



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

András Schmelczer

MULTIPLAYER 2D BROWSER GAME DEVELOPMENT WITH CIRCLE TRACING

ADVISOR

Dr. László Szécsi

BUDAPEST, 2020

Table of contents

Összefoglaló	4
Abstract.....	5
1 Introduction.....	6
1.1 Graphics	6
1.2 Game	7
1.3 Structure.....	8
2 SDF-2D.....	9
2.1 Overview.....	9
2.2 Concerns	10
2.3 Performance.....	11
2.4 Solution.....	16
2.5 Architecture	18
2.6 Implementation	22
2.7 Limitations	31
2.8 Results.....	31
3 The videogame.....	37
3.1 State of 2D games	37
3.2 Game mechanics	38
3.3 Architecture	39
3.4 Collision detection	43
3.5 Multiplayer.....	44
3.6 Commands	48
3.7 User interface	49
3.8 Limitations	51
3.9 Results.....	51
4 Conclusion	54
5 References.....	55
6 Appendix.....	58
6.1 Code for the noise renderer.....	58

HALLGATÓI NYILATKOZAT

Alulírott **Schmelczer András**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettettem, egyértelműen, a forrás megadásával megjelöltetem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervezet esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2020. 11. 20



Schmelczer András

Összefoglaló

A videójátékok hőskora óta töretlen népszerűségnek örvend az oldal-, és felülnézetes játékok stílusa. Ezek megjelenítése azonban nem követte szorosan a végfelhasználói eszközök számításiteljesítmény növekedését. A modern 2D játékok jellemzően nem használják ki a rendelkezésre álló hardver — kifejezetten grafikus — erőforrásokat, ezért az általuk nyújtott vizuális élmény javítható lenne. Erre megoldást jelenthet a sugárkövetés használata, ami viszont elsőre túl költségesnek tűnhet. Ez hatványozottan igaz mobilkészülékeken.

Szakdolgozatomban különböző optimalizációs eljárásokat és technikákat mutatok be, amikkel a valós idejű, két-dimenziós sugárkövetés szerényebb képességű eszközökön is elérhetővé válik. Az ötleteimet egy újrafelhasználható, webes technológiákra építő könyvtár formájában valósítom meg. Ennek alkalmazhatóságát és az ezáltal elérhető újszerű képi világot egy többjátékos online játék elkészítésén keresztül demonstrálom. Megvalósításomat számos, változatos képességű eszközön tesztelve arra következtethetünk, hogy az optimalizálatlan megoldásnál nagyságrendekkel hatékonyabb, a jelenlegi, átlagos 2D-s játékokéval pedig összemérhető teljesítményre képes, miközben vizuális megjelenítésben többet nyújt náluk.

Összességében, a fejlesztők számára egy új eszközt jelenthet a szoftverkönyvtáram, míg a felhasználóknak egy kipróbálási és visszajelzési lehetőséget ad a játékom. Utóbbi tapasztalatokból eldönthető, hogy érdemes-e a 2D sugárkövetésen alapuló grafika továbbfejlesztése.

Abstract

Multiple decades have passed since their first appearance, the genre of side and top-view games still enjoys significant popularity today. However, their visuals have not closely followed the increase in computing performance of end-user devices. Thus, modern 2D games typically do not take full advantage of available hardware — especially graphics — resources. Hence, the visual experience they provide could be improved. A solution may arise in the form of ray tracing, but at first sight it seems overly costly, even more so on smartphones.

In my thesis, I detail various optimisation techniques to make real-time, two-dimensional ray tracing a viable approach even on devices having modest resources. I implement my ideas in the form of a reusable library based on web technologies. Additionally, I demonstrate its applicability and the novel visuals that are achievable by using it through the development of an online multiplayer game. After testing my implementation on a plethora of devices of various capabilities, we can conclude that it is orders of magnitude more efficient than the unoptimised solution and its performance is comparable to that of current, mediocre 2D games, while providing better visual quality and more features.

Overall, my software library can be an innovative tool for developers, while my game gives users a chance to experience and form an opinion on this technology. From this feedback, it can be decided whether it is worthwhile to pursue the further development of 2D ray-traced graphics.

1 Introduction

Videogames are one of the universally liked items of technology. Not only they can teach and entertain players of every background, but they most often represent the cutting-edge technology and capabilities of consumer electronics. Certainly, games are not the only marvels of technology, however, they seemingly get the most exposure.

I am surely not a game designer, still, I believe, experimenting with and demonstrating the capabilities of modern graphics hardware is best done through a videogame. This way the appealing visuals can be enjoyed while having fun with the game itself.

In my thesis project I set out to come up with a solution to create a two-dimensional game which's imagery relies on the unprecedented performance of modern graphical processing units (GPU-s). This reliance is not without a reason, my rendering utilises ray tracing to draw scenes with — in my opinion — impressive quality.

My motivation comes from most 2D games having simple and similar visuals: only compositing sprites with little to no lighting effects. The power of video hardware grew considerably, whilst the prices kept decreasing continuously over the last few years [1]. Yet, the visual quality of 2D games did not follow this tendency closely. Without a doubt, there are exceptions to this rule, but these are rather uncommon.

1.1 Graphics

I would have liked to create a solution that I had not yet seen in practice. I enjoy watching lights that interact with the scene, casting shadows, penetrating objects, and giving a feeling of depth and reality. Using lights that partially illuminate objects is somewhat common in 2D games, but objects casting shadows are rare. Furthermore, I do not know of any 2D rendering engine supporting more complex lighting effects.

There is a reason for that, these effects can be expensive and hard to calculate using traditional approaches. From this statement, the issue of *hardness* can be solved by using a different rendering model, while the issue of *expensiveness* can be delegated to modern GPU-s having ever-increasing performance.

This different rendering model is called ray tracing, and it has a considerable history starting in the 16th century [2]. Recently, ray tracing has become a buzzword and with this, a large number of demos have showcased the potential of applying it for the

real-time illumination of 3D scenes. I truly am fond of watching these demos. That is why I aspired to implement something similar in my game.

The main part of my rendering process consists of an optimised implementation of circle tracing, a specific type of ray tracing. With this, a wide array of shading techniques becomes available. Additionally, it makes it possible to draw not just polygons but any complex shape which can be described mathematically. Partly the reason for ray tracing having to wait a couple of decades to become mainstream in real-time computer graphics is that it has some heavy performance costs. This somewhat holds true even in 2D, hence, effort needs to be invested in optimising it.

Since I focus heavily on the visuals, I am motivated to make it easily reusable in other software solutions. Not only by me, but by anyone else as well. Therefore, I have separated the graphics library into its own module, called *SDF-2D*. My desire with this is to expand the toolset of 2D game creators. For this library to be useful to anyone, extensive documentation and examples are also needed, thus, these are also part of my thesis.

For showcasing the raw possibilities of *SDF-2D*, I created a website where four animated demo scenes can be enjoyed. Additionally, the page inspects (and anonymously logs) the performance of the demos on different devices, making it possible to have some idea on the viability of the widespread use of this rendering method.

1.2 Game

With the advent of smartphones and facilitated by cheap mobile data plans, even playing multiplayer games on their phones is a realistic option for many. Still, consumers are also interested in gaming from their desktop or laptop personal computer (PC)-s. The landscape of available hardware (that can run videogames) is even more nuanced. There are tablets, game consoles, and even smart televisions.

The only common thing in these diverse platforms is that they all have browsers. These browsers, fortunately, aim to conform to the same specifications. This makes it possible to give the same input to all of them and be able to expect similar¹ results from each of them. This is the reason I chose to develop my videogame for browsers. Coincidentally, this also simplifies the issue of graphics, due to *WebGL* [3], and *WebGL2* [4],

¹ The word 'similar' may be a slight exaggeration.

which make it possible to implement demanding, real-time visuals on the web without having to tackle too many platform specific challenges.

It is important to keep in mind, that developing the game as a web application does not make it automatically usable on every platform. For instance, the controls have to work in every context, i.e. relying on keyboard input is not a viable solution on phones, while requiring touch input may not be suitable for PC-s. Thus, every aspect of the game has to be designed in a way that fits the platform it is played on.

One aspect of developing a browser game is that for downloading it, an internet connection is needed. But if the device has to be connected to the internet to play my game, then making it multiplayer does not make it less accessible to end-users. Hence, I made it multiplayer. Although playing online with your friends is a great experience, the technical aspects of multiplayer games can be less fun. Fortunately, a protocol called *WebSocket* [5], and technologies built on top of it make this easier to implement.

Because I am interested in 2D graphics, the game is obviously two-dimensional as well, this simplifies most facets of game design. Less assets are needed, the game logic need not be overly complicated, and it seems possible to build a game like this from the ground up under a single semester.

When designing the game, I aimed to make it as easy to use as possible. Continuously taking the feedback of my acquaintances into account, it turns out that a top-view, team-based conquest game mode satisfies the usability and enjoyability criteria alike; in short, the scene is set in space, two teams have to conquer small asteroids, while they can also shoot at the other team. Points are given based on the number of asteroids controlled, and the first team which reaches to a predefined score wins.

1.3 Structure

Between the graphics and the actual game, coupling is minimal. They are distributed in separate packages, and only the codebase of the game depends on that of the graphics. Their architecture, motivation and challenges are also quite distinct. I consider their implementation nearly independent of each other. Thusly, I introduce them one after the another. First, *SDF-2D*, then the videogame. I detail the background, architecture, implementation, and achieved results of both projects in their respective subsections.

2 SDF-2D

Hardware is getting increasingly faster year by year; this holds especially true for 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.

2.1 Overview

Circle tracing is the 2-dimensional simplification of sphere tracing, the latter was introduced by *Hart* [6]. Briefly, sphere tracing (a special form of ray tracing) 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 zero². In that case, an object should be visible under that pixel. So far, ray tracing need 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.

² Rudimentary antialiasing can be done by allowing finer levels of occlusion when the SDF has a value close to zero.

2.2 Concerns

2.2.1 Compatibility

To address the issue of compatibility in the domain of graphics, cross-platform development — based on my empirical understanding — is still in its infancy. A multitude of approaches 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. Moreover, 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, and later, the WebGL2 application programming interfaces (API-s)³. 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 [7]. The former is more feature-rich, but for achieving cross-browser compatibility both API-s should be employed because even with WebGL alone, significant performance gains can be achieved [8].

2.2.2 Reusability

At least a single framework exists providing 2D shadows, the 2D variant of the *Unity Universal Rendering Pipeline*. Still, it only has hard shadows which's casters also have to be manually specified. Additionally, *Unity* is more suitable for desktop applications due to its inherent performance requirements. Besides, the aim is to provide more sophisticated effects than simple hard shadows.

Looking through the available and even remotely 2D ray tracing related frameworks and software, it seems, they are rare. Although many graphics frameworks exist for the web, for example, *three.js* [9], 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* [10] or *Quilez* [11]. With Iñigo Quilez being a determining pioneer of this

³ The next step for graphics web API-s is *WebGPU* [30], but it is still a work in progress and not yet ready for universal adoption.

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) [12] 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 use of circle tracing in real-time rendering even on mobile devices. This list could be improved and extended, although, after implementing these proposals (the implementation is introduced in *section 2.6*), I found that the performance is no longer a bottleneck in providing a 2D experience.

2.3.1 Memoised 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 pixel⁴, then the second one is able to access these memoised 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 $n_{pixel} \cdot n_{light} \cdot n_{step}$ 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 texture⁵. From this, another advantage can be derived. The resolution of

⁴ For convenience, the base colour of the object occupying the pixel should also be calculated here.

⁵ The exact format and interpolation type might depend on the available WebGL version and extensions.

these textures need not be equal; hence, evaluating the SDF can be done at a lower resolution, while maintaining a higher resolution during shading. In real life scenarios, this comes with minimal noticeable deterioration of quality.

