Approximating Real-Time 2D Illumination with Cast Shadows and Rim Lighting

A Technical Overview of 2D Lighting and Cast Shadows in GameMaker

Vincent Hendriks
Software Engineer

Documented for Game Developers and Shader Enthusiasts

January 16, 2025

Abstract

This document presents a practical approach to implementing real-time 2D lighting in games, featuring dynamic point lights, cast shadows, and rim lighting. The system emphasizes simplicity and efficiency, making it adaptable to a variety of project types. Key components include a mathematical framework for light attenuation and shadow casting, as well as a detailed implementation in GameMaker. This work aims to provide developers with a replicable solution for enhancing visual depth and realism in 2D game environments.

Figure 1: Overview of the 2D lighting system showcasing dynamic lighting and shadow casting.

Contents

1 Introduction **3**

2 Mathematical Framework **4**
2.1 Point Light Attenuation 4
2.2 Shadow Casting 4
2.3 Rim Lighting 5

3 Implementation in GameMaker **6**
3.1 Lighting Logic 6
3.2 Shadow Casting 6
3.3 Shader Logic 7

4 Results and Visuals **11**
4.1 Dynamic Point Lights 11
4.2 Shadows and Light Interaction 11
4.3 Rim Lighting 12

5 Limitations **13**
5.1 Performance Limitations 13
5.2 Visual Limitations 13
5.3 Scalability Limitations 13
5.4 Engine-Specific Implementation 13

6 Future Work **14**
6.1 Multi-Light Source Support 14
6.2 Enhanced Shadow Casting 14
6.3 Streamlined Animation Workflow for Normal Maps 14
6.4 Performance Optimization 14
6.5 Portability to Other Engines 14

7 Conclusion **15**

1 Introduction

Lighting is a key element in game design, creating atmosphere and enhancing the player's experience. In 2D games, dynamic lighting adds depth and realism but can be challenging to implement, especially when balancing simplicity, performance, and visual quality.

While working on my own projects, I needed a 2D lighting system that could handle dynamic point lights, cast shadows, and highlight actors effectively. Existing solutions were either too complex or did not meet my specific needs, so I developed an approximation system. This system offers a simple and accessible approach to dynamic lighting for 2D games, tailored to smaller-scale projects.

This document outlines the core features of the system, including dynamic point lights, cast shadows, and rim lighting, along with the underlying mathematical framework and its implementation in GameMaker. The aim is to offer developers a practical and adaptable solution that is straightforward to understand and replicate.

2 Mathematical Framework

This section outlines the core mathematical principles that form the foundation of the 2D lighting system. These principles are designed to be simple yet effective, ensuring that the system can be implemented in various game engines.

2.1 Point Light Attenuation

Point light attenuation determines how the intensity of light decreases with distance from the light source. Based on the linear falloff in the implementation, the intensity is modeled as:

$$I(x, y) = L \cdot \max\left(0, 1 - \frac{d(x, y)}{\text{radius}}\right)$$

Where:

- $I(x, y)$: Intensity of light at a point (x, y),
- L: Strength of the light source,
- radius: Maximum distance the light affects,
- $d(x, y)$: Distance from the light source to the point, calculated as:

  $$d(x, y) = \sqrt{(x - x_L)^2 + (y - y_L)^2}$$

  where (x_L, y_L) is the position of the light source.

2.2 Shadow Casting

Shadows are calculated based on the direction and length of the shadow relative to the light source. The shadow direction is given by:

$$\theta = \text{atan2}(y_O - y_L, x_O - x_L)$$

Where:

- (x_O, y_O): Position of the object casting the shadow,
- (x_L, y_L): Position of the light source,
- θ: Angle of the shadow direction.

The shadow length is computed as:

$$\text{Length} = \text{clamp}\left(\frac{C}{d(x_O, y_O) + 1}, \text{min\_length}, \text{max\_length}\right)$$

Where:

- C: A constant defining the maximum shadow length,
- $d(x_O, y_O)$: Distance from the object to the light source,
- $\text{min\_length}, \text{max\_length}$: Bounds on the shadow length.

