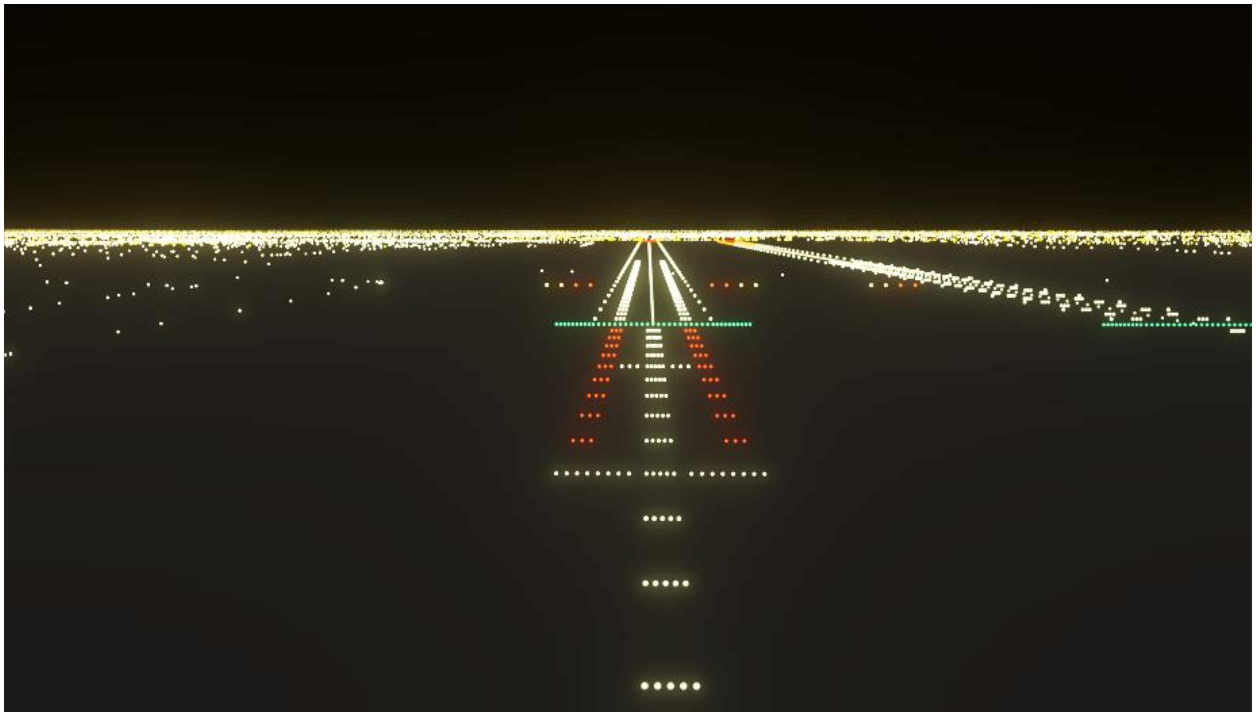
# Sprite Lights

Fast procedural lights



## Table of contents

## Introduction

The SpriteLights package consists of several shaders which can render up to 1.4 billion lights in one single batch. They are not regular Unity lights however, as it doesn't render light hitting a surface. Instead, it renders the light source itself.

The light changes brightness depending on the viewing angle, attenuates with distance, and is visible even if it is smaller than one pixel. This combination gives the lights a realistic appearance when viewed from a distance.

SpriteLights has the following features:

- Very fast. All light types run entirely on the GPU using pixel/vertex shaders.
- Up to 1.4 billion lights per mesh, rendered in only one batch!
- Physically Based.
- Lights punch through fog, just like in real life.
- Distance brightness attenuation.
- Several light radiation patterns (lens types) available.
- Distant lights visible even if they are smaller than one pixel.
- Lights can have random brightness for a more realistic look.
- The front and back of a light can each have a different color.
- Animated lights supported, running entirely in the shader.
- Sprites are translated upwards as they scale, so they do not intersect with the ground.
- Works in forward and deferred rendering.
- Does not require DX11 (but works with it).
- Works on mobile.
- Looks best with HDR and a 3rd party Physically Based bloom shader.
- Lights are not Unity lights and don't affect other objects.

## Installation

1. Import SpriteLights.unitypackage.

2. Go to Edit->Project Settings->Quality. Disable Anti Aliasing for each platform and quality setting. This is necessary to get HDR to work. Anti Aliasing can still be used, but it has to be implemented as a post process effect.