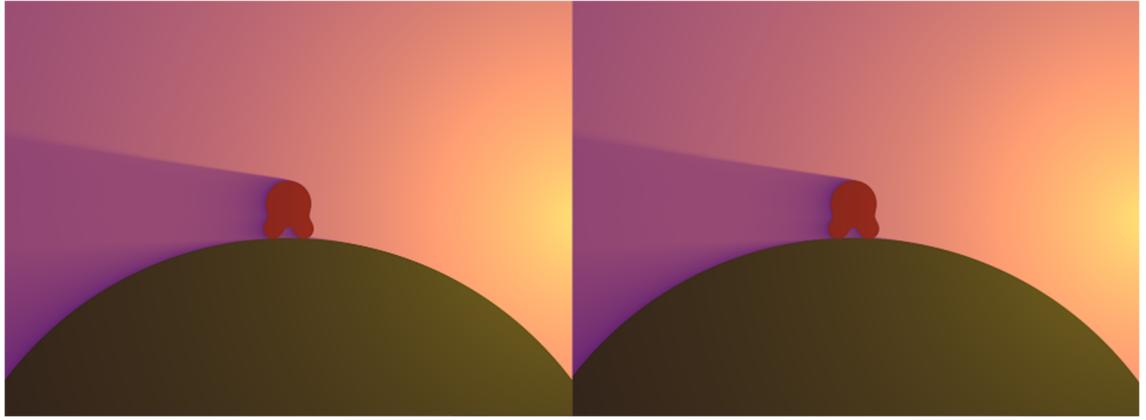


Figure 1 Original (left) and linearly $\frac{1}{3}$ undersampled, then interpolated distance field (right)

2.3.2 Tile-based rendering

Another interesting optimisation method can be the use of bounding areas. This technique is widely utilised in ray tracing 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 by the specific ray. Still, this comes with a non-negligible cost when 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.

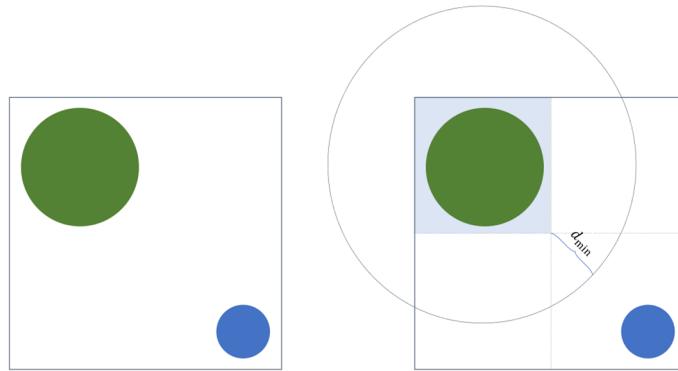


Figure 2 Dividing the screen-space into 4 tiles

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 only needs to 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 *Figure 2* two screens are visible containing two coloured circles. In the first case, the shader of each pixel must consider both circles. When the aforementioned pre-calculated tiles are used in the second case, the fragment shader of the upper left tile only needs to consider the green circle, halving execution time. It is important to notice that the distance function should not return greater values than the length of d_{\min} (indicated on the diagram) due to it having no information on how far the nearest object is beyond its bounding circle.

For instance, in *Figure 5*, while there are 200 objects on screen, on average, every single tile needs to know about approximately 23 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 type 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 precompiled, 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} object types, 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 the original purpose of the library. Shadow casting comes with the highest price in performance⁶; thus, great care needs to be taken when implementing it. Ray tracing based shadows could be drawn using the solution described by *Quilez* [13] or with a slightly differing algorithm implemented by the *Shadertoy* user *Maarten* [14]. Nevertheless, when using these approaches two shortcomings become apparent.

Firstly, for pleasing results, around 64 to 128 ray steps are needed per light source, which is not acceptable on 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-referenced soft shadow shader, on the right image, a simple hard shadow casting function was used with rays traveling up to 8 steps.



Figure 3 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. In short, it uses the heuristic of dividing the ray length with the light distance, then smoothing the results by raising them to an arbitrary power. This way, the closer the ray can get to the light source, the less shadow is drawn onto the starting point of the ray.

⁶ After all, this is the sole circle tracing part of this graphics application.

It only produces empirically acceptable results — like 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. Comparison of different values can be seen on *Figure 4*.

```
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
    );
}
```

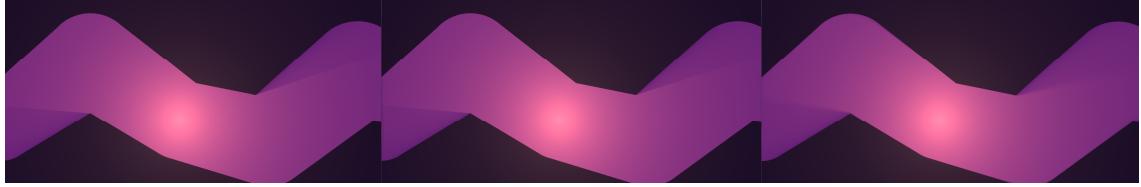


Figure 4 Images produced by applying the abovementioned functions, from left to right the iteration counts are 32 - 16 - 8, notice the shadow becoming softer with less iterations

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 luminosity 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 was rarely noticeable during testing.

2.3.5 Results of the optimisations

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

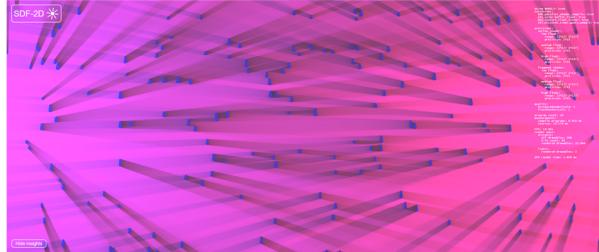


Figure 5 Scene used for performance testing

<i>Optimisations enabled</i>	Frames per second (FPS)⁷	GPU draw time⁸
<i>None</i>	7.5 FPS	130 ms
<i>Memoised distance field</i>	60 FPS	5 ms
<i>Memoised distance field</i> and <i>Tile-based rendering</i>	60 FPS	1.85 ms

Figure 6 Average values measured with different optimisations enabled

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 ray tracing.

2.4 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

⁷ It is capped by the browser at the display's refresh rate, in this case, 60 FPS.

⁸ As reported by the *EXT_disjoint_timer_query* extension [28].

automatic restoration, parallel shader compiling, and has a number of built in distance functions and lights as well, while still being easily extensible. It also supports textures.

A demo page utilising its features can be found at <https://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, <https://npmjs.com/package/sdf-2d>.

2.4.1 API

Not much is exposed for the clients. The API of the graphics library mainly consists of the **Renderer** interface, through which drawing **Drawables** is possible. To get a **Renderer** object, one must call either the **compile** or the **runAnimation** function. The former is tailored towards drawing static scenes and having fine control over when the actual rendering happens. The latter is made for easily running animations at the highest available framerate.

Using **runAnimation** is quite straightforward. For instance, the code of a minimal example application running an animation of a circle orbiting a light, could look like the following, given that the HTML page has a `<canvas>` element.

```
import { CircleFactory, CircleLight, hsl, runAnimation } from 'sdf-2d';

const canvas = document.querySelector('canvas');
const Circle = CircleFactory(hsl(180, 100, 40));

const draw = (renderer, time) => {
    renderer.addDrawable(
        new Circle(
            [
                150 + 50 * Math.cos(time / 1000),
                75 + 50 * Math.sin(time / 1000)
            ],
            25
        )
    );

    renderer.addDrawable(
        new CircleLight([150, 75], hsl(270, 100, 40), 0.1)
    );

    return true;
};

runAnimation(canvas, [Circle.descriptor, CircleLight.descriptor], draw);
```

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. This might seem redundant at first, but it is not. At start-up, the renderer has no information on the types of objects needed to be drawn, but the shaders have to be compiled then, to not hinder the performance later. Thus, these objects must be specified beforehand.

Most **Drawable** classes are accessed through factories. This way, it is possible to give the objects custom colours. The colour can be a single, hardcoded value (transparency is also supported), or a number that corresponds to a specific position of the colour palette, which can be changed between rendering frames. Helper methods are also available to help converting between different colour representations.

The **runAnimation** function has to be provided with a drawing function, which will be called by the library before rendering each frame. The calls to the renderer should be placed here. The render loop continues indefinitely until a *false* value is returned by the draw function.

The **Renderer** interface provides many more options than just scheduling to-be-drawn objects. Including but not limited to, updatable settings that can be changed at any time, insights describing the state of the renderer, the view area can be resized and moved, converting from screen to world coordinates and vice versa is also supported, and finally, the whole renderer can be conveniently disposed of by calling its destroy method.

Additionally, there is a helper class made publicly available for rendering noises to be used as textures, several built in **Drawable** implementations, and an **FpsAutoscaler** for automatically setting the rendering quality based on the framerate (which is utilised by default when using **runAnimation**).

More in-depth information and the specification of the API can be found in the documentation of *SDF-2D*. Nonetheless, with some understanding of the API and basic concepts of the library, we can start diving deeper into its structure and implementation.

2.5 Architecture

The main challenges that the architecture needs to be able to handle are the runtime generation and compiling of shaders, tile-based rendering, and multi-pass rendering. I found that these should be the main concerns when designing the structure of the library. Certainly, many other features are expected from the *SDF-2D* but those do

not require significant effort to implement regardless the layout of the program. The design of my library offers no surprises. Most WebGL objects are wrapped to provide their users with more functionality⁹, and some new abstractions have also arisen, e.g. for render passes, colour, and uniform handling.

2.5.1 Overview

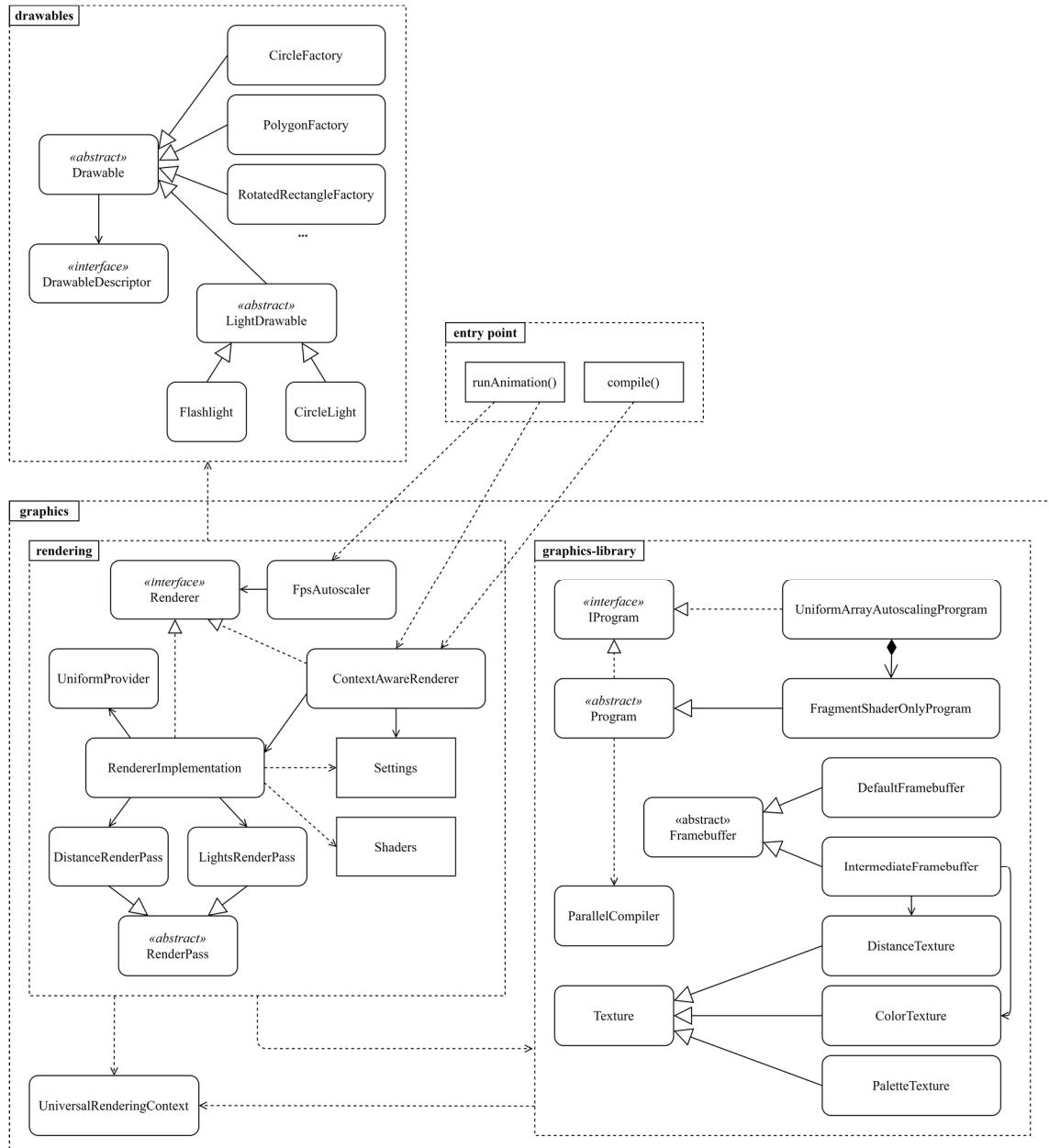


Figure 7 Overview of the architecture

⁹ Wrapping WebGL objects has the added benefit of increased type-safety, because passing around numeric identifiers is rather error prone.

