:

// Register a mask for gameObj with the move gizmo. The object can be moved along the X and Y, but can not be moved along the Z axis.

TranslationGizmo trGizmo = EditorGizmoSystem.Instance.TranslationGizmo;

trGizmo.SetObjectAxisMask(gameObj, new bool[] {true, true, false};

// Register a mask for gameObj with the rotation gizmo. The object can only be rotated around the X axis.

RotatinoGizmo rotGizmo = EditorGizmoSystem.Instance.RotationGizmo;

rotGizmo.SetAxisMask(gameObj, 0, false);

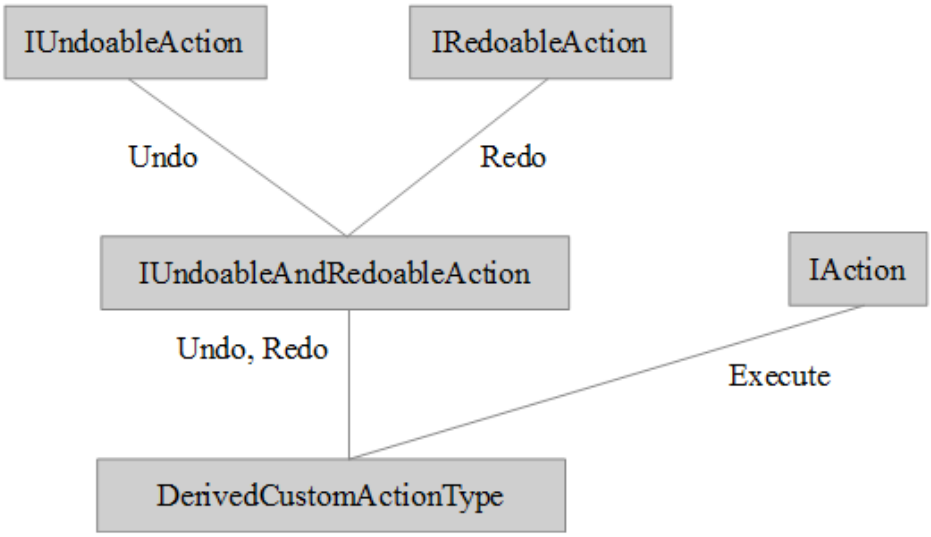
rotGizmo.SetAxisMask(gameObj, 1, true);

rotGizmo.SetAxisMask(gameObj, 2, true);

Note: Something to be aware of. Let's assume that you are using the move gizmo to move a single object and you have specified a mask for that object which stops it from being moved on any axes. In that case, when you drag the gizmo, the gizmo will still move, but the object will remain behind. After you end the drag, the gizmo will snap back where it is supposed to be.

6.3 Actions

Every action that can be executed and then undone or redone (i.e. Undo/Redo) 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:



These interfaces reside at the following location: **Assets\Runtime Transform Gizmos\Scripts\Editor Undo Redo System\Actions.**

For example, object duplication is performed using an action class which implements the Undo, Redo and Execute methods. All actions which can be undone/redone must be wrapped inside a custom action class.

6.3.1 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. **Note:** If no undo/redo is required, you don't need to implement an action class.

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);**

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

- 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);
            EditorUndoRedoSystem.Instance.RegisterAction(this);
        }
    }
    public void Undo()
    {
        _gizmo.SetAxisColor(_gizmoAxis, _oldColor);
    }
    public void Redo()
    {
        _gizmo.SetAxisColor(_gizmoAxis, _newColor);
    }
}
```

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();
```

This would change the color of the active gizmo's X axis and it would also allow for Undo/Redo. Again, if no Undo/Redo is required, there is no need to implement an action class. Using the Gizmo properties/methods directly will work just fine.

