Actions

Actions

In Phaser, Actions are functions that typically perform a single task by combining multiple Phaser features together, such as Placing a set of sprites along a Line, or rotating a set of sprites around a given point. Actions run across an array of Game Objects, allowing you to perform the same action on them all, often with optional offsets. Some Actions are meant to be called once, while others are designed to be called continuously as part of an update loop.

Actions are available under the `Phaser.Actions` namespace and are all static functions, so you can call them directly.

Applying an Action to a Group

If you wish to apply an Action to a Phaser Game Objects Group, you can do so by calling the Groups `getChildren()` method when invoking the Action. For example, to set the alpha of all children in a Group to 0.5 you could do:

Copy

```
Phaser.Actions.SetAlpha(myGroup.getChildren(), 0.5);
```

Step, Index and Direction

As well as the array of Game Objects to manipulate, lots of Actions also take the following optional parameters:

- `step`

- `index`
- `direction`

The `step` parameter is an optional amount that is added to the `value` passed into the Action, multiplied by the iteration counter. For example, if you pass in an array of 10 sprites to the `SetX` Action, and set the `value` to be 100, then every Sprite will given the same x coordinate of 100. But if you set the `step` parameter to 50, then the first sprite will have a position of 100, the second will be at 150, the third at 200 and so on, as the `step` value (50) is multiplied by the array iteration. This is extremely useful to quickly provide variety from a single Action without calling it repeatedly.

The `index` parameter controls the offset to start from in the given array. When you call an Action function, you pass in an array of Game Objects to modify. The `index` parameter allows you to skip 'n' number of items in the array, as defined by the starting `index` and only apply the Action to the remaining entries. By default, the `index` is zero, which means it will run the Action across every entry in the array. The `index` should never exceed the length of the Game Objects array.

The `direction` parameter allows you to adjust the order in which you iterate through the Game Objects array. The default value is `1` which means it will iterate from the start to the end of the array. If you set the `direction` to be `-1` it will iterate from the end to the start of the array instead. This is useful if you wish to reverse the order in which the Action is applied to the array, without modifying the array itself.

If an Action allows you to specify the Step, Index and Direction, please refer back to here for details.

Some Actions allow you to specify 2 step values, such as the Set Scale Action. This is because the step is split across both the `x` and `y` properties of the Action. In the case of Set Scale, the first step value is applied to the `scaleX` property and the second to the `scaleY` property. Please cross-reference the documentation for the Action you're using to see if it supports this.

Actions List

Align To

Takes an array of Game Objects and aligns them next to each other.

Copy

```
Phaser.Actions.AlignTo(items, position, [offsetX], [offsetY]);
```

- `items` : The array of items to be updated by this action.
- `position` : The position to align the items with. This is an align constant, such as `Phaser.Display.Align.LEFT_CENTER`.
- `offsetX` : Optional horizontal offset from the position.
- `offsetY` : Optional vertical offset from the position.

Angle

The Angle Action will *add* the given value to the `angle` property of an array of Game Objects.

Copy

```
Phaser.Actions.Angle(myObjects, 90);
```

As covered in the Game Objects guide, the `angle` sets the angle of rotation of a Game Object in degrees.

Because this Action is additive it means you can call it in an `update` loop, or similar, and keep incrementing the angle of the objects by small amounts.

[file: actions/angle update.js]

This Action supports the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Call

The Call Action will take an array of Game Objects and pass them to the given callback:

Copy

```
Phaser.Actions.Call(myObjects, callback, context);
```

The `callback` is a function that will be invoked for each Game Object in the array. It will be sent the Game Object itself as the first argument.

This Action isn't required if you are coding in ES6, because you can achieve the same thing with the native `forEach` method:

Copy

```
myObjects.forEach(callback, context);
```

However, if you're using ES5 then this Action is a useful shortcut.

