Getting Started with Grids for Game Maker

v1.2.0

Online Documentation	http://gamelogic.co.za/gridsgm/
API documentation	http://www.gamelogic.co.za/documentation/gridsgm/
Support	support@gamelogic.co.za
Twitter	https://twitter.com/gamelogicza

Contents

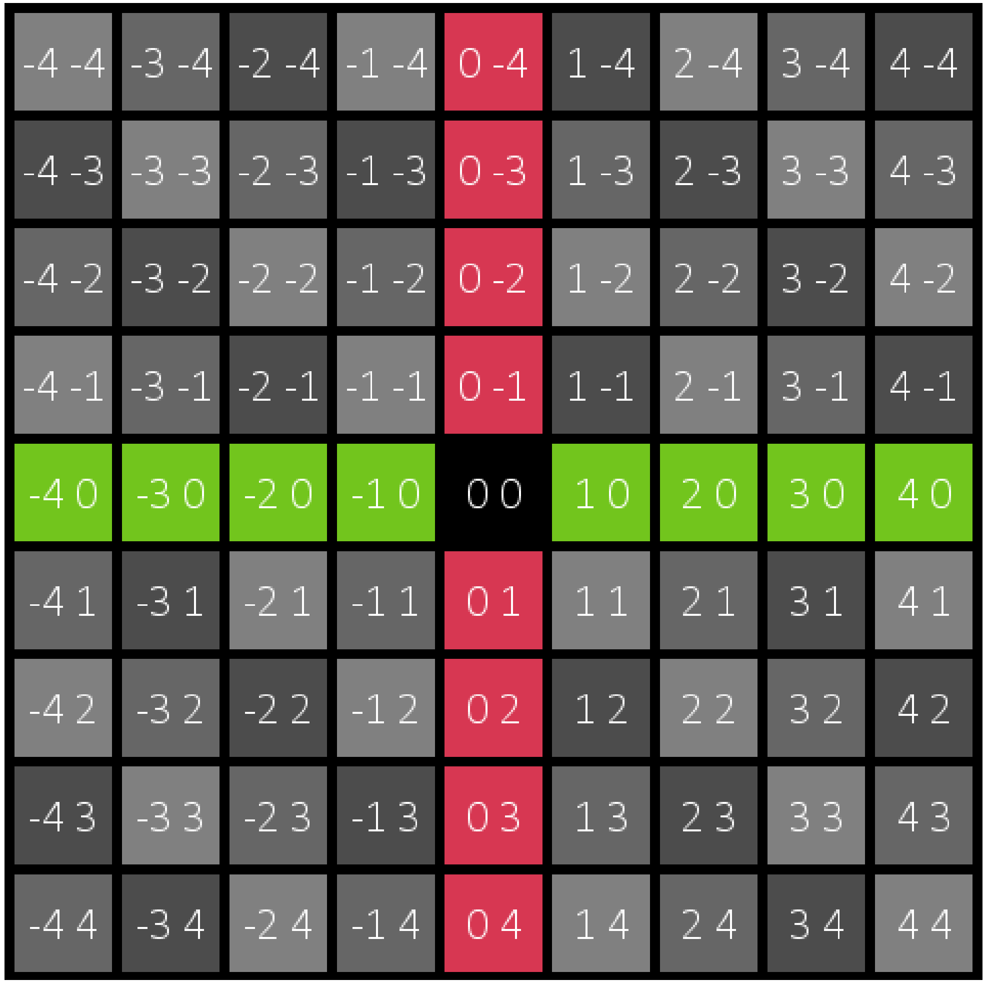
Overview of Important Concepts

Grids: These are data structures, a bit similar to Game Makers grids. They can "hold" data, and you can access the cells through two-dimensional coordinates. The most common use for grids like these are to hold Game Maker objects that make the tiles of your grid. But grids can also hold other data types, such as numbers, which is sometimes useful in algorithms.