3. Go to Edit->Project Settings->Player->Other Settings. Set Color Space to Linear.

4. Enable HDR on the main camera.

5. Add a good quality bloom shader to the main camera. The example project contains the Unity post processing stack bloom shader.

6. Make sure tone mapping is enabled (part of color grading in the Unity post processing stack).

7. Run the example scene to check if everything works correctly. Camera control is the same as in the Scene View. Fly speed can be change by pressing 1 or 2.

## Usage

The light mesh uses a custom format so it cannot be created with a 3d modeling program. Instead, the lights have to be generated by code. This can be done either offline in the Editor, or generated at runtime.

SpriteLights.cs contains the relevant functions for creating a light mesh.

### Initializing

Before displaying any lights, call the Init() function. This function should also be called each time you change the global brightness offset, or if the camera Field Of View, or resolution changes.

```
SpriteLights.Init(strobeTimeStep, globalBrightnessOffset, FOV, screenHeight);
```

`float strobeTimeStep`
The strobe time step is only required if you use the strobeLight material, otherwise it can be set to 0. See chapter **Strobe Setup** for more information.

`float globalBrightnessOffset`
Increase or decrease the global brightness of all lights from all materials.

`float FOV`
The camera Field Of View. If you use VR, make sure you get the correct Field Of View and screen height as using the Unity API to get this, might give incorrect results.

`float screenHeight`
The camera vertical resolution.

### Creating lights

To create a light mesh, call the CreateLights() function. Note that the maximum amount of lights in a mesh is defined by the meshIndexFormat (21.844 for 16 bit, and 1.431.655.765 for 32 bit). Multiple meshes will be automatically created if needed.

There are two function overloads available for the CreateLights function. The function overload with the positions variable is for the simple shader only It uses less data and is faster to compute. The function overload with the lightData variable is for all other shaders.

```
lightObjects = CreateLights(name, lightData, material, meshIndexFormat)
lightObjects = CreateLights(name, positions, lightSize, material, meshIndexFormat)
```

The CreateLights() function needs the following input:

`string name`
The name which will be given to the light game object.

`Material material`
The light material. The light mesh uses a custom format, so only a material with one of the supplied shaders can be used. See chapter **Material Settings** for more information.

`UnityEngine.Rendering.IndexFormat meshIndexFormat`
The mesh index format (32 or 16 bit), determining how many lights can fit into a single mesh.

`Vector3[] positions`
The positions of the lights.

`float lightSize`
The size of the lights.

`LightData[] lightData`
This array contains the properties of each individual light.

```
public struct LightData{
	public Vector3 position;
	public Quaternion rotation;
	public float size;
	public float brightness;
	public Color frontColor;
	public Color backColor;
	public float strobeID;
	public float strobeGroupID;
}
```

`Vector3 position`
The position of the light, relative to the light game object.

`Quaternion rotation`
The orientation of the light, relative to the light game object. The light sprite always faces the camera, but the rotation variable is used to determine the brightness, which changes depending on the viewing angle.

`float size`

The size of the light. It is important that this value is initialized, otherwise the light will not be visible. Not applicable for city lights (simple shader).

`float brightness`
Set the brightness a light. A value of 1 is bright, and a value of 0 is off. Note that this variable is stored in the shader color channel and therefore is automatically clipped to a range of 0 to 1. It is important that this value is initialized, otherwise the light will not be visible.

`Color frontColor`
The front color of the light. Only applicable for directional and omnidirectional lights. The color for the other lights is set on the material. The alpha value is ignored. It is important that this value is initialized, otherwise the light will not be visible.

`Color backColor`
The back color of the light. Only applicable for directional and omnidirectional lights. The color for the other lights is set on the material. Ignored if an Equal radiation pattern is used. The alpha value is ignored. To make the back side invisible, set the color to Color.clear

`float strobeID`
Only applicable for strobe lights. This value is used to identify the light. See the chapter **Shaders - Strobe** for more information.

`float strobeGroupID`
Only applicable for strobe lights. This value is used to identify the group the strobe light belongs to. See the chapter **Strobe Setup** for more information.

