

Enabling the performant rendering of 2D signed distance fields on the web

András Schmelczér

Budapest University of Technology and Economics

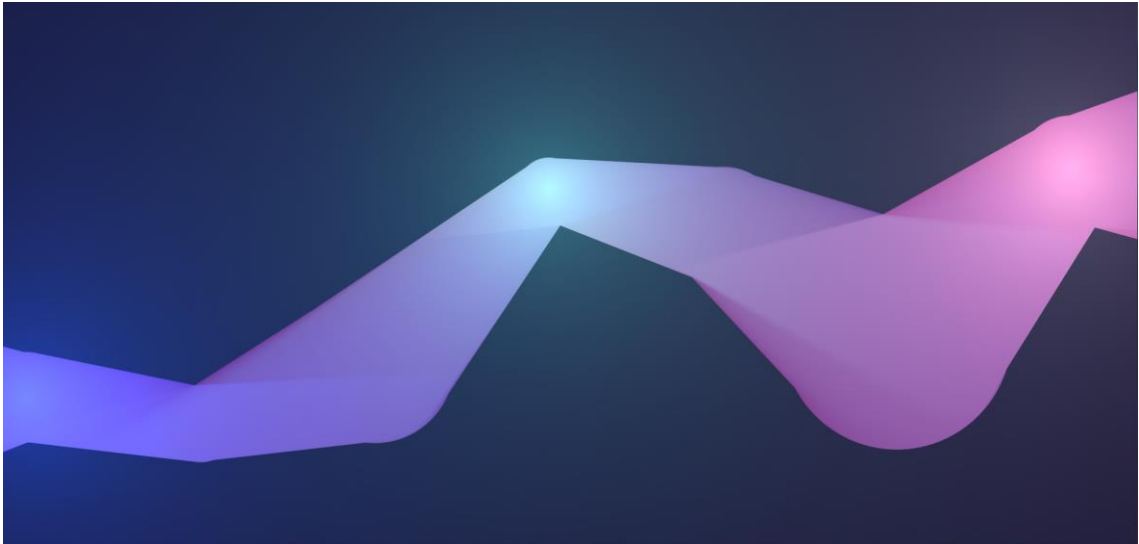


Figure 1 Screenshot of a scene rendered by SDF-2D

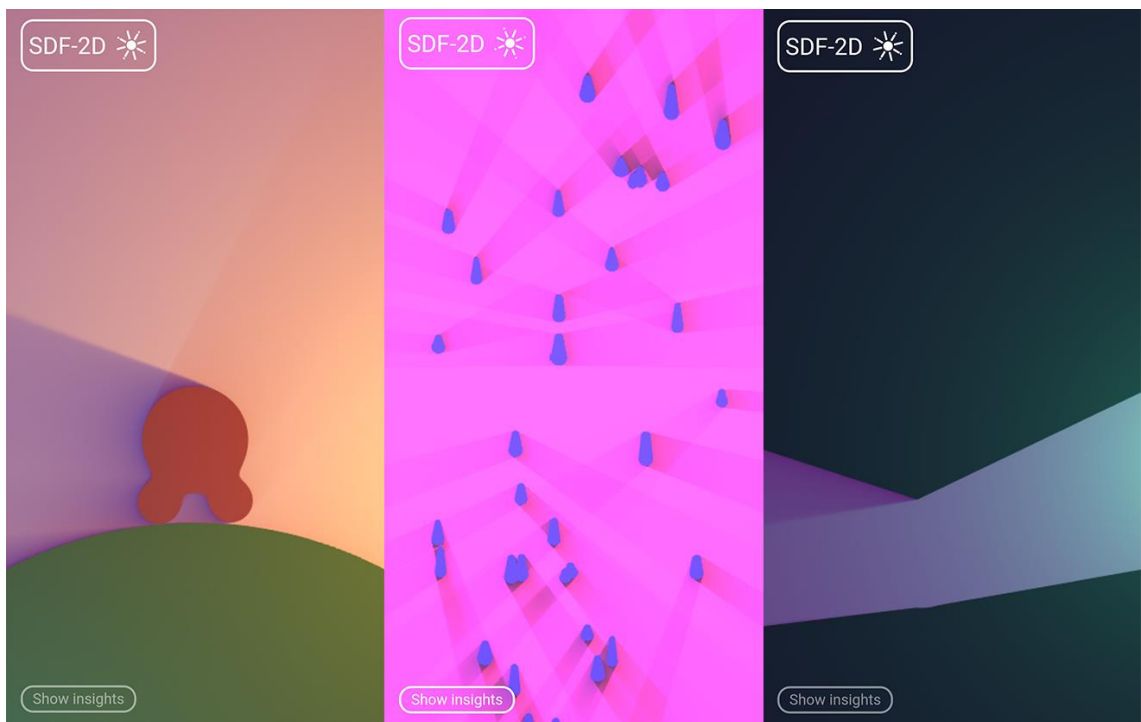


Figure 2 Three separate screenshots taken on a mobile phone running SDF-2D in real-time

1 Introduction

1.1 Motivation

Hardware is getting increasingly faster year by year; this holds especially true for Graphical Processing Units (GPU-s). Thus, modern consumer electronics (more specifically, smartphones, tablets, notebooks, and personal computers) are capable of performing some computationally heavy workloads that were not possible before, including real-time circle-tracing. Unfortunately, reusable software solutions utilising this technique are non-existent.

1.2 Overview

Circle-tracing is the 2-dimensional simplification of sphere tracing, the latter was introduced by *Hart* [1]. Briefly, sphere tracing (a special form of raytracing) makes it possible to render implicit surfaces defined by a signed distance field (SDF) or its lower bound. Rendering is done by marching along a ray with steps equal to the value of the scene's SDF estimate at the current end of the ray. In a 3-dimensional setting it opens various possibilities to render and shade in novel ways.

To render a 2D scene defined by an SDF, all one has to do is to evaluate the function at every pixel and check if its value is less than zero¹. In that case, an object should be visible under that pixel. So far, raytracing needs not be involved. However, tracing rays might be appropriate to achieve more sophisticated shading, for instance, shadows, area lights, reflections, volumetric effects, ambient occlusion, etc.