Maps: Grids do not know *where* cells should be placed in the world. This is where maps come in; they map Game Maker coordinates with grid coordinates, and vice versa. To emphasize: if we have a grid point, the map can convert it to a corresponding Game Maker world point. If we create a grid of tiles (made from Game Maker objects), then the map can tell us where each tile should be placed. Similarly, if we have a position in Game Maker coordinates, the map tells us which grid point corresponds.

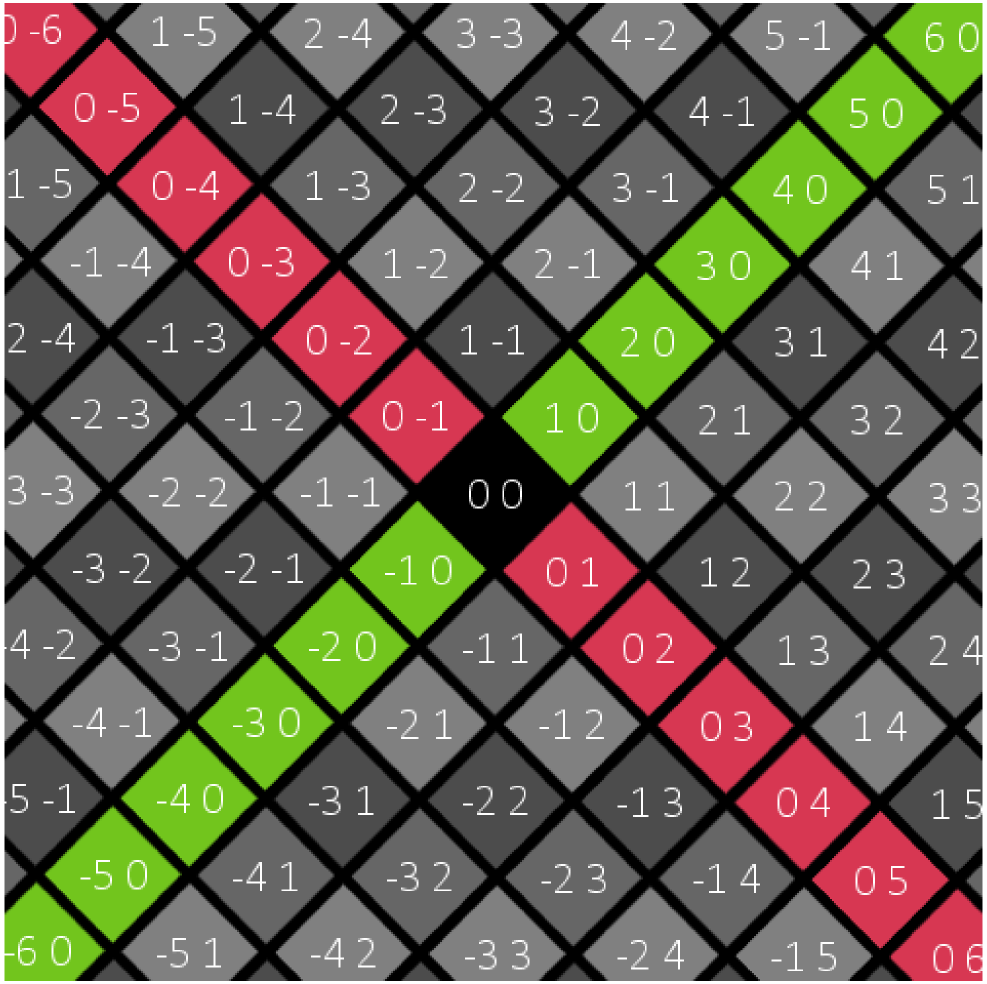
Vectors: As we said before, we use 2D coordinates to access grids. These coordinates are represented as an array with two components. When we use vectors to access grids, and the coordinates are always integers. We also use vectors to represent world coordinates.