The CreateLights() function has the following output:

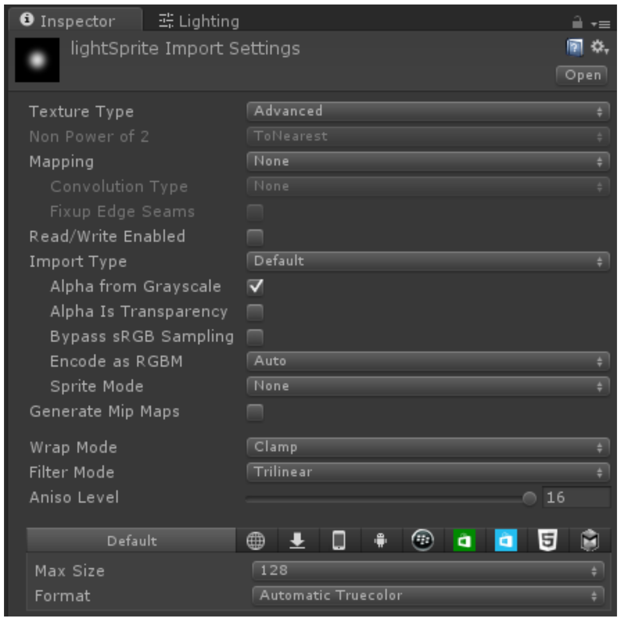
`GameObject[] lightObjects`
This array contains all light game objects which are created.

## Material Settings

Some materials share common shader properties. All shaders can be found on the Inspector shader dropdown list under Lights.

### Texture

The sprite texture. This has to be a small monochrome texture with the following import settings:



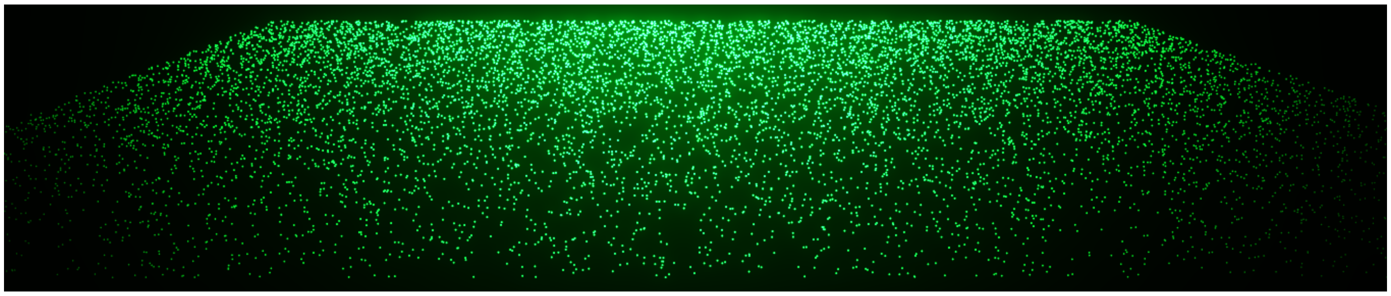
lightSprite Import Settings

| | |
|---|---|
| Texture Type | Advanced |
| Non Power of 2 | ToNearest |
| Mapping | None |
| Convolution Type | None |
| Fixup Edge Seams | ☐ |
| Read/Write Enabled | ☐ |
| Import Type | Default |
| Alpha from Grayscale | ✓ |
| Alpha Is Transparency | ☐ |
| Bypass sRGB Sampling | ☐ |
| Encode as RGBM | Auto |
| Sprite Mode | None |
| Generate Mip Maps | ☐ |
| Wrap Mode | Clamp |
| Filter Mode | Trilinear |
| Aniso Level | 16 |
| Max Size | 128 |
| Format | Automatic Truecolor |

If you want to use mip mapping, you have to generate your own mip maps. This is because the edges have to be full black, otherwise the light will have a triangle shape instead of the shape of the texture. Automatically generated mip maps do not have full black edges.

**Radiation pattern**

This determines how bright the light is depending on the viewing angle. It gives the following effect:



Lights which point straight at the camera are brighter then lights that don't. All directional lights have this behavior but the shape of the lobe depends on the type of lens fitted. These radiation patterns are available:

