• Index

AddLayer

```
static function AddLayer (mapSize : Int2,
                          addBorder : int,
                          tileSize : float (or Vector2),
                          zPosition : float,
                          layerLock : LayerLock) : void
```

Adds a new layer to the level. Since the layer is last in the list, it will be rendered on top of preceding layers. The project must have enough layers in the Tags and Layers manager to accommodate the new layer (see "How Layers Work").

mapSize: the dimensions, in tiles, of the level. Must be at least 1x1.

addBorder: the number of additional rows/columns that are added around the screen border, when using oversized tiles or camera rotation. Must be 0 or greater.

tileSize: the dimension, in units, of each cell in the level. Must be at least .001. For a non-square tile grid, use a Vector2 instead of a float.

zPosition: the distance from the origin along the Z axis. Only has an effect with perspective cameras. Must be at least 0.0.

layerLock: whether the layer should be prevented from moving on the X or Y axes. Uses the LayerLock enum:

LayerLock.None: the layer is not locked.

LayerLock.X: the layer is locked on the X axis, but can move on the Y axis.

LayerLock.Y: the layer is locked on the Y axis, but can move on the X axis.

LayerLock.XandY: the layer is locked on both the X axis and the Y axis.

```
    // Creates a 20x20 layer with no added border, a tile size of 1.0,
    // located at z = 0.0, with no layer lock
    Tile.AddLayer (new Int2(20, 20), 0, 1.0f, 0.0f, LayerLock.None);
    // Creates a 50x25 layer with a 1 tile added border, a non-square tile size
    // of (1.0, 2.0), located at z = 5.0, locked on the Y axis
    Tile.AddLayer (new Int2(50, 25), 1, new Vector2(1.0f, 2.0f), 5.0f, LayerLock.Y);
```

```
static function AddLayer (levelData : LevelData) : void
```

As above, but the layer parameters are contained in a variable with a LevelData type. The LevelData properties are mapSize (Int2), addBorder (int), tileSize (Vector2), zPosition (float), and layerLock (LayerLock). If you want a square grid, just use the same value for the X and Y tileSize.

```
    // Creates a 50x25 layer with a 1 tile added border, a non-square tile size
    // of (1.0, 2.0), located at z = 5.0, locked on the Y axis
    var myLevelData = new LevelData(new Int2(50, 25), 1, new Vector2(1.0f, 2.0f),
                                    5.0f, LayerLock.Y);
    Tile.AddLayer (myLevelData);
```

AnimateTile

```
static function AnimateTile (tileInfo : TileInfo,
                             range : int,
                             frameRate : float,
                             animType : AnimType = AnimType.Loop) : void
```

Animates all instances of tileInfo in all layers in the current level.

The range is the number of tiles, including the tile specified by tileInfo, that will be included in the animation sequence. The range added to tileInfo must not exceed the number of tiles in the current set, and must be at least 2.

The frameRate is how fast the animation sequence will be played back, in terms of frames per second. It must be at least 0.0 (although using 0.0 will of course not result in any actual animation).

The animType is AnimType.Loop by default, which plays the animation from the first frame to the last, then starts over. AnimType.Reverse will play the animation from the last frame to the first and then start over, and AnimType.PingPong will play the animation from the first frame to the last, back to the first, and then repeat the cycle.

Animation can be stopped with StopAnimatingTile.

```
    // Animates tile #15 in set 2, using tiles 15-20, at 5fps, with AnimType.Loop
    Tile.AnimateTile (new TileInfo(2, 15), 6, 5.0f);
    // Animates tile #10 in set 1, using tiles 10-19, at 15fps, in reverse
    Tile.AnimateTile (new TileInfo(1, 10), 10, 15.0f, AnimType.Reverse);
```

```
static function AnimateTile (tileInfo : TileInfo,
                             tileInfoArray : TileInfo[],
                             frameRate : float,
                             animType : AnimType = AnimType.Loop) : void
```

As above, but instead of specifying a range, the tile animation sequence is supplied as an array of TileInfo. The tiles can be from any set, in any order.

```
    // Animates tile #15 in set 2, using the supplied TileInfo array, at 5fps
    // Unityscript
    var tiles = [TileInfo(1, 4), TileInfo(1, 5), TileInfo(2, 8)];
    Tile.AnimateTile (TileInfo(2, 15), tiles, 5.0);
    // C#
    TileInfo[] tiles = {new TileInfo(1, 4), new TileInfo(1, 5), new TileInfo(2, 8)};
    Tile.AnimateTile (new TileInfo(2, 15), tiles, 5.0f);
```

AnimateTileRange

```
static function AnimateTileRange (tileInfo : TileInfo,
                                  range : int,
                                  frameRate : float,
                                  animType : AnimType = AnimType.Loop) : void
```

Similar to AnimateTile, but all tiles in the range are animated independently. All tiles use the range as specified by tileInfo + range, looping as necessary. For example, if tileInfo is TileInfo(1, 10) and the range is 3, then TileInfo(1, 10) will animate using tiles 10, 11, and 12. TileInfo(1, 11) will animate using tiles 11, 12, and 10. TileInfo(1, 12) will animate using tiles 12, 10, and 11.

The range is the number of tiles, including the tile specified by tileInfo, that will be included in the animation sequence. The range added to tileInfo must not exceed the number of tiles in the current set, and must be at least 2.

The frameRate is how fast the animation sequence will be played back, in terms of frames per second. It must be at least 0.0 (although using 0.0 will of course not result in any actual animation).

The animType is AnimType.Loop by default, which plays the animation from the first frame to the last, then starts over. AnimType.Reverse will play the animation from the last frame to the first and then start over, and AnimType.PingPong will play the animation from the first frame to the last, back to the first, and then repeat the cycle.

Animation can be stopped with StopAnimatingTileRange.

```
    // Animates tiles #5-9 in set 2, at 8fps, with AnimType.Loop
    Tile.AnimateTileRange (new TileInfo(2, 5), 5, 8.0f);
    // Animates tiles #10-19 in set 1, at 15fps, in reverse
    Tile.AnimateTile (new TileInfo(1, 10), 10, 15.0f, AnimType.Reverse);
```

```
static function AnimateTileRange (tileInfo : TileInfo,
                                  range : int;
                                  tileInfoArray : TileInfo[];
                                  frameRate : float;
                                  animType : AnimType = AnimType.Loop) : void
```

As above, but all tiles in the range will use a tile animation sequence that's supplied as an array of TileInfo. The TileInfo array doesn't need to be the same size as the range. The tiles can be from any set, in any order.

```
    // Animates tiles #10-15 in set 2, using the supplied TileInfo array, at 5fps
    // Unityscript
    var tiles = [TileInfo(1, 4), TileInfo(1, 5), TileInfo(2, 8)];
    Tile.AnimateTile (TileInfo(2, 10), 6, tiles, 5.0);
    // C#
    TileInfo[] tiles = {new TileInfo(1, 4), new TileInfo(1, 5), new TileInfo(2, 8)};
    Tile.AnimateTile (new TileInfo(2, 10), 6, tiles, 5.0f);
```

CameraRotationX

```
static function CameraRotationX (xRotation : float,
                                 camNumber : int = 0) : void
```

Applies a rotation (in degrees) to the specified camera. The xRotation is clamped between -90.0° and 90.0°. Any other rotation the camera may have is removed, so only rotation on the X axis will occur. Note that only perspective cameras can use CameraRotationX; if the camera is orthographic, an error is printed. Typically the AddBorder property of the level should be increased to prevent tiles from popping in during scrolling; how much depends on the amount of rotation.