Get First and Get Last

The Get First Action will take an array of Game Objects along with a comparison object. It will iterate the array and return the first Game Object that matches all conditions in the comparison object:

Copy

```
const first = Phaser.Actions.GetFirst(myObjects, { scaleX: 0.5, alpha: 1 });
```

In the example above, the first Game Object that has a `scaleX` value of 0.5 and an `alpha` value of 1 will be returned. If no Game Object in the array matches the conditions, `null` is returned.

The comparison can also include references to other objects. For example, to get the first Game Object that has a specific texture, you can do:

Copy

```
const fireTexture = this.textures.get('fire');

const first = Phaser.Actions.GetFirst(myObjects, { texture: fireTexture });
```

The comparison object can have as many properties as you like. Each property in the comparison object must map to a property on the Game Object, i.e. `visible`, `x`, `y`, `scaleX`, etc. If you include a property that Game Objects do not have, it will likely never return a match. The comparison cannot match against nested properties, such as `texture.name`. Comparisons are strict, so `===` is used to compare the values.

The Get Last Action works in exactly the same way, but returns the *last* Game Object from the array that matches the comparison object. It does this by starting at the end of the array and iterating towards the start.

Grid Align

Takes an array of Game Objects, or any objects that have public `x` and `y` properties, and then aligns them based on the grid configuration given to this action.

Copy

```
Phaser.Actions.GridAlign(items, options);
```

- `items` : The array of items to be updated by this action.
- `options` : The GridAlign Configuration object.
    - `width`: The width of the grid in items (not pixels). -1 means lay all items out horizontally, regardless of quantity.
    - `height` : The height of the grid in items (not pixels). -1 means lay all items out vertically, regardless of quantity.
    - `cellWidth` : The width of the cell, in pixels, in which the item is positioned.
    - `cellHeight` : The height of the cell, in pixels, in which the item is positioned.
    - `position` : The alignment position. One of the `Phaser.Display.Align` consts such as `TOP_LEFT` or `RIGHT_CENTER`.
    - `x` : Optionally place the top-left of the final grid at this coordinate.
    - `y` : Optionally place the top-left of the final grid at this coordinate.

Inc Alpha

The Inc Alpha Action will *add* the given value to the `alpha` property of an array of Game Objects.

Copy

```
Phaser.Actions.IncAlpha(myObjects, 0.1);
```

As covered in the Game Objects guide, the `alpha` property controls the alpha value of a Game Object, where 0 is fully transparent and 1 is fully opaque.

Because this Action is additive it means you can call it in an `update` loop, or similar, to keep incrementing the alpha of the objects by small amounts.

[file: actions/inc alpha update.js]

This Action supports the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Inc X, Inc Y and Inc XY

The Inc X Action will *add* the given value to the `x` property of an array of Game Objects.

Copy

```
Phaser.Actions.IncX(myObjects, 10);
```

In the example above, all Game Objects in the array will have 10 added to their current `x` value.

The Inc Y Action will *add* the given value to the `y` property of an array of Game Objects.

Copy

```
Phaser.Actions.IncY(myObjects, 10);
```

In this example, all Game Objects in the array will have 10 added to their current `y` value.

Both of these Actions are additive which means you can call them in an `update` loop, or similar, to keep incrementing the position of the objects by small amounts.

Both Inc X and Inc Y Actions support the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

The Inc XY Action will *add* the given values to the `x` and `y` properties of an array of Game Objects.

Copy

```
Phaser.Actions.IncXY(myObjects, 4, 6);
```

In the example above, all Game Objects in the array will have 4 added to their current `x` value and 6 added to their current `y` value.

This Action is additive which means you can call it in an `update` loop, or similar, to keep incrementing the position of the objects by small amounts.