6.4 Scene Management

6.4.1 Game Object Sphere Tree

In order to allow you to interact with objects independently of colliders (when **Use Unity Colliders** is unchecked in the Runtime Editor Application Inspector), the tool uses a tree structure to organize the objects in the scene. One action that this tree has to perform is to determine when an objects transform has changed. The system can do this in 2 ways:

- manually keep track of the object transform data (default) which works very well;
- use the **Transform.hasChanged** property of an object to determine if any transform specific data has changed. This is the fastest, but is not activated by default because the tool will then set

this flag to false and you probably want to manage this value yourself in other parts of the code. If you know for sure that you won't need to manually set this value in any parts of your code, you can activate this functionality by performing the following steps:

1. open the **Assets\Runtime Transform Gizmos\Scripts\Scene Management\Bounding Volume Hierarchies\Sphere Tree\Game Object Tree\GameObjectSphereTree.cs** file;
2. **uncomment** the line at the top of the file (i.e. **#define USE_TRANSFORM_HAS_CHANGED);**

The speed-up gained from this is not very significant, but it is something that you might want to do if you desire. Both methods work perfectly fine.

6.4.2 Picking Mesh Objects

The system will need access to the mesh data in order to allow you to pick mesh objects. In case a mesh is marked as non-readable, the system will perform the following steps:

- it will check to see if the game object has a mesh collider attached and use that to perform the pick operation;
- if no mesh collider is present, it will use the game object's box volume for picking.

6.5 Object Selection

The object selection mechanism is implemented in the **EditorObjectSelection.cs** script (**Assets\Runtime Transform Gizmos\Scripts\Editor Object Selection\).**

The selection module is a singleton class which means you can access it by writing code like the following: **EditorObjectSelection.Instance**.

6.5.1 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.

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

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:

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

6.5.2 Manually changing the object selection

Most of the times you will let the object selection module do its job and select objects using the mouse either by clicking or by using the selection rectangle. However, there may be times when you wish to

manually change the object selection. For example, if you wish to implement a CTRL + A functionality which selects all objects in the scene, you will need a way to tell the selection module what objects to select.

For this purpose, the **SetSelectedObjects** method can be used (of the **EditorObjectSelection** class). Its prototype is shown below:

```
public void SetSelectedObjects(List<GameObject> selectedObjects, bool allowUndoRedo)
```

Simply put, the method accepts the collection of objects that you wish to select and a boolean parameter which allows you to specify if the selection operation can be undone/redone.

Note: The method will take into account any restrictions that have been specified (object selection masks, layer masks, can select mesh objects, can select terrain objects etc). Objects which do not pass this filter, will not be selected.

Also, any objects which were previously selected, will be deselected if they don't reside in the 'selectedObjects' collection. So this method **does not append** to the current selection. It **changes** the selection.

You can also manually clear the object selection via a call to **ClearSelection:**

```
public bool ClearSelection(bool allowUndoRedo)
```

The parameter allows you to specify if the clear operation can be undone/redone.

2 other methods which you may find useful are:

- **bool AddObjectToSelection(GameObject gameObj, bool allowUndoRedo) –** appends an object to the current selection. The first parameter represents the object which must be added to the selection and the second parameter specifies whether or not the operation can be undone/redone;
- **bool RemoveObjectFromSelection(GameObject gameObj, bool allowUndoRedo)** – removes the specified object from the selection. The second parameter specifies whether or not the operation can be undone/redone.

6.5.3 Listening to Selection Changed events

It is possible to listen to selection change events by registering an even handler with the selection module, as shown in the following image:

```
EditorObjectSelection.Instance.SelectionChanged += MyHandler;
```

The handler must have the following prototype:

```
public void MyHandler(ObjectSelectionChangedEventArgs args)
```

The only parameter that the handler must accept is an instance of

ObjectSelectionChangedEventArgs.

