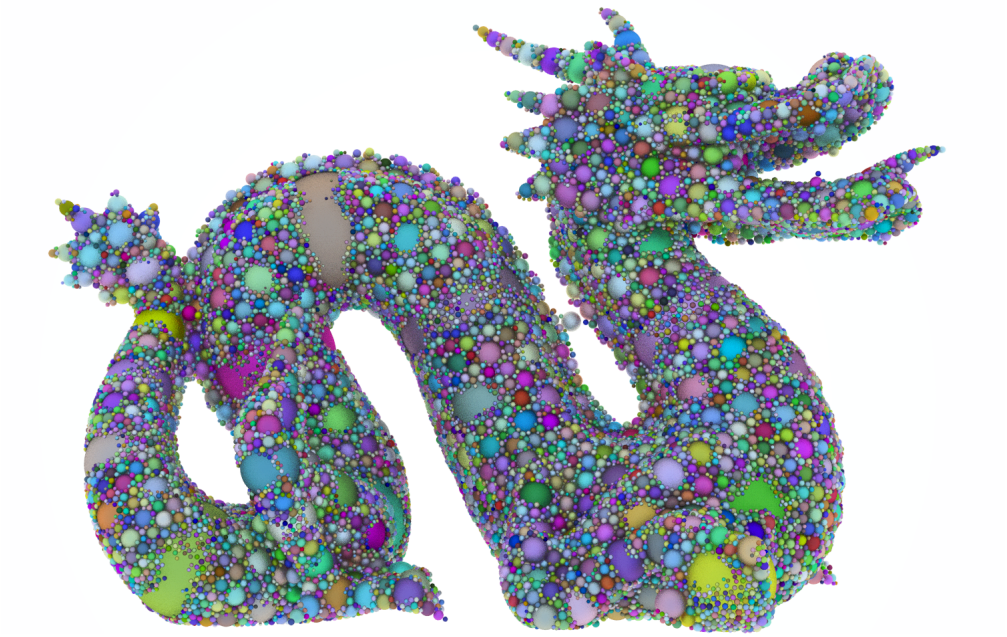
University of Bremen
Bibliothekstr. 1
28359 Bremen

Raytracing based Renderer for Sphere-Packings



bachelor report
to obtain the academic degree
Bachelor of Science

Author:	Dajo Frey
Supervisor:	Dr. Rene Weller
Personal identifier:	4326756
Date:	February 8, 2020, Bremen

Abstract

This bachelor report discusses the implementation of a program [1] for sphere packing rendering within specific and unspecific contexts. Most work was put into developing the software that goes along with this report. As a consequence, the implementation aspect of this specific piece of software will be at the heart of this document. Additionally, the report will also provide insight into other research on the topic, which can be considered the unspecific part of this report and may be helpful for future work.

Regarding the developed software: Rendering is based on the ray-tracing approach and different methods inside this approach were implemented. These methods range from simple ray-casting using only one ray per pixel to path-tracing, using multiple rays and multiple samples. Performance-tests on these methods are measured using a broad range of computer-systems to gather meaningful data. Acceleration structures were implemented at varying degrees with optional visualizers to dynamically witness the integration of these structures. Algorithms for Metaballs-rendering were implemented and discussed. Generally, rendering is based on a tiled approach across all algorithms to provide a stable and highly configurable experience.

Future work and improvements are discussed at the end of this report. In conclusion, the software fulfills it's purpose of visualizing sphere-packings. Rendering is mostly interactive depending on the sphere count and hardware in use. The software is a good base for any specialized directions that may come in the future.

Contents

1 Introduction

The introduction will give you some information 1.1 about the most important concepts which are useful to know when reading this report. In addition, the goal is explained 1.2 and an overview of this report is given 1.3

1.1 Background

1.1.1 Sphere Packings

The need to visualize (or render) sphere-packings arose from an application called *ProtoSphere* which is in use at the Computer Graphics AG of Bremen University. *ProtoSphere* fills 3D objects with spheres and the user can choose, for example, how many spheres the software should use to fill the object. The resulting aggregation of spheres is simply called sphere-packing. An example for how this might look can be seen on the title-page of this bachelor report. Important to mention is that the spheres inside these packings don't intersect each other or the object. Generally, the sphere packing algorithm tries to fill as much space using spheres inside the object as possible. Because of this, generated spheres have different sizes. The details of this filling process depend on the specific algorithm. The original algorithm can be studied in the corresponding paper [2]. *ProtoSphere* generates data in the form of files which contain a numerical representation of the spheres that fill the object. These files can be read by applications to visualize the generated spheres, which is basically the job of the application presented in this report (please see section 1.2 for more info). It's also interesting to note that these files contain the spheres in a ordered manner, which means that the first sphere on line 1 is the biggest sphere of the entire package.

