Let there be Light – 2D Dynamic Soft Shadows for SFML

V 1.3

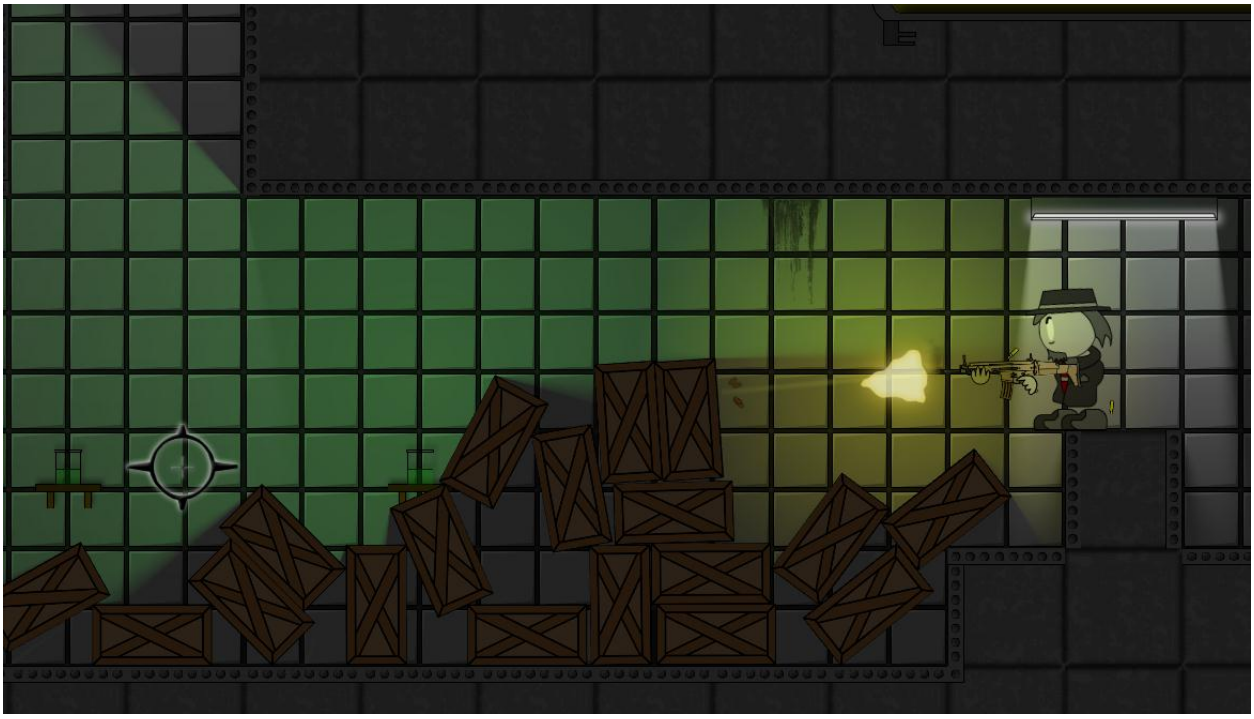

Image from the game Gore Factor

Eric Laukien

Table of Contents

A Walkthrough

The aim of this walkthrough is to explain how to recreate the included demo app.

In order to set up a project, you simply need to link to SFML and GLEW (which SFML actually uses internally), and include the source in your project (or link to it). You also need to put the contents of the data folder in your project directory, or change the file paths in the LightSystem class source.

Include the light system, and create a SFML render window like so (no Let There Be Light code yet):

```
#include "LightSystem.h"

#include <assert.h>

#include <SFML/Graphics.hpp>

int main(int argc, char* args[])
{
	sf::VideoMode vidMode;
	vidMode.Width = 800;
	vidMode.Height = 600;
	vidMode.BitsPerPixel = 32;
	assert(vidMode.IsValid());

	sf::RenderWindow win;
	win.Create(vidMode, "Let there be Light - Demo");

	// ---------------------- Background Image ---------------------

	sf::Texture backgroundImage;

	assert(backgroundImage.LoadFromFile("data/background.png"));

	// Tiling background
	backgroundImage.SetRepeated(true);

	sf::Sprite backgroundSprite(backgroundImage);
	backgroundSprite.SetTextureRect(sf::IntRect(0, 0, vidMode.Width, vidMode.Height));
```

This portion just creates a window and a sprite that will be our background image in the demo app.