This Action supports the optional `index` and `direction` parameters. See the "Step, Index and Direction" section for details. It also supports `step`, however this is specified as two different values, `stepX` and `stepY` which are used to multiply the `x` and `y` values respectively:

Copy

```
Phaser.Actions.IncXY(myObjects, 4, 6, 1.25, 1.5);
```

In the example above, the `x` value will be multiplied by 1.25 and the `y` value will be multiplied by 1.5.

If you wish to set the positions of the Game Objects rather than increment them, see the Set XY Actions instead.

Place On Circle

The Place on Circle Action takes an array of Game Objects and positions them on evenly spaced points around the circumference of the given circle.

Copy

```
const circle = new Phaser.Geom.Circle(400, 300, 200);

Phaser.Actions.PlaceOnCircle(myObjects, circle, startAngle, endAngle);
```

The `circle` argument can be any object that has public `x`, `y` and `radius` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Circle` object, or one that extends it.

The `startAngle` and `endAngle` arguments are optional and allow you to control where the placement starts and ends. The angles are given in radians where the value 0 is to the right and increases clockwise. So, for example, if you wanted to display an arc of Game Objects from the top of the circle to the bottom, you could use:

Copy

```
Phaser.Actions.PlaceOnCircle(myObjects, circle, Math.PI * 1.5, Math.PI * 0.5);
```

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at the `startAngle` point on the circle, with the next Game Object placed at the next point around the circle, and so on until it reaches the `endAngle` point. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

If you modify the underlying circle object after calling this Action, it will have no effect on the Game Objects that were positioned on it until you call this Action again. Should you wish to animate the process, you can call this Action in an `update`, tween or timed loop while modifying the circle shape to create some interesting realtime effects.

Place On Ellipse

The Place on Ellipse Action takes an array of Game Objects and positions them on evenly spaced points around the perimeter of the given ellipse.

Copy

```
const ellipse = new Phaser.Geom.Ellipse(400, 300, 200, 300);

Phaser.Actions.PlaceOnEllipse(myObjects, ellipse, startAngle, endAngle);
```

The `ellipse` argument can be any object that has public `x`, `y`, `width` and `height` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Ellipse` object, or one that extends it.

The `startAngle` and `endAngle` arguments are optional and allow you to control where the placement starts and ends. The angles are given in radians where the value 0 is to the right and increases clockwise. So, for example, if you wanted to display an arc of Game Objects from the top of the ellipse to the bottom, you could use:

Copy

```
Phaser.Actions.PlaceOnEllipse(myObjects, ellipse, Math.PI * 1.5, Math.PI * 0.5);
```

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at the `startAngle` point on the ellipse, with the next Game Object placed at the next point around the ellipse, and so on until it reaches the `endAngle` point. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

If you modify the underlying ellipse object after calling this Action, it will have no effect on the Game Objects that were positioned on it until you call this Action again. Should you wish to

animate the process, you can call this Action in an `update`, tween or timed loop while modifying the ellipse shape to create some interesting realtime effects.

Place On Line

The Place on Line Action takes an array of Game Objects and positions them on evenly spaced points along the length of the given line.

Copy

```
const line = new Phaser.Geom.Line(100, 100, 700, 500);

Phaser.Actions.PlaceOnLine(myObjects, line);
```

The `line` argument can be any object that has public `x1`, `y1`, `x2` and `y2` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Line` object, or one that extends it.

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at the `x1` and `y1` point on the line, with the next Game Object placed at the next point along the line, and so on until it reaches the `x2` and `y2` point. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

If you modify the underlying line object after calling this Action, it will have no effect on the Game Objects that were positioned on it until you call this Action again. Should you wish to animate the process, you can call this Action in an `update`, tween or timed loop while modifying the line shape to create some interesting realtime effects.

Place On Rectangle

The Place on Rectangle Action takes an array of Game Objects and positions them on evenly spaced points around the perimeter of the given rectangle.

Copy

```
const rect = new Phaser.Geom.Rectangle(100, 100, 600, 400);

Phaser.Actions.PlaceOnRectangle(myObjects, rect);
```

The `rect` argument should be an instance of a `Phaser.Geom.Rectangle` object, or one that extends it.