Real-life applications for sphere-packings could be for example the creation of art (in the form of statues) or collision detection between objects. The approach for sphere-packings by *ProtoSphere* is closely related to a paper which introduced a new geometric data structure [3], called *Inner Sphere Tree*. This data structure can be used to improve the speed and accuracy of collision detection.

1.1.2 Rasterization and Ray-tracing

Previous approaches for visualizing sphere-packings at the Computer Graphics AG of Bremen University were based on rasterization based pipelines, which is a common method for rendering and enjoys prevalent hardware support. Thus, this method is perfectly suitable for realtime rendering tasks

and can be considered a cornerstone of software applications such as computer games. Rasterization in it's essence tries to solve the visibility problem by projecting 3D geometry onto a 2D plane (for example the screen). Figure 1 visualizes the general approach. The projection is done by projecting vertices of triangles onto the 2D plane and after that, filling the inner space of the triangle. These two steps are computationally simple and that's one reason why the algorithm is fast [4] (even without hardware acceleration).

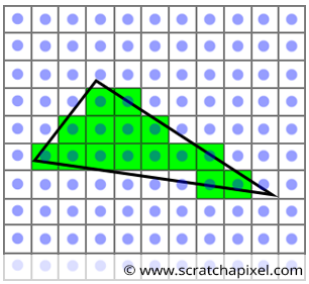


Figure 1: Rasterization of a triangle [1]

While this method is ubiquitous, it is also said that it can only achieve a limited visual fidelity when compared with other techniques, as an article on the Nvidia news center points out [5]. One of those other techniques is ray-tracing. Ray-tracing is fundamentally different in that it works by shooting a ray from a starting point (for example a camera, a human eye, etc.) into a scene. The ray then can be checked for intersections with geometry and thus, a 2D image can be created. Figure 2 shows the concept of ray-tracing. While the approach is fundamentally different it's also closer to how vision in the real world works. One conceptual difference is that rays aren't shot from light sources, which is called forward tracing. In contrast, ray-tracing is a backward tracing method.

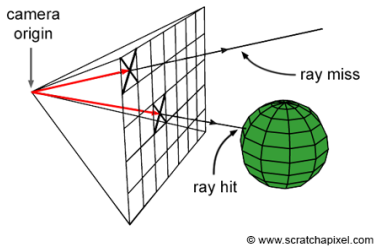


Figure 2: Concept of ray-tracing [2]

Because ray-tracing is closer to how reality works, it's possible to simulate higher fidelity visuals with less work compared to a rasterization based ap-

proach. Although ray-tracing was too computationally expensive for many realtime applications in the past, recent hardware accelerated solutions made partial ray-tracing feasible for a lot of applications.

1.2 Goal

So, what is the goal of this bachelor report? Basically, Dr. Rene Weller suggested that I could write a ray-tracing based renderer to visualize sphere-packings. One of the reasons for his suggestion was that ray-tracing lends itself perfectly to sphere rendering. For example, using a rasterization approach inherently requires us to approximate the sphere surface using triangulation while the ray-tracing approach is more straight forward. Thus, ray-tracing in comparison to rasterization can be simpler in terms of coding. For example, you can write a raytracer featuring global illumination using less than 100 lines of code [6]. The downside is that this approach is usually slower and acceleration structures are needed. But we will talk about that later. In summary, the goal is split into two parts. The first part is developing the software and the second part is writing a document that explains, measures and expands the software that was developed in terms of educational value. This document (the one your reading right now) is not meant to be a comprehensive documentation of software functionality, it's rather an overview and and will go only into detail at critical code sections.

1.3 Report Overview

The report follows the regular pattern of its kind. Right now, you're reading the introduction, which is supposed to introduce you to the theme and explain why this bachelor report came into existence. You should also gain a good understanding of what the overall goal is. Next up, we have a section about related work, which gives insight about interesting and relevant research regarding this report 2. The section is divided into sub-sections each talking about a specific paper. Then I talk about the approaches I make for implementing the software in the follow up section 3. This is a kind of high level talk about the most crucial algorithms and systems I use. Closely related to this is the implementation section that immediately follows 4. Here I discuss important things regarding the actual implementation of the previous mentioned approaches. I also talk about architecture details and dependencies that are noteworthy. At the end of the document I present measurements in the results-section 5 and comment on overall findings and results in the conclusion-section 6. The last section discusses future directions and possibilities based on the results presented in this report 7.

One word regarding *image credits*: Most images I used are either by me or in the public domain. Those images that do not have a footnote in the caption are by me. The footnotes can be found at the end of the section where the image belongs to. To be able to use images which are not in the public domain, I asked permission of the respective owners. Images from scientific papers reference the paper accordingly.