Here is where it starts to get interesting:

```
// --------------------- Light System Setup ---------------------

	ltbl::LightSystem ls(AABB(Vec2f(0.0f, 0.0f),
Vec2f(static_cast<float>(vidMode.Width), static_cast<float>(vidMode.Height))), &win);

	// Create a light
	ltbl::Light* testLight = new ltbl::Light();
	testLight->center = Vec2f(200.0f, 200.0f);
	testLight->radius = 500.0f;
	testLight->size = 30.0f;
```

```
testLight->spreadAngle = 2.0f * static_cast<float>(M_PI);
testLight->softSpreadAngle = 0.0f;
testLight->CalculateAABB();

ls.AddLight(testLight);

// Create a hull by loading it from a file
ltbl::ConvexHull* testHull = new ltbl::ConvexHull();

if(!testHull->LoadShape("data/testShape.txt"))
	abort();

// Pre-calculate certain aspects
testHull->CalculateNormals();
testHull->GenerateAABB();

testHull->SetWorldCenter(Vec2f(300.0f, 300.0f));

ls.AddConvexHull(testHull);
```

Now, we will go over the light system code step by step.

```
ltbl::LightSystem ls(qdt::AABB(Vec2f(0.0f, 0.0f),
Vec2f(static_cast<float>(vidMode.Width), static_cast<float>(vidMode.Height))),
&win);
```

First, we are creating the light system object `ls`, whose constructor requires the region (as an **AABB – axis aligned bounding box**) in which the lights will exist. In this case, the lights are only in the window, so we size it exactly to the window. The lights will still render outside of the region, but it will not receive the benefits of the spatial partitioning. Resizing it currently is a very slow operation since it regenerates the entire tree. A self-expanding quad tree is currently still in development, there are still a lot of bugs to iron out. However, a self-expanding quad tree is not required if the map has a finite size (so, everything that isn't procedurally generated works). The second parameter is simply a reference to the render window.

```
// Create a light
ltbl::Light* testLight = new ltbl::Light();
testLight->center = Vec2f(200.0f, 200.0f);
testLight->radius = 500.0f;
testLight->size = 30.0f;
testLight->spreadAngle = 2.0f * static_cast<float>(M_PI);
testLight->softSpreadAngle = 0.0f;
testLight->CalculateAABB();

ls.AddLight(testLight);
```

Next, we add a light to the scene. These **MUST** be allocated on the heap. You register it with the light system (with `ls.AddLight(Light)`), and **it will take care of deleting it for you.** So, only `new` it, and then pass it to the light system. There are several parameters you can change with the light; in this case we are initializing the light center point, the radius of the light, and the light size. Radius and size is not the same thing, the former is the distance at which the light fades out and the latter is the physical radius of the light source (a circle). The spread angle parameter defines what the angle the light cone should be

(for directional lights). We set it to 2 * PI, so it goes all the way around (which is actually the default). We therefore also set the soft spread angle to 0 in order to have it render the light in a continuous fashion. If it were a cone, this parameter would determine the cone edge softness as an angle of the soft portion. But, since it is not a cone, we set it to 0 so that we don't get a weird looking soft wedge in our light.

After we configured all of our light settings, we have to calculate the AABB for the light. If possible, do this before you add it to the light system, since if you change certain parameters later on, you have to not only recalculate the AABB but also update the quad tree status. Whenever the AABB of something is changed somehow in a way that effects the physical dimensions or position of the light (e.g. by moving the light, resizing it, or rotating a light cone), you must update its tree status. This will be shown in the game loop in a bit, since we want to move the light around. The Light class contains several functions (setters) **that will perform the updates for you** (e.g. `SetCenter`, `SetRadius`, etc.), so you do not have to worry about it. However, since we are setting up the light initially, we do not want to recalculate everything every time a single parameter is set even though we are not rendering it yet.

```
// Create a hull by loading it from a file
ltbl::ConvexHull* testHull = new ltbl::ConvexHull();

if(!testHull->LoadShape("data/testShape.txt"))
	abort();

// Pre-calculate certain aspects
testHull->CalculateNormals();
testHull->GenerateAABB();

testHull->SetWorldCenter(Vec2f(300.0f, 300.0f));

ls.AddConvexHull(testHull);
```

After adding the light to the light system, we want something that will cast shadows. So, we have to create a convex hull. As the name suggests, individual hulls **cannot be concave**. **However**, you can still have concave objects cast shadows by making them out of multiple convex shapes. If your game has a lot of concave shapes with a lot of vertices, you may want to look into a concave shape decomposition algorithm to generate the convex shapes for you.

As with the lights, hulls **MUST** be allocated on the heap and added to the light system (using `AddConvexHull`), which will destroy the hulls for you. After creating the hull, you can either manually specify hull vertices by accessing the vertex array in the hull or by loading it from a file, as we do here. The file must contain the coordinates of the vertices, which must be specified in a **counter clockwise order.** This holds true for when you add coordinates to the vertex array manually, too. When that is done, you have to calculate the normals for the shape with `CalculateNormals`. You have to do this every time the vertices are changed somehow. Finally, you have to generate and AABB for it, with `GenerateAABB`, so that the quad tree can properly sort it in space.