If the camNumber is omitted, the default of 0 is used. SetCamera must be called before using CameraRotationX, so the camNumber corresponds to any cameras that were set up with SetCamera.

```
    // Sets the X axis rotation of the camera to 20°
    Tile.CameraRotationX (20.0f);
    // Sets the X axis rotation of the second camera to -10°
    Tile.CameraRotationX (-10.0f, 1);
```

CameraRotationY

```
static function CameraRotationY (yRotation : float,
                                 camNumber : int = 0) : void
```

This works like CameraRotationX, but the rotation is applied to the Y axis.

```
    // Sets the Y axis rotation of the camera to 20°
    Tile.CameraRotationY (20.0f);
    // Sets the Y axis rotation of the second camera to -10°
    Tile.CameraRotationY (-10.0f, 1);
```

CameraRotationZ

```
static function CameraRotationZ (zRotation : float,
                                 camNumber : int = 0) : void
```

Applies a rotation (in degrees) to the specified camera. Any other rotation the camera may have is removed, so only rotation on the Z axis will occur. CameraRotationZ works with both perspective and orthographic cameras, and the rotation is not clamped. Typically the AddBorder property of the level should be increased to prevent tiles from popping in during scrolling; how much depends on the amount of rotation.

If the camNumber is omitted, the default of 0 is used. SetCamera must be called before using CameraRotationZ, so the camNumber corresponds to any cameras that were set up with SetCamera.

```
    // Sets the Z axis rotation of the camera to 180°
    Tile.CameraRotationZ (180.0f);
    // Sets the Z axis rotation of the second camera to -45°
    Tile.CameraRotationZ (-45.0f, 1);
```

CopyGroupToPosition

```
static function CopyGroupToPosition (position : Int2,
                                     offset : Vector2 = Vector2.zero,
                                     layer : int = 0,
                                     groupSet : int,
                                     groupNumber : int) : void
```

Copies a specified group to a specified position, with an optional offset. Groups must be loaded using LoadGroups first.

The position is a location within a layer. It must be within bounds of the layer. However, if the group would extend beyond the layer bounds at that position, it will be clipped appropriately without errors.

The optional offset can be used to set the “pivot point” or center of the group. That is, the group will be placed at the specified position, and then moved by the offset. The offset must be zero or negative; positive offsets will generate an error. For example, a 5X5 cell group with an offset of (-2, -2) will be placed where the position is the middle tile of the group.

If the layer is omitted, then CopyGroupToPosition will work on layer 0 by default.

The groupSet is the set number, as shown in the TileEditor. The groupNumber is the group number within that set, as shown in the TileEditor.

```
    // Copies group 5 in set 4 to (10, 15) on layer 1, with an offset of (-2, -2)
    Tile.CopyGroupToPosition (new Int2(10, 15), new Int2(-2, -2), 1, 4, 5);
    // Copies group 2 in set 1 to (5, 5) on layer 0, with no offset
    Tile.CopyGroupToPosition (new Int2(5, 5), 1, 2);
```

DeleteTile

```
static function DeleteTile (position : Int2,
                            layer : int = 0,
                            removeCollider : boolean = false) : void
```

Removes the tile in the cell at the coordinates specified by the position. Other properties of the cell are not affected. The position can’t be lower than (0, 0), which is the lower-left corner of the map, and must be within bounds of the map.

If the layer is omitted, then DeleteTile will work on layer 0 by default.

If removeCollider is omitted, then the collider state of the cell is left alone. If it’s set to true, then the collider is removed.

```
    // Removes the tile in the cell at coords (10, 25) in layer 0
    Tile.DeleteTile (new Int2(10, 25));
    // Same thing, but uses layer 1
    Tile.DeleteTile (new Int2(10, 25), 1);
    // Removes the tile and collider in the cell at coords (10, 25) in layer 0
    Tile.DeleteTile (new Int2(10, 25), true);
    // Same thing, but uses layer 1
    Tile.DeleteTile (new Int2(10, 25), 1, true);
```

DeleteTileBlock

```
static function DeleteTileBlock (position1 : Int2,
                                 position2 : Int2,
                                 layer : int = 0,
                                 removeCollider : boolean = false) : void
```

Like DeleteTile, except it works on a block of cells—defined from position1 at one corner of the block up to and including position2 of the opposite corner—of the map in layer 0. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then DeleteTileBlock will work on layer 0 by default.

If removeCollider is omitted, then the collider state of all the cells in the block is left alone. If it's set to true, then the collider state of the block is removed.

```
    // Removes the tiles in a block of cells in layer 0, defined by (10, 18)
    // at one corner and (30, 25) at the other
    Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25));
    // Same thing, but uses layer 1
    Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25), 1);
    // Removes the tiles and colliders in a block of cells in layer 0
    Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25), true);
    // Same thing, but uses layer 1
    Tile.DeleteTileBlock (new Int2(10, 18), new Int2(30, 25), 1, true);
```

EraseLevel

```
static function EraseLevel () : void
```

Deletes all tiles on all layers, removes all colliders and triggers, and sets all rotation and order-in-layer values to 0. A level must be loaded or created before this function can be used. The level itself is not deleted, and keeps the same number of layers and so on, but all tiles are empty.

GetCollider

```
static function GetCollider (position : Int2,
                             layer : int = 0) : boolean
```

Returns the collider state of the specified map cell.

If the layer is omitted, then GetCollider will work on layer 0 by default.

```
    // Checks if the cell at (10, 20) in layer 0 is a collider
    var collider = Tile.GetCollider (new Int2(10, 20));
    if (collider) {
         Debug.Log ("None may pass");
    }
    // Same thing, but uses layer 1
    collider = Tile.GetCollider (new Int2(10, 20), 1);
```

```
static function GetCollider (position : Vector2,
                             layer : int = 0) : boolean
```

As above, but the position is in world space rather than map coordinates. Since Unity automatically converts Vector3 to Vector2, the position can be a Vector3 as well, such as a transform's position. In this case the Z is ignored.

```
    // Checks if the cell at the current world position of this transform
    // is a collider, using layer 0
    var collider = Tile.GetCollider (transform.position);
    if (collider) {
         Debug.Log ("None may pass");
    }
    // Same thing, but uses layer 1
    collider = Tile.GetCollider (transform.position, 1);
```

GetColor

```
static function GetColor (position : Int2,
                          layer : int = 0) : Color32
```

Returns a Color32 value for the specified position in the map. Note that if UseTrueColor has been set to false (which is the default), the color returned may not exactly match the color that was used with SetColor. Also note that Color32 implicitly converts to Color, and vice versa.

If the layer is omitted, then GetColor will work on layer 0 by default.

```
    // Gets the color at (5, 5) in layer 0 and stores it in a Color32 variable
    var myColor = Tile.GetColor (new Int2(5, 5));
    // Same thing, but uses layer 1
    myColor = Tile.GetColor (new Int2(5, 5), 1);
```

GetLevelBytes

```
static function GetLevelBytes () : byte[]
```

Returns an array of bytes that contains the current level. This array can then be saved to disk or uploaded to a website using various functions in Unity.

```
    // Gets the bytes for a level and saves it in the project folder as “MyFile.bytes”.
    var levelBytes = Tile.GetLevelBytes();
    System.IO.File.WriteAllBytes (Application.dataPath + "/MyFile.bytes", levelBytes);
```

GetMapBlock

```
static function GetMapBlock (position1 : Int2,
                             position2 : Int2,
                             layer : int = 0) : MapData
```

Returns a MapData object that contains all the map data in a block defined by position1 at one corner up to and including position2 at the opposite corner. Both positions must use valid coordinates inside the map, but can be in any order; that is, position1 doesn't have to be less than position2. The map data includes tile, rotation, order in layer, collider, trigger, and material (if any per-tile materials have been set up). GetMapBlock would typically be used in combination with SetMapBlock.