[1]From scratchapixel.com/rasterization-stage with the owners consent
[2]From scratchapixel.com/ray-tracing-overview with the owners consent

2 Related Work

This section collects and summarizes related work. Sphere-packing rendering is not a big or even medium sized field in the area of computer graphics, but there are a lot of papers that can be placed between volume rendering and sphere-packing rendering which are relevant. The first section will provide a overview of the most prominent and most cited relevant papers. The sections afterwards will provide detailed insight of papers which I personally found to be very interesting or promising in the context of this report.

2.1 Sphere rendering

Sphere rendering can be approached from many different directions. One obvious direction is to render spheres using the rasterization pipeline, where one has to approximate spheres using triangles. This is something I didn't explore further since other applications already use this approach.

Another direction is to render spheres using point primitives. Point primitives are nowadays commonly used in the form of point clouds, which are useful for recreating surfaces based on real world data. One important characteristic of point clouds is, that there is no connectivity between primitives. In 1985, Whitted et al. [7] were among the first to render using point clouds rather than using modeling geometry. The problem was, that the proposed algorithms didn't quite capture the look of solid looking objects. This can be achieved by using *splatting*, which was introduced by Lee Westover [8] in 1989. This technique projects the points into 2D space and uses a so called *footprint* principle, where each point contributes it color to the neighboring pixels. The evolution of that were the *surfels*, introduced by Pfister et al. [9] in 2000, which hold attributes like depth, normal etc. for each point. Surfels have the shapes of disks in 3D space and can be used to approximate 3D surfaces. These approaches are not better suited for sphere rendering than other methods, since they would need to approximate the sphere surfaces, which creates overhead. It would be possible to render a sphere using just a single point, but this would trait off a lot of render quality compared to other methods.

A different take and maybe the most elegant way is to use ray-tracing. Ray-tracing in computer graphics became relevant when in 1980, Turner Whitted [10] expanded on previous ideas and introduced recursive ray casting. Before this breakthrough, only simple ray-casting was known, which was introduced by Arthur Appel [11] in 1968. Ray-tracing paved the way for the

famous rendering equation from James T. Kajiya [12], which most realistic rendering techniques attempt to solve. A lot of work was (and is) put into accelerating these base concepts. For example Gunther et al. [13] developed a parallel BVH traversal algorithm which makes use of modern graphics hardware. The BVH implementation explained in this report is similar to the approach that is explained in that particular paper.

As the goal already mentioned 1.2, the approach explained in this report will be based on ray-tracing, mainly because its natural ability to render spheres and to study the performance on recent hardware. Also an approach based on ray-tracing can be easily extended to render other primitives such as cubes or cones.

2.2 Metaballs

Metaballs as a implicit modeling technique was first introduced by Jim Blinn [14] in 1982. Basically, the approach uses the summation of Gaussian density distributions to model density maps of molecular structures and renders these, using the ray- tracing algorithm developed two years earlier [10]. My approach is based on this principle. Over the years, the metaballs technology has matured and achieved commercial success. In general, metaballs are nothing more than *iso-surfaces* which are defined by so called *scalar field functions*. Agata et al. [15] provide a good overview of iso-surfaces which is still valid today.

2.3 Multi-Scale Modeling and Rendering of Granular Materials

Meng et al. [16] present an interesting method for rendering large amounts of granular materials. The paper is relevant for this report because the presented approach uses sphere-packings in the process. For example, the method they are using for filling a scene description with granular materials as well as some rendering steps are based on sphere-packings. They call the approach 'tiled sphere packings' which are quite similiar to our packings. One major difference is that spheres inside these packings all have the same radius. They also do not directly render spheres, they are only used as bounding volumes. These bounding volumes contain randomly instanced grains which they render with a technique called Explicit Path Tracing (EPT). For this rendering method they first voxelize the mesh. The generated voxels are basically the same as the tiles from the sphere-packings. They then intersect a ray with the voxel grid to get a voxel that is a valid candidate for rendering.

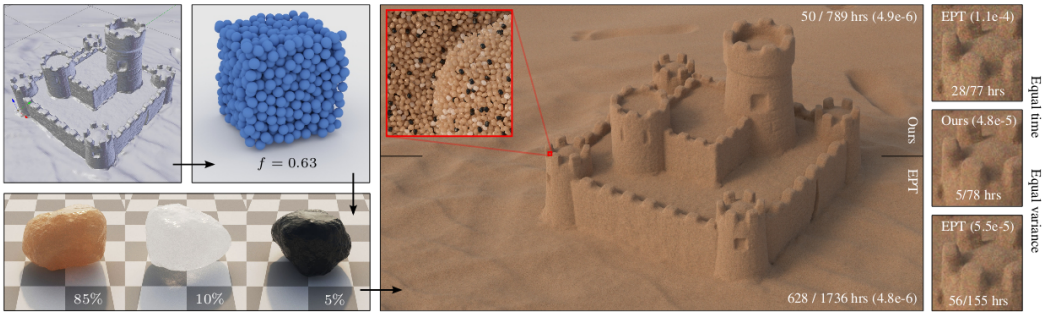


Figure 3: Overview of the multi-scale approach [16]

They then intersect the bounding spheres that overlap this voxel. After finding a valid intersection they test if the randomized grain inside the bounding sphere is intersected. If yes, the color of the grain at the intersection will be drawn.