On *Figure 7* I present a diagram of the simplified architecture of *SDF-2D*. The aim of this chart is to provide a high-level overview on the structure of the internals. It is not a UML diagram and is also missing some classes / functions / objects that are present in the codebase. For example, the box named *Settings* on the diagram represents the *RuntimeSettings* interface, the *StartupSettings* interface, and their default values as well. Or, inside *drawables*, it looks like the factories inherit from the *Drawable* abstract base class; this is not precisely the case: these are just functions that return different subclasses of *Drawable*. I find it limiting to go into more detail or be more pedantic when providing the overview of the library.

The classes can be nicely separated into two main categories, the *drawables* and the *graphics*, while the latter into another two distinct groups: *rendering* and the *graphics-library*. The folders are also structured according to this separation.

On the overview, a component named *entry point* is also visible, containing two functions that the clients of the library mainly use to start interacting with it. Distance functions, object primitives, and light sources, i.e. the high-level rendering elements are all in the *drawables*. The *rendering* draws these using the capabilities provided by the *graphics-library*.

2.5.2 Components

Regarding the *graphics*, the low-level primitives can be found in the *graphics-library*, these are the wrappers that make it more convenient and less error-prone¹⁰ to interact with the WebGL API. To illustrate the level of abstraction it provides, I included the code of the *renderNoise* utility function in the *Appendix 6.1*. There, it is visible that the *ParallelCompiler* takes care of compiling all the shaders, the program queries the available uniforms, and handles the entire rendering process when its *draw* method is called (the values of the queried uniforms can be passed in here). Apart from these, the frame buffers (used for passing data between rendering phases), and textures are also implemented by this module.

The *FragmentShaderOnlyProgram* draws a screen sized rectangle on which the fragment shaders can be evaluated. Meanwhile, the *UniformArrayAutoscalingProgram*

¹⁰ Mainly, these facilitate the *DRY* [17] -ness of the codebase.

aggregates these programs and chooses which one to use just before it renders a given scene. Its decision is based on the number of *drawables* it is asked to draw.

The ***UniversalRenderingContext*** is a common abstraction over both WebGL and WebGL2 and it even handles context lost events in a clean way. The exact working of this is described in the *Implementation* section. Even though it is also located in the *graphics-library* folder, considering that most classes depend on and interact with it, I extracted it in the diagram to not clutter it with an excessive number of arrows.

Using the features provided by *graphics-library*, *rendering* can implement most application specific constructs. Handling the two render passes (distance and lighting, as explained in *Memoised distance field*) is one of its main responsibilities. A proxy can be found here as well, named ***ContextAwareRenderer***, which wraps the ***RendererImplementation*** (the most notable class of this section), and robustly handles when the latter loses its WebGL rendering context.

The calculation of some application specific uniforms, such as, the camera transformation, is governed by the ***UniformProvider***. ***FpsAutoscaler*** uses the public ***Renderer*** interface to execute its responsibilities. Also, the options (both compile and runtime) expected by the ***RendererImplementation*** are extracted into interfaces.

Lastly, the shaders reside in this module as well. The division of the classes is mostly straightforward, but there are two exceptions to this. The responsibilities of ***IntermediateFramebuffer*** are in-between of *rendering* and *graphics-library*, still it represents a WebGL framebuffer, thus I put it in the latter category. The place of shaders is also not cleanly defined, they could be either in *rendering* or *drawables*. Since *rendering* already contains shader snippets for distance calculations, and the shaders of the box named *shaders* are rather unlike these aforementioned ones, I opted for leaving them in the *rendering* module.

What is left is *drawables*, this module is the closest to the clients and it contains the actual drawable shapes. On top of that, a base class is defined here, from which all drawable must inherit. Additionally, ***DrawableBase*** subclasses also have to have a static property of type ***DrawableDescriptor***, in this, the exact manner of using and drawing the given object is specified. For objects with distance functions, the GLSL code of this function must be provided in this descriptor. Additionally, the expected number of occurrences in a frame (tile) has to be given too, so that the library can precompile shaders

to handle every situation. *DrawableBase* subclasses can optionally override a *distance* method to enable the tile-based rendering optimisation for that object type.

2.6 Implementation

The most challenging part of the implementation, aside from optimising, was the uniform 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 done with various mobile and desktop operating systems and browsers.

I take the view that statically enforcing types is key for creating reliable applications. It also helps the integrated development environment (IDE)¹¹ and overall enhances the developer experience. That is why I chose to implement the library in TypeScript.

In the following sections I gather the specific details of my software solution and the more challenging adversities I have encountered during my development process and the decisions made to mitigate them.

2.6.1 Dependencies

For understandable reasons, vector maths is heavily used in the library. For instance, when calculating distance fields, or converting between coordinate systems. Some of these calculations should happen on the central processing unit (CPU), that is why a high-performance vector math solution is needed. For that I have chosen *glMatrix* [15], which served me well, especially with its *mat2d* feature, which optimises 2D affine transformations by only storing a 2 by 3 matrix. Some bugs have arisen from the mutable handling of vectors and matrices, but the performance gains outweigh these drawbacks.

The project has two dependencies, the other being *resize-observer-polyfill* [16]. In short, it injects the resize observer API into older browsers that do not have it by default. The need for this comes from using resize observer to keep track of the actual size of the canvas onto which the renderer draws. It could be done using other, simpler methods, but it turned out that those had serious performance issues. Namely, possibly triggering a content reflow in the wrong time.

¹¹ In this case, I used Visual Studio Code.

Many other dependencies were used during development, but they are not required at runtime. The landscape of JavaScript libraries is wild and diverse, as a result I will only mention some of the more noteworthy ones. I created the build pipeline with *Webpack*, the transpiling (from TS to JS) is done with *Terser*. *Eslint* and *Prettier* is used for linting and formatting respectively. To generate the documentation from the comments in the code (that every public class and method has), *TypeDoc* is utilised.

Additionally, version control was provided by *git* and *GitHub*, the documentation is hosted on *GitHub Pages*, and the demo website is hosted via *Firebase Hosting*. The demo page also contains a miniature script for measuring performance, from that, insights can be gained on the real-world performance of the library. To log these metrics, *Firebase Cloud Functions* and *Firebase Cloud Firestore* is used. The charts I generated from these values were rendered using *ApexCharts*.

2.6.2 Universal rendering context

For rendering in the browser, first a rendering context must be obtained (in this case, from any available canvas element), which can be of multiple types depending on the capabilities of the specific browser and operating system. The rendering contexts that are relevant for my application are the WebGL¹² and WebGL2 ones. The context needs to be made available to most objects of my library, because interacting with WebGL is only possible through these. Two problems arise from this fact.

First, should this context be made globally accessible by every object? I do not think so. Shared state can be the root of many evil. Additionally, supporting multiple renderers should be a feature of *SDF-2D*. This is the reason I opted for passing the context object to most of my classes on their creation. To be more precise, I pass a wrapped context, due to the second issue.

Secondly, the API-s of the two WebGL versions are more or less identical, except the second version has more methods, and also some methods have divergent behaviour. For instance, the available extensions are very different between the two versions. Certainly, *WebGLRenderingContext2* has more and better features, still both versions should

¹² A *WebGLRenderingContext* can be acquired in two ways, either by using *webgl* or *experimental-webgl* as an argument for the *getContext* method. The latter is appropriate for legacy browsers and might not provide an acceptable WebGL implementation, thus should only be used as a last resort fallback. [31]

be supported to let us be able to support the widest range of devices¹³. Duplicating the *graphics-library* component of *SDF-2D* to achieve the support of both versions seems wasteful, not DRY, and most importantly, not maintainable. I solved this issue by creating a type called ***UniversalRenderingContext*** which is defined as the following TS type.

```
export type UniversalRenderingContext = (
  | WebGLRenderingContext & { isWebGL2: true })
  | (WebGL2RenderingContext & { isWebGL2: false })
);
```

Leveraging the advanced typing features of TS, this type makes it possible to easily use the common methods of both interfaces without knowing which version of WebGL we are using. When a WebGL2 feature is needed, checking the *isWebGL2* property lets both us and the TS compiler know whether it is possible to access that specific feature. When it is indeed false, a fallback solution can be implemented.

Additionally, an object containing insights about the state of the renderer is also piggybacking on the context object. This is included in the type used in the source code but was redacted from the above example for the sake of simplicity. This seemed the easiest way of making this object easily accessible but not global.

2.6.3 Context lost event

However, after obtaining a rendering context, we cannot take it for granted that we are going to have it for the rest of the page's lifecycle. A plethora of reasons can be behind a context getting lost during rendering. Oftentimes it is the result of the GPU getting overwhelmed, but it can also happen after a driver update finished in the background [17]. Although rare they be, their robust handling should be a feature of my library.

To accomplish this goal, I utilise two proxies. The first one is a JS concept called *Proxy* [18], introduced in the language quite recently. It makes it possible to trap property accessing calls. That is to say, that additional code can be ran transparently when a client tries to access the properties of some object. This is a truly powerful construct. The other proxy is the conventional and well-known design pattern.

The issue with the default context lost handling is that no exceptions are thrown, the WebGL API calls just start to return *null* values. If a context lost should happen, it is

¹³ Most notably, iPhones.

usually only found out later. Another concern is that when the context is finally restored, the state machine — that is WebGL — is completely reset, i.e. every single state changing API call should be repeated in this case. The easiest way of achieving this is to just re-create the whole renderer.

Fortunately, in JS it is possible to subscribe functions to context lost/restored events. When the former event happens the current rendering process needs to be stopped immediately. JS proxies come in handy in this situation. When obtaining the *WebGLRenderingContext*, a proxy is created, which has a flag signalling whether the context is lost. At that time a function subscribes to the *contextlost* event that sets the flag accordingly when the event happens. The context is wrapped in the aforementioned proxy which checks the flag before each property access, and when it is appropriate, throws an exception that propagates through the entire rendering process making it come to a halt.

The responsibility of finally catching this exception is of the *ContextAwareRenderer*. Which is actually a traditional proxy for the *RendererImplementation*, they both implement the *Renderer* interface, and *ContextAwareRenderer* passes each request to its contained *RendererImplementation* instance. Each operation is wrapped into a try-catch waiting for a context lost exception. When that happens, the methods still work, for example the result of update settings gets cached, and when the context is restored and after the new renderer has been created, the settings are passed onto that new instance. The other methods also have reasonable fallbacks.

An interesting (and worthwhile) journey led to achieving graceful context lost handling. I created a context lost simulator, which artificially makes the context lost then restored in short, random intervals. Then I let it ran and looked for errors, fixed them, and restarted the simulator ad infinitum. Some bugs were uncovered relating to globally shared state that did not seem to be that at first glance. In the end, clients of the library are saved from bothering with context lost events and apart from some dropped frames, they will not even notice it.

2.6.4 Setting up for rendering

To start the rendering process, first, some more initial setup is needed. This is orchestrated by the *RendererImplementation*. It acquires a context, then uses it as an argument for the following constructors. First, it creates the framebuffers, one for the distance field and one representing the current canvas (it is called the default frame buffer

in WebGL). The ***IntermediateFrameBuffer*** also creates the textures needed to be rendering targets. These frame buffers are then wrapped by ***RenderPass*** subclass instances. These operations are all done synchronously.

During the creation of many of the objects, asynchronous methods are needed. A constructor cannot be async, so async ***initialize*** methods are common in the codebase. ***RendererImplementation*** has such function as well. Its responsibility is to create the palette texture (used for cheaply and dynamically assigning colours to objects), creates the parallel compiler, then initialises the render passes with the given options and the to-be-used shader templates. The render passes create the autoscaling programs, which in turn create many child programs.

A sidenote for the parallel compiler. JavaScript applications are single-threaded. Multithreading can only be achieved by using subprocesses (workers). At first, the parallel compiling does not seem to be advantageous, however it is not actually done by JS, but by the browser and the GPU driver, which, fortunately, can utilise multiple threads. That is why it is much faster to compile this way, provided the appropriate software/hardware support is present.

In short, these are the more interesting happenings occurring during a ***compile*** or ***runAnimation*** call. The machinery is now ready, and the rendering can start by giving some ***Drawable*** instances to the program.