2.3 Rim Lighting

Rim lighting enhances the edges of objects that are illuminated by a nearby light source. The intensity of the rim lighting is proportional to the cosine of the angle between the light direction and the object's surface normal, calculated using the dot product:

$$R = \max(0, \vec{N} \cdot \vec{L})$$

Where:

- R: Rim lighting intensity,
- $\vec{N}$: Normalized surface normal vector,
- $\vec{L}$: Normalized light direction vector.

This approach avoids the computational overhead of explicitly calculating $\phi = \arccos$, instead using the dot product directly to determine the cosine of the angle.

—

This mathematical framework ensures that the system is flexible and can adapt to different scenes while maintaining simplicity and efficiency.

3 Implementation in GameMaker

This section explains how the lighting system is implemented in GameMaker, focusing on the key components: light rendering, shadow casting, and rim lighting. Each part is designed to be efficient and easy to adapt.

3.1 Lighting Logic

The lighting logic handles the rendering of dynamic point lights, including their intensity and range. The following GameMaker function is responsible for rendering a light source:

Listing 1: Point Light Rendering Function

```
function render() {
    if (is_active_at_time(global.time, time_start, time_end)) {
        var _draw_x = x - camera_get_view_x(view_camera[0]);
        var _draw_y = y - camera_get_view_y(view_camera[0]);

        // Calculate the blended light color based on intensity
        var _blended_light = merge_color(c_light, c_black, 1.0 -
            intensity);

        // Draw the light circles with intensity adjustment
        draw_circle_color(_draw_x, _draw_y, radius,
            _blended_light, c_black, 0);
        draw_circle_color(_draw_x, _draw_y, radius / 3,
            _blended_light, c_black, 0);
    }
}
```

Where:

- `radius`: Determines the range of the light source.
- `c_light`: Specifies the color of the light at its center.
- `c_black`: Gradually fades the light intensity to black at the edges.
- `intensity`: A value between 0.0 and 1.0, controlling the brightness of the light.
  - `intensity = 1.0`: The light is at full brightness.
  - `intensity = 0.0`: The light is completely dimmed.
- `_blended_light`: The result of merging the light color `c_light` with black (`c_black`) based on the intensity.
- The position is adjusted based on the camera's view to ensure correct rendering.

3.2 Shadow Casting

Shadows are calculated on the basis of the position of objects relative to the light source. The following code snippet demonstrates the shadow casting logic:

Listing 2: Shadow Casting Logic

```
var _nearest_light = noone;
var _min_dist = 300; // Initial large distance
var _max_dist = 64;  // Maximum effective shadow distance

```

```
// Find the nearest light source
with (obj_light_source) {
    var _dist = point_distance(other.x, other.y, x, y);
    if (_dist < _min_dist) {
        _nearest_light = id;
        _min_dist = _dist;
    }
}

// Skip shadow calculations if the light is too far
if (_dist_to_light > _max_dist) {
    continue;
}

// Shadow direction and length
var _dir_to_light = point_direction(other.x, other.y + 8, x, y);
var _shadow_direction = _dir_to_light + 180;
var _shadow_length = clamp(0 + (500 / (_dist_to_light * 0.1 +
    1)), 10, 300);
```

Where:

- `_nearest_light`: Identifies the light source closest to the object.
- `_shadow_direction`: Calculated using the `point_direction` function, which finds the angle between the light source and the object.
- `_shadow_length`: Adjusts dynamically based on the distance to the light source, with clamped minimum and maximum values.

3.3 Shader Logic

Rim lighting relies on normal maps to define the surface geometry of objects. In this project, normal maps were generated using EdgeNormals, a script that converts 2D sprites created in Aseprite into compatible normal maps. The rim lighting implementation uses two shaders: a vertex shader and a fragment shader. The vertex shader processes vertex positions, texture coordinates, and colors, passing interpolated values to the fragment shader. The fragment shader then calculates the lighting effect based on the normal map, light position, and other lighting properties.