The paper argues that this method is impractical for higher order scattering, which is why they transition to a hybrid approach comprised of Volumetric Path Tracing (VPT) and diffusion approximation. This decision is also made because of the wide range of shots they needed to support (ranging from long distance to close up shots). Efficient rendering based on camera distance is out of scope for the application, but the proposed paper gives a very good explanation on how to do it and it even involves sphere-packings. So further work on efficiency could be inspired by this paper. It is also interesting that in the paper they use tiled sphere-packings instead of just sphere-packings. My understanding is that, according to the authors methodology, the implementation presented in this report also uses tiled sphere-packing when combining rendering with the grid acceleration structure 3.3.

2.4 GPU-Based Ray-Casting of Quadratic Surfaces

Sigg et al. [17] present an interesting hybrid method which combines rasterization and ray-casting to render large amounts of quadric surfaces. Spheres are also quadric surfaces and thus it's a interesting paper to consider in this report. In summary, the authors use the vertex shader stage of the gpu-rasterization pipeline to compute a screen-space bounding box for every vertex. Important to note is that each vertex corresponds to a quadric primitive. They then compute the ray-quadric intersection based on the rasterized bounding boxes which in turn colors the fragment. This step is done in the fragment shader. An image produced by the method can be seen in figure 4.

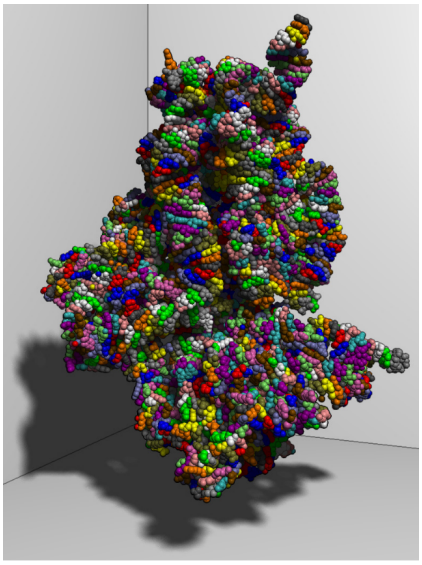


Figure 4: Molecular structure consisting of 52k atoms [17]

In detail, the authors use a bilinear form for the implicit definition of quadrics which can be projected into screen space by a linear transformation. This also makes it easy to compute bounding box and ray intersection later on. For each quadric the authors use one distinct basis. The specific basis is determined by the normalized diagonal form of the quadric surface which is obtained from a basis transformation. The basis is used to simplify other computations such as intersections. The basis transformation is based on a transformation matrix the authors call *variance matrix*. This matrix is used additionally to determine the shape of the quadric in object space. The authors use point sprite primitives because they can be computed with a single vertex call. The parameters of these point sprites are obtained in the vertex shader. The first step is to compute a bounding box. Next, the authors compute the center position and the radius of the point sprite using the bounding box and from there, perform post processing tasks in the fragment shader.

The approach is very fast but also complex and riddled with details that make it not fitted for sphere-packing rendering, such as the general quadric approach.

3 Approach

The approach defines how I went about the goal that was set out for this bachelor report 1.2. Because the main goal was to develop an application, this section talks a lot about algorithms but also about other considerations such as usability. The general approach 3.1 overshadows every other aspect of this discussion because it defines the fundamentals that should be honored by each approach. Next, I discuss the rendering approach 3.2 which explains the different algorithms that are used for rendering. After that, you can read about some considerations and algorithms regarding acceleration 3.3. In the end, I give a brief discussion about the approach for memory management 3.4 and the GUI 3.5

3.1 General

Because I didn't have a specific use-case other than sphere visualization in mind when development started, I tried to keep the general approach as extensible as possible. This comes with a few up and down sides. The biggest upside is that future work on the implementation can literally be in any direction without the need to rework specific parts. It also enabled me to try out different approaches (for example regarding acceleration) because some might be more appropriate than others in certain scenarios. Because I didn't have a specific scenario in mind I was able to use a broad spectrum of approaches for different kinds of problems.

While this is great for future extensions, it also limited how deep I could implement specific systems. This is because for every approach there exists usually a lot of improvements that build up on the original idea. These take time to implement and only make sense (from an economical perspective) when the approach you are using is something you are going to stick with for a appropriate amount of time, which means that I refrained from implementing optimizations when there just wasn't enough reason to justify it.