Rect Coordinates



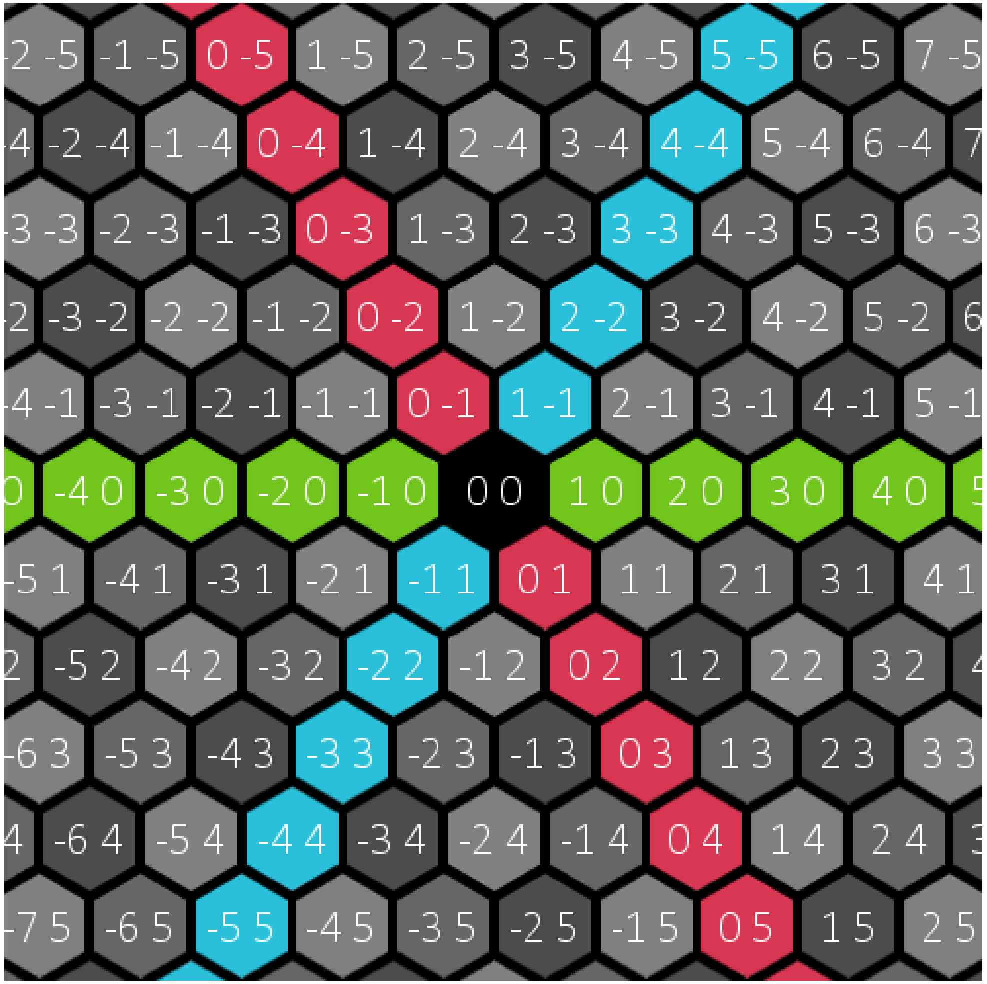
-4 -4	-3 -4	-2 -4	-1 -4	0 -4	1 -4	2 -4	3 -4	4 -4
-4 -3	-3 -3	-2 -3	-1 -3	0 -3	1 -3	2 -3	3 -3	4 -3
-4 -2	-3 -2	-2 -2	-1 -2	0 -2	1 -2	2 -2	3 -2	4 -2
-4 -1	-3 -1	-2 -1	-1 -1	0 -1	1 -1	2 -1	3 -1	4 -1
-4 0	-3 0	-2 0	-1 0	0 0	1 0	2 0	3 0	4 0
-4 1	-3 1	-2 1	-1 1	0 1	1 1	2 1	3 1	4 1
-4 2	-3 2	-2 2	-1 2	0 2	1 2	2 2	3 2	4 2
-4 3	-3 3	-2 3	-1 3	0 3	1 3	2 3	3 3	4 3
-4 4	-3 4	-2 4	-1 4	0 4	1 4	2 4	3 4	4 4