For providing users with cutting-edge 2D graphics, using circle-tracing seems to be an ideal candidate for its simplicity and its ability to facilitate aesthetic lighting effects. Unfortunately, a naïve implementation of this would only produce a real-time experience on high-end graphics cards. Additionally, to deliver a universally usable product, compatibility with most devices is also a requirement. These concerns are not unique to any single application, consequently, investing into solving them can benefit a wide range of operations.

¹ Rudimentary antialiasing can also be provided by allowing finer levels of occlusion when the SDF returns a value close to zero.

2 Concerns

2.1 Compatibility

To address the issue of compatibility, cross-platform development — based on my empirical understanding — is still in its infancy. A multitude of solutions exist, but very few of them are truly cross-platform and easily accessible for end-users. The only viable solution seems to be web-based applications for the reasons that browsers mostly conform to the same web specifications and are also available on every platform. Furthermore, clicking a link is by far the easiest way of accessing an application. Hence, the ideal solution should be web-based.

In the context of a webpage, communicating with the GPU was made possible by the *WebGL* [2], and later, the *WebGL2* [3] application programming interfaces (API-s). These are the ports of the OpenGL ES 2.0 and OpenGL ES 3.0 API-s respectively and are widely supported as of September 2020. WebGL2 is available for 78.13 per cent of the world’s users, while WebGL is supported for 97.23 per cent of them [4]. The former is more feature-rich, but for achieving cross-browser compatibility both API-s should be employed because even with WebGL, significant performance gains can be achieved [5].