Another thing I wanted to focus on in this approach is usability. Because of the aforementioned broad spectrum approach, the software needs to be intuitive to use for the user to leverage all possibilities. This meant for me that I had to pay attention to the user interface in order to make it as accessible as possible. It also meant that changing the underlying system (for example changing render techniques, acceleration structures, etc.) should be very easy to do from a user perspective.

To sum it up, the implementation is based on a general approach that tries

to deliver a broad variety of solutions to different kind of problems that come up when rendering sphere-packings while keeping in mind usability.

3.2 Rendering

The approach I took for rendering is rooted in the general approach. Which means that I made the decision to support different kinds of rendering techniques as opposed to only implementing one type of rendering technique. These techniques are all based on ray-tracing or ray-marching. Because the algorithms all have a common base it was possible to share specific sections of these implementations between different techniques. For example the routine for generating a ray for a specific pixel based on the camera location is the same across all techniques.

3.2.1 Ray-tracing based Techniques

Ray-tracing based techniques for rendering can be divided into 3 different approaches.

1st approach:
First of I implemented a naive ray-tracing approach which basically shoots one ray per pixel into the scene and tests if a intersection occurs. If so, the pixel will be colored using some randomizing function. If not, the pixel will occur white. This is the simplest technique for rendering when using the ray-tracing approach and can also be expected to be the fastest when combined with acceleration structures. The technique is sometimes called ray casting because it doesn't involve tracing secondary/tertiary/... rays.

2nd approach:
Based on the first approach I implemented another technique that produces images which appear smoother. This technique doesn't really change much of the behavior of the first technique, except that it applies Anti-Aliasing. This was needed because straight up ray-tracing produces very sharp but also jagged looking images. An example of what I mean can be seen in section 5.1.1. The approach basically scales the image by 2 (4xAA) or by 3 (9xAA) and averages the corresponding pixel colors. The pseudo code shows the general approach in the context of 4xAA 1.

Listing 1: Determining pixel-color using AA in combination with tiled rendering

```
1 if (pixel inside tile)
2 {
```

```
3     pixel coordinates *= 2;
4
5     uv[4]; // these texture coordinates of the scaled image
          correspond to the original pixel coordinates
6     uv[0] = pixel coordinates upper left / (viewport
          resolution * 2);
7     uv[1] = pixel coordinates upper right / (viewport
          resolution * 2);
8     uv[2] = pixel coordinates lower left / (viewport
          resolution * 2);
9     uv[3] = pixel coordinates lower right / (viewport
          resolution * 2);
10
11    for each uv {
12        pixel color += calculate pixel color at uv;
13    }
14
15    averaged pixel color = pixel color / 4;
16
17    set pixel color of last sample to averaged pixel color;
18    set pixel color of this sample to averaged pixel color;
19 }
20 else {
21    set pixel color of this sample to pixel color of last
          sample;
22 }
```

3rd approach:
The 3rd approach is based on a path-tracing technique which shoots multiple rays per pixel and produces a believable image but comes at a greater performance cost than previous techniques. The algorithm has to average samples in order to converge to a less noisy image. The noise that can be seen in figure 32 is actually the variance caused by the pseudo-random nature of the ray generation. The pseudo code 2 illustrates the concept of determining pixel color via path-tracing in combination with tiled rendering 3.2.3.

Listing 2: Determining pixel-color using path-tracing in combination with tiled rendering

```
1 if (pixel inside tile)
2 {
3     get random seed;
```

```
4     get texture coordinates;
5     get ray based on camera settings;
6     get pixel color of this sample by tracing the ray;
7
8    averaged pixel color = ((last pixel color * frame count) +
        pixel color) / (frame count + 1);
9
10    set pixel color of last sample to averaged pixel color;
11    set pixel color of this sample to averaged pixel color;
12 }
13 else {
14    set pixel color of this sample to pixel color of last
        sample;
15 }
```

In summary, these ray-tracing based rendering techniques cover up a lot of use-cases that might come up in the future and should prove to be a good starting-point for further optimization and visual tweaking. Because of the interchangeable nature of these techniques, it made sense to allow the user to switch between them as dynamically as possible. This can be achieved by either preparing everything render related for each rendering technique upfront or by only preparing the necessary amount based on which rendering technique is in active use. I decided to use the first approach because it leads to simpler and less code while not causing significant decrease of performance.

3.2.2 Metaballs

I implemented a technique which tries to render implicit surfaces (or iso-surfaces) by using the metaballs approach. This technique is vastly different from the previous ones because it not only renders spheres, but also the empty space between those spheres. This gives the impression of a continuous surface even though it's technically not.

