# Neo Impressive Title
## Map Making Guide
## (last revision 2/1/2022)

Inside the “Game” project folder, you will find a folder called “maps”. That is the folder that contains all the data for every map in the game. It’s structure looks like this:

```
maps/
  images/
    [all the heightmaps and textures for the maps]
  materials.json
  weather.json
  [all the JSON files for the maps]
```

Whenever the game needs to load a map, it will search the “maps” folder for a JSON file with the same name as the map. As the game loads a map, it will search the “images” folder within the “maps” folder for heightmaps, masks, and textures needed by the map that is being loaded.

To create a new map, start by placing its heightmap, masks, and textures into the “images” folder within the “maps” folder. Then create a new JSON file with the same name as the map. Each map’s JSON file must contain at least the following:

```
{
  “heightmap”: “MyHeightmap.png”,
  “size”: [5000, 300, 5000],
  “material”: “Terrain/MyMap”,
  “spawn_pos”: [2500, 500, 2500]
}
```

The “heightmap” property must be set to the name of a greyscale image file located in the “images” folder. It will determine the contour of the terrain. Each heightmap should have a width and height that are both powers of 2 plus 1 such as 513 x 513. In the heightmap image, black is the lowest elevation and white is the highest elevation.

The “size” property determines the width, maximum height, and depth of the map. The “material” property is the name of the material to use for shading the terrain. It must be the name of a material that exists in the “materials.json” file. I will talk more about the “materials.json” file later.

And last but not least, the “spawn_pos” property determines where in the map the player will spawn after entering via a portal. Gates ignore the map spawn position and provide their own.

With these 4 properties alone, you will get an empty map that just has the terrain. However, most maps have other objects such as portals and scenery. I will explain how to add those next:

```
{
    “heightmap”: “MyHeightmap.png”,
    “size”: [5000, 300, 5000],
    “material”: “Terrain/MyMap”,
    “spawn_pos”: [2500, 500, 2500],
    “portals”: [],
    “gates”: [],
    “water_planes”: [],
    “objects”: [],
    “particles”: [],
    “lights”: [],
    “billboards”: [],
    “walls”: [],
    “random_objects”: [],
    “collision_shapes”: [],
    “music”: []
}
```

scenery meshes), particles, lights, billboards, walls, random objects, collision shapes, and music. Each type of object has its own corresponding property this is a list containing objects of that type. If you do not need a certain category of objects, you can simply set its property to an empty list or omit it altogether.

Let’s start by taking a look at portals:

```
{
    “pos”: [0, 0, 0],
    “radius”: 25,
    “dest_map”: “Default”
}
```

The “pos” property determines where in the map the portal will be located, the “radius” property determines the size of the portal, and the “dest_map” property determines which map the portal leads to. Remember that you need to place each portal inside the square brackets of the “portals” property list and separate them with commas like this:

```
{
   “heightmap”: “MyHeightmap.png”,
   “size”: [5000, 300, 5000],
   “material”: “Terrain/MyMap”,
   “spawn_pos”: [2500, 500, 2500],
   “portals”: [
      {
         “pos”: [0, 0, 0],
         “radius”: 25,
         “dest_map”: “Default”
      },
      {
         “pos”: [80, 0, 500],
         “radius”: 25,
         “dest_map”: “Playground”
      }
   ],
   “gates”: [],
   “water_planes”: [],
   “objects”: [],
   “particles”: [],
   “lights”: [],
   “billboards”: [],
   “walls”: [],
   “random_objects”: [],
   “collision_shapes”: [],
   “music”: []
}
```

Next we will look at gates:

```
{
   “material”: “GateMatBlack”,
   “pos”: [0, 0, 0],
   “dest_map”: “Default”,
   “dest_vec”: [0, 0, 0]
}
```

The “material” property sets the material for the gate. It is usually “GateMatBlack” or “GateMatWhite”. The “pos” property determines where in the world the gate will be, the “dest_map” property determines where the gate will take the player, and the “dest_vec” property determines where the player will spawn in the destination map.

Now let’s take a look at water planes:

```
{
    “pos”: [0, 0, 0],
    “scale”: [100, 100],
    “material”: “Terrain/Water”,
    “sound”: “water1.wav”,
    “is_solid”: false
}
```

The “pos” property determines where in the world the water plane will be located, the “scale” property determines the width and height of the water plane, the “material” property determines the appearance of the water plane, the “sound” property determines the sound effect to play when near the water plane, and “is_solid” determines if the water is solid (ice) or liquid (water). “sound” and “is_solid” are optional.