Now let's discuss the properties/information which is stored in this object:

```
public ObjectSelectActionType SelectActionType { get { return _selectActionType; } }
public List<GameObject> SelectedObjects { get { return new List<GameObject>(_selectedObjects); } }
public ObjectDeselectActionType DeselectActionType { get { return _deselectActionType; } }
public List<GameObject> DeselectedObjects { get { return new List<GameObject>(_deselectedObjects); } }
public GizmoType GizmoType { get { return _gizmoType; } }
public bool IsGizmoActive { get { return _isGizmoActive; } }
```

The first property is an enumerated type called **ObjectSelectActionType**, which is shwon below:

```
namespace RTEditor
{
    public enum ObjectSelectActionType
    {
        Click = 0,
        ClickAppend,
        MultiSelect,
        Undo,
        Redo,
        SetSelectedObjectsCall,
        AddObjectToSelectionCall,
        None
    }
}
```

So it tells the client code that **if any objects** were selected**,** this is the way in which they were selected. And now let's talk about the possible values:

- **Click –** the object was selected via a mouse click;
- **ClickAppend –** the object was selected via a click but with the append to selection hotkey active which caused the object to be appended/added to the current selection;
- **MultiSelect –** the objects were selected via the selection rectangle;
- **Undo –** the objects were selected as part of an Undo operation. This happens when you change the object selection (in any way, shape or form), and then Undo which will restore the object selection to what it as before. So the previously selected objects are re-selected by the Undo mechanism;
- **Redo –** Same as Undo, but it applies to Redo operations;
- **AddObjectToSelectionCall –** the object was selected via a call to **EditorObjectSelection.Instance .AddObjectToSelection;**
- **SetSelectedObjectsCall –** the objects were selected via a call to **EditorObjectSelection.Instance.SetSelectedObjects**.
- **None –** useful in certain scenarios to indicate that no objects were selected.

The second property is a collection which holds all objects which were selectected. Together with the select action type property, these 2 can be used to find out what objects have been selected and in what way. **Note:** If the **SelectActionType** property is set to **None** it means no objects were selected and

when this is the case the object collection is empty.

The next 2 properties are almost the same as the first 2, but they give information about which objects were deselected and in what way. Let's see how the **ObjectDeselectActionType** looks like:

```
namespace RTEditor
{
    public enum ObjectDeselectActionType
    {
        ClearSelectionCall = 0,
        SetSelectedObjectsCall,
        RemoveObjectFromSelectionCall,
        ClearClickAir,
        ClickAlreadySelected,
        MultiDeselect,
        Undo,
        Redo,
        DeselectInactive,
        None
    }
}
```

- **ClearSelectionCall –** the objects were deselected via a call to **EditorObjectSelection.Instance.ClearSelection**;
- **SetSelectedObjectsCall –** the objects were deselected via a call to **EditorObjectSelection.Instance.SetSelectedObjects**. This happens because this method changes the object selection based on the specified collection, so the objects which were previously selected, will be deselected;
- **RemoveObjectFromSelectionCall –** the object was removed from the selection via a call to E**ditorObjectSelection.Instance.RemoveObjectFromSelection**;
- **ClearClickAir –** the objects were deselected when the user clicked in the air. This causes the object selection to be cleared by the selection module;
- **ClickAlreadySelected** – this happens when the user clicks on an object with the append to selection hotkey active and the object is already selected. In this case, the module will deselect it;
- **MultiDeselect –** the objects were deselected using the selection rectangle while the multi-deselect hotkey was active;
- **Undo –** the objects were deselected as part of an Undo operation;
- **Redo –** the objects were deselected as part of a Redo operation;
- **DeselectInactive –** it may happen sometimes that after objects are selected, some of them may become inactive. The module will check for inactive objects every frame and remove them from the selection. When these objects are removed from the selection, this is the type of deselect action that is used;
- **None –** useful in some situations to indicate that no objects were deselected.

So back to the **ObjectSelectionChangedEventArgs,** the **DeselectActionType** and **DeselectedObjects**

tell you which objects were deselected and in what way. If **DeselectActionType** is set to **None** it means no objects were deselected and **DeselectedObjects** is empty.

Next, we have a **GizmoType** property which tells you the type of gizmo which was used when the selection has changed. The last property, **IsGizmoActive,** specifies whether or not the gizmo is active. The gizmo can be inactive when the user turned off the gizmos via the corresponding hotkey.