2.2 Reusability

Looking through the available and even remotely 2D raytracing related frameworks and software, it seems, they are noticeably rare. Although many graphics frameworks exist for the web, for example, *three.js* [6], their support for circle-tracing is absent. On the other hand, sphere tracing in the browser has been attempted many times before, e.g. by *Jiarathanakul* [7] or *Quilez* [8]. With Iñigo Quilez being a determining pioneer of this field. Yet these applications are still concerned with 3D. In conclusion, a web-based graphics framework capable of handling SDF-s in 2-dimensions is missing from the landscape.

An ideal graphics library — apart from supporting the above-mentioned features — should be available as a Node Package Manager (NPM) package, which is the de facto format for sharing code written for the web. This way it could be easily downloaded and bundled with any JavaScript (JS) and TypeScript (TS) application helping later reuse in all kinds of applications.

2.3 Performance

Without a doubt, performance should be the top priority; for this, optimisations are needed. Below I propose various approaches to enable the usage of circle-tracing in real-time rendering even on mobile devices. This list could be improved and extended, although, after implementing these propositions (the implementation is introduced in section 3), I found that performance is no longer a bottleneck in providing a 2D experience.

2.3.1 Memoized distance field

Firstly, deferred shading can be beneficial in this situation. Using two render passes, the first one can calculate the value of the SDF at every pixel, then the second one is able to access these memoized values for calculating the actual shading. This comes with two distinct advantages.

For instance, given that from every pixel a ray is shot towards every light source with each ray having n_{step} steps, the SDF would need to be evaluated $n_{pixel} * n_{light} * n_{step}$ times. Instead, with this method, it is only evaluated n_{pixel} times. The negligible overhead of accessing the results of the previous pass does not change the fact that a significant performance increase has been gained.

In practice, the first render pass can draw to a texture and the second can read the values of this texture ². From this, another advantage can be derived. The resolution of these textures need not be equal, hence, evaluating the SDF can be done in a lower resolution, while maintaining a higher resolution during shading. In real life scenarios, this comes with minimal noticeable deterioration of quality.

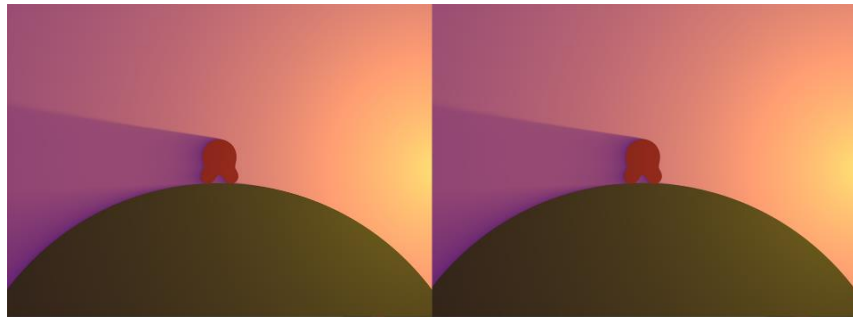


Figure 3 Original (left) and $\frac{1}{2}$ undersampled, then linearly interpolated distance field (right)

² The exact format and interpolation type might depend on the available WebGL version and extensions.

2.3.2 Tile-based rendering

Another interesting optimisation method can be the usage of bounding areas. This technique is widely utilised in raytracing contexts. It is oftentimes implemented in a way that each complex object is assigned a bounding sphere, the SDF of which gets evaluated first. Further evaluation of the object's possibly complex distance function happens only if the bounding sphere has been penetrated. Still, this comes with a non-negligible cost if multiple rays are shot from every pixel. A significant speedup can be achieved by shifting this overhead earlier in the pipeline. Fortunately, when rendering the distance field into a texture, it can be done.