**Directional lights**
Teardrop
Round
Egg

**Omnidirectional lights**

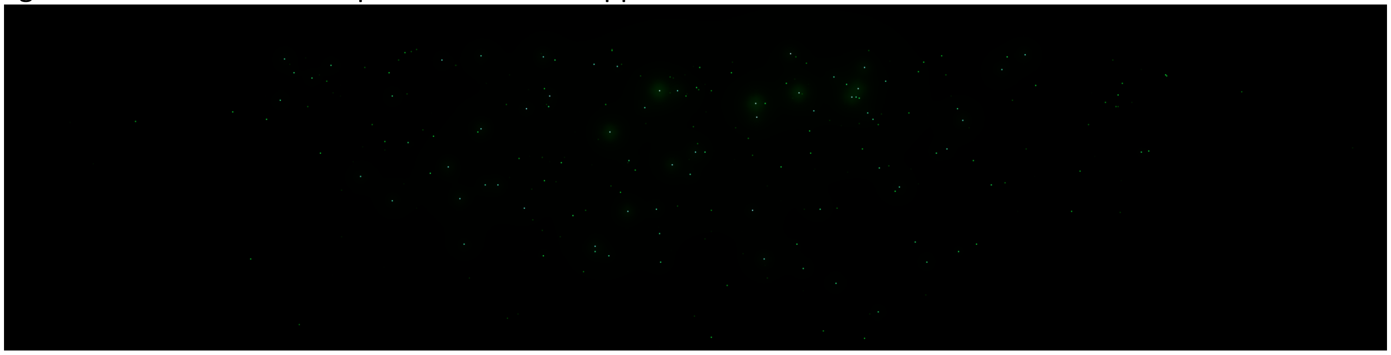
Doughnut
Equal

**OmniSimpe lights**
Equal

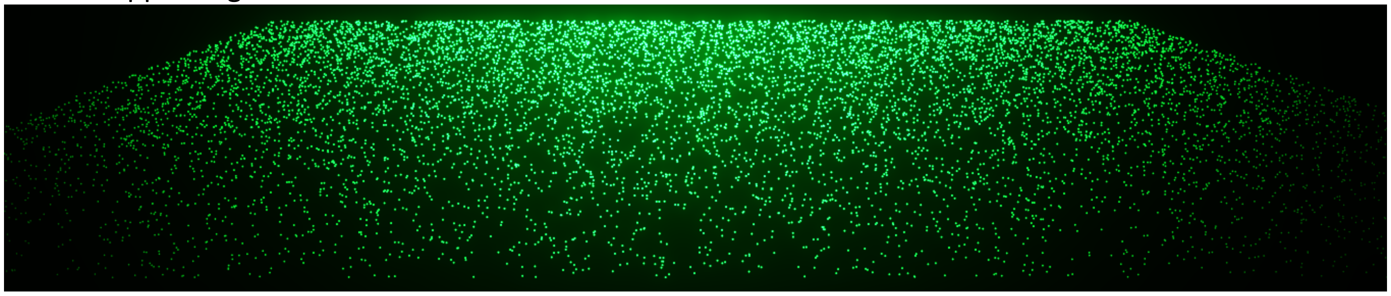
Lights with an equal radiation pattern have the same brightness, regardless of the viewing angle. This is the cheapest shader.

**Minimum screen size**

Scales up the lights if they are smaller than a screen space pixel. If lights are too far away and therefore too small, they will shimmer or disappear completely. In the screenshot below you can see lights which are far away and have no minimum screen space pixel size. Many of the lights are smaller than one pixel and have disappeared.



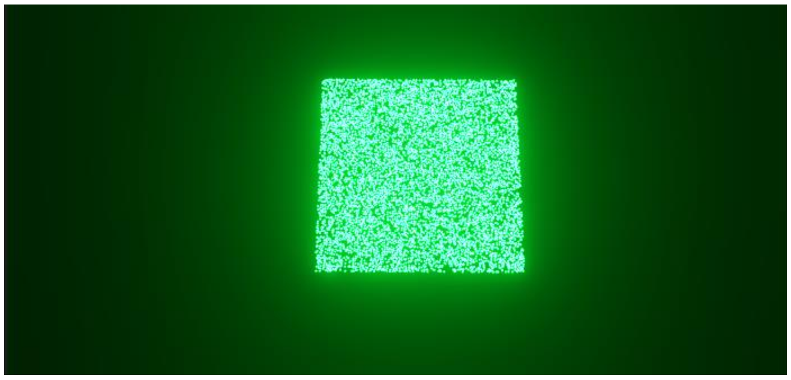
In the screenshot below, the lights have a minimum screen size of 5, which prevents the lights from disappearing.