Diamond Coordinates

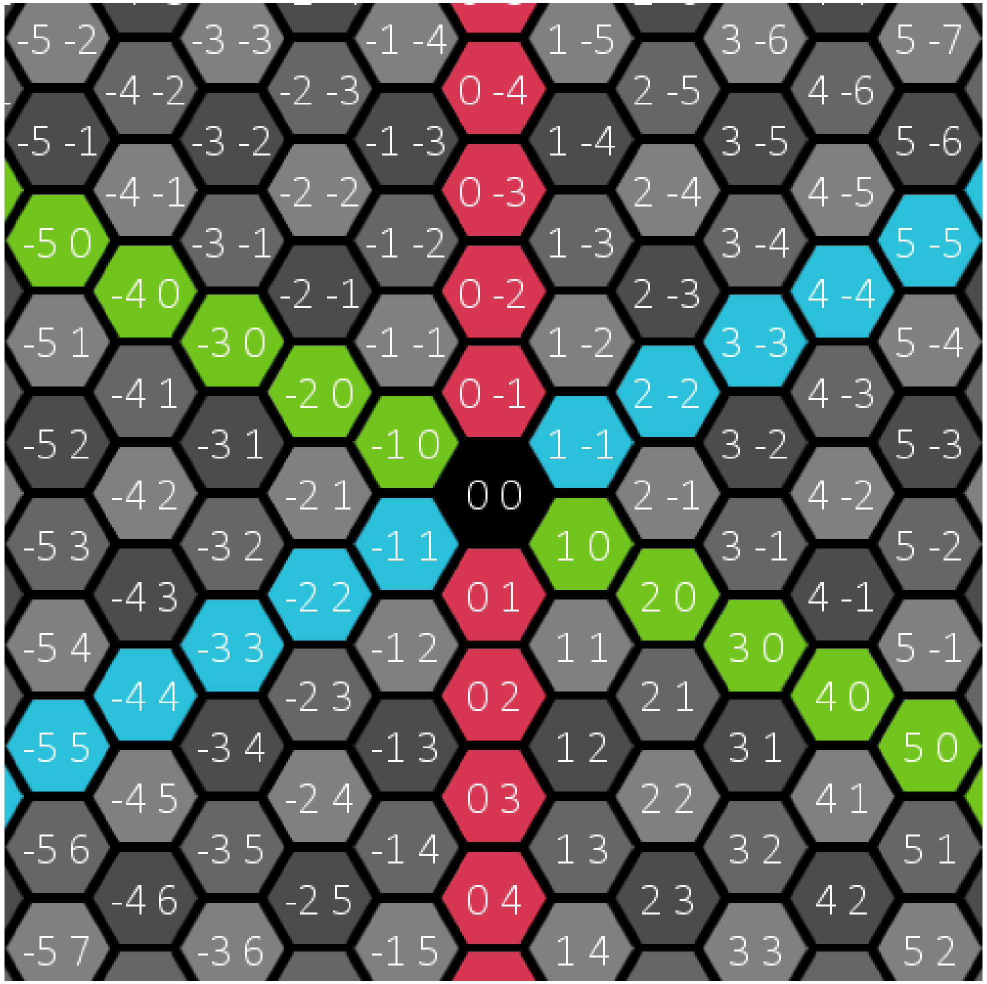


0 -6 1 -5 2 -4 3 -3 4 -2 5 -1 6 0
0 -5 1 -4 2 -3 3 -2 4 -1 5 0
1 -5 0 -4 1 -3 2 -2 3 -1 4 0 5 1
-1 -4 0 -3 1 -2 2 -1 3 0 4 1
2 -4 -1 -3 0 -2 1 -1 2 0 3 1 4 2
-2 -3 -1 -2 0 -1 1 0 2 1 3 2
3 -3 -2 -2 -1 -1 0 0 1 1 2 2 3 3
-3 -2 -2 -1 -1 0 0 1 1 2 2 3
4 -2 -3 -1 -2 0 -1 1 0 2 1 3 2 4
-4 -1 -3 0 -2 1 -1 2 0 3 1 4
5 -1 -4 0 -3 1 -2 2 -1 3 0 4 1 5
-5 0 -4 1 -3 2 -2 3 -1 4 0 5
-6 0 -5 1 -4 2 -3 3 -2 4 -1 5 0 6