Listing 3: Vertex Shader for Rim Lighting

```
attribute vec3 in_Position;                  // (x, y, z)
    coordinates of the vertex
attribute vec4 in_Colour;                    // (r, g, b, a)
    color values of the vertex
attribute vec2 in_TextureCoord;              // (u, v) texture
    coordinates

varying vec2 v_vTexcoord;                    // Passed to the
    fragment shader
varying vec4 v_vColour;                      // Passed to the
    fragment shader

void main()
{
    // Transform vertex position into clip space
    vec4 object_space_pos = vec4(in_Position.x, in_Position.y,
        in_Position.z, 1.0);
```

```
    gl_Position = gm_Matrices[MATRIX_WORLD_VIEW_PROJECTION] *
        object_space_pos;

    // Pass color and texture coordinates to the fragment shader
    v_vColour = in_Colour;
    v_vTexcoord = in_TextureCoord;
}
```

Listing 4: Fragment Shader for Rim Lighting

```
varying vec2 v_vTexcoord;
varying vec4 v_vColour;

uniform sampler2D u_NormalMap;
uniform vec2 u_LightPos;
uniform vec2 u_PlayerPos;
uniform vec3 u_LightColor;
uniform float u_LightIntensity;

void main() {
    vec4 normalMap = texture2D(u_NormalMap, v_vTexcoord);

    // Discard pixels with no alpha
    if (normalMap.a == 0.0) {
        discard;
    }

    vec3 normal = normalize((normalMap.rgb * 2.0) - 1.0);
    vec2 lightDir = normalize(u_LightPos - u_PlayerPos);
    float diffuse = max(dot(normal.xy, lightDir), 0.0);
    float attenuation = smoothstep(u_LightIntensity,
        u_LightIntensity * 0.5, length(lightDir));

    vec3 highlight = diffuse * attenuation * u_LightColor * 3.0;

    gl_FragColor = vec4(highlight, attenuation);
}
```

Where:

- `normalMap`: Stores the object's normal data, used for rim lighting calculations.
- `lightDir`: Direction from the light source to the object.
- `diffuse`: Lighting intensity based on the angle between the surface normal and the light direction.
- `attenuation`: Controls how light fades with distance, calculated using `smoothstep`.
- `highlight`: Combines the lighting and attenuation for the rim effect.

GameMaker Integration Code: The following GameMaker code demonstrates how to integrate rim lighting using the above GLSL shader:

Listing 5: GameMaker Rim Lighting Logic

```
shader_set(sh_rim_lighting);

// Bind the normal map texture
texture_set_stage(1, normal_map_texture);
```

```
shader_set_uniform_i(shader_get_uniform(sh_rim_lighting,
    "u_NormalMap"), 1);

// Set uniform variables for light position and intensity
shader_set_uniform_f(shader_get_uniform(sh_rim_lighting,
    "u_LightPos"), light_x, light_y);
shader_set_uniform_f(shader_get_uniform(sh_rim_lighting,
    "u_PlayerPos"), x, y);
shader_set_uniform_f(shader_get_uniform(sh_rim_lighting,
    "u_LightIntensity"), light_radius);
shader_set_uniform_f(shader_get_uniform(sh_rim_lighting,
    "u_LightColor"),
     color_get_red(light_color) / 255,
     color_get_green(light_color) / 255,
     color_get_blue(light_color) / 255
);

// Draw the normal map sprite using the shader
draw_sprite(normal_map, image_index, x, y);

shader_reset();
```

Where:

- `sh_rim_lighting`: The GLSL shader used for rim lighting.
- `normal_map_texture`: The texture containing the normal map for the current object.
- `light_x, light_y`: The position of the light source.
- `light_radius`: The intensity or range of the light source.
- `light_color`: The RGB color values of the light source.
- `shader_reset`: Resets the shader after drawing the sprite.