Placement starts from the top-left of the rectangle and continues around it in a clockwise direction. The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at the top-left of the rectangle, with the next Game Object placed at the next point around the rectangle, and so on until it wraps around to the top-left point again. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

This Action also has an optional `shift` parameter. This allows you to control the starting point of the placement. By default, the first Game Object in the array is positioned at the first point on the perimeter. But the `shift` parameter allows you to start the placement from a different point. For example, if you set `shift` to be 1, then the first Game Object in the array will be placed at the second point on the perimeter, instead of the first. If you set `shift` to be 2, then the first Game Object in the array will be placed at the third point on the perimeter, and so on. The Game Objects will still wrap around the full perimeter of the rectangle, regardless of the `shift` value.

If you modify the underlying rectangle object after calling this Action, it will have no effect on the Game Objects that were positioned on it until you call this Action again. Should you wish to

animate the process, you can call this Action in an `update`, tween or timed loop while modifying the rectangle shape to create some interesting realtime effects.

Place On Triangle

The Place on Triangle Action takes an array of Game Objects and positions them on evenly spaced points around the perimeter of the given triangle.

Copy

```
const triangle = new Phaser.Geom.Triangle(400, 100, 100, 500, 700, 500);

Phaser.Actions.PlaceOnTriangle(myObjects, triangle);
```

The `triangle` argument can be any object that has public `x1`, `y1`, `x2`, `y2`, `x3` and `y3` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Triangle` object, or one that extends it.

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at the `x1` and `y1` point on the triangle, with the next Game Object placed at the next point around the triangle, and so on until it reaches starting point again. The points on each face of the triangle are calculated using the Bresenhams Line Algorithm, with overlaps removed.

The Game Objects are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

This Action has an optional `stepRate` parameter. This allows you to control the density of the packing of the points. This default to 1. Change it to lower or higher values to modify the spacing between the points.

If you modify the underlying triangle object after calling this Action, it will have no effect on the Game Objects that were positioned on it until you call this Action again. Should you wish to animate the process, you can call this Action in an `update`, tween or timed loop while modifying the triangle shape to create some interesting realtime effects.

Play Animation

The Play Animation Action will start playing the given animation on all Game Objects in the array.

Copy

```
Phaser.Actions.PlayAnimation(myObjects, key);
```

This Action will check to ensure that the Game Object has an Animation State component before trying to play the animation. If it doesn't, it will be skipped.

The Action has an optional `ignoreIfPlaying` parameter. If set to `true` it will not start the animation if the Game Object is already playing the given animation. The default value is `false`.

Property Value Inc

Takes an array of Game Objects, or any objects that have a public property as defined in `key`, and then *adds* the given value to it. The optional `step` property is applied incrementally, multiplied by each item in the array.

Copy

```
Phaser.Actions.PropertyValueInc(items, key, value, [step], [index],
[direction]);
```

- `items` : The array of items to be updated by this action.
- `key` : The property to be updated.
- `value` : The amount to be added to the property.
- `step` : This is added to the value amount, multiplied by the iteration counter.
- `index` : An optional offset to start searching from within the items array.
- `direction` : The direction to iterate through the array. 1 is from beginning to end, -1 from end to beginning.

Property Value Set

Takes an array of Game Objects, or any objects that have a public property as defined in `key`, and then sets it to the given value. The optional `step` property is applied incrementally, multiplied by each item in the array.

Copy

```
Phaser.Actions.PropertyValueSet(items, key, value, [step], [index],
[direction]);
```

- `items` : The array of items to be updated by this action.
- `key` : The property to be updated.
- `value` : The amount to be added to the property.
- `step` : This is added to the value amount, multiplied by the iteration counter.
- `index` : An optional offset to start searching from within the items array.
- `direction` : The direction to iterate through the array. 1 is from beginning to end, -1 from end to beginning.