If the layer is omitted, then GetMapBlock will work on layer 0 by default.

```
    // Copies a block from (5, 10) to (15, 20) in layer 0,
    // and pastes it to location (50, 60) in layer 1
    var mapData = Tile.GetMapBlock (new Int2(5, 10), new Int2(15, 20));
    Tile.SetMapBlock (new Int2(50, 60), 1, mapData);
```

GetMapBlockBytes

```
static function GetColor (mapData : MapData) : byte[]
```

Returns an array of bytes for a block of MapData, such as returned by GetMapBlock. The byte array can then be saved to disk or uploaded to a website using various functions in Unity. The saved block can be loaded with LoadMapBlock, and it can also be loaded as a level in the TileEditor (though the tile size will always be 1.0 x 1.0 regardless of what was used in the level).

```
    // Gets the bytes for a block in level 0, from (5, 5) to (10, 10),
    // and saves it in the project folder as “Block1.bytes”.
    var myBlock = Tile.GetMapBlock (new Int2(5, 5), new Int2(10, 10));
    var blockBytes = Tile.GetMapBlockBytes (myBlock);
    System.IO.File.WriteAllBytes (Application.dataPath + "/Block1.bytes", blockBytes);
```

GetMapPosition

```
static function GetMapPosition (position : Vector2,
                                layer : int = 0) : Int2
```

Returns the map coordinates from the supplied world coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

If the layer is omitted, then GetMapPosition will work on layer 0 by default.

```
    // Prints "Lower left corner" if the map position is Int2(0, 0)
    var pos = Tile.GetMapPosition (transform.position);
    if (pos == Int2.zero) {
        Debug.Log ("Lower left corner");
    }
    // Same thing, but uses layer 1
    pos = Tile.GetMapPosition (transform.position, 1);
```

GetMapSize

```
static function GetMapSize (layer : int = 0) : Int2
```

Returns the map size as an Int2, where x is the number of cells on the X axis and y is the number of cells on the Y axis.

If the layer is omitted, then GetMapSize will work on layer 0 by default.

```
    // Checks if layer 0 is over 500 cells wide
    var mapSize = Tile.GetMapSize();
    if (mapSize.x > 500) {
        Debug.Log ("That’s a wide layer!");
    }
    // Same thing, but uses layer 1
    mapSize = Tile.GetMapSize (1);
```

GetNumberOfLayers

```
static function GetNumberOfLayers () : int
```

Returns the number of layers in the current level.

```
    // Gets the number of layers in the level
    var layerCount = Tile.GetNumberOfLayers();
    Debug.Log ("This level has " + layerCount + " layers");
```

GetOrder

```
static function GetOrder (position : Int2,
                          layer : int = 0) : int
```

Returns the order-in-layer number for the specified map cell. The order number will be an int between -32768 and 32767.

If the layer is omitted, then GetOrder will work on layer 0 by default.

```
     // Increases the order-in-layer for the cell at (10, 20) in layer 0 by 1
     var order = Tile.GetOrder (new Int2(10, 20));
     order++;
     Tile.SetOrder (new Int2(10, 20), order);
     // Same thing, but uses layer 1
     order = Tile.GetOrder (new Int2(10, 20), 1);
```

```
static function GetOrder (position : Vector2) : int
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
     // Increases the order-in-layer at the current world position of this transform
     // by 1, using layer 0
     var order = Tile.GetOrder (transform.position);
     order++;
     Tile.SetOrder (transform.position, order);
     // Same thing, but uses layer 1
     order = Tile.GetOrder (transform.position, 1);
```

GetRotation

```
static function GetRotation (position : Int2,
                             layer : int = 0) : float
```

Returns the rotation for the specified map cell.

If the layer is omitted, then GetRotation will work on layer 0 by default.

```
    // Rotates the cell at (10, 20) in layer 0 by 90 degrees
    var rotation = Tile.GetRotation (new Int2(10, 20));
    rotation += 90.0f;
    Tile.SetRotation (new Int2(10, 20), rotation);
    // Same thing, but uses layer 1
    rotation = Tile.GetRotation (new Int2(10, 20), 1);
```

```
static function GetRotation (position : Vector2,
                             layer : int = 0) : float
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
    // Rotates the cell at the current world position of this transform
    // by 90 degrees, using layer 0
    var rotation = Tile.GetRotation (transform.position);
    rotation += 90.0f;
    Tile.SetRotation (transform.position, rotation);
    // Same thing, but uses layer 1
    rotation = Tile.GetRotation (transform.position, 1);
```

GetSortingLayerName

```
static function GetSortingLayerName (layerNumber : int) : String
```

Returns the name of the specified sorting layer. This enables the sorting layer for sprites to be set via numerical order. The layerNumber must be at least 0, and less than the total number of sorting layers in the project. The sorting layer names must be set up first by selecting the “Assets / Set SpriteTile Sorting Layer Names” menu item.

```
    // Sets the object’s sorting layer to 1
    // Unityscript
    GetComponent(Renderer).sortingLayerName = Tile.GetSortingLayerName (1);
    // C#
    GetComponent<Renderer>().sortingLayerName = Tile.GetSortingLayerName (1);
```

GetTile

```
static function GetTile (position : Int2,
                         layer : int = 0) : TileInfo
```

Returns the set and tile numbers for the specified map cell in a TileInfo struct.

If the layer is omitted, then GetTile will work on layer 0 by default.

```
     // Sees if the cell at (10, 20) in layer 0 contains tile 3 in set 0
     var thisTile = Tile.GetTile (new Int2(10, 20));
     if (thisTile.set == 0 && thisTile.tile == 3) {
          Debug.Log ("Found tile 3");
     }
     // Same thing, but uses layer 1
     thisTile = Tile.GetTile (new Int2(10, 20), 1);
```

```
static function GetTile (position : Vector2,
                         layer : int = 0) : TileInfo
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
     // Sees if the cell in layer 0 at the current world position of this transform
     // contains tile 3 in set 0
     var thisTile = Tile.GetTile (transform.position);
     if (thisTile.set == 0 && thisTile.tile == 3) {
          Debug.Log ("Found tile 3");
     }
     // Same thing, but uses layer 1
     thisTile = Tile.GetTile (transform.position, 1);
```

GetTilePositions

```
static function GetTilePositions (tileInfo : TileInfo,
                                  layer : int = 0,
                                  ref positions : List<Int2>) : void
```

Fills an Int2 List with all the positions in the layer where the specified TileInfo value occurs.

If the layer is omitted, then GetTilePositions will work on layer 0 by default.

The positions variable is a generic List of type Int2. It should be declared before being passed into GetTilePositions. It can be null, or not; any existing entries will be cleared. If no cells containing the TileInfo value exist, then positions will have a Count of 0.

```
    // Gets all positions for set 3, tile 1 in layer 0
    var tilePositions : List.<Int2>; // Unityscript
    Tile.GetTilePositions (TileInfo(3, 1), tilePositions);
    for (var i = 0; i < tilePositions.Count; i++) {
        Debug.Log ("Found at " + tilePositions[i]);
    }
    // Same thing, but uses layer 1
    Tile.GetTilePositions (new TileInfo(3, 1), 1, tilePositions);

    List<Int2> tilePositions;  // C#
    Tile.GetTilePositions (new TileInfo(3, 1), ref tilePositions);
    for (var i = 0; i < tilePositions.Count; i++) {
        Debug.Log ("Found at " + tilePositions[i]);
    }
    // Same thing, but uses layer 1
    Tile.GetTilePositions (new TileInfo(3, 1), 1, ref tilePositions);
```

GetTileSize

```
static function GetTileSize (layer : int = 0) : Vector2
```

Returns the size of the tiles for a specified layer, in world units. If the layer is omitted, then GetTileSize will work on layer 0 by default. If a square tile grid has been used, the X and Y will be the same.

```
    // Prints the equivalent in world units of 50 cells along the x axis in layer 0
    var tileSize = Tile.GetTileSize();
    Debug.Log ("50 cells is " + tileSize.x*50 + " units");
    // Same thing, but uses layer 1
    tileSize = Tile.GetTileSize (1);
```

GetTrigger

```
static function **GetTrigger** (**position** : Int2,
                               **layer** : int = **0**) : int
```

Returns the trigger of the specified map cell. The trigger is an int between 0 and 255.