The approach is based on the function for the strength of a electric field at a certain point. The point can be called a point-charge and defines where the electric field is the strongest. For a point-charge at the origin, the function looks like this: $f(x, y, z) = r/(x^2 + y^2 + z^2)$ where r is the radius of the point charge. Which makes this interesting is that the function returns a value of 1.0 at a distance of r from the point-charge. At a distance greater than r the function will return values lesser than 1.0 in a continuous manner based on its distance to r. The function will generally return values greater than r if

the point lies within the point-charge radius. For a great explanation of the principle visit Ryan Geiss's website [18].

The characteristics of the function can be used for rendering. To render metaballs we need to shoot rays into the scene, much like the aforementioned ray-tracing techniques. But instead of checking for intersections, we compute the value of the electric field function along the ray in incremental steps. We do this for each sphere and add up the values in each step. If the sum is bigger than a certain threshold at a step along the ray, we color the pixel. There are a lot of variables in this approach (step size, step count, threshold, method of coloring the pixel) which hugely impact the outcome. The approach I implemented allows the user to change the threshold and contribution weight, but the possibility to change the other variables at runtime may be useful too. The pseudo code 3 shows the general execution of the approach. Please note, that the pseudo code abstracts from a lot of details (e.g. keeping track of charge contributions) in order to focus on the skeleton algorithm. As you might have observed, I do not compute the electric field function directly. This is because I think splitting the function across two computations makes it easier to understand.

Listing 3: Determining pixel-color in the first Metaballs shader

```
1  ray position = ray origin;
2  for each step {
3      for each sphere {
4          distance = distance between ray position and sphere
               position;
5          charge = sphere radius / distance; // will compute a
               value similar to the electric field function;
6          if (charge GREATER OR EQUAL internal Threshold)
7          {
8              if (charge GREATER OR EQUAL 1.0) { // if true, the
                   ray position must be inside the sphere according
                   to the electric field function
9                  inside sphere = true;
10             }
11             charge sum += charge;
12         }
13     }
14     if (charge sum GREATER OR EQUAL user defined threshold OR
           inside sphere) {
15         determine pixel color;
```

```
16         return pixel color;
17     }
18 }
19 return default pixel color;
```

3.2.3 Tiled Rendering

All these rendering techniques operate with different execution speeds depending on the hardware. This is a challenge because while one technique may be unproblematic, some other may lead to a phenomenon what I call *pseudo-freeze*. A *pseudo-freeze* may happen when the operating system uses the same hardware for rendering tasks as the software, which may render other more complex things than a GUI. A *pseudo-freeze* more specifically happens when a rendering task takes so long to complete, that rendering the GUI from the operating system is blocked. In this case the operating system still responds to input but there is no visual feedback, giving of the impression that the system is on halt. Some drivers react in those situations by interrupting the execution of the workload. To tackle this I decided to use a approach which is called tiled rendering. This approach basically divides the rendering task into manageable chunks that can be computed in a timely manner. The user can identify tiled rendering when the image is not rendered at once but only sections of it, as can be seen in figure 5.

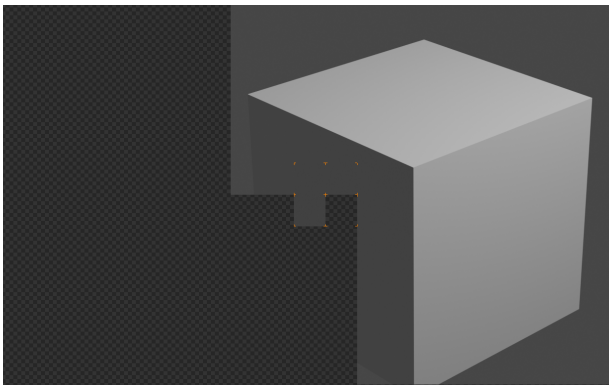


Figure 5: Tiled rendering in action (Blender Cycles)

However, this approach comes with a downside because it introduces overhead. This overhead is caused by the software having to, figuratively speaking, send a lot of packages to the computing unit as opposed to only having to send one package. As a consequence the computing unit needs to unwrap these packages and the sending software needs to prepare them, which costs

time on both sides. This means that more chunks also means more overhead and this essentially makes the process of choosing the tile size a balancing act between overhead and performance. This is also the one of the reasons why I decided to let the user interactively change the tile size. It's also coherent with the general approach of high usability.

Determining the tile position/size is easy and very portable 4. The one thing that is not portable is the method of sending the position and size data to the GPU. This can be done using an *UBO* or (in the case of Vulkan) using *PushConstants*. I chose the latter approach because of some performance benefits.