2.6.5 Drawables

For the library to be useful, it is of utmost importance to make it easily extensible. The most probable subject of extending seems to be the *drawables*. In short, these contain the renderable distance fields and the specifics of how to render them. A simple example for the definition of a circle-shaped drawable object (which, by the way, is coloured opaque yellow) could look like the following.

```

class Circle extends Drawable {
    public static descriptor: DrawableDescriptor = {
        sdf: {
            shader: `
                uniform vec2 circleCenters[CIRCLE_COUNT];
                uniform float circleRadii[CIRCLE_COUNT];

                float circleMinDistance(vec2 target, out vec4 color) {
                    color = vec4(1.0, 1.0, 0.0, 1.0);
                    float minDistance = 1000.0; // some large number

                    for (int i = 0; i < CIRCLE_COUNT; i++) {
                        minDistance = min(
                            minDistance,
                            distance(circleCenters[i], target) - circleRadii[i]
                        );
                    }

                    return minDistance;
                },
                distanceFunctionName: 'circleMinDistance',
            },
            propertyUniformMapping: {
                center: 'circleCenters',
                radius: 'circleRadii',
            },
            uniformCountMacroName: 'CIRCLE_COUNT',
            shaderCombinationSteps: [0, 1, 2, 3, 8, 16],
            empty: new Circle(vec2.create(), 0),
        };
    }

    public minDistance(target: vec2): number {
        return vec2.dist(this.center, target) - this.radius;
    }

    protected getObjectToSerialize(transform2d: mat2d, transform1d: number) {
        return {
            center: vec2.transformMat2d(vec2.create(), this.center, transform2d),
            radius: this.radius * transform1d,
        };
    }
}

```

At first, it might look intimidating to the uninitiated, yet, basically, it only contains two distance functions, and a way of transferring data from the central memory to the GPU memory. Additionally, some other data is present, namely *descriptor.empty*, and *descriptor.shaderCombinationSteps*. These are discussed in the next section.

The distance function must be specified as a GLSL function in order to make it interpretable by the GPU. For extensibility, not only the distance function needs to be specified but also the method of joining the different instances of the objects. This solution enables, for instance, the use of smooth merge algorithms. To please the aesthetic needs, colour (or its palette index) also has to be provided here.

Next, some metadata is required in order to transfer the data to the GPU. For that, a mapping has to be given detailing the name of the properties of the values returned by *getObjectToSerialize* and the names of their aggregating counterparts in the GLSL code. Passing these values is only supported for arrays by default. Nevertheless, it is not strictly enforced, because by overriding the *serializeToUniforms* method of the base class, it is possible to implement arbitrary uniform uploading logic. Still, for speedy uniform passing, it is desirable to use arrays.

Addressing the issue of floating-point precision is essential. 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. As 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 to *getObjectToSerialize*, but the appropriate transformations need to be handled by the **Drawable** subclasses.

Lastly, *minDistance* does not necessarily have to have an actual implementation. If it returns 0 or anything below, the visual output of the rendering will stay unchanged. Giving it a sensible implementation is beneficial because of tile-based rendering. Stemming from this, it can be a lower-bound of the actual distance field, and it may sometimes be advisable if calculating the exact value would be too expensive.

I implemented some **Drawable** subclasses which come with my library. The distance functions used were mostly derived by *Quilez* [19]. Using these out-of-the-box classes, it is reasonable to use the library without having to write a single line of GLSL.

2.6.6 Autoscaling

Finally, let us look at the last step of the rendering process. The main appeal of my graphics solution¹⁴ comes from its tile-based approach to rendering which increases its performance greatly. This, combined with the use of shaders that are optimised for the

¹⁴ Apart from the memoised distance field, which can be quite easily implemented using the features provided by WebGL.

number of onscreen objects is one of the main concerns of the *rendering* component. The former is handled by the ***DistanceRenderPass*** and is quite straightforward; the location of the tiles are calculated alongside the objects in their close vicinity, then a matrix is sent to the program specifying the region it should draw into.

By autoscaling, I mean the decision-making process of choosing the appropriate shader for each tile. For that, first, multiple shaders have to be compiled. This needs to be done beforehand as to not hinder the performance later. The current implementation generates every permutation of the *shaderCombinationSteps* specified for each drawable. Having an exponential complexity, this method does not scale well for many objects having a large number of *shaderCombinationSteps*¹⁵. With careful planning I did not find it hard to keep the number of generated shaders under a couple of hundred¹⁶.

Some macro substitutions are done by my library when generating the shaders. Through this, the number of to be drawn elements can be specified for the GLSL code. The substitution pipeline is also used for other purposes, such as the injection of colour constants and rendering-related constants as well.

The shader having the fewest drawables that can handle the current scene is chosen and used during rendering. If there are too many objects, a warning is written to the console and the largest shader program is used as a fallback. Thanks to this and the preceding sections, most of my library has been covered. At least the intriguing parts.

2.6.7 Extensions

On a related note, I digress from the previous train of thought. To increase the visual quality, rendering speed, and visual performance multiple extensions are used. Some of these may not be available in every context. The following fallback methods are used to handle the extensions' lack thereof.

Extension	Fallback
EXT_disjoint_timer_query_webgl2	the stopwatch is disabled

¹⁵ Mitigating the large growing shader count could be done by using a different render pass for each object, but that would incur an immense performance hit.

¹⁶ Looking at this number, it becomes apparent why I had to implement parallel shader compiling.

WEBGL_lose_context	it is expected to be available [20]
OES_vertex_array_object	it is expected to be available [20]
EXT_color_buffer_float	rendering is done to a RGBA8 buffer
OES_texture_float_linear	rendering is done to a RGBA8 buffer
WEBGL_debug_renderer_info	no renderer info is provided in the insights object
KHR_parallel_shader_compile	shaders are compiled synchronously

Figure 8 Fallbacks for used extension

Although it works great for extensions, handling the lack of WebGL2 cannot be summarised in such a neat table. For that, multiple shader variants, macros, and alternative function implementations are used. Nonetheless, the implementation revolves around the `isWebGL2` property of the context introduced in section 2.6.2 *Universal rendering context*.

2.6.8 Demo page

As I have already mentioned, a demo page comes with my library. It serves three purposes. Mainly, it showcases the capabilities and performance of *SDF-2D*, additionally it also gathers data enabling the improvement of this performance. Lastly, testing shaders is not an easy challenge to undertake, and at the small (single-developer) scale of my operation, manual testing seems to be most suitable one. Usually, while working on the library, I use the demo page to immediately see the results of my changes.

The demo website is implemented in its own NPM package, which has *SDF-2D* as a dependency. For the ease of local development, it is possible to link the latter to the former with the `npm link` command. After that, if the build tool (`webpack`) of both packages is running in watch mode, a change to a file in the graphics library triggers the rebuild of the whole library, which triggers the rebuild of the demo page, which automatically reloads the browser tab containing the website. Though it takes around 3 to 5 seconds for these changes to propagate, it still provides me with a usable development experience.

For deployment, *Firebase Hosting* is employed. On every push to the main (master) branch, a simple script builds the webpage and pushes the generated artifacts to *Firebase*, which serves it over a content delivery network (CDN) with nodes all around

the world. As a sidenote, I experimented with various hosting solutions, however, *Firebase* turned out to be the winner in most aspects.

2.7 Limitations

Because uniforms are used for passing data related to onscreen objects, and the number of available uniform can be limited by the GPU, the operating system, the graphics driver, and even the browser, their number may not be enough for rendering a really complex scene. Although such complex scenes should be avoided for other performance reasons, this is still a limitation. A solution could be the use of textures when passing the objects, but that might incur some pricey overhead when reading from them.

Another solution could be the use of a non-uniform tile grid. Changing the size of a tile is basically free; accordingly, more, smaller tiles could be used on the complex parts of the scenes, where are more objects, and fewer, larger tiles could be used elsewhere. Not only would this make it possible to render (in theory) an infinite number of objects, but could also lead to some performance increase as well. In a later release of *SDF-2D*, this feature might be implemented.

Apart from this, custom lighting shaders cannot be defined, and currently there are only two lighting models (a circle area light, and a flashlight-like directional one). These light sources are also optimised for low-performance devices. More complex and custom lighting models should be available to the library's clients. Mitigating this issue is also on the roadmap.

I also find the banding effects surrounding the lights distracting. It could be solved by using some kind of dithering, but at first sight it seems overly expensive.

2.8 Results

2.8.1 Visual

Overall, I am satisfied with the results. During the development of my videogame, when I used the API of my package, I found it both intuitive and powerful. Regarding the visual quality, I am more than pleased; to illustrate the capabilities of *SDF-2D*, I present some screenshots below.

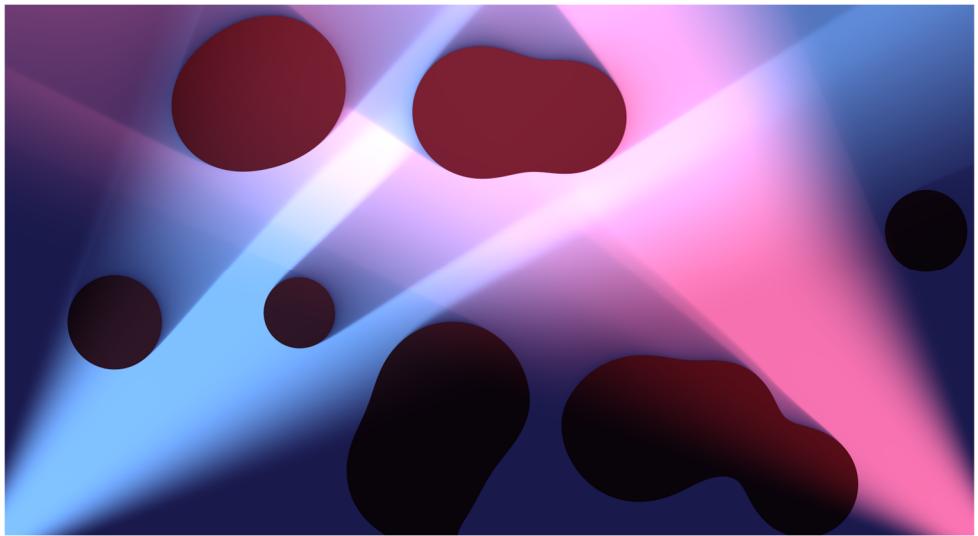


Figure 9 Screenshot of a demo scene from sdf2d.schmelczer.dev

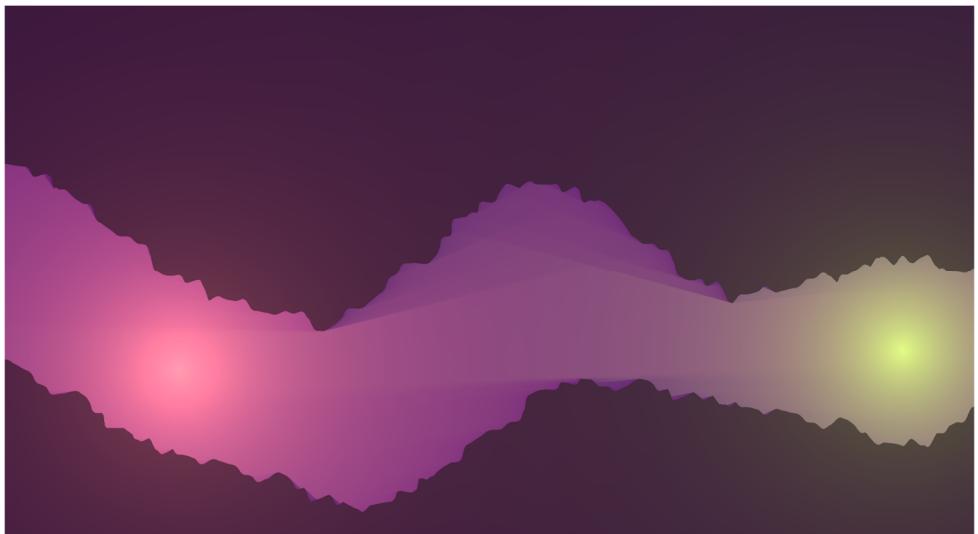


Figure 10 Screenshot of a demo scene from sdf2d.schmelczer.dev

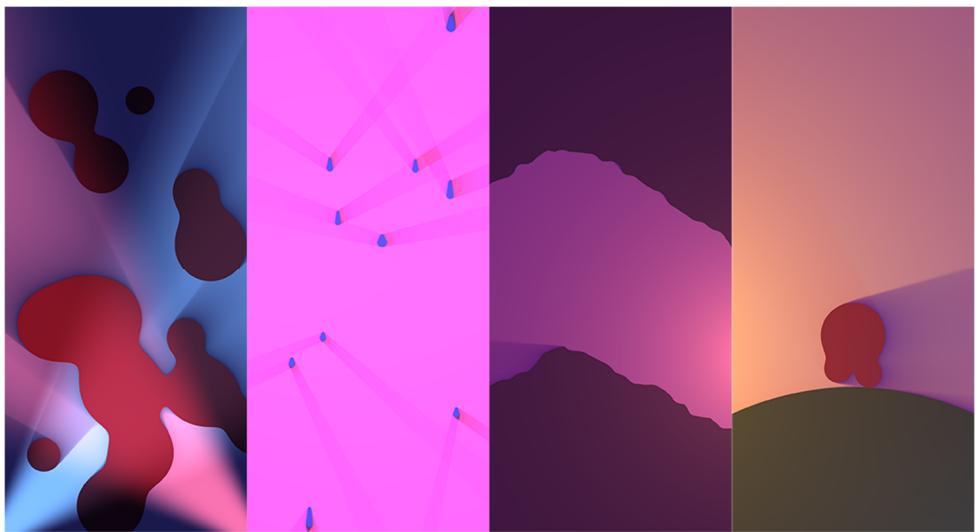


Figure 11 Screenshots taken on a smartphone

2.8.2 Performance

The aforementioned *Demo page* contains a script for logging; it sends hardware and software information (anonymously) when the page loads and then after each scene it sends the key performance metrics: drawn frames per seconds (FPS), and the quality settings of the framebuffers. The latter is automatically set by the default *FpsAutoscaler*.