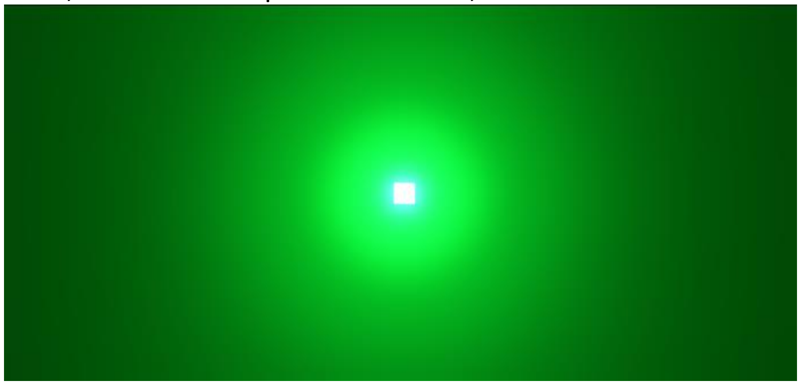
**Attenuation**

Far away lights are less bright than close up lights. This is simulated using the inverse square law. Increasing the attenuation factor will make the brightness falloff steeper as the light gets further away from the camera.

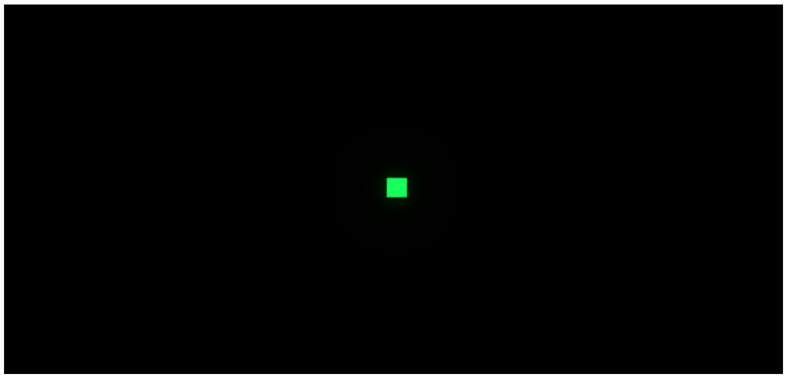
To demonstrate this effect, here is a square of 300x300 meter with 10.000 lights, viewed at a distance of 1 km.



Here, the camera is placed at 10 km, with no attenuation set. The lights are way too bright.



Here is the same scene with brightness attenuation applied, which looks much more realistic at this distance.



**Brightness offset**

Increase or decrease the brightness of the lights on a per-material basis.

**Brightness**

This is for is for PAPI, Strobe, and Debug lights only. It is used to set the brightness of the lights on a per-material basis. Note that the brightness is not affected by the brightness variable in the LightData class.

**Scale**

This is for is for PAPI, Strobe, and Debug lights only. It is used to set the scale of the lights on a per-material basis. Note that the size is also affected by the size variable in the LightData class, which is used to create the lights mesh.

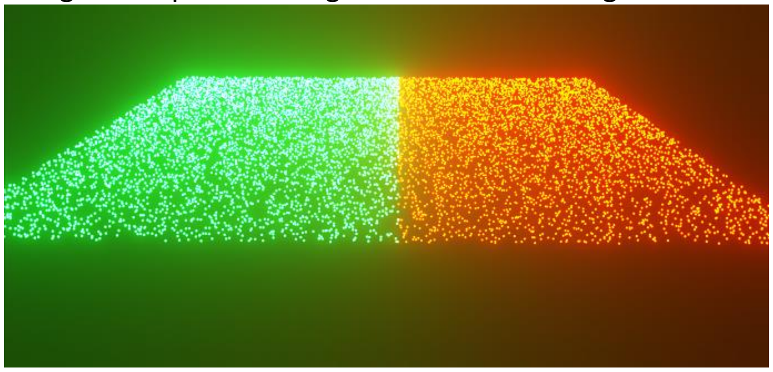
**Front Color**

This is for PAPI, Strobe, and Debug lights only. It is used to set the front color (or top color in case of a PAPI light) of the lights on a per-material basis. Note that the front color is not affected by the front color in the LightData struct.

**Back Color**

This is for is for PAPI and Debug lights only. It is used to set the back color (or bottom color in case of a PAPI light) of the lights on a per-material basis. Note that the back color is not affected by the back color in the LightData struct.