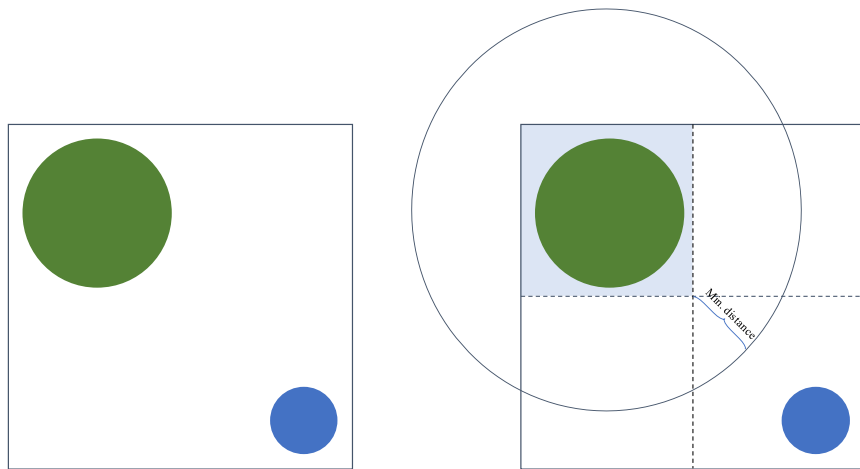


Figure 4 Dividing the screen-space into 4 tiles

Let us divide the screen into a grid of tiles, then for each tile fetch the objects intersecting with its close vicinity. After determining the maximum distance within which no other objects can be, the drawing code needs to only examine the few objects near its currently shaded pixel's tile. In practice, an 8 by 8 grid seemed to achieve the best results.

On the diagram, two screens are visible, containing two coloured circles. In the first case, the shader of each pixel must consider both circles. When the aforementioned precalculated tiles are used in the second case, the fragment shader of the upper left tile needs to only consider the green circle, halving the execution time. It is important to notice, that the distance function should not return greater values than the length of *min. distance*, indicated on Figure 4, due to it having no information on how far the nearest object is beyond its bounding circle. For instance, in the middle picture of Figure 2, while there are 36 objects on screen, on average every single tile needs to know about around 4 of them. Clearly, this method can only be used when the SDF evaluation and lighting happen in different passes.

2.3.3 Dynamic shader generation

Owing to the GPU-s highly parallel nature (and to branching being rather expensive) loop unrolling is heavily utilised when compiling shader code. Accordingly, having a fixed number of iterations for for-loops is beneficial when using WebGL2 and is required when using WebGL.

For instance, for-loops can be used to iterate over the objects of a scene, however, the number of visible objects is not always fixed. We can opt for finding the largest object count and specifying that as the iteration count, but that would result in non-existent objects being needlessly considered and that is costly.

Determining some arbitrary values for possible counts of each object on screen and using these values to compile multiple shaders with them being hardcoded into for-loops could prove beneficial. If multiple programs with differing object limits are pre-compiled, at render time it is possible to choose the program that most closely matches the current number of objects, therefore, reducing draw time.

For many different types of objects, this method does not scale well, because, for n_{object} objects, each having n_{counts} steps, $(n_{counts})^{n_{object}}$ number of shader programs are required. Consequently, a trade-off must be made when using a multitude of different objects. Nonetheless, combined with tile-based rendering — where some tiles may need to consider a larger cluster of objects, while other might even be empty — it can produce meaningful improvements.

2.3.4 Efficient lighting algorithm

Aesthetic lighting has to be implemented, in order to achieve the original purpose of the library. Shadow casting comes with the highest price in performance³; hence, great care needs to be taken in implementing it. Raytracing based shadows could be drawn using the solution described by *Quilez* [9] or with a slightly differing algorithm implemented by the Shadertoy user *Maarten* [10]. Nevertheless, when using these approaches two shortcomings become apparent.

³ After all, this is the sole circle-tracing part of this graphics application.

Firstly, for pleasing results, around 64 to 128 ray steps are needed, which is not acceptable for low-performance devices. Secondly, these algorithms rely heavily on the SDF being exact (not just a lower bound). For this reason, unappealing artifacts can occur. Additionally, shadows cast by sharp edges can also pose a challenge for both techniques. This could be fixed with smaller steps but that would incur the additional cost of even more steps. These artifacts are visualised below. The left image was produced using the above soft shadow shader, on the right image, a simple hard shadow casting function was used with rays traveling up to 8 steps.