If the layer is omitted, then GetTrigger will work on layer 0 by default.

```
     // If the cell at (10, 20) in layer 0 has a trigger of 1, set the cell to tile 10
     var trigger = Tile.GetTrigger (new Int2(10, 20));
     if (trigger == 1) {
          Tile.SetTile (new Int2(10, 20), 0, 10);
     }
     // Same thing, but uses layer 1
     trigger = Tile.GetTrigger (new Int2(10, 20), 1);
```

```
static function **GetTrigger** (**position** : Vector2,
                               **layer** : int = **0**) : int
```

As above, but the position is in world space rather than map coordinates. A Vector3 can be used instead of Vector2, in which case the Z is ignored.

```
     // If the cell at the current world position of this transform in layer 0
     // has a trigger of 1, set the cell to set 0, tile 10
     var trigger = Tile.GetTrigger (transform.position);
     if (trigger == 1) {
          Tile.SetTile (transform.position, 0, 10);
     }
     // Same thing, but uses layer 1
     trigger = Tile.GetTrigger (transform.position, 1);
```

GetTriggerPositions

```
static function GetTriggerPositions (trigger : int,
                                     layer : int = 0,
                                     ref positions : List<Int2>) : void
```

Fills an Int2 List with all the positions in the layer where the specified trigger value occurs.

If the layer is omitted, then GetTriggerPositions will work on layer 0 by default.

The positions variable is a generic List of type Int2. It should be declared before being passed into GetTriggerPositions. It can be null, or not; any existing entries will be cleared. If no cells containing the trigger value exist, then positions will have a Count of 0.

```
    // Gets all positions for tiles with a trigger value of 25 in layer 0
    var triggerPositions : List.<Int2>; // Unityscript
    Tile.GetTriggerPositions (25, triggerPositions);
    for (var i = 0; i < triggerPositions.Count; i++) {
        Debug.Log ("Found at " + triggerPositions[i]);
    }
    // Same thing, but uses layer 1
    Tile.GetTriggerPositions (25, 1, triggerPositions);

    List<Int2> triggerPositions;  // C#
    Tile.GetTriggerPositions (25, ref triggerPositions);
    for (var i = 0; i < triggerPositions.Count; i++) {
        Debug.Log ("Found at " + triggerPositions[i]);
    }
    // Same thing, but uses layer 1
    Tile.GetTriggerPositions (25, 1, ref triggerPositions);
```

GetWorldPosition

```
static function GetWorldPosition (position : Int2,
                                  layer : int = 0) : Vector3
```

Returns the world position in units that corresponds to the specified cell.

If the layer is omitted, then GetWorldPosition will work on layer 0 by default.

```
    // Moves the transform to the world position for (10, 20) in layer 0
    var worldPos = Tile.GetWorldPosition (new Int2(10, 20));
    transform.position = worldPos;
    // Same thing, but uses layer 1
    var worldPos = Tile.GetWorldPosition (new Int2(10, 20), 1);
```

GetZPosition

```
static function **GetZPosition** (**layer** : int = **0**) : float
```

Returns the distance from the origin along the Z axis.

If the layer is omitted, then GetZPosition will work on layer 0 by default.

```
     // Gets the z position of layer 0
     var zPos = Tile.GetZPosition();
     transform.position = new Vector3(5, 10, zPos);
     // Same thing, but uses layer 1
     var zPos = Tile.GetZPosition (1);
```

GrabSprite

```
static function GrabSprite (position : Int2,
                            layer : int = 0,
                            name : String = "Sprite",
                            deleteTile : boolean = true) : GameObject
```

Returns a GameObject derived from the tile at the specified position. The GameObject has a SpriteRenderer component with the appropriate sprite, and a SpriteTileInfo component, which is used to store the tile set info for use with PutSprite. It will have the same order-in-layer value and rotation as the tile. The GameObject's sorting layer corresponds to the layer specified in GrabSprite. If the tile has a physics collider, a PolygonCollider2D component with the appropriate shape will be added to the sprite. Note that empty tiles can't be grabbed.

If the layer is omitted, then GrabSprite will work on layer 0 by default.

The name for the GameObject is "Sprite" by default, but this can be changed by supplying a string.

By default, the tile is removed from the specified position and replaced with an empty tile; if the cell has a collider, it's removed. Supplying false for the deleteTile parameter will make GrabSprite leave the tile alone.

```
    // Converts the tile at (5, 5) in layer 0 to a sprite
    var mySprite = Tile.GrabSprite (new Int2(5, 5));
    // Same, but uses layer 1
    mySprite = Tile.GrabSprite (new Int2(5, 5), 1);
    // As above, but names the sprite "MySprite"
    mySprite = Tile.GrabSprite (new Int2(10, 10), 1, "MySprite");
    // As above, but doesn't remove the tile
    mySprite = Tile.GrabSprite (new Int2(15, 15), 1, "MySprite", false);
```

```
static function GrabSprite (position : Int2,
                            layer : int = 0,
                            go : GameObject,
                            deleteTile : boolean = true) : void
```

Similar to the above method, but instead of returning a new GameObject, it uses an existing GameObject. The supplied GameObject must have a SpriteRenderer component and a SpriteTileInfo component. This way, GameObjects can be re-used. This can be useful for creating a pooling system, if GrabSprite and PutSprite are used frequently, rather than repeatedly destroying and creating new objects.

```
    // Converts the tile at (5, 5) in layer 0 to a sprite
    var mySprite = Tile.GrabSprite (new Int2(5, 5));
    // Uses the grabbed sprite to convert the tile at (10, 10)
    Tile.GrabSprite (new Int2(10, 10), mySprite);
    // As above, but uses layer 1
    Tile.GrabSprite (new Int2(10, 10), 1, mySprite);
    // As above, but doesn't remove the tile
    Tile.GrabSprite (new Int2(15, 15), 1, mySprite, false);
```

LoadGroups

```
static function LoadGroups (level : TextAsset) : void
```

Loads a SpriteTile group file from a TextAsset file. If groups have been loaded previously, they will be replaced by the new groups. Once loaded, groups can then be used with the CopyGroupToPosition function.

```
var myGroups : TextAsset; // Unityscript

function Start () {
     Tile.LoadGroups (myLevel);
}
```

```
     public TextAsset myGroups; // C#

     void Start () {
          Tile.LoadGroups (myGroups);
     }
```

```
static function LoadGroups (bytes : byte[]) : void
```

Loads a SpriteTile group file from a byte array. The byte array can be obtained from external files, downloaded from the WWW, or some other method.

```
var pathToGroupFile : String; // Unityscript

function Start () {
     var bytes = System.IO.File.ReadAllBytes (pathToGroupFile);
     Tile.LoadGroups (bytes);
}
```

```
     public string pathToGroupFile; //C#

     void Start () {
          var bytes = System.IO.File.ReadAllBytes (pathToGroupFile);
          Tile.LoadGroups (bytes);
     }
```

LoadLevel

```
static function LoadLevel (level : TextAsset) : void
```

Loads a SpriteTile level from a TextAsset file. If a level already exists, it will be replaced by the new level. Loading a level will reset the position of any layers, so if SetLayerPosition had been used, it will need to be called again. Also, any materials set up with SetTileMaterial will need to be set up again.

```
var myLevel : TextAsset; // Unityscript

function Awake () {
     Tile.SetCamera();
     Tile.LoadLevel (myLevel);
}
```

```
     public TextAsset myLevel; // C#

     void Awake () {
          Tile.SetCamera();
          Tile.LoadLevel (myLevel);
     }
```

```
static function LoadLevel (bytes : byte[]) : void
```