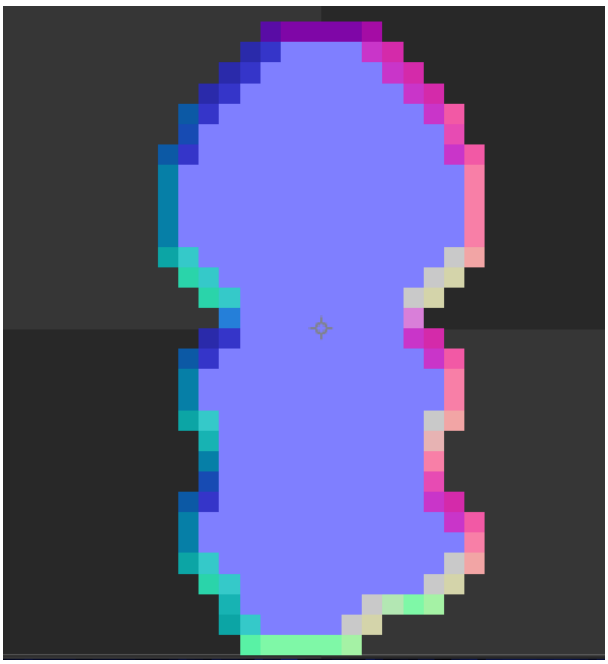

Figure 2: Example normal map generated from a sprite using Aseprite. The colors encode surface normals: **Cyan** represents edges facing outward and slightly upward, **Magenta** indicates edges facing outward and sideways, and **Purple** denotes flat surfaces facing directly outward.

4 Results and Visuals

This section showcases the outcomes of the implemented 2D lighting system, demonstrating its core features: dynamic point lights, cast shadows, and rim lighting.

4.1 Dynamic Point Lights

Figure 3 demonstrates the dynamic point lights, showing how intensity and radius affect the scene.

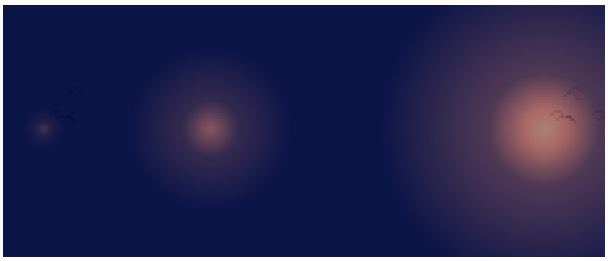

Figure 3: Dynamic point lights with varying intensity and radius.

4.2 Shadows and Light Interaction

Figure 4 highlights how shadows are cast based on the position of objects relative to the light source. The shadow length and direction are dynamically calculated.

Figure 4: Shadows cast by actor under dynamic lighting.

4.3 Rim Lighting

Rim lighting enhances the edges of objects illuminated by a nearby light source. Figure 5 shows a comparison of objects with and without rim lighting applied.

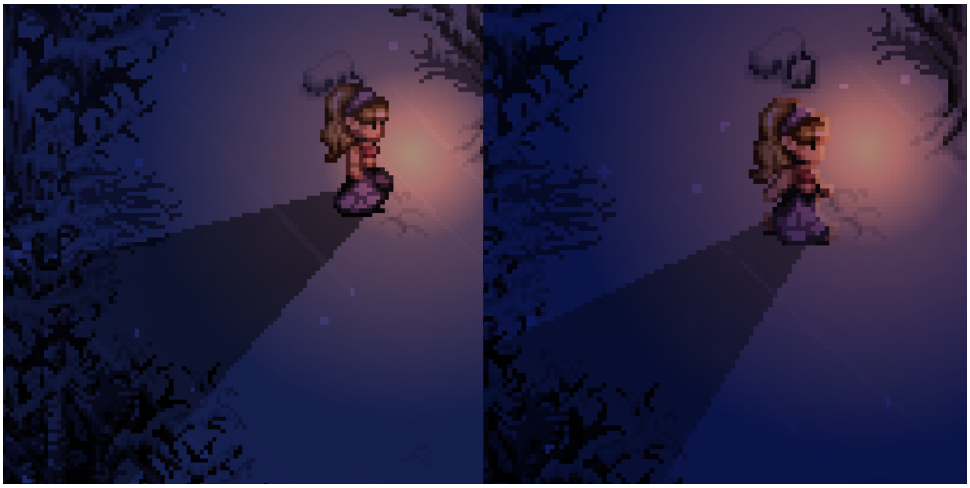

Figure 5: Comparison of actors with (right) and without (left) rim lighting.

5 Limitations

While the implemented 2D lighting system provides dynamic point lights, cast shadows, and rim lighting, there are several limitations that constrain its functionality and scalability. These are outlined below:

5.1 Performance Limitations

The system's performance can degrade when handling a large number of light sources or objects:

- **Multiple Light Sources:** The current implementation does not fully account for overlapping light sources. Shadows and rim lighting may not combine seamlessly when multiple lights affect the same object.
- **Computational Overhead:** Real-time calculations of lighting and shadows inherently require more computational resources compared to baked lighting. While this is a trade-off common to all dynamic lighting systems, optimizing the system for larger scenes or more complex interactions would require additional performance tuning.