Pointy Hex Coordinates



Flat Hex Coordinates



Initializing Grids

Grids uses a variety of constants, which are setup in using the gl_init function. These functions should also be useful for other purposes in your game, and sometimes must be used as parameters to functions such as the map alignment function. The library must be initialized like so:

```
gl_init();
```

It is only necessary to do this once per game, and must be called before using any of the library's constants or functions.

Building a Grid

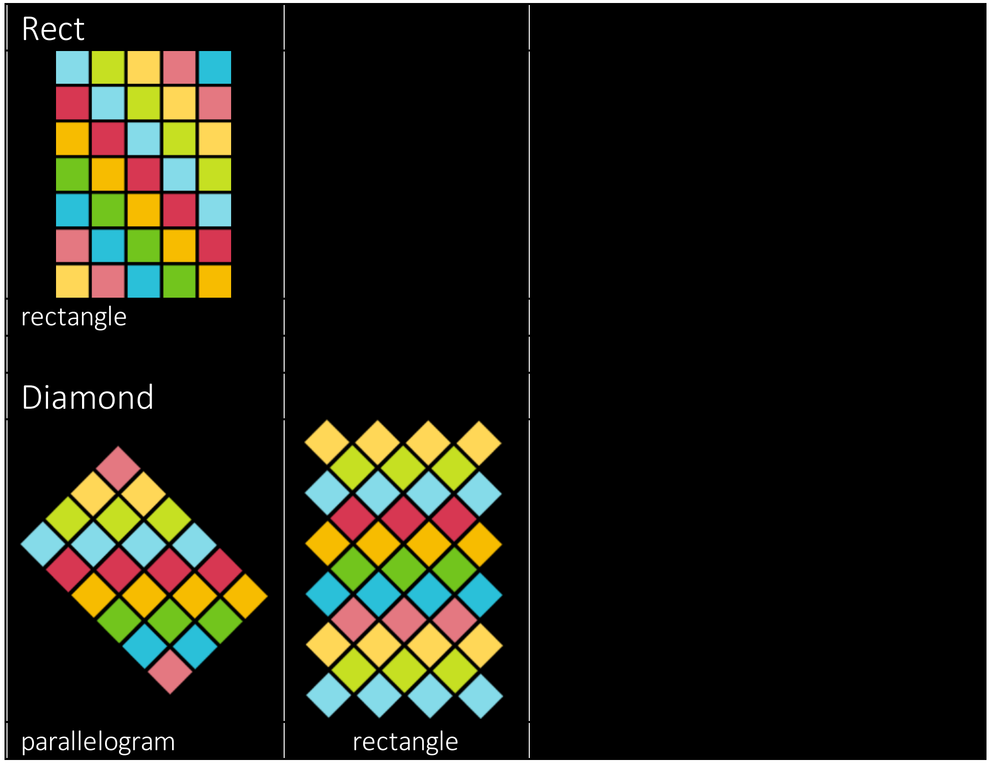
Grids support four grid types: rect grids, diamond grids, and two types of hex grids. The two types of hex grids correspond to the two most common orientations of the cells. We call them "pointy" and "flat".