Loads a SpriteTile level from a byte array. The byte array can be obtained from external files, downloaded from the WWW, or some other method.

```
var pathToLevelFile : String; // Unityscript

function Awake () {
     Tile.SetCamera();
     var bytes = System.IO.File.ReadAllBytes (pathToLevelFile);
     Tile.LoadLevel (bytes);
}
```

```
     public string pathToLevelFile; //C#

     void Awake () {
          Tile.SetCamera();
          var bytes = System.IO.File.ReadAllBytes (pathToLevelFile);
          Tile.LoadLevel (bytes);
     }
```

LoadMapBlock

```
static function LoadMapBlock (byte[] : bytes) : MapData
```

Converts a byte array to a MapData block, which can be used with SetMapBlock. The byte array can be loaded from disk using IO functions, loaded from the web using WWW functions, or obtained from TextAsset.bytes. Typically GetMapBlockBytes would have been used to save the byte array. It must be a valid SpriteTile file.

```
// Loads a level, then loads a block specified by pathToBlockFile and pastes it
// into the level in layer 0 at position (5, 5)
var pathToBlockFile : String; // Unityscript
var level : TextAsset;

function Awake () {
     Tile.SetCamera();
     Tile.LoadLevel (level);
     var blockBytes = System.IO.File.ReadAllBytes (pathToBlockFile);
     var myBlock = Tile.LoadMapBlock (blockBytes);
     Tile.SetMapBlock (Int2(5, 5), myBlock);
}
```

```
     public string pathToBlockFile; // C#
     public TextAsset level;

     void Awake () {
          Tile.SetCamera();
          Tile.LoadLevel (level);
          var blockBytes = System.IO.File.ReadAllBytes (pathToBlockFile);
          var myBlock = Tile.LoadMapBlock (blockBytes);
          Tile.SetMapBlock (new Int2(5, 5), myBlock);
     }
```

NewLevel

```
static function NewLevel (mapSize : Int2,
                          addBorder : int,
                          tileSize : float (or Vector2),
                          zPosition : float,
                          layerLock : LayerLock) : void
```

Creates a new level with one layer, using the specified parameters. If a level already exists, it's erased. Any layers set with SetLayerPosition will be reset, and any materials set with SetTileMaterial will also be reset.

mapSize: the dimensions, in tiles, of the level. Must be at least 1x1.

addBorder: the number of additional rows/columns that are added around the screen border, when using oversized tiles or camera rotation. Must be 0 or greater.

tileSize: the dimension, in units, of each cell in the level. Must be at least .001. For a non-square tile grid, use a Vector2 instead of a float.

zPosition: the distance from the origin along the Z axis. Only has an effect with perspective cameras. Must be at least 0.0.

layerLock: whether the layer should be prevented from moving on the X or Y axes. Uses the LayerLock enum:

LayerLock.None: the layer is not locked.

LayerLock.X: the layer is locked on the X axis, but can move on the Y axis.

LayerLock.Y: the layer is locked on the Y axis, but can move on the X axis.

LayerLock.XandY: the layer is locked on both the X axis and the Y axis.

```
     // Makes a new level with one layer of 100x50 tiles, no added border tiles,
     // a tile size of 2.0, positioned at 5.0 on the z axis, and locked on the X axis
     Tile.NewLevel (new Int2(100, 50), 0, 2.0f, 5.0f, LayerLock.X);
     Tile.SetCamera();
```

```
static function NewLevel (levelData : LevelData[]) : void
```

Creates a new level with multiple layers, using an array of the LevelData class. Each entry in the array contains properties for one layer, with layer 0 being the topmost layer. The LevelData properties are mapSize (Int2), addBorder (int), tileSize (Vector2), zPosition (float), and layerLock (LayerLock). If you want a square grid, just use the same value for the X and Y tileSize.

```
     // Creates a new level that has two layers
     // Layer 0 is 100x100 with an added border of 1, tile size 1.0, at z position 0.0
     // Layer 1 is 60x20 with no added border, tile size 2.0 X 4.0, at z position 10.0
     var levelData = new LevelData[2];
     levelData[0] = new LevelData(new Int2(100, 100), 1, new Vector2(1.0f, 1.0f), 0.0f,
LayerLock.None);
     levelData[1] = new LevelData(new Int2(60, 20), 0, new Vector2(2.0f, 4.0f), 10.0f,
LayerLock.Y);
     Tile.NewLevel (levelData);
     Tile.SetCamera();
```

PutSprite

```
static function PutSprite (go : GameObject,
                           layer : int = 0,
                           disposeType : DisposeType = DisposeType.Destroy) : void
```

Converts a sprite created with GrabSprite back into a tile. The supplied GameObject must have SpriteRenderer and SpriteTileInfo components attached. The world position of the GameObject is converted to the closest tile coordinate, and must be within bounds of the level. The tile will have the same rotation and order-in-layer values as the GameObject. If the sprite has a PolygonCollider2D component, the tile will have a physics collider set with the appropriate collider shape which corresponds to the tile that the sprite was grabbed from.

If the layer is omitted, then PutSprite will work on layer 0 by default.

The disposeType is DisposeType.Destroy by default, which means the sprite GameObject is destroyed after calling PutSprite. Other values are DisposeType.Deactivate, which deactivates the sprite rather than destroying it by calling SetActive (false) on the GameObject; and DisposeType.LeaveAlone, which leaves the sprite untouched. DisposeType.Deactivate could be used to create a pooling system, so that sprites used with GrabSprite can be reused, rather than repeatedly creating and destroying GameObjects.

```
    // Converts the tile at (5, 5) in layer 0 to a sprite
    var mySprite = Tile.GrabSprite (new Int2(5, 5));
    // Move the sprite 5 units over, then clone it into the level in layer 1
    mySprite.transform.Translate (Vector3.right * 5);
    Tile.PutSprite (mySprite, 1, DisposeType.LeaveAlone);
    // Put the sprite into layer 0 and destroy it
    Tile.PutSprite (mySprite);
```

ScreenToMapPosition

```
static function ScreenToMapPosition (screenPos : Vector2,
                                     layer : int = 0,
                                     out mapPos : Int2
                                     camNumber : int = 0) : bool
```

Converts screen coordinates (such as supplied by Input.mousePosition) to a map position. The mapPos variable must be declared before calling the function. The function returns true if the screen position is inside the map, or false if outside. This way, attempting to refer to out-of-bounds coordinates for the map can be avoided.

If the layer is omitted, then ScreenToMapPosition will work on layer 0 by default.

If the camNumber is omitted, then camera 0 is used by default. If only one camera was used with SetCamera, then this should always be 0. If multiple cameras were used, then the camNumber refers to the respective entry in the Camera[] array.

```
    // Unityscript
    // Prints the tile coords that the mouse cursor is over, for layer 0
    var mapPos : Int2;
    if (Tile.ScreenToMapPosition (Input.mousePosition, mapPos)) {
         Debug.Log ("Mouse is over tile coords " + mapPos);
    }
```

```
    // C#
    // Prints the tile coords that the mouse cursor is over, for layer 0
    Int2 mapPos;
    if (Tile.ScreenToMapPosition (Input.mousePosition, out mapPos)) {
         Debug.Log ("Mouse is over tile coords " + mapPos);
    }
```

SetBorder

```
static function SetBorder (layer : int = 0,
                           set : int,
                           tile : int,
                           setCollider : bool) : void
```

Creates a 1-tile border, using the specified set and tile numbers, around the perimeter of the map. The collider cells are set as well if setCollider is true. This is particularly useful for character controllers that use GetCollider: if there's always a border around the map, then you can avoid having to check for out-of-bounds movement.