Below I present some charts generated from the acquired analytics data after some pre-processing which included a formatting script for human-friendly renderer¹⁷ / user agent parsing, deduping and the sorting based on the values. The data gathering was done using the organic visitors of the website, by asking my acquaintances to visit it, and by going into an electronics store and trying the webpage out on each device. The latter could have been replaced with an online testing tool but for testing on physical devices (which is a requirement when assessing real-time rendering performance) testing companies set the price too high.

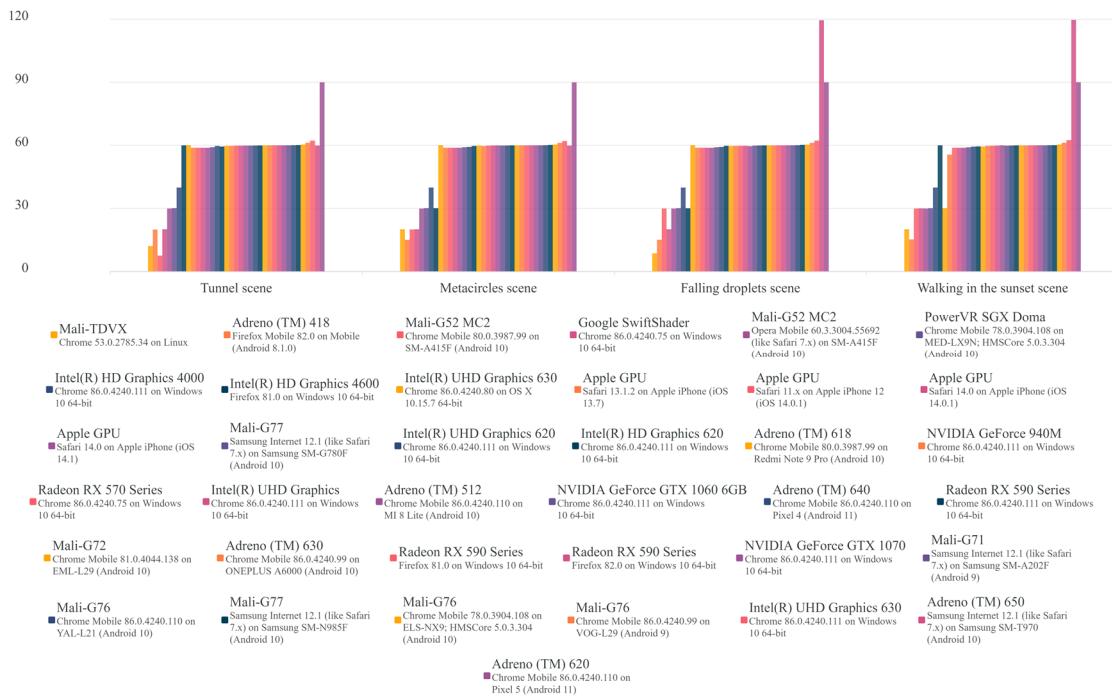


Figure 12 Frames per second

For rendering, the *requestAnimationFrame* API is used which caps the FPS at the display's refresh rate (it is usually 60 Hz). Two outliers are visible on *Figure 12*, a

¹⁷ The name of the renderer (oftentimes a GPU, see later) is reported by the *WEBGL_debug_renderer_info extension* [29].

Samsung tablet, and the *Pixel 5* smartphone. These have 120 Hz, and 90 Hz displays respectively. The FPS stayed around or above 30 frames per second for 33 out of the 37 unique devices. For non-interactive content, this is adequate, but the 60 FPS limit was reached by more than three-quarters of the tested devices on every scene, which makes me confident in the capabilities of my software solution.



Figure 13 Distance field scaling

Of course, FPS is not everything. The exact purpose of the *FpsAutoscaler* is to sacrifice the quality to achieve at least 50 drawn frames per second. This can be done either by changing the resolution of the first rendering pass, or the second (or obviously both). First happens the rendering of the distance field, which need not be done at full resolution to provide excellent visual quality as illustrated by *Figure 1*. For this reason, the autoscaler sets the initial scaling to 0.5, where — subjectively — no visual artifacts are present. The demo scenes only last for 8 seconds, hence, the values are not quite stable after this few samples, yet it is pleasing to see the distance scale, presented on *Figure 13*, staying around its initial value.



Figure 14 Scaling of final framebuffer

Finally, the most meaningful chart. The visual quality correlates closely with the overall scaling. Its value starts from 1.0 and drops sharply if the FPS values are not adequately high. Almost every desktop GPU was able to render the scenes at the highest quality, and overall, few renderers had to go through a large quality loss. To put the values in context, most tests were ran on smartphones, for which it is not unusual to have screen resolutions well over FullHD. Although this combined with their relatively lower performance hardware results in decreased quality, their smaller physical screen size still makes it possible to enjoy a non-pixelated result even at lower rendering resolutions.

As a sidenote, two interesting records can be found in the charts above, firstly a CPU-based renderer (*Google SwiftShader*), which has the performance we expect form it¹⁸. Secondly, the GPU of a modern 4K TV makes appearance as well, which for some reason runs a Google Chrome version from 4 years ago.

Using the data gathered, I modified the *FpsAutoscaler* to achieve a higher overall performance on the lower end of the spectrum. I lowered the target FPS to 30 to stop the

¹⁸ It would be possible to disable rendering on such renderer using *failIfMajorPerformanceCaveat*, but having a low-quality output is still better than having no visual output at all.

quality from extreme deterioration. I also implemented motion blur to make the movement more fluid even at lower framerates. Additionally, I tweaked with the steps and starting values of the autoscaler. After some initial measurements with my modified setup, it seems, that end-user experience has improved. After all, the performance logging proved to be useful.

To conclude, it was never a goal of *SDF-2D* to run perfectly on every device. Though, it runs adequately on most of them and exceptionally well on modern appliances. The performance could be further improved, for instance with the methods detailed in *Limitations*. Still, I am more than satisfied with the results achieved so far.

3 The videogame

First of all, to be able to easily reference my videogame, a title is needed. I stumbled upon a great name when I was searching through the list of English words ending in a valid top-level domain (TLD) that are also not yet registered as domain names. Turns out, *red* is a valid TLD, and *decla.red* had not been registered, so I registered it. The game is available at <https://decla.red>, and to ease the confusion, the game is also titled *decla.red*. In the following sections I describe the mechanics, architecture, implementation, and details of the videogame *decla.red*.

3.1 State of 2D games

As I have mentioned earlier, I was dissatisfied with the current state of the 2D videogame landscape. Flat textures without illumination effects are rather prevalent. Certainly, there is no issue with this method, on the contrary, it has little to no performance restrictions, and is also easier to interpret for the user, however, it does makes it easier for the game to lack uniqueness. Though, this blanket statement cannot be applied to each and every 2D game, for instance, games like *Thomas Was Alone*, and *Ori and the Blind Forest* are some noteworthy exceptions. Still, the exceptions are the rarer.

As for multiplayer browser games, the more well-known ones are the *.io* games, i.e. games of which domain has the TLD of *io*. These games also represent the massively popular multiplayer browser games. To name a few, *agar.io*, *slither.io*, and *hole.io* are quite known. Although they can be entertaining to play, they too have the visual symptoms described above. Nevertheless, their multiplayer capabilities increase their enjoyability by orders of magnitude.

My game is unlike any other I know of. Obviously, this does not mean it is better than them. Combining existing mechanics and experimenting with new ones is a great experience for the programmer but does not automatically guarantee a great game. To be honest, stemming from the fact that I am not a game designer, I do not find it strictly necessary to have in-depth game design considerations in my thesis. Nevertheless, considering the amount of effort involved in creating this game, the least I can do is to make it as exciting for the players as I can.

The issue of the visuals is solved by using *SDF-2D*. After I came up with the distance functions for some game object, it will instantly look great in-game. Nonetheless, the matter of multiplayer and overall enjoyability still needs to be tackled.

3.2 Game mechanics

Without a doubt, I was inspired by the titles listed earlier. Yet, my chosen design was also heavily influenced by 3D shooter games, like the *Battlefield* series. The prospect of competitive multiplayer is enticing, to triumph over other human players is an exhilarating experience. Subjectively I miss the lack of teamwork when everyone is against each other. Hence, my game has two teams. This also has the added benefit of making the gameplay more rewarding for individuals, since around half the players come out victorious each round. In the *.io* games, combat is mostly done by devouring one another, which is less spectacular than shooting. I opt for spectacularism.

To provide a goal other than shooting everywhere, a conquest style mechanic is used. The game is set in space, inside a dense asteroid field; each asteroid can be conquered by a team. Conquered asteroids generate points over time. To balance the game, killing opponents also gives some points for your team. Unlike in the aforementioned shooter games, *declared* should have much more conquerable places; in the range of hundreds. However, teams should be discouraged from going to a far-off place of the map and sneakily conquering some asteroids there, while the other team has little chance of finding these hidden places. For this reason, under less than half a minute, the asteroids stop being conquered if there is no one near.

Another aspect I consider uncommon in existing games are the kill/death counts shown next to the names and health bars of characters. It facilitates decision-making when considering whether one should opt for fight or flight in an encounter.

In many games, to rate-limit the number of shots a player can make, a cooldown period has to elapse after each shooting action. I find it frustrating to not be able to shoot whenever I want to. Certainly, a lack of rate-limiting can be exploited by pressing the shoot button in fast succession. Solving both issues can be done in a simple way. First, each character is given an “energy level” that has a maximum, and when lower than this maximum, increases linearly until it reaches it. Then, each shot has a “strength” equal to the half of the initiating character’s current “energy level”, and also halves this level. This

way the amount of possibly inflicted damage stays mostly the same regardless the shooting frequency, whilst the players are less restricted in their actions.

There is an interesting caveat to this approach. Shooting as fast as possible is still slightly beneficial, yet I would like to deter users from it. The solution is to apply a linear decrease over time to the “strength” of these projectiles. The less one shoots, the less their possibly inflicted damage (which comes from the remaining strength of the projectiles when they hit the opponents) is hindered by the accumulated drag.

As a final touch, “wonky physics” should also be utilised. Physics laws as we are used to them, need not apply here. After some experimentation and user feedback, I concluded that the projectiles bouncing off surfaces but not affected by gravity, while the players affected by gravity (that is much stronger than expected from the size of the asteroids) result in optimum satisfaction. Also, the legs are attached to the player mimicking a spring joint but this just for aesthetic pleasure.

As for the user inputs, on PC, movement should be initiated by keypresses, while shooting and aiming with the mouse. On touch screens a dynamic joystick should be used for controlling the character, while the second touch should be interpreted as a target for the next shot. As mobile users are also targeted and chatting in-game is not really a convenient option for them, chat will not be enabled. Besides, the conversations have a tendency to become overly negative and toxic in some games.

3.3 Architecture

Players have high expectations when it comes to online games. Multiplayer titles are expected to have great visuals, minimal latency, fast loading times, and also have to be fun to play. These are demanding functional and non-functional requirements. In satisfying them, the architecture has a decisive role.

Two popular methods of hosting multiplayer games are *peer-to-peer (P2P)* [21] and traditional client-server based approaches. The former scales better and requires considerably less hardware from the publisher, while the latter has superior performance. I prefer the latter.