Listing 4: Tile position and size generation

```
1  multiplicator = divide user defined tile size by 100; //
       should give us something between 0% and 100%
2  tile width = viewport width * multiplicator;
3  tile height = viewport height * multiplicator;
4
5  if (tile reached maximum viewport-width)
6  {
7      tile x coordinate = 0;
8  } else {
9      tile x coordinate += tile width:
10 }
11 if (tile reached maximum viewport-width AND maximum
       viewport-height)
12 {
13     tile y coordinate = 0;
14 } else {
15     tile y coordinate += tile height;
16 }
```

3.3 Acceleration

While we talk about the rendering approach, we also need to consider the acceleration approach. These 2 topics come usually hand in hand because acceleration is a crucial factor for achieving interactive frame rates. I approached acceleration the same as I approached rendering in that it is based on the general approach that I wanted to cover a broad spectrum of techniques. For that I decided to implement 2 fundamental acceleration tech-

niques that are common for ray-tracing based rendering.

Considering the overarching approach, I also wanted to support the ability to dynamically change the underlying acceleration structures without losing usability (similarly to the rendering approach). By 'dynamically changing' I mean the ability to interactively change, for example, the amount of bounding volumes of the BVH. This enables the user to develop a good understanding of said techniques while it also makes it easier to measure performance for specific configurations, which is helpful for deciding which acceleration technique using which configuration gives the best performance.

3.3.1 BVH

The first technique is called Bounding Volume Hierarchy (BVH) and works roughly by generating bounding volumes that together enclose the object and minimize useless intersections while rendering. Figure 6 visualizes how the ray is sent through those bounding volumes. While the image does not visualize a specific hierarchy, it is still clearly visible that most of the bounding volumes are not hit by the ray and thus discarded, which is exactly what's causing the speed up when rendering using a BVH.

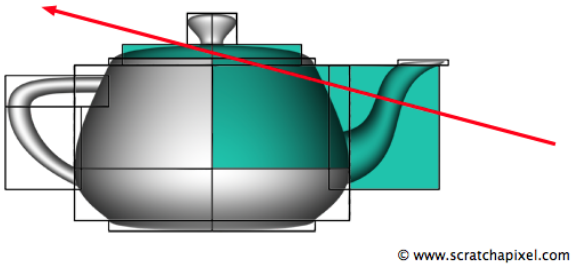


Figure 6: Bounding Box - Ray Intersection [3]

The volume type I use for the bounding volumes is called Axis Aligned Bounding Boxes (AABB). The reason I chose AABB's is, that it's considerably easy to test for a ray-AABB intersection, which also makes it relatively fast. The generation of the hierarchy is done in iterated steps where you begin to first define the root of the hierarchy. The root is nothing else than a AABB that completely encapsulates the object for which you want to speed up rendering. Next you partition the space inside the parent (in case of the 2nd iteration this would be the root AABB) and create a new AABB child that completely encapsulates this new space inside the parent AABB. This goes on and on until the desired BVH depth is achieved. However,

the current approach is limited to space partition along the x-axis. This is a conscious decision because space partition along all dimensions is something that the second acceleration technique does natively. It also improves the code-maintainability for future work. Ultimately, rendering speed up is achieved by intersecting the hierarchy instead of the actual geometry. Only if the ray intersects a leaf inside this hierarchy will the geometry encapsulated by the leaf be tested for intersection. For a visualization of the bounding boxes see figure 34.

The basic idea of rendering using a BVH can be seen in the corresponding pseudo-code 5. A lot of optimization comes solely from the discarded intersections caused by not fulfilling the first if-clause. After that, only a part of the spheres will be intersected depending on the AABB distribution and size.

Listing 5: Determining pixel-color using a BVH in combination with tiled rendering

```
1  if (pixel inside tile)
2  {
3      // abstracted from ray generation etc.
4
5      if (bounding box count GREATER 0 AND ray intersects BVH
           root)
6      {
7          for each bounding box
8          {
9              if (ray intersects bounding box)
10             {
11                 get pixel color;
12                 if (hit AND ray intersection distance smaller
                       than previous) {
13                     set pixel color;
14                 }
15             }
16         }
17     }
18
19     set pixel color of last sample to pixel color;
20     set pixel color of this sample to pixel color;
21 }
22 else {
```

```
23      set pixel color of this sample to pixel color of last
            sample;
24  }
```

3.3.2 Grid

The second acceleration technique is simply called a Grid and partitions space evenly across all 3 dimensions. The resulting structure can be considered a 3 dimensional grid where each cell has the same size. The visualizer in figure 36 shows how this looks when coloring the cells. The cells are of course nothing more than AABB's, which means that intersecting these cells is quite fast. One major difference to the BVH technique is that objects which are intersected by cells need to be inserted into each of the cells that intersect the object for correct rendering. This makes it a little bit harder to implement compared to the first technique. When rendering, the current approach is quite similar to the pseudo code in listing 5. However, because of the well defined nature of the grid there is another possible optimization which is called Digital Differential Analyser (DDA). This optimizes the way we traverse the grid by preventing unnecessary AABB intersections. I consider this a minor improvement so I did not implement it in the current approach, though it is a possibility for future optimization.