Next we will look at scenery objects:

```
{
    “mesh”: “waterfall”,
    “pos”: [0, 0, 0],
    “scale”: [1, 1, 1],
    “rot”: [0, 0, 0],
    “sound”: “water1.wav”,
    “material”: “waterfall”
}
```

The “mesh” property determines what mesh will be used for the object, the “pos” property determines where in the world the object will be located, the “scale” property determines the size of the object, the “rot” property determines the rotation of the object, the “sound” property determines what sound effect should play when the player is near the object, and the “material” property determines the appearance of the object’s surface. “sound” and “material” are optional.

You can also specify a group of objects that use the same mesh and material like this:

```
{
   “mesh”: “acaciaTree”,
   “material”: “bark”,
   “instances”: [
      {
         “pos”: [0, 0, 0],
         “scale”: [1, 1, 1],
         “rot”: [0, 0, 0]
      },
      {
         “pos”: [80, 0, 300],
         “scale”: [1, 1, 1],
         “rot”: [0, 0, 0]
      }
   ]
}
```

In this form, we specify the “mesh” and optionally the “material” property. Then we have a property called “instances” which is a list of object instances. Each object instance has “pos”, “rot”, and “scale” properties.

Now we will look at particles:

```
{
   “name”: “WaterSplash”,
   “pos”: [0, 0, 0],
   “sound”: “water1.wav”
}
```

The “name” property is determines the name of the particle system to use. I will discuss particle systems later. The “pos” property determines where in the world the particles will appear and the “sound” property determines the sound effect that should play when the player is new the particles. “sound” is optional.

Next we will look at lights:

```
{
   “pos”: [0, 0, 0],
   “color”: [0, 0, 1]
}
```

The “pos” property determines where in the world the light is located and the “color” property determines the color of the light. Each color value is RGB where each component is between 0 and 1 inclusive.

Next we will look billboards:

```
{
   “pos”: [0, 0, 0],
   “scale”: [20, 10],
   “material”: “Sign”
}
```

The “pos” property determines where in the world the billboard will be located, the “scale” property determines the size of the billboard, and the “material” property determines the appearance of the billboard. Billboards always face the camera and therefore do not need a rotation property.

Next we will look at walls:

```
{
   “shape”: “sphere”,
   “pos”: [0, 0, 0],
   “radius”: 25,
   “is_inside”: false
}
```

The “shape” property determines the shape of the wall. It can be “sphere” or “box”. The “pos” property determines where in the world the wall is located, the “radius” property determines the size of the sphere, and “is_inside” determines whether the player’s movement should be limited to the inside or outside of the wall. If the shape is a box, the syntax changes to:

```
{
   “shape”: “box”,
   “pos”: [0, 0, 0],
   “extents”: [50, 50, 50],
   “is_inside”: false
}
```

The “extents” property determines the size of the box in this case.

Now Let’s look at random objects:

```
{
   “meshes”: [
      “mesh1”,
      “mesh2”,
      “mesh3”
   ],
   “instance_count”: -1
}
```

The “meshes” property defines a list of meshes to randomly choose from and the “instance_count” property determines how many instances to randomly generate. If the

instance count is -1, a random number of random instances will be generated. This is useful for generating random trees and other foliage.

Next we will take a look at collision shapes:

```
{
  “shape”: “box”,
  “pos”: [0, 0, 0],
  “size”: [50, 50, 50]
}
```

The “shape” property determines the shape of the collision volume. It can be either “box” or “sphere”. The “pos” property determines where in the world the collision shape will be located and the “size” property determines the size of the collision shape. If the shape is “sphere”, the syntax is:

```
{
  “shape”: “sphere”,
  “pos”: [0, 0, 0],
  “radius”: 25
}
```

In this case, the “radius” property determines the size of the collision shape.

Now we will look at music:

```
“song1”
```

Each entry in the music list is simply the name of an audio file (minus the extension). The game will randomly choose a song from this list to play and when it finishes playing one song, it will randomly choose another song from the list to play.

Now that we have covered the various map objects, let’s take a look at some optional map features that can be added directly to the map properties. The first is weather:

```
“weather”: “MyWeather”
```

Adding this to the map properties allows you to specify the weather cycle for a map. I will discuss the weather file later.