Random Circle

The Random Circle Action takes an array of Game Objects and positions them at random locations within a circle shape.

Copy

```
const circle = new Phaser.Geom.Circle(400, 300, 200);

Phaser.Actions.RandomCircle(myObjects, circle);
```

The `circle` argument can be any object that has public `x`, `y` and `radius` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Circle` object, or one that extends it.

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at a random point within the circle, with the next Game Object placed at the next random point within the circle, and so on until it reaches the end of the array. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

This Action does not try to fit the bounds of the Game Object within the circle, it simply places them by their position, so it's entirely possible that your Game Objects will extend outside of the circle depending on their size. The random positions chosen from the circle are uniformly distributed, meaning that it's just as likely to return a point within the center of the circle as it is on the edge of the circle.

If you modify the underlying circle object after calling this Action, it will have no effect on the Game Objects that were positioned within it until you call this Action again. Note that calling this Action multiple times will always result in you getting different random positions back.

Random Ellipse

The Random Ellipse Action takes an array of Game Objects and positions them at random locations within an ellipse shape.

Copy

```
const ellipse = new Phaser.Geom.Ellipse(400, 300, 200, 300);

Phaser.Actions.RandomEllipse(myObjects, ellipse);
```

The `ellipse` argument can be any object that has public `x`, `y`, `width` and `height` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Ellipse` object, or one that extends it.

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at a random point within the ellipse, with the next Game Object placed at the next random point within the ellipse, and so on until it reaches the end of the array. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

This Action does not try to fit the bounds of the Game Object within the ellipse, it simply places them by their position, so it's entirely possible that your Game Objects will extend outside of the ellipse depending on their size.

If you modify the underlying ellipse object after calling this Action, it will have no effect on the Game Objects that were positioned within it until you call this Action again. Note that calling this Action multiple times will always result in you getting different random positions back.

Random Line

The Random Line Action takes an array of Game Objects and positions them at random locations along the length of a line segment.

Copy

```
const line = new Phaser.Geom.Line(100, 100, 700, 500);

Phaser.Actions.RandomLine(myObjects, line);
```

The `line` argument can be any object that has public `x1`, `y1`, `x2` and `y2` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Line` object, or one that extends it.

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at a random point on the line, with the next Game Object placed at the next random point on the line, and so on until it reaches the end of the array. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

If you modify the underlying line object after calling this Action, it will have no effect on the Game Objects that were previously positioned on it until you call this Action again. Note that calling this Action multiple times will always result in you getting different random positions back.

Random Rectangle

The Random Rectangle Action takes an array of Game Objects and positions them at random locations within a rectangle shape.

Copy

```
const rect = new Phaser.Geom.Rectangle(100, 100, 600, 400);

Phaser.Actions.RandomRectangle(myObjects, rect);
```

The `rect` argument can be any object that has public `x`, `y`, `width` and `height` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Rectangle` object, or one that extends it.

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at a random point within the rectangle, with the next Game Object placed at the next random point within the rectangle, and so on until it reaches the end of the array. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

This Action does not try to fit the bounds of the Game Object within the rectangle, it simply places them by their position, so it's entirely possible that your Game Objects will extend outside of the rectangle depending on their size.

If you modify the underlying rectangle object after calling this Action, it will have no effect on the Game Objects that were positioned within it until you call this Action again. Note that calling this Action multiple times will always result in you getting different random positions back.

Random Triangle

The Random Triangle Action takes an array of Game Objects and positions them at random locations within a triangle shape.

Copy

```
const triangle = new Phaser.Geom.Triangle(400, 100, 100, 500, 700, 500);

Phaser.Actions.RandomTriangle(myObjects, triangle);
```

The `triangle` argument can be any object that has public `x1`, `y1`, `x2`, `y2`, `x3` and `y3` properties, however for strongly-typed languages this Action will expect you to pass a `Phaser.Geom.Triangle` object, or one that extends it.