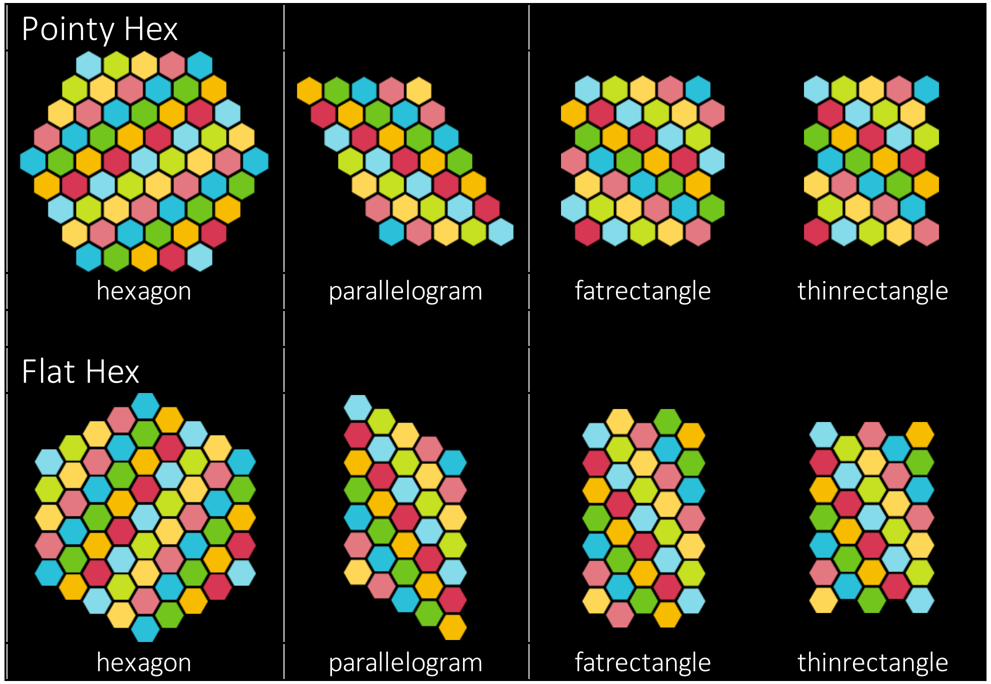
In this tutorial, we will go over setting up a pointy hex grid. The process is the same for other grids.

To create a grid, you use one of the following functions:

```
var grid = gl_pointyhex_grid_create(shape, width, height);
var grid = gl_flathex_grid_create(shape, width, height);
```

The first parameter denotes the grid shape, and must be a literal string of one those shown in the shapes table.





The next two parameters are the dimensions of the grid. In the case if the hexagon shape, only the first parameter is used, and it is the length of each side. Grids use memory, and as with other Game Maker data structures, you need to destroy a grid when you are done with it. To do this, you can use the following function:

```
gl_grid_destroy(grid);
```

Accessing Cells & Using Vectors

Once you have created the grid, you can set and get cells. You can use the following functions to do that:

```
gl_grid_set(grid, vector, value);
var value = gl_grid_get(grid, vector);
```

The first parameter is simply the object returned by the grid creation function. The second parameter is a vector. The value parameter is the value to store in cell, and can be a string, number, object or any Game Maker value.

A vector is a special array created by a function:

```
var vector = gl_vector_create(x, y);
```

The first parameter is the x-coordinate and the second parameter is the y-coordinate. When you use a vector as a grid point, the coordinates must always be integers.

You can access the *x* and *y* components of the resulting vector by calling the different indexes of the vector array:

```
var x_coordinate = vector[0];
var y_coordinate = vector[1];
```

To set a grid vector to a value, the full call would look like this:

```
gl_grid_set(grid, gl_vector_create(x, y), value);
```

This is cumbersome, especially if you want to set a value for every cell. A better way would be to *iterate* over all points in the grid. We provide a handy set of functions that allows you to process all the points of the grid, without you having to worry about the shape of the grid.

Here is how the iteration pattern looks:

```
var iter = gl_grid_get_iter(grid);

while(gl_iter_hasnext(iter))
{
  var vector = gl_iter_next(iter);
  //Do something useful with the vector
}

gl_iter_destroy(iter);
```

For instance, if you want to create new instances of an object to put in the grid, you could do it using that pattern:

```
var iter = gl_grid_get_iter(grid);

while(gl_iter_hasnext(iter))
{
  var vector = gl_iter_next(iter);
  var tile = instance_create(tile, 0, 0);
  gl_grid_set(grid, vector, tile); //put the tile in the grid.
}

gl_iter_destroy(iter);
```

The snippet of code above creates new instances, and put them all at the point 0, 0. To position the tiles correctly, we need to use a map.