After setting up the system, we get to the game loop:

```
	// ------------------------- Game Loop -------------------------

	sf::Event eventStructure;

	bool quit = false;

	while(!quit)
	{
		while(win.PollEvent(eventStructure))
			if(eventStructure.Type == sf::Event::Closed)
			{
				quit = true;
				break;
			}

		sf::Vector2i mousePos = sf::Mouse::GetPosition(win);

		// Update light
		testLight->SetCenter(Vec2f(static_cast<float>(mousePos.x),
static_cast<float>(vidMode.Height - mousePos.y)));

		win.Clear();

		// Draw the background
		win.Draw(backgroundSprite);

		// Calculate the lights
		ls.RenderLights();

		// Draw the lights
		ls.RenderLightTexture(0.0f);

		win.Display();
	}

	win.Close();
}
```

In the game loop, we move the light to follow the mouse like so:

```
sf::Vector2i mousePos = sf::Mouse::GetPosition(win);

		// Update light
		testLight->SetCenter(Vec2f(static_cast<float>(mousePos.x),
static_cast<float>(vidMode.Height - mousePos.y)));
```

The light is aligned by moving its center point. Had we not used the `SetCenter` function, we would have to update the AABB and tree status manually. Only avoid the use of these when you are first setting up the light, and configure multiple parameters at once. The function

```
		// Calculate the lights
		ls.RenderLights();
```

does all the light **processing**. However, it does not actually render it yet. The function call

```
// Draw the lights
ls.RenderLightTexture();
```

takes care of that. The parameter you see is the depth at which to render, which will **always be zero** if you are not messing with OpenGL directly and simply using SFML.

And that's it!

Setup

Let there be Light contains only a few core classes that you have to worry about:

- Light – A circular light source. Also a base class for other light types.
- ConvexHull – Something that can obstruct the light and cast shadows.
- EmissiveLight – A light that doesn't give off shadows.
- LightSystem – Class with which all lights (emissive and non-emissive) as well as all convex hulls must be registered in order to render them.

Light_Point

Parameters

The light class allows you to create a round light. If desired, you can also have it only go in one direction, to form a sort of cone. This can be used for flash lights or something. The parameters that affect the way the light looks are as follows:

```
float radius;
Vec2f center;
float directionAngle;
float spreadAngle;

float intensity;

float size;

float softSpreadAngle;

Color3f color;

float bleed;
float linearizeFactor;
```

The parameters describe the following:

- Radius – how far the light travels before falling off
- Center – the center point of the light source
- Direction angle – for use with direction lights, defines what angle the light is pointing
- Spread angle – used to make the light direction. Defines the cone angle of a direction light
- Intensity – how bright the light is. **Set this to 0 to keep the light from rendering entirely.**
- Size – the radius of the circular light emitter (physical light source size). Making this larger will make lights softer
- Soft Spread Angle – the angle with which to soften the edges of the cone of a directional light
- Color – color of the light source, in RGB, where the components are in the range [0, 1]
- Bleed – How large the light halo is
- Linearization factor – How linear the light attenuation is

Modifying the Parameters

In order to change these parameters, you can either change them and manually by generating a new AABB (using `CalculateAABB`) and updating the quad tree status, or you can use the setters, which do that stuff for you:

```
void SetRadius(float Radius);
void IncRadius(float increment);

void SetCenter(Vec2f Center);
void IncCenter(Vec2f increment);

void SetDirectionAngle(float DirectionAngle);
void IncDirectionAngle(float increment);

void SetSpreadAngle(float SpreadAngle);
void IncSpreadAngle(float increment);
```

I recommend using the former method for the initial set up of the light, and the latter for when you are modifying it for everything else.

Static Lights

In order to make you lights **static** (cached on a texture), you simply have to call `SetAlwaysUpdate(false)` (true would make it not static). **You must call this after you register the light with the light system.** This can greatly improve performance. **However**, you cannot do anything to the light once set static besides **moving it around**, since other paramaters can affect the dimensions of the light (which would require a different size cache).

New Light Types

You can overload the following functions in the Light class (from which Light_Point inherits)

```
virtual void RenderLightSolidPortion(float depth);
virtual void RenderLightSoftPortion(float depth);
virtual void CalculateAABB();
```

in order to create new light types. You can tell the light system to not use the attenuation shader by changing `bool shaderAttenuation` to false. One such type, included with this distribution, is the light beam. All you change is how the lights themselves are drawn. Look at the source for the light and the light beam as a guide for writing your own new light types. For most applications, you will probably not need new light types.