If the layer is omitted, then SetBorder will work on layer 0 by default.

```
     // Makes a border around the edge of the map in layer 0, using set 3, tile 1,
     // and sets the collider cells for the border
     Tile.SetBorder (3, 1, true);
     // Same thing, but uses layer 2 and sets the collider cells of the border to false
     Tile.SetBorder (2, 3, 1, false);
```

```
static function SetBorder (layer : int = 0,
                           tileInfo : TileInfo,
                           setCollider : bool) : void
```

As above, but uses a TileInfo struct for the set and tile info.

```
     // Makes a border around the edge of the map in layer 0, using set 3, tile 1,
     // and sets the collider cells for the border
     var tInfo = new TileInfo(3, 1);
     Tile.SetBorder (tInfo, true);
     // Same thing, but uses layer 2 and sets the collider cells of the border to false
     var tInfo = new TileInfo(3, 1);
     Tile.SetBorder (2, tInfo, false);
```

SetCamera

```
static function SetCamera () : void
```

Initializes a camera or cameras for use with SpriteTile. This should be called first, before using any other SpriteTile functions such as LoadLevel or NewLevel. Without arguments, any camera tagged “MainCamera” is used. If there are multiple cameras tagged “MainCamera”, all are used.

If other scripts depend on SpriteTile being set up, make sure they run after SetCamera is called. For example, multiple Start functions have no defined order unless the script execution order in set explicitly in Unity, so using Awake is typically a good idea.

```
static function SetCamera (camera : Camera) : void
```

As above, but uses a specific camera instead of the MainCamera tag.

```
static function SetCamera (cameras : Camera[]) : void
```

As above, but uses a specified array of cameras instead of the MainCamera tag.

```
var myLevel : TextAsset; // Unityscript
var myCam : Camera;
var myCameras : Camera[];

function Awake () {
     // Sets the camera to any camera or cameras tagged MainCamera
     Tile.SetCamera();
     // This would use myCam instead:
     // Tile.SetCamera (myCam);
     // This would use an array of cameras:
     // Tile.SetCamera (myCameras);
     Tile.LoadLevel (myLevel);
}
```

```
     public TextAsset myLevel; // C#
     public Camera myCam;
     public Camera[] myCameras;

     void Awake () {
          // Sets the camera to any camera or cameras tagged MainCamera
          Tile.SetCamera();
          // This would use myCam instead:
          // Tile.SetCamera (myCam);
          // This would use an array of cameras:
          // Tile.SetCamera (myCameras);
          Tile.LoadLevel (myLevel);
     }
```

SetCollider

```
static function SetCollider (position : Int2,
                             layer : int = 0,
                             active : bool) : void
```

Sets the collider to either active (true) or inactive (false), of the cell at the coordinates, specified by the position, of the map. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. If the appropriate tile in the TileEditor has the "Use physics collider" setting checked, then a polygon collider is created from the sprite shape of this tile.

If the layer is omitted, then SetCollider will work on layer 0 by default.

```
     // Set the cell in layer 0 at coords (10, 25) to an active collider
     Tile.SetCollider (new Int2(10, 25), true);
     // Same thing, but uses layer 1
     Tile.SetCollider (new Int2(10, 25), 1, true);
```

SetColliderBlock

```
static function SetColliderBlock (position1 : Int2,
                                  position2 : Int2,
                                  layer : int = 0,
                                  active : bool) : void
```

Like SetCollider, except it works on a block of cells in the map, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetColliderBlock will work on layer 0 by default.

```
     // Set the cells in layer 0, using a block defined by (10, 18) at
     // one corner and (30, 25) at the other, to active collider cells
     Tile.SetColliderBlock (new Int2(10, 18), new Int2(30, 25), true);
     // Same thing, but uses layer 1
     Tile.SetColliderBlock (new Int2(10, 18), new Int2(30, 25), 1, true);
```

SetColliderBlockSize

```
static function SetColliderBlockSize (size : int) : void
```

Sets the size of collider blocks used for polygon colliders. (See Collider Blocks in the How Colliders Work section of the SpriteTile documentation.) This must be at least 1, with no particular upper limit. SetColliderBlockSize must be called before any level setup is done. If no polygon colliders are used, this function has no effect.

```
function Start () {  // Unityscript
     // Sets the collider block size to 10x10
     Tile.SetColliderBlockSize (10);
     Tile.LoadLevel (myLevel);
     Tile.SetCamera();
}
```

```
     void Start () { // C#
          // Sets the collider block size to 10x10
          Tile.SetColliderBlockSize (10);
          Tile.LoadLevel (myLevel);
          Tile.SetCamera();
     }
```

SetColliderLayer

```
static function SetColliderLayer (layer : int) : void
```

Sets the GameObject layer of all physics colliders used by SpriteTile to the specified layer. The layer must be at least 0 and not greater than 31.

```
     // Sets the physics colliders to Unity’s IgnoreRaycast layer, which is layer 2
     Tile.SetColliderLayer (2);
```

SetColliderMaterial

```
static function SetColliderMaterial (material : PhysicsMaterial2D) : void
```

Sets the PhysicsMaterial2D used for polygon colliders in the level. If no material is set, the default Unity PhysicsMaterial2D is used.

```
var colliderMaterial : PhysicsMaterial2D; // Unityscript
var myLevel : TextAsset;

function Start () {
     Tile.LoadLevel (myLevel);
     Tile.SetCamera();
     Tile.SetColliderMaterial (colliderMaterial);
}
```

```
     public PhysicsMaterial2D colliderMaterial; // C#
     public TextAsset myLevel;

     void Start () {
          Tile.LoadLevel (myLevel);
          Tile.SetCamera();
          Tile.SetColliderMaterial (colliderMaterial);
     }
```

SetColliderTag

```
static function SetColliderTag (tag : String) : void
```

Sets the GameObject tag used for all polygon colliders in the level. The tag should be defined in the Unity tag manager, or else a runtime exception will occur if a non-existent tag is used.

```
     // Sets the tag for all SpriteTile colliders to "Background",
     // which should be set up in the Unity tag manager
     Tile.SetColliderTag ("Background");
```

SetColor

```
static function SetColor (p : Int2,
                          layer : int = 0,
                          color : Color32) : void
```

Sets the tile at the specified position to the supplied color. Color and Color32 convert implicitly to each other, so values such as Color.red are acceptable. Alpha values will work as long as the tile material allows transparency.

If the layer is omitted, then SetColor will work on layer 0 by default.

```
     // Sets the tile at (5, 5) in layer 0 to red
     Tile.SetColor (new Int2(5, 5), Color.red);
     // Sets the tile at (5, 5) in layer 1 to a semi-transparent medium purplish color
     Tile.SetColor (new Int2(5, 5), 1, new Color32(100, 10, 115, 50));
```

SetColorBlock

```
static function SetColor (p1 : Int2,
                          p2 : Int2,
                          layer : int = 0,
                          color : Color32) : void
```

Like SetColor, except it works on a block of cells in the map, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetColorBlock will work on layer 0 by default.

```
     // Set the cells in layer 0, using a block defined by (10, 18) at
     // one corner and (30, 25) at the other, to red
     Tile.SetColorBlock (new Int2(10, 18), new Int2(30, 25), Color.red);
     // Same thing, but uses layer 1
     Tile.SetColorBlock (new Int2(10, 18), new Int2(30, 25), 1, Color.red);
```

SetLayerActive

```
static function SetLayerActive (layer : int,
                                active : boolean) : void
```

Activates or deactivates the rendering for a given layer. Polygon colliders, if any, are not affected.

```
     // Turns layer 1 off
     Tile.SetLayerActive (1, false);
```

SetLayerPosition

```
static function SetLayerPosition (layer : int,
                                  position : Vector2) : void
```

Sets the specified layer to a particular position in world space. This is primarily useful for layers that use a LayerLock other than LayerLock.None, such as fixed background layers, so you can place them as desired.