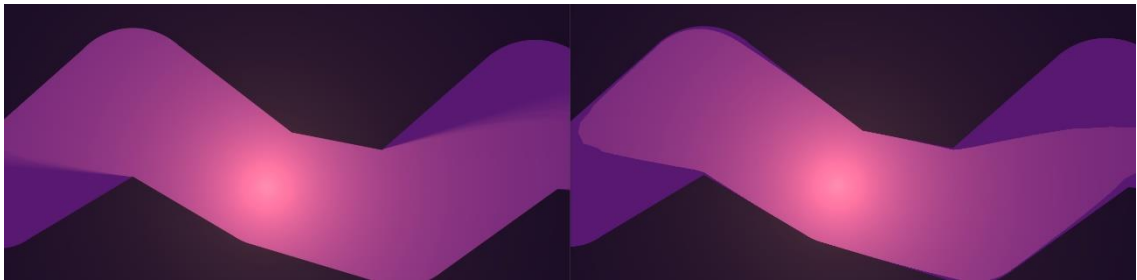


Figure 5 Artifacts from sharp edges (left) and insufficient number of steps (right)

I propose an alternative shadow casting function (implemented in the OpenGL Shading Language (GLSL) function *shadowTransparency*), the code of which is available below. It only produces empirically acceptable results — similar to the previous solutions —, still it runs much faster than the previous methods, while not relying on the SDF being exact.

In the code, 16 iterations are used, but this could be set anywhere between 8 and 32 to change the look of the result. Comparisons of different values can be seen on Figure 6.

```
float shadowTransparency(
    float lightCenterDistance,
    vec2 lightDirection
) {
    float rayLength = 0.0;

    for (int j = 0; j < 16; j++) {
        rayLength += max(0.0, getDistance(
            uvCoordinates + lightDirection
            * rayLength
        ));
    }

    return min(1.0, pow(
        rayLength
        / lightCenterDistance,
        0.3
    ));
}
```

```
vec3 colorInPosition(
    int lightIndex,
    out float lightCenterDistance
) {
    int i = lightIndex;

    lightCenterDistance = distance(
        circleLightCenters[i],
        position
    );

    return circleLightColors[i] / pow(
        lightCenterDistance
        / circleLightIntensities[i] + 1.0,
        2.0
    );
}
```



Figure 6 Images produced by applying the above-mentioned functions, from left to right the iteration counts are 32 – 16 – 8

For the results above, some other techniques were utilised as well. Namely, anti-aliasing and letting the light penetrate objects with decreased intensity. Overall, I found this approach more useful in this context in every aspect, except one.

Long shadows are getting increasingly lighter as they reach their end, which somewhat holds true in real life; more light rays have the chance to illuminate the area under the shadow, and the shadow casting light loses its power the farther we get from it. These combined can create a similar feeling to what the shader is producing. The only issue with the shader is that objects in the shadow of another object can cast a new shadow onto the previous one; thus, a single light source can cast two overlapping shadows. This is undesirable, though it occurred rarely during testing.

2.3.5 Results

Assembling the information presented in the preceding sections, a table is provided below containing some performance comparisons for the following scene. The picture contains exactly 200 objects and 2 light sources and is rendered at a resolution of 2560 by 1080 pixels using an RX 590 GPU.