5.2 Visual Limitations

The visual fidelity of the system has its constraints:

- **Simplified Shadow Casting:** Shadows are limited to a single direction and length per light source, which can result in unrealistic visual artifacts in scenes with complex geometries.
- **Static Normals:** The rim lighting system relies on normal maps generated for static objects. For animated objects, a sprite sheet of normal maps can be precomputed and used in conjunction with an animation system to properly index the correct normal for each frame. However, this requires additional effort in asset creation and implementation.

5.3 Scalability Limitations

The system is designed with simplicity in mind, making it less suitable for large-scale or highly dynamic scenes:

- **Scene Size:** The lighting and shadow system assumes a relatively small and contained environment. Scaling it to larger scenes would require some optimization.
- **Dynamic Objects:** Handling large or fast moving dynamic objects can result in visual inconsistencies due to reliance on simplified calculations.

5.4 Engine-Specific Implementation

The current implementation is tailored to GameMaker, which may limit its portability to other engines without significant modifications:

- **Language and API Dependencies:** Features such as `point_direction` and `draw_circle_color` are specific to GameMaker and would need to be adapted for use in other frameworks or engines.

These limitations highlight areas where the system could be improved or expanded in future iterations. Despite these constraints, the lighting system is a practical and accessible solution for many small- to medium-sized 2D projects. With careful optimization and resource management, the system can effectively handle moderately complex scenes while maintaining performance and visual quality.

6 Future Work

While the current implementation provides a practical and accessible approach to 2D dynamic lighting, there are several areas where the system could be expanded or optimized:

6.1 Multi-Light Source Support

The system could be enhanced to handle overlapping light sources more effectively. This would involve:

- Combining light intensities and colors from multiple sources.
- Ensuring shadows blend realistically when affected by multiple lights.

6.2 Enhanced Shadow Casting

To increase visual realism, shadow casting could be refined by:

- Implementing support for multi-cast shadows, allowing objects to cast shadows from multiple light sources simultaneously.
- Dynamically adjusting shadow edges to simulate penumbra effects.

6.3 Streamlined Animation Workflow for Normal Maps

While the system already supports animated normal maps through sprite sheets, the workflow could be further streamlined. This could include:

- Automating the generation of normal map sprite sheets from animation frames.
- Developing tools or scripts to integrate this workflow seamlessly into game development pipelines.

6.4 Performance Optimization

Optimizations could reduce the computational overhead, such as:

- Implementing light culling to skip calculations for lights outside the camera's view.
- Using precomputed look-up tables or simpler approximations for frequently repeated calculations.

6.5 Portability to Other Engines

Adapting the system for other game engines, such as Unity or Godot, would increase its accessibility and application scope. This would involve:

- Replacing GameMaker-specific functions with equivalents in other engines.
- Writing reusable GLSL shaders or adapting them for engine-specific formats.

7 Conclusion

This document presented an accessible and practical approach to the implementation of dynamic 2D lighting, including point lights, cast shadows, and rim lighting. The system was designed to balance simplicity, performance, and visual quality, making it well-suited for small- to medium-sized 2D projects.

The mathematical framework and implementation details provided in this document demonstrate how the system can be recreated or adapted in GameMaker or similar engines. Key features, such as dynamic point light attenuation and rim lighting based on normal maps, enhance the visual depth and realism of 2D scenes.

Despite limitations like simplified shadow casting and computational overhead, the system offers a robust foundation for future enhancements. Future work could focus on improving multi-light source handling, enhancing shadow realism, and optimizing performance for larger or more complex scenes.

Overall, the lighting system offers a straightforward yet effective solution for 2D dynamic lighting, empowering developers to bring depth and atmosphere to their games.

Credits

The sprites shown in this document are licensed assets from Mana Seed.[1]

[1]These assets are part of a project currently in development and are used here solely for demonstration purposes. Learn more about Mana Seed at Mana Seed Website.

References

- Aseprite, "Pixel Art Tool," https://www.aseprite.org/.
- Securas, "EdgeNormals: An Aseprite script to compute fake normal maps at the edges of sprites," GitHub, https://github.com/securas/EdgeNormals.