3.3.1 Overview

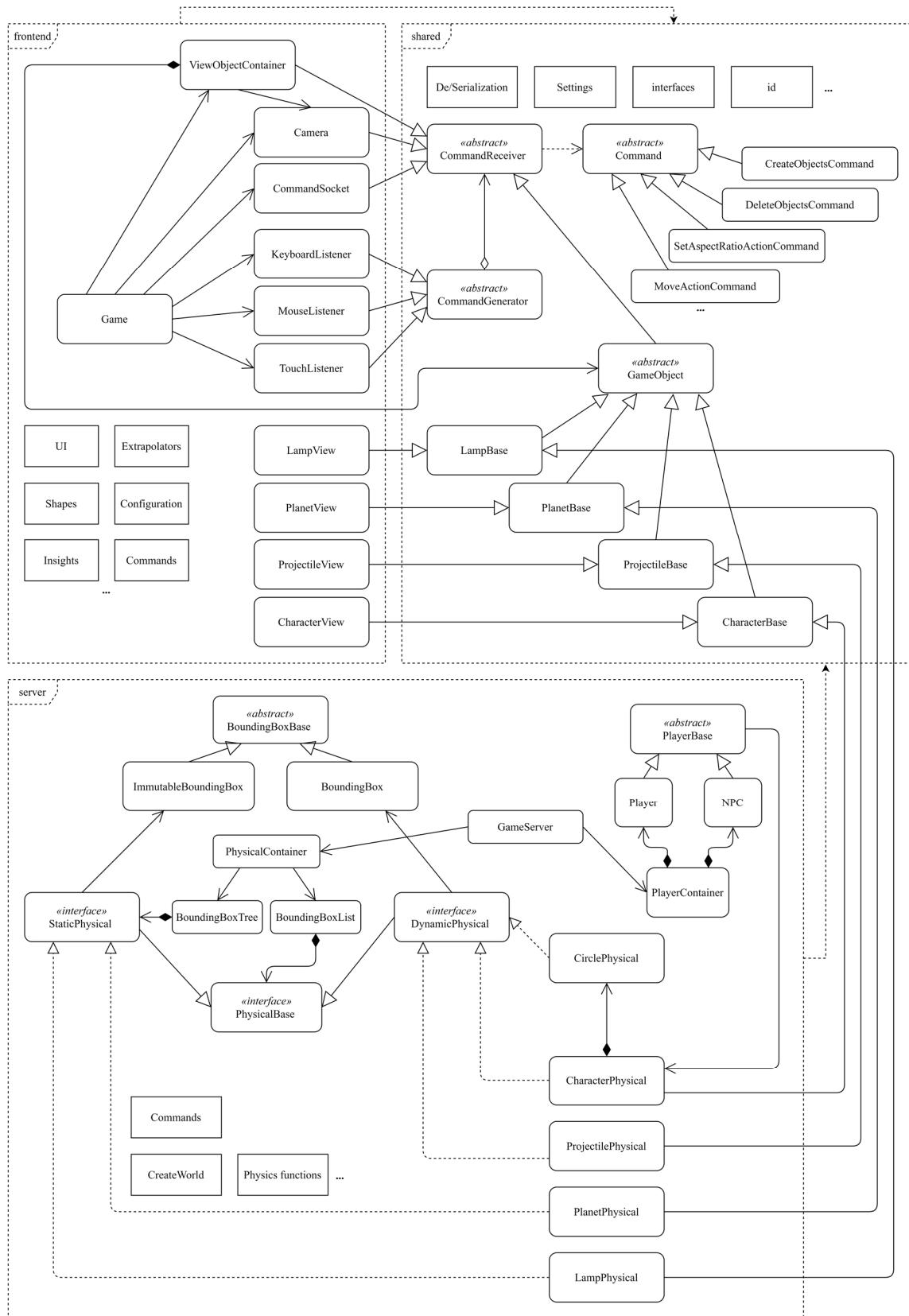


Figure 15 Overview¹⁹ of the architecture of *decla.red*

My architectural choice clearly separates the client and the server. The client is the thinnest it can be, while most calculations happen server-side. To facilitate communication between them, they share the *shared* library. This way, I have three NPM packages, namely: *frontend*, *shared*, and *server*.

The *server* is a Node.JS application, thus it is possible to easily reuse the codebase of *shared*, which is a library package. Lastly, the *frontend* is also an NPM package having a website as its build output.

3.3.2 Shared

Starting with the library, the following responsibilities become visible; the command handling constructs detailed in *Commands*, the base game objects defining the models of the possible entities found in the game, the de/serialisation facilities, and settings, helpers, and interfaces easing the coordination between the packages.

GameObject is the base of every game entity. Its uniqueness is guaranteed by its *id* property, which should be generated only by the server²⁰. Some convenience methods are also located here related to remote procedure calls from the server-side to the client-side version of the same object. Each game entity is defined here as a subclass of ***GameObject***. Other than their properties and methods required for serialisation, they do not really contain much more.

Lastly, the constants of the game, such as the size of the character or its maximum health is also located here, since both the server and the client may depend on these values. Every noteworthy configuration is aggregated in the settings object. Alongside it, the interfaces used for non-WebSocket communications can also be found here.

3.3.3 Frontend

To save on both CPU and download time, the website is — architecturally speaking — thin. Making use of *SDF-2D*, it does not have a considerable amount of standalone code. The user interface (UI) related items are not detailed in the overview diagram. Apart

¹⁹ I stand by my remarks — regarding the accuracy of the diagram — detailed in 2.5.1. In this case, the non-shared Command subclasses are missing, like ***StepCommand***, and ***ReactToCollisionCommand***.

²⁰ This requirement could be partly avoided by using universally unique identifiers (UUID-s) instead of an incremented integer value, but I see no added advantage in switching to UUID-s.

from handling the server finding and joining tasks, it only stores the objects got from the server, and updates them accordingly. Using the features of the de/serialisation solution, it loads the game objects into view objects. Each game object has a view object counterpart and the latter expands its command reaction capabilities by handling the render command. To compensate for changing network environments, it does some extrapolation as well.

As of yet, only linear inter/extrapolation is used. Since the future actions of other players cannot be predicted, a linear extrapolation of their state seems adequate. A more complex, physics-based system would not give overly better results. Besides, with this solution there is no noticeable lag, thus I conclude my solution viable.

Additionally, the inputs acquired from the player are also processed by this module. The processing merely consists of deriving a tangent vector; all else is handled server-side. It can be argued that this reduces the possibility of cheating, but the real reason behind the simple frontend code is the desire to have a clean architecture.

3.3.4 Server

Physics and game mechanics. This is how I could describe this package in four words. Each *GameObject* subclass has a *physical* (implementing *Physical*) subclass. These objects are stored efficiently in a four-dimensional *k-d tree* [22]. More precisely, their axis-aligned bounding boxes (AABB) are stored in a tree structure that make it possible to query the objects possibly intersecting with some other AABB. This high-performance querying mechanism is crucial for providing real-time physics even on lower-end server machines.

Simulating the interactions of the physical objects and deriving the points of each team is initiated and handled by *GameServer*, for calculating the physics, some helper functions are available. Apart from the WebSocket endpoint, another one is also available for querying the current state of the game. It also keeps a list of players and non-player characters (NPC). The artificial intelligence of the latter operates based on a simple table of rules.

The player count limit, NPC count, and basic variables related to the gameplay (world size, point goal) can be given as command line arguments. This prospect of customisation helps in giving the servers a unique feel.

3.4 Collision detection

There are two kinds of objects in the game: static and dynamic. The latter can move, the former cannot. Most game objects have a fixed position, for instance, the asteroids and lights stay in place during a game. If we want to detect collisions, it is enough to only check for them during the movement of a dynamic object.

I decided to use a k-d tree based storage for static objects, and a simple list for dynamic ones. This way, the tree does not need to be reorganised after each physical frame, while — coming from the large number of static objects — it can still provide significant performance gains. Querying possibly intersecting objects is quite simple; after receiving the AABB of the potential collision's other party, the k-d tree is used to fetch the static objects which's AABB intersects the previously mentioned AABB. The result of this is merged with the entire list of dynamic objects and finally it is returned.

Collision detection is only implemented for circular dynamic objects (which can collide with any other object, even non-circular ones). It is appropriate, since each character is made up of three circles, while projectiles are just a single circle, therefore there is no need for any other kind of collision detection.

Before a circle is moved, its possible future state is calculated. The future circles' AABB is computed, the objects possibly intersecting with it are queried, the value of the SDF generated by these shapes is evaluated at the future centre of the circle. If this value is smaller, than the radius of the circle, a collision happens²¹. On collision, both objects are notified of the event. An approximation of the collision's normal also gets calculated by numerically differentiating the SDF.

The efficiency of collision / intersection detection is of utmost importance. From profiling measurements, it seems that most of the time is spent on querying and fetching objects. Developing the rest of the server-side logic becomes easy utilising this method. Apart from its main purpose, which is related to moving objects, it is also used for keeping the objects known by each client in sync with the position of their camera, for calculating the gravity in a point, and even for creating new characters in an empty position.

²¹ This method is only accurate when the SDF is exact, which it is in *declared*.

3.5 Multiplayer

3.5.1 Protocol

State has to be transferred over the network between the server and the clients, and vice versa. Our options are rather limited when it comes to the available methods. There are two options for communication, apart from polling style Hypertext Transfer Protocol (HTTP) connections (which are just not adequate for a real-time experience): WebSocket and *WebRTC* [23]. The latter has some real potential, it mostly uses User Datagram Protocol (UDP) and is aimed at P2P, especially multimedia, applications. The former uses a special feature of HTTP that makes it possible to keep a connection open indefinitely.

I believe both solutions are great, but I have more experience with WebSockets, also I chose not to have a P2P multiplayer, hence WebSocket seemed more appropriate to me. Besides, having no lost messages and reordering helps to keep the complexity lower as well. To be more precise, I use a wrapper library, namely *Socket.IO* [24]. It adds various features, such as rooms, and automatic reconnection, additionally, a fallback solution if WebSocket is not available.

3.5.2 Matchmaking

To prepare my videogame for fluctuating demand, multiple servers can be chosen by the users. The addresses of these servers should not be hardcoded into the frontend. Though, due to the swift continuous deployment cycle, it would be acceptable. It is still nicer to be able to instantly update the list of game servers. For achieving this, *Firebase Remote Configuration* is used.

Originally, I planned on creating a simple solution from scratch, however, since I am already using *Firebase* for hosting and data gathering, it was a reasonable decision to try out the *Remote Configuration* service. It works as expected, has a nice UI, propagates the changes instantly, and is simple to access from the frontend. Although I may not be satisfied with the large NPM package that has to be included to access this feature, in return it saved me many hours of development. For shipping a minimum viable product, it is an ideal solution.

Using this, the server search starts by querying the server list, then a connection is initiated to each server's *status* endpoint, if it responds, the returned values are used to

assemble a card for the server. It contains its name, progress of the game, player count, and player limit. Additionally, a WebSocket connection is initiated towards each server, making the updating of these values dynamic. After the user has decided their preferred server, these connections are dropped and a new connection is created with the chosen one, and with that the game session starts.

With a small enough player base the continuous updating of server information is feasible with the current setup. It is glaring, that $n_{player} \cdot n_{server}$ connections are open as a result of this method. This complexity could be reduced to just $n_{player} + n_{server}$ using an intermediary server just for aggregating and serving the game server information. The downside of that would be the single point of failure and possible bottleneck introduced in the form of the intermediary. These could be then mitigated by using a flat array of multiple aggregators, or some more sophisticated hierarchical structure. As of yet, spending resources of coming up with a service like this would not have an acceptable return on investment. If the player base should grow exponentially, this topic might need to be revisited.

3.5.3 Serialisation

Marshalling JavaScript objects is as easy as calling `JSON.stringify` on them. The issue is that some information is lost, namely the objects lose their prototypes²². This means that, for instance, they lose their methods defined in their classes and base classes. The `instanceof` operator also stops functioning properly after deserialisation. This is not surprising; no type information was output by `JSON.stringify`. By default, it is also not possible to do so.

Although multiple JS serialisation solutions exist, for example, `class-transformer` and `serializr`, I did not find them sufficient for my purposes. Minimising the breadth and depth of the `shared` package is a perfect way to reduce both coupling and build size. This holds true even for the model classes. Using the aforementioned solutions, it seems cumbersome to serialise a class instance, then deserialise it to another instance of another class as shown on *Figure 16*. It would make the codebase more concise if this was possible.