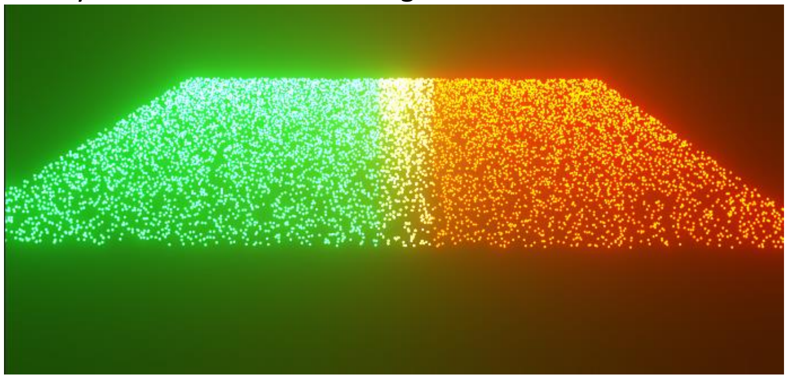
Here you can see the effect of lights with a different front and back color, viewed from the side. All lights are part of a single mesh. The coloring is done in the shader.



**Transition degrees**

This is for the debug shader only. It is used to change the amount of degrees where the front and back color will be blended. To change the transition degree variable in the other shaders, enable the appropriate GetDegreeTransition() function in the shader source.

Here you can see transition degrees set to 10:



**Translate up**

This is for the debug shader only. It is used to enable or disable the feature which translates the light upward as it is scaled up. This is used to prevent the light from intersecting with the ground as it scales up.

**Speed**

This is for the strobe lights only. It is used to change the speed at which the strobes flash.

**Persistence**

This is for strobe lights only. It is used to change the time how long the strobe should be switched on.

## Shaders

Only materials with one of the supplied shaders can be used for SpriteLights due to the custom mesh format used. The following shaders are available.

**Directional**

Directional lights can be single or double sided. The radiation pattern is directional, like with a flashlight.

Here is a real world example of a directional light with a different front and back color.

**Omnidirectional**

Omnidirectional lights can also be single or double sided, but the radiation pattern is doughnut shaped. Brightness is only reduced when the light is viewed from the top or bottom.

Lights with an equal radiation pattern have the same brightness, regardless of the viewing angle. This is the cheapest shader.

Here is a real world example of an omnidirectional light with a different front and back color.

**OmniSimple**

This shader only supports a single color, single brightness, omnidirectional lights. It is less flexible than the other shaders but it is faster to compute and uses a smaller mesh. It is most suited for city lights.

**PAPI**

PAPI lights are directional lights which emit two different colors, split vertically. These are specialized lights which are normally only found at airports.

Here is a real world example of a PAPI light.

**Strobe**

Strobe lights can be configured as walking strobe lights, or as a group of lights which flash at the same time. To get the strobes work correctly, a couple of variables must be set. See the chapter **Strobe Setup** for more information.

**Debug**

Debug lights should only be used for finding the optimum variables, as this shader is not optimized for speed. Use it to quickly tweak the shader variables to find the best configuration.

## Strobe Setup

To get the strobes work correctly, these variables must be set:

-strobeTimeStep, using the SpriteLights.Init() function.
-strobeID, in the lightData class.
-strobeGroupID, in the lightData class.

**strobeTimeStep**

The strobeTimeStep variable is used for the strobe flashing logic in the shader. It is set with the SpriteLights.Init() function. The strobeTimeStep variable must be calculated as follows:

strobeTimeStep = 1 / strobeAmountInRow

The strobeAmountInRow variable is the amount of strobes used in a walking strobe sequence. For example, if there are 22 strobes in a walking strobe sequence, strobeTimeStep must be:
1 / 22 = 0.0454545

All walking strobe light sequences must have the same amount of strobes each. If you want some walking strobes each to have a different amount of strobes, you must use a different material and set the strobeTimeStep for each material individually instead of globally. This can be done by calling these functions:

```
Renderer renderer = gameObject.GetComponent<Renderer>();
renderer.material.SetFloat("_StrobeTimeStep", strobeTimeStep);
```

If you use these functions, make sure you don't overwrite the strobeTimeStep variable in the shader with the SpriteLights.Init() function.

If all strobes should flash at the same time instead of execute in a sequence, the strobeTimeStep variable should still be set to a certain number, for example 20.