The Game Objects are positioned in the order in which they appear in the array. The first Game Object in the array will be placed at a random point within the triangle, with the next Game Object placed at the next random point within the triangle, and so on until it reaches the end of the array. They are positioned by adjusting their `x` and `y` coordinates, so their Origin will play a role in how they are displayed in relation to these points.

This Action does not try to fit the bounds of the Game Object within the triangle, it simply places them by their position, so it's entirely possible that your Game Objects will extend outside of the triangle depending on their size.

If you modify the underlying triangle object after calling this Action, it will have no effect on the Game Objects that were positioned within it until you call this Action again. Note that calling this Action multiple times will always result in you getting different random positions back.

Rotate

The Rotate Action will *add* the given value to the `rotation` property of an array of Game Objects.

Copy

```
Phaser.Actions.Rotate(myObjects, 0.5);
```

As covered in the Game Objects guide, the `rotation` sets the angle of rotation of a Game Object in radians.

Because this Action is additive it means you can call it in an `update` loop, or similar, and keep incrementing the rotation of the objects by small amounts.

This Action supports the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Rotate Around

Rotates each item around the given point by the given angle in radians.

Copy

```
Phaser.Actions.RotateAround(items, point, angle);
```

- `items` : An array of Game Objects. The contents of this array are updated by this Action.
- `point` : Any object with public x and y properties.
- `angle` : The angle to rotate by, in radians.

Rotate Around Distance

Rotates an array of Game Objects around a point by the given angle and distance.

Copy

```
Phaser.Actions.RotateAroundDistance(items, point, angle, distance);
```

- `items` : An array of Game Objects. The contents of this array are updated by this Action.
- `point` : Any object with public x and y properties.
- `angle` : The angle to rotate by, in radians.
- `distance` : The distance from the point of rotation in pixels.

Scale X, Y and XY

The Scale Actions will *add* the given value to the `scaleX` or `scaleY` properties of an array of Game Objects.

Copy

```
Phaser.Actions.ScaleX(myObjects, 0.5);
Phaser.Actions.ScaleY(myObjects, 0.5);
Phaser.Actions.ScaleXY(myObjects, 0.5, 0.25);
```

In the example above, all Game Objects in the array will have 0.5 added to their current `scaleX` value. The Scale Y Action will do the same for the `scaleY` property. The Scale XY Action will do the same for both the `scaleX` and `scaleY` properties. You can pick whichever action suits your needs.

Because these Actions are additive it means you can call them in an `update` loop, or similar, and keep modifying the scale of the objects by small amounts.

These three Actions support the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Set Alpha

The Set Alpha Action will set the `alpha` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetAlpha(myObjects, 0.5);
```

As covered in the Game Objects guide, the `alpha` property controls the alpha value of a Game Object, where 0 is fully transparent and 1 is fully opaque.

The Set Alpha Action does not test to see if the Game Object has an Alpha component. It just sets the property regardless, so be careful about the array of items you pass.

This Action supports the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Set Blend Mode

The Set Blend Mode Action will set the `blendMode` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetBlendMode(myObjects, Phaser.BlendModes.ADD);
```

Please see the Blend Modes section of the guide for details about the different modes available.

The Set Blend Mode Action does not test to see if the Game Object supports blend modes. It just sets the property regardless, so be careful about the array of items you pass.

This Action supports the optional `index` and `direction` parameters. See the "Step, Index and Direction" section for details. Note that it does not include `step` support.