²² Additionally, some values cannot be serialised, for instance `undefined`, `Nan`, or functions. Though that comes from the restrictions of the Javascript Object Notation (JSON).

After some consideration, I concluded that it is not unreasonably difficult to create a basic solution satisfying my needs. In the following, I detail my approach.

A class (having a `toArray` method) can be marked with the `serializable` decorator as a class that has special serialisation behaviour. The `serialize` function is an extension of `JSON.stringify`, and it can create the textual representation of any object, not just `serializable` instances. However, such objects are handled specially. They get output in the following format `[object.type, ...object.toArray()]`. Any level of recursion is supported, and there are no restrictions on the top-level objects, they need not be `serializable`.

The property `object.type` gets injected automatically by the aforementioned decorator. It is actually not called `type`, instead it has a mangled name to avoid name collisions. The decorator also adds a record to a map containing type names and their respective constructors.

The inverse is done by `deserialize`, and it is an extension of `JSON.parse`, except it looks for arrays having a string as their first element, which also match a type name in the type-to-constructor map compiled earlier. If that is found, the constructor gets called with the `new` operator, and is given the rest of the array as its arguments. That is why it is critical for `toArray` to return the appropriate number and type of values that the constructor expects.

Two methods are available to achieve the desired manner of operation, the `serializesTo` decorator, and the `overrideDeserialization` function. The former acts similar to `serializable`, but also expects a class as its argument and the injected type property will have a value derived from that argument. The latter just replaces a constructor in the type-constructor mapping with the one given as its argument.

A real example from the codebase can be seen below. `ProjectileBase` is in the `shared` package and ultimately ends up in the artifacts of both the `server` and the `frontend`. It is imperative, that the name of `ProjectileBase` cannot be changed by the build pipeline, but it is easy to disable this optimisation step and its repercussions are nearly non-existent if compression is used when serving the files.

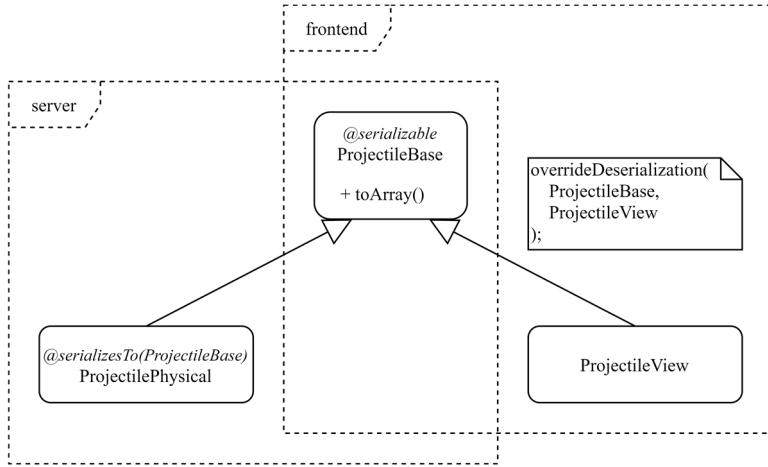


Figure 16 Serialisation from the client's point of view

The following can happen in the example: someone calls *serialize* on a **ProjectilePhysical**, transfers the resulting string to the *frontend*, there *deserialize* is called with the aforementioned string as its argument and it will return a **ProjectileView** instance. This alone can already be quite helpful, but combined with recursion, plain objects, and arrays, it amounts to a great developer experience.

To show the full picture, in the case of a plain array that by chance has a first element that is the name of a *serializable*, that array cannot be deserialised. I do not find this an issue, because I do not even send plain arrays. On the other hand, my solution requires minimal bandwidth, is very slim, and to tell the truth, was genuinely exhilarating to work on.

3.5.4 Server deployment

To make the game playable, servers are required to run the *server* package. Although the servers do not have tremendous resource requirements, some performance is required from them. I tried many approaches, and ultimately ended up with the simplest solution.

My journey started with creating a *Docker Swarm* cluster with virtual machines rented from a cloud provider. It worked great, except the management of the cluster had such overhead that required too many and too high-performance (compared to my budget) machines making it an expensive solution. Then, I experimented with the *DigitalOcean App Platform*. With minimal effort it enables the continuous deployment of docker images. Turns out that even though the price of a deployed container is the same as of a virtual machine, it has magnitudes worse performance.

Ultimately, I created a simple script that can be ran on a fresh *Linux* virtual machine that sets up a server. This solution is the cheapest, provides the greatest multiplayer fidelity, but on the other hand requires some (though minimal) manual action. It consists of a *NGNIX* reverse proxy handling the transport layer security, *certbot* to automatically update its certificate and, above all, the NPM package for the *declared* server.

3.6 Commands

Commands are used almost everywhere in the codebase. They are an abstraction of function calls, which at first sight might seem like needless overengineering, fortunately, they are not. In short, a command wraps a function call into an object; it contains its abstract function type (for instance, *MoveCommand*, or *RenderCommand*), and its arguments. Using these wrapped function calls, it becomes possible to, for example, send them over the network. It is also beneficial from an architectural point of view.

Imagine keeping a heterogenous collection of game objects, out of which we would like to render the ones having a visual representation. Let us say that rendering is done by calling an object's *render* method. A problem may arise from the fact that not every object has such method. This could be mitigated by the base class having an empty *render* implementation, which can be overridden by the renderable objects. Or somehow the capabilities of the object could be queried, for instance, if they have a method named *render*, they must be renderables. Separating the objects based on capabilities is also a solution, but then we would not have a heterogenous collection and that contradicts our assumption.

The architectural problems with the previous approaches are the following: in the first case, we might not be able to find a common base class (as is the case in my game), and since multiple inheritance is not a JS feature, we might be out of luck here. The next best solution is to define an empty implementation for every possible capability in every single class which is far from a well-engineered approach. Querying capabilities is not at all a terrible idea, however, implementing it in a way that avoids false positives is not straightforward (though, by using JS Symbols, it is doable). Besides, it would result in subjectively unpleasing class diagrams.

Another solution can be the *Visitor pattern* [25], but it would require both a stable class hierarchy and the separation of model and methods, which is not in line with my design approach. A document object model (DOM) event like system could also be used,

however, it has the downside of requiring subscribers not only to depend on the events, but also the providers / generators of these events as well.

In conclusion, the aforementioned approaches all have some shortcomings in the context of my application. This motivated me to come up with another solution. In the end, I decided to use a *Smalltalk*-like message passing mechanism [26] implemented by **Command**, and **CommandReceiver**. Every object capable of handling commands has to be an instance of a subclass of **CommandReceiver**. Subclasses can override a property and a method; the former is the mapping of command types to the actual implementations of the command handlers. The latter is the default action that should be taken when no appropriate implementation is found for the given command. Sending a command to a receiver is as easy as calling its *handleCommand(c: Command)* method.

It allows the objects to only depend on the **Commands** they actually use, the command generators not to worry about the callees' capabilities, and the network code to effortlessly serialise function calls. In my experience, its sole drawback is the lack of built-in return value handling. For a **Command** “call” to have a return value, a new command has to be defined and instantiated by the callee.

3.7 User interface

One of the most frustrating aspects of using software can be poor-quality user experience (UX). No matter the technical achievements of the product, its users will despise it nonetheless. Thus, UX should not be just an afterthought.

First, the website has to be responsive, that is to say, it must be usable on a wide range of display formats²³. On the landing page, the actual content is quite slim, it is just a form containing the servers and input for the player's name. The background animation of the landing page fills out the available place, while the form has a mostly fixed size that fits even the smallest displays. The full-screen control button and the settings toggler button are anchored to opposite corners of the screen.

Even during gameplay, there are no black bars: the display's aspect ratio is respected. First, I dive into a simple example of a traditional design element, then I summarise the key points of the UX of playing the game.

²³ Though, for evident reasons, printed and slow-refresh rate (e-ink) media need not be targeted.

3.7.1 Settings

The rotating gear in the top-right corner represents the settings. To not clutter the view with needless options all the time, they only show up after that icon has been clicked. As a result of a second click, they disappear.

Universal accessibility is essential for games. For instance, I met a four-year-old relative of mine who loves to play videogames on his tablet. He does not speak English, neither can he read, still, he is able to operate and enjoy games. This is due to the simple and icon-based user interfaces that are prevalent in modern mobile games.

After some consideration, I concluded that there need not be many options in *declared*. As of yet, a toggler for the sound, music, and vibration is present. During game-play, an exit button is also placed in the settings panel. The aforementioned three options have intuitive behaviour: when the sound is turned off, the music also gets turned off, or the vibration toggler is only shown on capable devices²⁴. Fortunately, there are universally recognisable symbols for every item on the settings panel. Their lack is represented by making them somewhat transparent and a crossing them out.

The next step is to provide the users with a rewarding experience. Each toggle event is animated, the slash appears as it was drawn. When enabling vibration, the phone vibrates a little, and when the sound is enabled and the buttons are clicked (or touched) a soft sound is played.

The settings panel has two more noteworthy features. It uses a frosted-like blur as its background; however, this is not available in all modern browsers, thus as a fallback, a darker transparent background is used. Lastly, on the landing screen on mobile screens, the vertical space is limited due to the large join form, hence the panel needs to open sideways. From this simple example it becomes apparent that great effort needs to be invested even into the minute details to create a polished product.

²⁴ There is no universal querying mechanism for vibration capabilities, hence, a heuristic is used, combining the check for the existence of the Vibration API and whether the device has touch capabilities.

3.7.2 Gameplay

During gameplay, a fair scaling mechanism is used, every player sees an area from the game world having the same area size. However, the shape of the area depends on the aspect ratio of the user's display.

Over each player's head, their names are shown along with their health, kill and death counts. On the top of the screen, located a progress bar illustrating the amount of points each team has. Additionally, the points generated by the asteroids, and the conquering state of the asteroids are also depicted. Lastly, to facilitate finding other players, arrows are shown on the sides of the screen pointing in the direction of both allies and enemies.

Since in *section 2* I have already written at great length about the graphics library, and the videogame uses the features of that library, I found my earlier coverage adequate and will not go into more detail about the graphics here.

3.8 Limitations

Although the game meets the quality and performance requirements we expect from it, I still find its networking aspect an ideal subject for improvement. Some occasional lag spikes can be experienced, which can hinder immersion. Maybe swapping the protocol stack to a UDP-based one could help with this issue. I might try it out in the future.

Making use of the capabilities of modern websites, namely, progressive web apps (PWA-s), it could be feasible to create an offline version of the game as well. It would certainly be a great feature, but in our connected world it might also be unnecessary.

3.9 Results

3.9.1 Performance

With *declared*, a performance measuring script is also downloaded which provided the values shown in the histogram below. The diagram was created with *ApexCharts*, following a basic pre-processing of the input data consisting of flattening and bin packing the gathered FPS values.

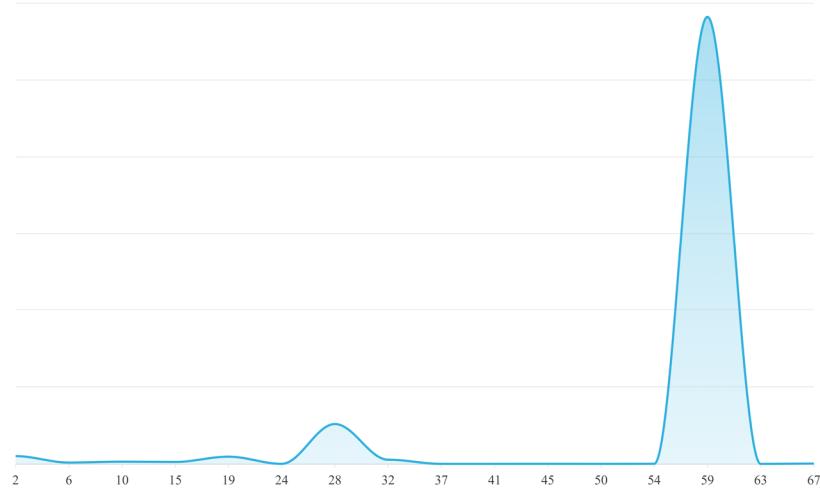


Figure 17 Histogram of FPS values over 66511 samples

In this case, FPS values are sent once every 15 second. The data comes from 1319 game sessions lasting for at least 45 seconds. It is promising to see the large clusters centred around 60 and just below 30.

This, combined with the data presented in *Performance* proves the potential usability of my solution. Further testing and weighing of the results to match the device distribution of the target audience more closely could be the next step in performance analysis. More in-depth network performance and user experience testing would also be a welcome addition to my list of measurements.