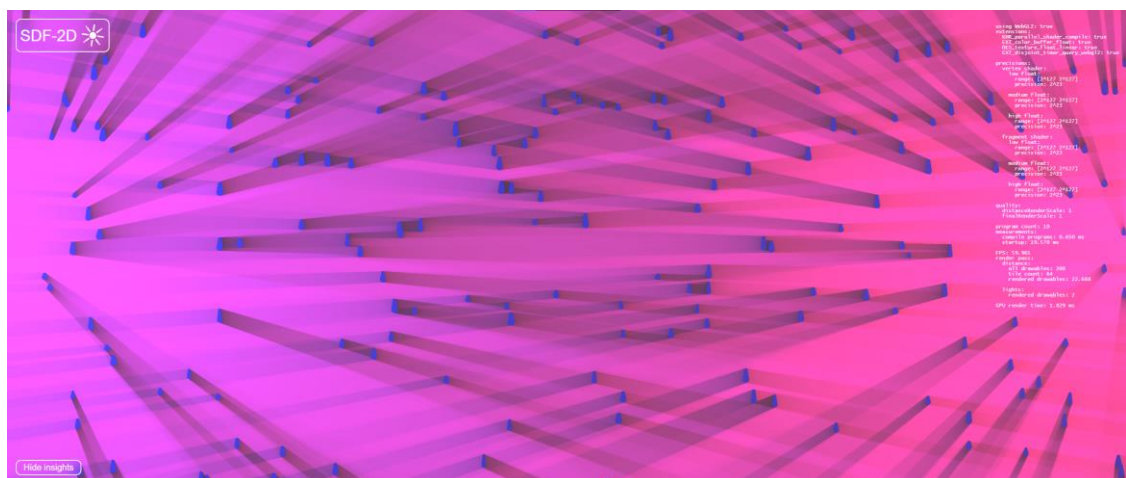


Figure 7 Scene used for performance testing

The lighting code was consistent between the experiments, with rays shooting from each pixel towards both light sources, taking 16 steps with each ray. The measurements were quite stable with minimal fluctuations of less than 5%. The chart demonstrates that the proposed optimisations — achieving a 70-fold increase in rendering speed — are capable of enabling real-time 2D raytracing.

<i>Optimisations enabled</i>	Frames per second (FPS) ⁴	GPU draw time ⁵
<i>None</i>	7.5 FPS	130 ms
<i>Memoized distance field</i>	60 FPS	5 ms
<i>Memoized distance field and Tile-based rendering ⁶</i>	60 FPS	1.85 ms

Figure 8 Average values measured with different optimisations enabled

⁴ It is capped by the browser at the displays refresh rate, in this case its maximum is 60 FPS.

⁵ As reported by the *EXT_disjoint_timer_query* extension [11].

⁶ For tile-based rendering an 8 by 8 grid was used.

3 Solution

For satisfying the requirements laid out so far, I have created the SDF-2D library. It is available as an NPM package (types included), it is compatible with both WebGL and WebGL2, takes care of many boilerplate code, such as lost context handling and automatic restoration or parallel shader compiling, and has a number of built in distance functions and lights as well, while still being easily extendible.

A demo page utilising its features can be found at sdf2d.schmelczer.dev. Its possibilities, features and performance has been showcased before, predominantly during the Performance section, however a thorough documentation of its API and use-cases — along with the library itself — can be found on its website, npmjs.com/package/sdf-2d. The source code can be found on GitHub, at github.com/schmelczerandras/sdf-2d.

3.1 Minimal example

Using the library is quite straightforward. For instance, the code of a minimal example application could look like this. Given that the HTML page has a `<canvas>`.

```
import { compile, Circle, CircleLight } from 'sdf-2d';

const main = async () => {
  const canvas = document.querySelector('canvas');
  const renderer = await compile(
    canvas, [Circle.descriptor, CircleLight.descriptor]
  );

  renderer.addDrawable(new Circle([200, 200], 50));
  renderer.addDrawable(new CircleLight([500, 300], [1, 0.5, 0], 0.5));

  renderer.renderDrawables();
};

main();
```