Set Depth

The Set Depth Action will set the `depth` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetDepth(myObjects, 100);
```

As covered in the Game Objects guide, the `depth` property controls the rendering order of a Game Object, where the higher the value, the more it will be rendered on-top of other Game Objects.

The Set Depth Action does not test to see if the Game Object has a Depth component. It just sets the property regardless, so be careful about the array of items you pass.

This Action supports the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Set Hit Area

Passes all provided Game Objects to the Input Manager to enable them for input with identical areas and callbacks.

Copy

```
Phaser.Actions.SetHitArea(items, [hitArea], [callback]);
```

- `items` : An array of Game Objects. The contents of this array are updated by this Action.
- `hitArea` : Either an input configuration object, or a geometric shape that defines the hit area for the Game Object. If not given it will try to create a Rectangle based on the texture frame.
- `callback` : The callback that determines if the pointer is within the Hit Area shape or not. If you provide a shape you must also provide a callback.

Set Origin

Set Rotation

The Set Rotation Action will set the `rotation` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetRotation(myObjects, 1.5707);
```

As covered in the Game Objects guide, the `rotation` sets the angle of rotation of a Game Object in radians.

The Set Rotation Action does not test to see if the Game Object has a Transform component. It just sets the property regardless, so be careful about the array of items you pass.

This Action supports the optional `step`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Set Scale

The Set Scale Action will set the `scaleX` and `scaleY` properties of an array of Game Objects to the given values:

Copy

```
Phaser.Actions.SetScale(myObjects, 0.5, 0.25);
```

In the example above, all Game Objects in the array will have their `scaleX` property set to 0.5 and their `scaleY` property set to 0.25.

The Set ScaleX Action will set the `scaleX` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetScaleX(myObjects, 0.5);
```

In the example above, all Game Objects in the array will have their `scaleX` property set to 0.5. Their `scaleY` properties will remain untouched.

The Set ScaleY Action will set the `scaleY` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetScaleY(myObjects, 0.25);
```

In the example above, all Game Objects in the array will have their `scaleY` property set to 0.25. Their `scaleX` properties will remain untouched.

The Set Scale Actions do not test to see if the Game Object has a Transform component. It just sets the properties regardless, so be careful about the array of items you pass.

These Actions support the optional `stepX`, `stepY`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Set Scroll Factor

The Set Scroll Factor Action will set the `scrollFactorX` and `scrollFactorY` properties of an array of Game Objects to the given values:

Copy

```
Phaser.Actions.SetScrollFactor(myObjects, 1, 0);
```

In the example above, all Game Objects in the array will have their `scrollFactorX` property set to 1 and their `scrollFactorY` property set to 0.

The Set ScrollFactorX Action will set the `scrollFactorX` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetScrollFactorX(myObjects, 1);
```

In the example above, all Game Objects in the array will have their `scrollFactorX` property set to 1. Their `scrollFactorY` properties will remain untouched.

The Set ScrollFactorY Action will set the `scrollFactorY` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetScrollFactorY(myObjects, 0);
```

In the example above, all Game Objects in the array will have their `scrollFactorY` property set to 0.25. Their `scrollFactorX` properties will remain untouched.

The Set Scroll Factor Actions do not test to see if the Game Object has a Transform component. It just sets the properties regardless, so be careful about the array of items you pass.

These Actions support the optional `stepX`, `stepY`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

Set Tint

The Set Tint Action take an array of Game Objects and call the `setTint` method on each of them:

Copy

```
Phaser.Actions.SetTint(myObjects, 0xff0000);
```

As covered in the Game Objects guide, tinting allows you to change the color of a Game Object by applying a tint to it. A tint works by taking the original pixel color values from the Game Object, and then multiplying each of them by the color value of the tint. You can provide either one color value, in which case the whole Game Object will be tinted in that color. Or you can provide a color per corner:

Copy

```
Phaser.Actions.SetTint(myObjects, 0xff0000, 0x00ff00, 0x0000ff, 0xff00ff,
0x00ffff);
```

In the example above the Game Object will be tinted with a red color in the top-left corner, green in the top-right, blue in the bottom-left and purple in the bottom-right. The colors are blended together across the extent of the Game Object.