A good way to figure out what exact position to use is to first run a level without using SetLayerPosition, and while the level is running, move the SpriteTileLayerX object (where X corresponds to the appropriate layer) until it lines up as desired. Make note of the layer's Transform.Position numbers, stop play mode in Unity, and use those numbers in SetLayerPosition.

```
    // Moves layer 1's X position to -5.0 and Y position to -3.0 in world space
    Tile.SetLayerPosition (1, new Vector2(-5.0, -3.0));
```

SetMapBlock

```
static function SetMapBlock (position : Int2,
                             layer : int = 0,
                             mapData : MapData) : void
```

Sets a block of MapData to a specified position in the map. The position is the lower-left corner of the MapData block, and the size of the block at that position must not exceed the bounds of the map.

If the layer is omitted, then SetMapBlock will use layer 0 by default.

The mapData is typically retrieved by GetMapBlock.

```
    // Copies a block from (5, 10) to (15, 20) in layer 0,
    // and pastes it to location (50, 60) in layer 1
    var mapData = Tile.GetMapBlock (new Int2(5, 10), new Int2(15, 20));
    Tile.SetMapBlock (new Int2(50, 60), 1, mapData);
```

SetMapTileset

```
static function SetMapTileset (layer : int = 0,
                               set : int) : void
```

Sets the tileset of all the tiles in a layer to a specified set. Useful for quickly switching between similar tilesets, such as day/night changes. The set must be a valid tileset as set up in the TileEditor. All the tile numbers in the layer must exist in the set that's being switched to. In other words, if the layer currently uses tiles in tileset 0, which contains 50 tiles, and SetMapTileset is used to switch to tileset 1, then tileset 1 must also contain at least 50 tiles.

If the layer is omitted, then SetMapTileset will use layer 0 by default.

```
    // Switch layer 0 to tileset 2
    Tile.SetMapTileset (2);
    // Switch layer 1 to tileset 3
    Tile.SetMapTileset (1, 3);
```

SetOrder

```
static function SetOrder (position : Int2,
                          layer : int = 0,
                          order : int) : void
```

Sets the order-in-layer number of the cell at the coordinates, specified by the position, of the map in layer 0. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. The order number must be between -32768 and 32767.

If the layer is omitted, then SetOrder will work on layer 0 by default.

```
    // Set the order in layer of the cell in layer 0 at coords (10, 25) to -3
    Tile.SetOrder (new Int2(10, 25), -3);
    // Same thing, but uses layer 1
    Tile.SetOrder (new Int2(10, 25), 1, -3);
```

SetOrderBlock

```
static function SetOrderBlock (position1 : Int2,
                               position2 : Int2,
                               layer : int = 0,
                               order : int) : void
```

Like SetOrder, except it works on a block of cells of the map, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetOrderBlock will work on layer 0 by default.

```
    // Set the order in layer of the cells in layer 0, using a block defined
    // by (10, 18) at one corner and (30, 25) at the other, to -3
    Tile.SetOrderBlock (new Int2(10, 25), -3);
    // Same thing, but uses layer 1
    Tile.SetOrderBlock (new Int2(10, 25), 1, -3);
```

SetRotation

```
static function SetRotation (position : Int2,
                             layer : int = 0,
                             rotation : float) : void
```

Sets the rotation of the cell at the coordinates of the map, specified by the position. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. The rotation is between 0.0 and 360.0. Any numbers outside that range are repeated so they can be represented by the 0.0 to 360.0 range; that is, -45.0 would become 315.0.

If the layer is omitted, then SetRotation will work on layer 0 by default.

```
    // Set the rotation of the cell in layer 0 at coords (10, 25) to 45°
    Tile.SetRotation (new Int2(10, 25), 45.0);
    // Same thing, but uses layer 1
    Tile.SetRotation (new Int2(10, 25), 1, 45.0);
```

SetRotationBlock

```
static function SetRotationBlock (position1 : Int2,
                                  position2 : Int2,
                                  layer : int = 0,
                                  rotation : float) : void
```

Like SetRotation, except it works on a block of cells of the map, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetRotationBlock will work on layer 0 by default.

```
    // Set the rotation of the cells in layer 0, using a block defined
    // by (10, 18) at one corner and (30, 25) at the other, to 45°
    Tile.SetRotationBlock (new Int2(10, 18), new Int2(30, 25), 45.0);
    // Same thing, but uses layer 1
    Tile.SetRotationBlock (new Int2(10, 18), new Int2(30, 25), 1, 45.0);
```

SetTile

```
static function SetTile (position : Int2,
                         layer : int = 0,
                         set : int,
                         tile : int) : void
```

```
static function SetTile (position : Int2,
                         layer : int = 0,
                         tileInfo : TileInfo) : void
```

Sets the cell at the coordinates of the map, specified by the position, to the tile specified by the set and tile numbers. Other properties of the cell are not affected. The set and tile numbers can be seen in the TileEditor window. The position can't be lower than (0, 0) and must be within bounds of the map.

If the layer is omitted, then SetTile will work on layer 0 by default.

The set and tile numbers must refer to sets and tiles that exist in the TileManager. They can either be specified separately as ints, or by using a TileInfo struct.

```
     // Sets the cell in layer 0 at coords (10, 25) to set 2, tile 5
     Tile.SetTile (new Int2(10, 25), 2, 5); // Using ints
     Tile.SetTile (new Int2(10, 25), new TileInfo(2, 5)); // Using TileInfo
     // Same thing, but uses layer 1
     Tile.SetTile (new Int2(10, 25), 1, 2, 5); // Using ints
     Tile.SetTile (new Int2(10, 25), 1, new TileInfo(2, 5)); // Using TileInfo
```

```
static function SetTile (position : Int2,
                         layer : int = 0,
                         set : int,
                         tile : int,
                         setCollider : boolean) : void
```

```
static function SetTile (position : Int2,
                         layer : int = 0,
                         tileInfo : TileInfo,
                         setCollider : boolean) : void
```

As above, but the collider of the cell will also be set, depending on the value of setCollider.

```
     // Sets the cell in layer 0 at coords (10, 25) to set 2, tile 5,
     // and sets the corresponding collider to true
     Tile.SetTile (new Int2(10, 25), 2, 5, true); // Using ints
     Tile.SetTile (new Int2(10, 25), new TileInfo(2, 5), true); // Using TileInfo
     // Same thing, but uses layer 1
     Tile.SetTile (new Int2(10, 25), 1, 2, 5, true); // Using ints
     Tile.SetTile (new Int2(10, 25), 1, new TileInfo(2, 5), true); // Using TileInfo
```

SetTileBlock

```
static function SetTileBlock (position1 : Int2,
                              position2 : Int2,
                              layer : int = 0,
                              set : int,
                              tile : int) : void
```

```
static function SetTileBlock (position1 : Int2,
                              position2 : Int2,
                              layer : int = 0,
                              tileInfo : TileInfo) : void
```

Like SetTile, except it works on a block of cells, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetTileBlock will work on layer 0 by default.