Building a Map

You can create a map using one of the following two functions:

```
var map = gl_pointyhex_map_create(cell_dimensions);
var map = gl_flathex_map_create(cell_dimensions);
```

The single parameter is a vector that contains the dimensions (height and width) of a cell in pixels.

```
// Create a map where the cells are 94 pixels wide by 108 pixels high.
var cell_dimensions  = gl_vector_create(94, 108);
var map = gl_ pointyhex _map_create(cell_dimensions);
```

You need to destroy maps once you are done with them, using the following function:

```
gl_map_destroy(map);
```

The most important functions are the conversions from grid points to world points, and world points to grid points. Here they are:

```
var world_vector = gl_map_grid2world(map, grid_vector);
var grid_vector = gl_map_world2grid(map, world_vector);
```

Let us combine this with the snippets shown earlier to position our tiles.

```
var grid = gl_pointyhex_grid_create(“hexagon”, 3, 0);

var cell_dimensions = gl_vector_create(108, 94);
var map = gl_pointyhex_map_create(cell_dimensions);

var iter = gl_grid_get_iter(grid);

while(gl_iter_hasnext(iter))
{
  var point = gl_iter_next(iter);
  var world_point = gl_pointyhex_grid2world(point);

  var tile = gl_instance_create(tile, world_point);

  gl_grid_set(grid, point, tile);
}

gl_iter_destroy(iter);
gl_map_destroy(map);
gl_grid_destroy(grid);
```

We have replaced the function call `instance_create` with `gl_instance_create` as the latter uses a vector instead of separate coordinates.

Map behaviours

By default, maps map the grid point 0, 0 to the Game Maker world point 0, 0. But there are several functions available to change the map’s behaviour. To make the map align the grid in a rectangle, we can use the following function:

```
gl_map_align_grid_in_rect(map, grid, rect, alignment_type);
```

This function takes: the map, the grid to align, a rectangle in which to align the grid, and a constant denoting the type of alignment.

To create a rectangle, use the following:

```
var rect = gl_rectangle_create(x, y, width, height);
```

We can now modify our snippet of code to center the grid on screen:

```
gl_init();

var grid = gl_pointyhex_grid_create(“hexagon”, 3, 0);
var map = gl_pointyhex_map_create(cell_dimensions);
var screen_rect = gl_rect_create(0, 0, 800, 600);

gl_map_align_grid_in_rect(map, grid, rect, global.gl_alignment_middlecenter);
gl_rect_destroy(rect);

var iter = gl_grid_get_iter(grid);

while(gl_iter_hasnext(iter))
{
  var grid_vector = gl_iter_next(iter);
  var world_vector = gl_map_grid2world(grid_vector);
  var tile = gl_instance_create(tile, world_vector);

  gl_grid_set(grid, grid_vector, tile);
}

gl_iter_destroy(iter);
gl_map_destroy(map);
gl_grid_destroy(grid);
```

The script above is all that is necessary to build a basic grid. It can be added to a controller in our room, and called on the Create event.

Accessing Grids & Maps

It is often useful have access to the grid and map objects. To allow this, we make them instance variables, and don't destroy them in this script.

```
// Controller Create
gl_init();

grid = gl_pointyhex_grid_create(“hexagon”, 3, 0);
map = gl_pointyhex_map_create(cell_dimensions);

var screen_rect = gl_rect_create(0, 0, 800, 600);

gl_map_align_grid_in_rect(map, grid, rect, global.gl_alignment_middlecenter);
gl_rect_destroy(rect);

var iter = gl_grid_get_iter(grid);

while(gl_iter_hasnext(iter))
{
  var grid_vector = gl_iter_next(iter);
  var world_vector = gl_map_grid2world(vector);
  var tile = gl_instance_create(tile, world_vector);

  gl_grid_set(grid, grid_vector, tile);
}

gl_iter_destroy(iter);
```

We put destruction in a separate script that we can then call from the object's Destroy event.

```
// Controller Destroy
gl_map_destroy(map);
gl_grid_destroy(grid);
```

Handling Mouse Input

It is common to manipulate a cell that the player clicks on. You *could* implement click events for the cell objects as you normally do. However, in many cases it is better to handle mouse input globally, figure out the corresponding grid point, and access the grid to get hold of the cell. (One reason is that this frees objects in grids to have more interesting shapes, without affecting the clicking behavior. It also makes it easier to access multiple grids.)