The Set Tint Action does not test to see if the Game Object has a Tint component. It just sets the property regardless, so be careful about the array of items you pass.

Set Visible

The Set Visible Action will set the `visible` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetVisible(myObjects, false);
```

As covered in the Game Objects guide, the `visible` property controls if a Game Object is rendered or not.

This Action supports the optional `index` and `direction` parameters. See the "Step, Index and Direction" section for details. Note that it does not include `step` support.

Set XY

The Set XY Action will set the `x` and `y` properties of an array of Game Objects to the given values:

Copy

```
Phaser.Actions.SetXY(myObjects, 100, 200);
```

In the example above, all Game Objects in the array will have their `x` property set to 100 and their `y` property set to 200.

The Set X Action will set the `x` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetX(myObjects, 100);
```

In the example above, all Game Objects in the array will have their `x` property set to 100. Their `y` properties will remain untouched.

The Set Y Action will set the `y` property of an array of Game Objects to the given value:

Copy

```
Phaser.Actions.SetY(myObjects, 200);
```

In the example above, all Game Objects in the array will have their `y` property set to 200. Their `x` properties will remain untouched.

The Set XY Actions do not test to see if the Game Object has a Transform component. It just sets the properties regardless, so be careful about the array of items you pass.

These Actions support the optional `stepX`, `stepY`, `index` and `direction` parameters. See the "Step, Index and Direction" section for details.

If you wish to increment the positions of the Game Objects rather than set them, see the Inc X, Inc Y and Inc XY Actions instead.

Shift Position

The Shift Position Action takes an array of Game Objects and then iterates through them, adjusting their `x` and `y` positions to be that of the previous entry in the array. The first Game Object in the array is known as the 'head' item. The position of the head is set to the `x` and `y` values you pass to this Action:

Copy

```
Phaser.Actions.ShiftPosition(myObjects, 400, 300);
```

In the example above, the first Game Object in the array will be positioned at 400 x 300. The next Game Object in the array will have its position set to whatever position the first Game Object had before it was moved, and so on. At the end of the array, the last Game Object will have its position set to that of the second-last Game Object.

You can pass in the `direction` parameter, which controls the iteration order. The default value is 0, which means from first to last in the input array. If you pass in a value of 1 it iterates backwards, from the final element in the array to the first.

The Action returns a Vector2 that contains the position of the last Game Object in the array before the Action was called.

A good way to think about this Action is with the classic Nokia Snake game. When the snake moves, the 'head' of the snake moves in one direction and then all tail pieces shift to be in the position of the previous tail segment. This Action can be used to replicate this effect.

Shuffle

The Shuffle Action will take an array, which can actually be on anything, but is primarily meant for Game Objects, and shuffles the contents of it:

Copy

```
Phaser.Actions.Shuffle(myObjects);
```

The contents of the array are shuffled in place. No new array is created.

Smooth Step and Smoother Step

This is convenient for creating a sequence of transitions using `smoothstep` to interpolate each segment as an alternative to using more sophisticated or expensive interpolation techniques.

Copy

```
Phaser.Actions.SmoothStep(items, property, min, max, [inc]);
Phaser.Actions.SmootherStep(items, property, min, max, [inc]);
```

- `items` : An array of Game Objects. The contents of this array are updated by this Action.
- `property` : The property of the Game Object to interpolate.
- `min` : The minimum interpolation value.
- `max` : The maximum interpolation value.
- `inc` : Should the property value be incremented (`true`) or set (`false`)?

Spread

Toggle Visible

The Toggle Visible Action will take an array of Game Objects and then toggle the 'visible' state of each of them:

Copy

```
Phaser.Actions.ToggleVisible(myObjects);
```

If the Game Object is visible it will be set to invisible. If it is invisible it will be set to visible.

The Toggle Visible Action does not test to see if the Game Object has a visible property. It just sets the property regardless, so be careful about the array of items you pass.