**strobeID**

This ID is used to identify each individual strobe light. It is set with the LightData class. The strobeID must be calculated as follows:

strobeID = ID * strobeTimeStep

The ID is the strobe number in the walking strobe sequence, starting at 0. For example, if there are 22 strobe lights, the ID's will be:

0, 1, 2, 3, (...), 21

If all the strobe lights should flash at the same time, set the StrobeID of all lights to 0.

**strobeGroupID**

This ID is used to identify each individual strobe group. It is set with the LightData class. This variable has a range of 0 to 1. It is recommended to give each strobe group a random number to give it a more natural appearance, like this:

```
strobeGroupID = Random.Range(0, 1);
```

## Examples

There are 3 example scripts included which show how to build a light mesh. All scripts are attached to a game object called SpriteLights in the example scene. The example scene can be found at Assets->SpriteLights->Scenes->SpriteLights.

SprightLights scene camera controls:

A,W,S,D = move camera.
Q,E = change altitude.

Right click + move mouse = rotate camera.
1,2 = change fly speed.

**Runway Lights**

This script creates runway lights by using a combination of SVG files and code to place the lights. If the script is enabled and all materials are correctly set up, it will automatically generate the lights at runtime.

**City Lights**

This script generates street lights from an OpenStreetMap file while in Play mode. It requires the Map-ity asset. The light mesh is saved in the project folder and can be re-used, so the lights only have to be created once. The example scene includes a city light mesh which has already been generated, so Mapity is not needed to run the example scene.

If you imported the Mapity Asset, follow these steps to enable offline generation of a light mesh using OpenStreetMap and Mapity.

1. Enable the code in CityLights.cs (this is disabled to prevent compile errors).

2. Add the Mapity prefab to the scene.

3. Download an OpenStreetMap data file, give it a file extension called *.mapity, and place it into the project folder at this location: Assets/StreamingAssets/MapData.
Do not give the file the name "map.mapity" because this file name is already in use by Mapity.

3. Go to Hierarchy->Mapity->Inspector->Mapity->Settings.
-Disable "Auto Load Data".
-Disable "Download Map Data".
-Set "Offline map filename" to the downloaded OpenStreetMap data file, but without the extension. So if the file is called "london.mapity", enter the name "london".

4. Press Play. Then go to Hierarchy->Mapity->Inspector->Mapity and click on "Build Map". The CityLights.cs script will execute after the map is loaded and will build a light mesh based on the street locations. A prefab and a mesh file is automatically created and placed in the Assets folder. The lights are also placed in the scene, but this will disappear when exiting Play mode. The generated prefab can be dragged into the Hierarchy in Edit mode and now it will not disappear anymore when exiting Play mode.

**Debug Lights**

This script generates a square of random lights in play mode when using the example scene. It can be used to interactively set light variables. It is more flexible than the other light shaders but not as fast.

## Graphics

### Fog

The lights are rendered in Forward mode, regardless whether Forward or Deferred rendering mode is used. Therefore the lights are only affected by Linear fog, which can be set at Window->Lighting->Scene->Fog.

If Deferred rendering is used, non-transparent objects are only affected by Global Fog (an image effect script that has to be added to the camera). So in this case, both Global Fog and Linear fog must be used. The variables of both fog modes must be matched up to create a consistent appearance between the lights and the rest of the world.

Lights in the real world are visible through fog at a greater distance than normal objects. SpriteLights also have this feature.

### Shimmer

If the lights are too small and the bloom set too high, the lights will appear to shimmer when the camera is in motion. To reduce this artifact, either increase the light size or reduce the amount of bloom.

If the light is smaller than one screen space pixel, it will shimmer as well (or completely disappear). To solve this, increase the "Minimum screen size" variable on the material.

### Colors

In order to create a realistic light with the center having a "hotter" color then the edge, never set an RGB channel to full bright (255) or full dim (0).

Here you can see lights which have a different color at the center, giving a realistic look.

There are several light presets available in Assets->SpriteLights->Editor->Lights, which can be used as a reference.

**Day/Night cycle**

Lights at daytime appear less bright than lights at night time. This can be simulated using a physically correct dynamic range, or by simply reducing the brightness of the lights at daytime. This feature is not included as it depends on the type of day/night cycle package used.

## Mobile

SpriteLights works on mobile but there are currently not many mobile devices which support HDR, so do not expect it to work with advanced bloom.

## Support

For bug reports and feature requests, contact bitbarrelmedia@gmail.com