Next we will look at interior:

```
“interior”: {
  “color” [1, 0, 0],
  “height”: 300,
  “material”: “RockCeiling”
}
```

The “interior” property is used to define a ceiling for caves etc when present. The “color” property is optional and defines the color of the ceiling, the “height” property

determines the height at which the ceiling is, and the “material” property determines the appearance of the ceiling.

Next up is map effects:

```
“map_effect”: “Exhale”
```

The “map_effect” property turns on an optional map effect such as visible breath (aka “Exhale”). The possible map effects are game specific and defined in the game code.

Now let’s look at grass:

```
“grass”: {
  “material”: “GrassMat”,
  “grass_map”: “MyGrassMap.png”,
  “grass_color_map”: “MyGrassColorMap.png”
}
```

The “material” property determines the appearance of the grass patches, the “grass_map” determines which parts of the map are covered by grass and how dense the grass is, and the “grass_color_map” is used to adjust the grass color.

Next is the “freeze time” feature:

```
“freeze_time”: 0
```

This feature allows us to freeze the day-night cycle at a set time. And last but not least, we have critter spawns:

```
“critters”: {
  “limit”: 40,
  “roam_areas”: [],
  “critters”: []
}
```

When this property is present, it overrides the default critter spawns defined in the “spawn.json” file. The “limit” property determines the maximum number of critters allowed on the map at one time, the “roam_areas” property defines a list of areas that critters can spawn in, and the “critters” property defines a list of critters that can spawn on the map. The format for each roam area is:

```
{
  “start”: [0, 0, 0],
  “extents”: [50, 50, 50]
}
```

The “start” property determines the origin of the roam area and the “extents” property determines the size of the roam area. The format for each critter is:

```
{
   “type”: “penguin”,
   “rate”: .1,
   “roam_area”: -1
}
```

The “type” property determines the critter species. It must be one of the species defined in “critters.json”. The “rate” property determines how often the critter spawns and the “roam_area” property is an index into the roam areas list. If the roam area is -1, the critter can spawn anywhere on the map.

Now that I have finished describing the format of a map file, I will discuss the format of the “materials.json” file. This file contains a dictionary of every material needed by a map. Each material entry contains a list of texture units:

```
{
   “texture_units”: []
}
```

Each texture unit has this format:

```
{
   “texture”: “water1.png”,
   “scale”: [.1, .1],
   “scroll”: [.01, .01],
   “type”: “base”
}
```

The “texture” property determines what image will be used for the texture, the “scale” property determines the scaling to be applied to the texture, the “scroll” property determines the velocity at which the texture should scroll, and the “type” property determines the texture unit mode. The possible texture unit modes are “base”, “layer_mask”, “layer”, and “countour”. Base textures are simply blended directly with each other. A layer mask determines which parts of the following layer will be blended with the base texture. A layer defines a texture that will be used along with the preceding layer mask. A contour texture will be used to apply a contour pattern the base texture. “scale” and “scroll” are optional.

Next I will describe the format of a the “weather.json” file. The weather file contains a list of weather definitions and a list of weather cycles as shown below:

```
{
   “weathers”: {},
   “cycles”: {}
}
```

The “weathers” dictionary contains weather type definitions that define the properties of a weather system. Here is a sample weather type definition entry:

```
{
   “particle”: “rain”,
   “offset”: [0, 0, 0],
   “sound”: “rain.wav”
}
```

The “particle” property determines which particle system to use for the weather, the “offset” property determines the offset of the particle system, and the “sound” property determines which sound effect should be played while that particle weather is active.

The “cycles” dictionary contains weather cycle definitions that define the properties of a weather cycle. Here is a sample weather cycle definition entry:

```
[
   {
      “start”: 0,
      “end”: 1200,
      “sky”: {
         “shader”: [.5, .5, .5],
         “adder”: [0, 0, 0]
      },
      “weather”: “rain”,
      “rate”: 1500
   }
]
```

## Important Considerations

1. Godot does not support DDS format images, so they must be converted to another format such as PNG.
2. Godot does not support MP3 files, so they must be converted to OGG format.
3. Godot does not support Ogre meshes and materials. Install the Better COLLADA Exporter from https://github.com/godotengine/collada-exporter into Blender and export your meshes as COLLADA scenes. Then import them into Godot via the Import menu.
4. This version of Impressive Title does not directly support the old Ogre based game’s map format. Use the included map importer to convert your old maps to the new map format.