The set and tile numbers must refer to sets and tiles that exist in the TileManager. They can either be specified separately as ints, or by using a TileInfo struct.

```
    // Sets a block of cells in layer 0, defined by (10, 18) at one corner and
    // (30, 25) at the other, to set 2, tile 5
    Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), 2, 5);
    var tInfo = new TileInfo(2, 5);
    Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), tInfo);
    // Same thing, but uses layer 1
    Tile.SetTileBlock (new Int2(10, 25), new Int2(25, 18), 1, tInfo);
```

```
static function SetTileBlock (position1 : Int2,
                              position2 : Int2,
                              layer : int = 0,
                              set : int,
                              tile : int,
                              setCollider : boolean) : void
```

```
static function SetTileBlock (position1 : Int2,
                              position2 : Int2,
                              layer : int = 0,
                              tileInfo : TileInfo,
                              setCollider : boolean) : void
```

As above, but the collider of the cells in the block will also be set, depending on the value of setCollider.

```
    Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), 2, 5, true);
    Tile.SetTileBlock (new Int2(10, 18), new Int2(30, 25), new TileInfo(2, 5), true);
```

SetTileLayerScale

```
static function SetTileLayerScale (layer : int = 0,
                                   scale : float) : void
```

Sets the scale of all tiles in the specified layer to the value specified by scale. This works the same as SetTileScale, except that it only affects the specified layer and overrides any default scale value supplied by SetTileScale. Note, however, that any new sprites created in the layer (such as by zooming out) will use the default, so SetTileLayerScale may need to be called again in that case.

If the layer is omitted, then SetTileLayerScale will use layer 0 by default.

```
     // Sets the scale of all tiles in layer 0 to 1.5
     Tile.SetTileLayerScale (1.5f);
     // Sets the scale of all tiles in layer 1 to 1.25
     Tile.SetTileLayerScale (1, 1.25f);
```

SetTileMaterial

```
static function SetTileMaterial (material : Material) : void
```

Sets the material used for all tiles in the level. If no material is set, the default Unity sprite material is used, unless overridden by the “non-transparent” or “use dynamic lighting” options in the TileEditor.

```
    // Makes tile sprites use Sprites/Diffuse shader
    var newMaterial = new Material(Shader.Find ("Sprites/Diffuse"));
    Tile.SetTileMaterial (newMaterial);
```

```
static function SetTileMaterial (set : int,
                                 tile : int,
                                 material : Material) : void
```

```
static function SetTileMaterial (tileInfo : TileInfo,
                                 material : Material) : void
```

Sets the default material for all tiles using the specified set and tile numbers. This will override the non-transparent or dynamic lighting options in the TileEditor.

```
    // Makes all tile sprites in set 3, tile 12 use the supplied material
    Tile.SetTileMaterial (new TileInfo(3, 12), myMaterial);
```

```
static function SetTileMaterial (layer : int = 0,
                                 position : Int2,
                                 material : Material) : void
```

Sets the material for the tile at the specified position. If the layer is omitted, layer 0 will be used by default. This will override all other material settings from either the TileEditor or by using SetTileMaterial with a set/tile number. Note that this usage of SetTileMaterial will cause cells in the map to use 8 bytes instead of 7, and a maximum of 256 different materials are allowed.

```
    // Makes the tile at (5, 10) in layer 1 use the supplied material
    Tile.SetTileMaterial (1, new Int(5, 10), myMaterial);
```

SetTileRenderLayer

```
static function SetTileRenderLayer (layer : int) : void
```

Sets the layer used for all tiles in the level. This is the GameObject layer as opposed to the sorting layer. The layer value must be between 0 and 31.

```
    // Makes tile sprites use the IgnoreRaycast layer
    Tile.SetTileRenderLayer (2);
```

SetTileScale

```
static function SetTileScale (scale : float) : void
```

Sets the scale of all tiles to the value specified by scale. The scale affects both the X and Y axes. This can be called at any time, even before SetCamera. SetTileScale can be used for special effects, and it can also be used if occasional 1-pixel gaps between tiles are visible when the camera is moved. Specifying a value slightly greater than 1.0 in this case will typically eliminate any such gaps.

```
    // Sets the scale of all tiles to 1.001
    Tile.SetTileScale (1.001f);
```

SetTrigger

```
static function SetTrigger (position : Int2,
                            layer : int = 0,
                            trigger : int) : void
```

Sets the trigger number of the cell at the coordinates in the map, specified by the position. Other properties of the cell are not affected. The position can't be lower than (0, 0) and must be within bounds of the map. The trigger number must be between 0 and 255.

If the layer is omitted, then SetTrigger will work on layer 0 by default.

```
    // Set the trigger of the cell in layer 0 at coords (10, 25) to 3
    Tile.SetTrigger (new Int2(10, 25), 3);
    // Same thing, but uses layer 1
    Tile.SetTrigger (new Int2(10, 25), 1, 3);
```

SetTriggerBlock

```
static function SetTriggerBlock (position1 : Int2,
                                 position2 : Int2,
                                 layer : int = 0,
                                 trigger : int) : void
```

Like SetTrigger, except it works on a block of cells, defined from position1 at one corner of the block up to and including position2 of the opposite corner. Both positions are clamped to the size of the map if necessary, and can be in any order; that is, position1 doesn't have to be less than position2.

If the layer is omitted, then SetTriggerBlock will work on layer 0 by default.

```
    // Set the trigger of the cells in layer 0, using a block defined
    // by (10, 18) at one corner and (30, 25) at the other, to 3
    Tile.SetTriggerBlock (new Int2(10, 18), new Int2(30, 25), 3);
    // Same thing, but uses layer 1
    Tile.SetTriggerBlock (new Int2(10, 18), new Int2(30, 25), 1, 3);
```

StopAnimatingTile

```
static function StopAnimatingTile (tileInfo : TileInfo) : void
```

Stops a tile from animating. If AnimateTile has been called for the specified tileInfo, then the animation will be halted immediately. If tileInfo is currently not animating, then nothing will happen.

```
    // Stops tile #10 in set 1 from animating
    Tile.StopAnimatingTile (new TileInfo(1, 10));
```

StopAnimatingTileRange

```
static function StopAnimatingTileRange (tileInfo : TileInfo,
                                        range : int) : void
```

Stops a range of tiles from animating. If AnimateTileRange has been called for the specified tileInfo and range, or if AnimateTile has been called for any of the tiles in the range, then the animation for the tile or tiles will be halted immediately. If any of the tiles in the range are currently not animating, then they will be ignored.

```
    // Stops tiles 20-29 in set 1 from animating
    Tile.StopAnimatingTileRange (new TileInfo(1, 20), 10);
```

UseTileEditorDefaults

```
static function UseTileEditorDefaults (useDefaults : boolean) : void
```

Normally SetTile and SetTileBlock will use 0 as the defaults for rotation and order-in-layer, or if useDefaults is false. If useDefaults is true, however, then the SetTile and SetTileBlock functions will use the per-tile defaults set in the TileEditor for the appropriate tile (see Tile Defaults in the Tile Editor: Tiles section of the SpriteTile documentation). So any defaults set for the collider, order-in-layer, and rotation are also set. For example, if set 3, tile 1 had defaults of 2 for order-in-layer, 90 for rotation, and Collider was checked, then this:

```
    Tile.UseTileEditorDefaults (true);
    Tile.SetTile (new Int2(5, 5), 3, 1);
```

is the equivalent of this:

```
    Tile.UseTileEditorDefaults (false);
    Tile.SetTile (new Int2(5, 5), 3, 1, true);
    Tile.SetOrder (new Int2(5, 5), 2);
    Tile.SetRotation (new Int2(5, 5), 90.0f);
```

UseTrueColor

```
static function **UseTrueColor** (**trueColor** : boolean) : void
```

By default, SpriteTile uses 4-bit per channel for tile colors, rather than the standard 8-bit per channel. This allows each tile to only use two bytes for color values instead of four. If more accurate color values are needed, then UseTrueColor can be set to use 8-bit per channel for colors at the expense of memory usage.

```
     // Make colors be 32-bit instead of 16-bit
     Tile.UseTrueColor (true);
```