3.9.2 Screenshots

Regarding the visual quality, the finished UI, and overall feeling of the end product, I present some screenshots from *decla.red*.



Figure 18 Screenshot of the landing page on a 21:9 screen

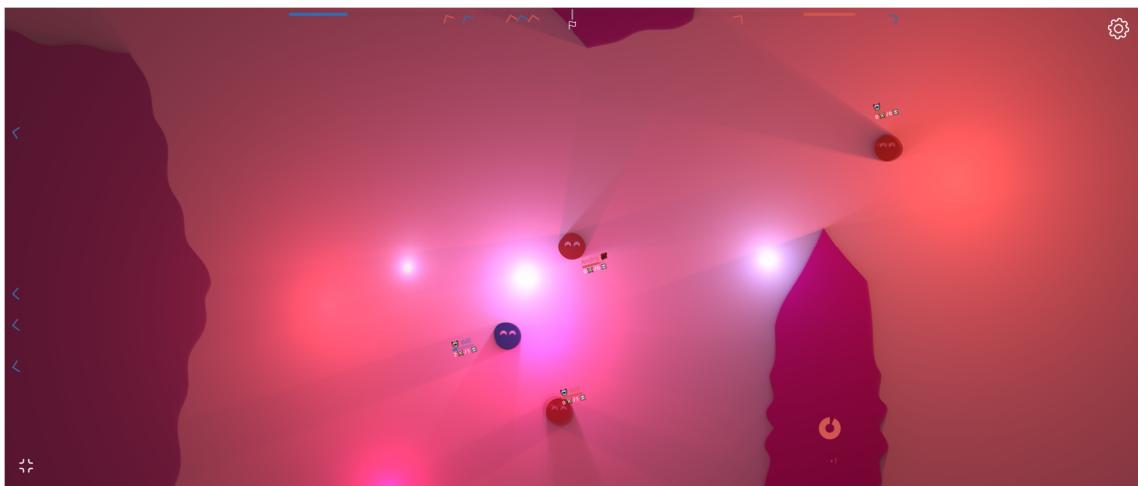


Figure 19 Screenshot of the game on a 21:9 screen

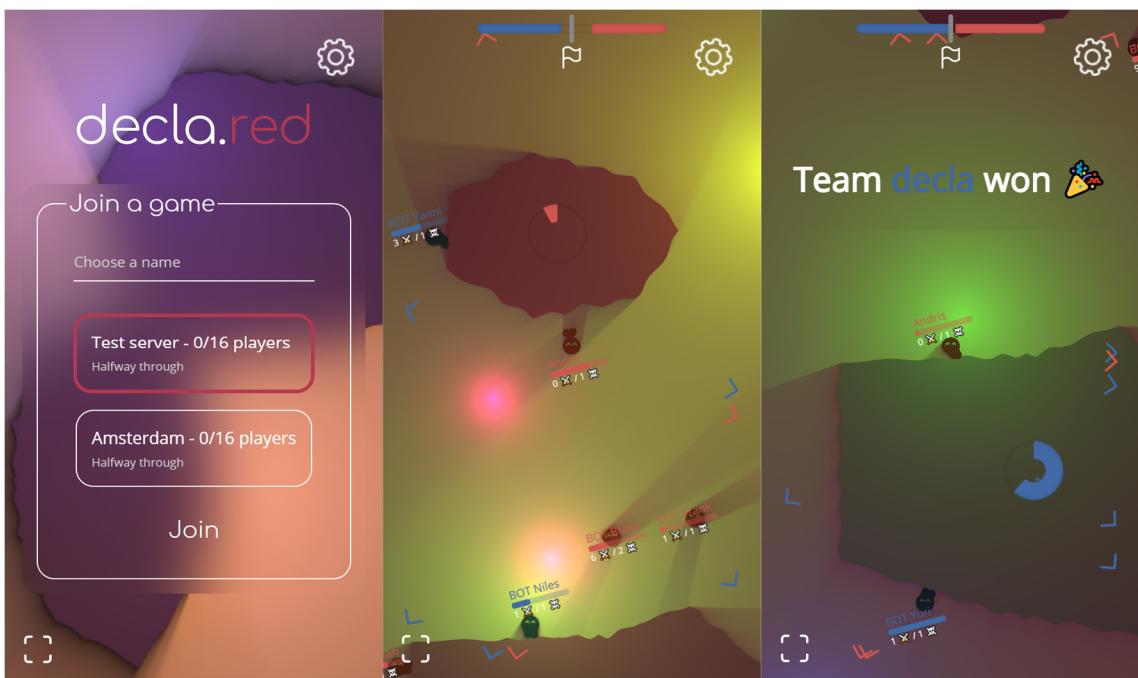


Figure 20 Screenshots taken at the screen size of iPhone SE

4 Conclusion

I came up with multiple ways of improving 2D ray tracing performance, which I then implemented as a library. Using remote performance testing, these — even slightly to my own surprise — turned out to be enough to work adequately even on cheaper mobile phones. Learning this, I extended the features of this *SDF-2D* library to make it a robust and reusable graphic framework which I then documented and published for the world to use.

With an interesting graphics solution at my disposal, I developed a multiplayer game. First, I compiled some ideas for an enjoyable gameplay, then implemented them. After seeing the architecture of my design, I heavily refactored it. To tell the truth, at one point three of the solutions described in *Commands* were present in my codebase. Nevertheless, merging and unifying these approaches while finding the strengths and weaknesses of my previous implementations was a fruitful learning experience.

To conclude, I am more than satisfied with my achieved results so far. Not only it has become clear to me that I immensely enjoy tackling problems relating to the domain of computer graphics and software architecture, I have also been able to extend the breadth and depth of my expertise in numerous ways. I learnt that relying on pre-existing solutions can be beneficial in many situations and the “but it works on my machine” cliché is more applicable in the context of shader programming than anywhere else.

It was thoroughly exhilarating to see the reactions of my friends and acquaintances when they first encountered my demos and videogame. Certainly, their praise may be somewhat biased, but that does not stop me from appreciating it. As for my projects, *SDF-2D* has more than a hundred weekly downloads on a regular basis on *npmjs*, which may not be too much, but I like to imagine that someone is actually using it. I have some *GitHub* stars as well. Regardless the fame or the lack of it, I would like to continue the development of both the library and *declared*. I just find the prospect of it fun and motivating.

All in all, the process leading to this point was not without adversity. After all, engineering feats are called that for a reason. I consider my efforts worthwhile, if not for the intrinsic value of the results (though, they certainly do not lack it), then for the invaluable experience this journey has given me.

5 References

- [1] “How rapidly are GPUs improving in price performance?,” Median Group, 2018.. [Online]. Available: <http://mediangroup.org/gpu.html>. [Accessed 30. October 2020.].
- [2] G. R. Hofmann, “Who invented ray tracing,” *The Visual Computer*, vol. 3., no. 6, pp. 120-124., 1990.
- [3] “WebGL Specification,” Khronos Group, 9 April 2020. [Online]. Available: <https://www.khronos.org/registry/webgl/specs/latest/1.0/>. [Accessed 25. September 2020].
- [4] “WebGL 2.0 Specification,” Khronos Group, 2. January 2020.. [Online]. Available: <https://www.khronos.org/registry/webgl/specs/latest/2.0/>. [Accessed 25. September 2020.].
- [5] F. &. Melnikov, “The WebSocket Protocol,” December 2011. [Online]. Available: <https://tools.ietf.org/html/rfc6455>. [Accessed 20. October 2020].
- [6] 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.
- [7] “caniuse,” [Online]. Available: <https://caniuse.com/?search=webgl>. [Accessed 24. September. 2020.].
- [8] 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.
- [9] J. Dirksen, Learning Three. js--the JavaScript 3D Library for WebGL, Packt Publishing Ltd, 2015.
- [10] P. Jiarathanakul, “Ray marching distance fields in real-time on webgl,” Citeseer.
- [11] I. Quilez, “Modeling with distance functions,” [Online]. Available: <https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>. [Accessed 12. September 2020.].

- [12] “TypeScript,” Microsoft, [Online]. Available: <https://www.typescriptlang.org/docs>. [Accessed 10. August 2020.].
- [13] I. Quilez, “Soft Shadows in raymarched SDF-s,” 2010. [Online]. Available: <https://iquilezles.org/www/articles/rmshadows/rmshadows.htm>. [Accessed 26. September 2020.].
- [14] Maarten, “2d signed distance functions,” Shadertoy, 17. August 2015.. [Online]. Available: <https://www.shadertoy.com/view/4dfXDn>. [Accessed 26. September 2020.].
- [15] t. a. sinisterchipmunk, “glMatrix,” 1. April 2020.. [Online]. Available: <http://glmatrix.net/>. [Accessed 10. August 2020].
- [16] que-etc, “resize-observer-polyfill,” [Online]. Available: <https://www.npmjs.com/package/resize-observer-polyfill>. [Accessed 5. November 2020.].
- [17] “HandlingContextLost,” Khronos Group, [Online]. Available: <https://www.khronos.org/webgl/wiki/HandlingContextLost>. [Accessed 10. August 2020.].
- [18] “ECMAScript® 2021 Language Specification,” 30. October 2020.. [Online]. Available: <https://tc39.es/ecma262/#sec-proxy-object-internal-methods-and-internal-slots>. [Accessed 30. October 2020.].
- [19] I. Quilez, “2D distance functions,” [Online]. Available: <https://www.iquipelzles.org/www/articles/distfunctions2d/distfunctions2d.htm>. [Accessed 30. July 2020.].
- [20] J. Gilbert, “WebGL feature levels,” [Online]. Available: <https://jdashg.github.io/misc/webgl/webgl-feature-levels.html>. [Accessed 1. November 2020.].
- [21] H. L. W. X. a. B. H. B. Knutsson, “Peer-to-peer support for massively multiplayer games,” *IEEE INFOCOM 2004*, vol. INFCOM, no. 10.1109, p. 107, 2004.
- [22] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, pp. 509-517., 1. September 1975..

- [23] “WebRTC 1.0: Real-Time Communication Between Browsers,” W3C, 15. October 2020.. [Online]. Available: <https://www.w3.org/TR/webrtc/>. [Accessed 30. October 2020.].
- [24] R. Rai, *Socket. IO Real-time Web Application Development*, Packt Publishing Ltd., 2013..
- [25] R. H. R. J. J. V. Erich Gamma, “Visitor,” in *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994., p. 331.
- [26] S. Ducasse, “Evaluating Message Passing Control Techniques in Smalltalk,” *Journal of Object-Oriented Programming*, June 1999..
- [27] “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.].
- [28] M. o. t. W. w. group, “WebGL WEBGL_debug_renderer_info Khronos Ratified Extension Specification,” Khronos Group, 15. July 2014.. [Online]. Available: https://www.khronos.org/registry/webgl/extensions/WEBGL_debug_renderer_info/. [Accessed 5. October 2020.].
- [29] J. F. K. N. Dzmitry Malyshau, “WebGPU,” W3C, 29. October 2020.. [Online]. Available: <https://gpuweb.github.io/gpuweb/>. [Accessed 30, October 2020.].
- [30] “WebGL Specification,” Khronos Group, 9. April 2020.. [Online]. Available: <https://www.khronos.org/registry/webgl/specs/latest/1.0/#2.1>. [Accessed 30. September 2020.].
- [31] D. T. Andrew Hunt, *The Pragmatic Programmer*, Addison Wesley, 1999.

6 Appendix

6.1 Code for the noise renderer

```
/**  
 * Create a renderer,  
 * draw a 2D noise texture with it,  
 * then destroy the used resources  
 * while returning the generated texture in the form of a canvas.  
 */  
  
export const renderNoise = async (  
    textureSize: ReadonlyVec2,  
    scale: number,  
    amplitude: number,  
    ignoreWebGL2 = false  
) : Promise<HTMLCanvasElement> => {  
    const canvas = document.createElement('canvas');  
    const gl = getUniversalRenderingContext(canvas, ignoreWebGL2);  
  
    const frameBuffer = new DefaultFrameBuffer(gl, textureSize);  
    const program = new FragmentShaderOnlyProgram(gl);  
    const compiler = new ParallelCompiler(gl);  
  
    const pogramPromise = program.initialize(  
        gl.isWebGL2  
            ? [randomVertex, randomFragment]  
            : [randomVertex100, randomFragment100],  
        compiler  
    );  
  
    await compiler.compilePrograms();  
    await pogramPromise;  
  
    frameBuffer.bindAndClear();  
    program.draw({  
        scale,  
        amplitude,  
    });  
  
    frameBuffer.destroy();  
    program.destroy();  
  
    return canvas;  
};
```