ConvexHull

The convex hull is a shape that blocks light. Convex means that each inner angle of the polygon the shape describes is <= 180 degrees. If you want concave shapes, simply put multiple convex shapes together.

Creating the Hull

You have two options for creating a convex hull: You can either load them from a file that has the coordinates of each vertex, or you can add them to the `vertices` array manually. The vertices are relative to the center of the hull, so it is typically convenient to make the point 0,0 around the center of the shape. In both cases, you must have the vertices specified in a counter-clockwise order. After the vertices are loaded, you **MUST** call:

```
pHull->CalculateNormals();
pHull->GenerateAABB();
```

Modifying the Hull

For any modifications to the vertices, you must call the two previously given functions in order to update the hull. However, you can move the hull with the following two functions (there is also an associated getter):

```
void SetWorldCenter(const Vec2f &newCenter);
void IncWorldCenter(const Vec2f &increment);
```

This does not modify the vertices, so no update is required. You do not have to recalculate normals or the AABB when doing this (the AABB is updated internally).

Helper Functions

If desired, you can determine if a point intersects the hull using: `PointInsideHull`
You can also get a world vertex using `GetWorldVertex.`

Emissive Lights

Emissive lights do not emit shadows. They simply brighten up an area. Therefore, they run **faster** than normal lights, and since the region the light up is defined by a texture, you can form them into all sorts

of shapes. Emissive lights are good for things such as glowing computer screens (in game, I mean) and buttons. They can also be used for a sort of **fake HDR effect.**

Creating an Emissive Light

You **MUST**, as with hulls and lights, allocate them on the heap, and register them with the light system. The light system will take care of deleting it for you.

You specify the texture of the emissive light using the `SetTexture(sf::Texture* texture)` function. The emissive lights use the **alpha layers** of the texture to determine how bright something is. So the color does not matter, making it easier to edit the emissive light shapes in an image editing program.

Emissive Light Parameters

The emissive light has only 1 parameter, and that is the color.

Transforming the Emissive Light

To transform the emissive light (position, scale, rotation) simply use the following setters:

```
void SetCenter(const Vec2f &newCenter);
void SetScale(const Vec2f &newScale);
void IncCenter(const Vec2f &increment);
void SetRotation(float angle);
void IncRotation(float increment);
```

This automatically updates their AABB's.

Light System

This is the main rendering class with which all lights, hulls, and emissive lights must be registered.

Creating the Light System

The constructor requires an AABB for the area which your light system will cover. If you make this too small, you will negate the benefits of the spatial partitioning system (a quad tree). It will still work; it will just not run quite as fast. Making the light system match the dimensions of the game map is probably a good choice.

Registering Lights, Emissive Lights, and Hulls

The objects added to the light system **MUST** be allocated on the heap. The light system will take care of deleting them. Add objects using the following functions:

```
void AddLight(Light* newLight);
void AddConvexHull(ConvexHull* newConvexHull);
void AddEmissiveLight(EmissiveLight* newEmissiveLight);
```

Remove them with:

```
void RemoveLight(Light* pLight);
void RemoveConvexHull(ConvexHull* pHull);
void RemoveEmissiveLight(EmissiveLight* pEmissiveLight);
```

This **deletes the object** in the process.

Parameters

The light system has very few parameters. Here they are:

```
AABB view;
sf::Color ambientColor;
bool checkForHullIntersect;

void SetView(const sf::View &view);
```

You can set the camera with the AABB, or use the `SetView` function to create one from an sf::View.

The view should be synced with the one you use for the rest of your game. Since the view defines the camera, you need to move it in order to have the lights move with the main view. So, you need to update it (either directly, or with `SetView`) every frame.

The ambient color is simply the ambient light color.

The `checkForHullIntersect` setting indicates whether the light system will check to see if a light intersects a hull (thereby not rendering it). Runs slightly slower when this is enabled (which is the default).

Pre-building Static Lights

You can pre-cache static lights by adding them to the build list using the function

```
void BuildLight(Light* pLight)
```

This allows you to pre-render the lights. You must still register the light normally.

Deleting Everything

You can delete all lights, emissive lights, and hulls using

```
void ClearLights();

void ClearConvexHulls();

void ClearEmissiveLights();
```

Debugging

The light system also provides a function for drawing all internal quad trees and AABB's. This can be useful for seeing if you position the quad tree / AABB's properly, and if the tree is partitioning properly.

Render the quad trees (there are 3, one for each light type) using:

```
void DebugRender();
```