The script to do this is very simple. Like the grid creation and destroy methods, we add it to the controller. Here is how it looks:

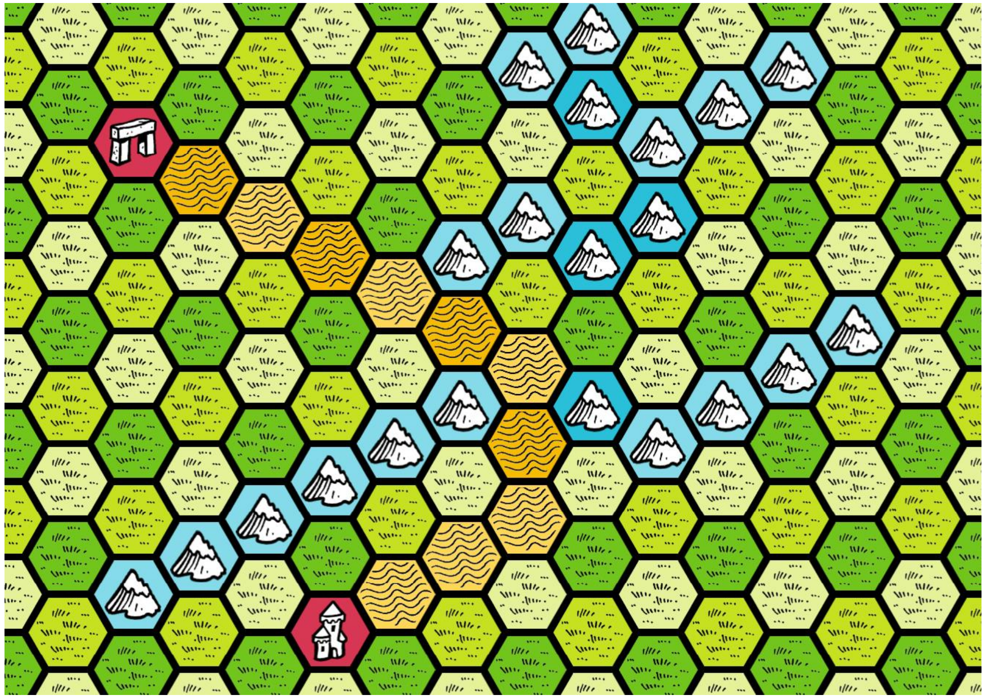
```
// On Click
var mouse_position = gl_mouse_get_position();
var grid_vector = gl_map_world2grid(map, mouse_position);
var clicked_cell = gl_grid_get(grid, grid_vector);

//Now do something cool with the clicked_cell
```

Using the A* Algorithm

To find a path between two nodes, you can use the following function:

```
var path = gl_grid_astar(grid, map, start_grid_vector, goal_grid_vector);
```

To use it, the cells in the grid must be Game Maker objects, and each cell must have the variable `accessible` set to `true` or `false`, depending on whether it can be reached or not. The algorithm returns a list of points from start to goal.



Typical Grid Game Architecture

To build a grid game, you will typically use the following:

A **controller object** that:

- initializes the Grids library
- creates the grid and map, and store them in variables in the controller
- initializes cells (as instances)
- process mouse input
- destroys the grid and map when the room is exited or the controller object is destroyed

One or more **cell objects** that serve as cells for the grid. It is often convenient to keep cell states in these objects, and update their presentation whenever the state changes. Other than that, cells are dumb and do not perform any other logic.

More help

We will add more topics to our web site; stay up to date by following us on Twitter: https://twitter.com/gamelogicza.

And if you get stuck, send as an email: support@gamelogic.co.za.