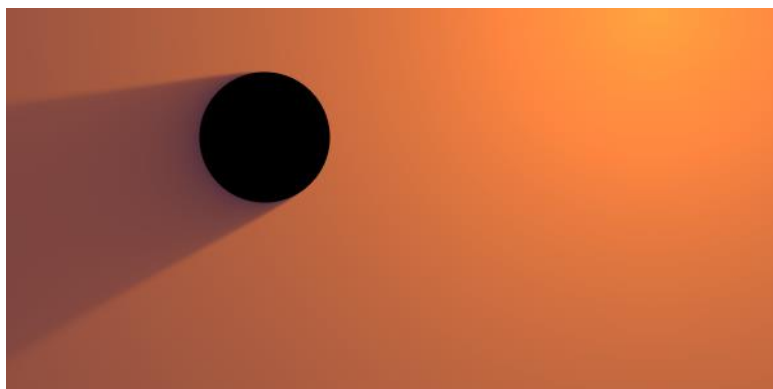


Figure 9 The result of the above code looks like this on a 600 by 300 pixel canvas.

Asynchronous code is required for parallel shader compiling. The descriptors of the to-be-drawn objects are required before creating the renderer, allowing the compiler to only create the shaders that will actually be used. The library offers more sophisticated commands and the fine-tuning of settings, to learn more about this, refer to the documentations mentioned in section 3. The way of adding the distance field definitions of new objects is also described there.

3.2 Insights

The architecture of the library offers no surprises. Most WebGL objects are wrapped to provide their users with more functionality, and some new abstractions have also arisen, e.g. for render passes, colour handling, and uniform handling.

The most challenging part of the implementation, aside from optimising, was the homogenous handling of both WebGL and WebGL2 contexts, the restoration of lost contexts, and using non-blocking shader compiling. Fortunately, cross-browser compatibility was not an issue based on my testing with various virtualised mobile operating systems.

As a side note, it is important to address the issue of floating-point precision. Using less precise float values is advisable for compatibility and performance reasons alike. That means, in our shaders large-growing world coordinates cannot be used, because unwanted artifacts can occur with great enough values. Because values are exclusively passed using uniform variables, conversion to Normalized Device Coordinates (NDC) does not happen automatically, thus, extra care must be taken to convert them to a coordinate system having a fixed range. The library takes care of this by making the appropriate transformation matrix available for drawable objects.

3.3 Limitations

Though colours are, texturing is not yet supported. Additionally, there are future plans for supporting more distance functions and more refined lighting effects as well.

4 References

- [1] J. Hart, "Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12., no. 10., pp. 527-545, 1996.
- [2] "WebGL Specification," Khronos Group, 9 April 2020. [Online]. Available: <https://www.khronos.org/registry/webgl/specs/latest/1.0/>. [Accessed 25. September 2020].
- [3] "WebGL 2.0 Specification," Khronos Group, 2. January. 220. [Online]. Available: <https://www.khronos.org/registry/webgl/specs/latest/2.0/>. [Accessed 25. September 2020.].
- [4] "caniuse," [Online]. Available: <https://caniuse.com/?search=webgl>. [Accessed 24. September. 2020.].
- [5] E. e. a. Aho, "Towards real-time applications in mobile web browsers," *IEEE 10th Symposium on Embedded Systems for Real-time Multimedia*, pp. 57-66., 2012.
- [6] "three.js," [Online]. Available: <https://threejs.org/>. [Accessed 24. September 2020.].
- [7] P. Jiarathanakul, "Ray marching distance fields in real-time on webgl," Citeseer.
- [8] I. Quilez, "Modeling with distance functions," [Online]. Available: <https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>. [Accessed 12. September 2020.].
- [9] I. Quilez, "Soft Shadows in raymarched SDF-s," 2010. [Online]. Available: <https://iquilezles.org/www/articles/rmshadows/rmshadows.htm>. [Accessed 26. September 2020.].
- [10] Maarten, "2d signed distance functions," Shadertoy, 17. August 2015.. [Online]. Available: <https://www.shadertoy.com/view/4dfXDn>. [Accessed 26. September 2020.].
- [11] "WebGL EXT_disjoint_timer_query Extension Specification," Khronos Group, 30. September 2016.. [Online]. Available: https://www.khronos.org/registry/webgl/extensions/EXT_disjoint_timer_query/. [Accessed 26. September 2020.].