The pseudo-code below 6 shows the general idea for the grid generation. It starts by defining the initial outer boundaries of the grid. This is hard-coded in the implementation but doesn't really affect anything, since sphere-packings do not even come close to reaching these values. Next, the algorithm computes the number of bounding boxes depending on the grid divisions. Last but not least, the grid frame is computed, which allows us to calculate the bounding box positions in the last step.

Listing 6: Calculating the grid

```
1  // set the initial boundaries
2  root minimum = {1000, 1000, 1000};
3  root maximum = {-1000, -1000, -1000};
4
5  if (number of grid divisions is 0) {
6      bounding box count = 0;
7  } else {
8      bounding box count = (number of grid divisions + 1)^3 + 1;
9  }
```

```
10
11 // calculate grid frame
12 for each sphere
13 {
14     if ((x position of sphere) MINUS (sphere radius) SMALLER
          THAN (x position of root minimum)) {
15         x position of root minimum = (x position of sphere
             number j) MINUS (sphere radius);
16     } else if ((x position of sphere) PLUS (sphere radius)
          GREATER THAN (x position of root maximum)) {
17         x position of root maximum = (x position of sphere
             number j) PLUS (sphere radius);
18     }
19     // repeat for height and depth values
20 }
21
22 // calculate grid
23 for each bounding box
24 {
25     x position of bounding box minimum = x position of root
          minimum PLUS (((i - 1) % (number of grid divisions +
          1)) * (width of root) / (number of grid divisions +
          1)));
26     x position of bounding box maximum = x position of
          bounding box minimum PLUS ((width of root) / (number of
          grid divisions + 1));
27 }
28
29 // repeat in a slightly more complex form for height and depth
     values
```

3.4 Memory Management

The approach for memory management is, of course, also based on the general approach. First of all, it is necessary to mention that multi-threading is an important concept in rendering applications. Just consider using only one thread for both input handling and rendering. That would not be fun since input handling would be totally dependent on how long rendering takes, or vice versa. Multi-threading helps with that, but it also introduces new problems, namely race conditions. *Race conditions* are events where multiple

threads compete for something in such a way, that the final outcome becomes non-deterministic. This is obviously undesired since it can lead to undefined behavior. In order to minimize race conditions regarding memory, I use a approach which completely avoids sharing memory between threads where it isn't necessary. In this approach, each thread manages its own memory which it can expand in unlimited manner (theoretically speaking). While sharing memory between threads is possible, the approach does not involve synchronisation (for example via semaphores, mutexes, fences, ...) because currently shared memory is read only. This would probably have to be expanded on in future work. The memory is divided in chunks and the size of these chunks can be adjusted. Chunk based memory management can increase performance because it avoids memory allocation calls, depending on the size of the chunks.

3.5 Graphical User Interface

I already talked a little bit about the usability and accessibility approach in section 3.1. The GUI follows this approach. Early into development I used separate windows inside the main window for different parts of the GUI. These windows could be toggled and moved. Only the toolbar at the top of the main window was always visible. The reason for this approach was that I wanted the user to be able to customize the arrangement of GUI elements. In the end it proved to complicate the matter because it was not as useful as I had thought and the user ended up having unnecessary options for a very simple matter. Having realised this I reworked the GUI approach by using a static window arrangement. The final arrangement can be seen in the corresponding figure 21. The approach still allows the user to toggle the visibility of everything but the windows are locked in place.

Something that goes hand in hand with the GUI are the input methods. I decided to use a common approach which is heavely based on the mouse. Most GUI elements can only be interacted with via mouse while a few also support keyboard input. I also integrated key-bindings which, for example, allow for faster scene navigation.

[3]From scratchapixel.com/bounding-volume with the owners consent

4 Implementation

The implementation is made up of different components in the sense of divide and conquer. These components will be individually explained, beginning at section 4.2.1 with a general overview. Before that, dependencies and other relevant technologies are introduced to get a picture of the backdrop of the application.

4.1 Dependencies

As a rule of thumb I tried to keep the dependencies as minimal as possible, because it makes building the application easier and avoids including unnecessary functionality.

4.1.1 Vulkan

Figure 7: Vulkan Logo [4]

Vulkan is a cross-platform graphics and compute API. It enables developers to use hardware (mostly GPU) accelerated routines. The Vulkan 1.1 specification launched on March 7th, 2018. The API is supported by AMD, NVIDIA, Intel etc. on a lot of devices. More info can be found on the official website [19].

Common graphics hardware API's nowadays can be divided into 2 groups (not counting pure compute API's). On the one hand we have API's like OpenGL or Microsoft DirectX11 which generate a lot of overhead and do not allow multi-threaded graphics submission. On the other hand there are newer API's like Vulkan or Microsoft DirectX12 which have low overhead and support application side multi-threading. Using these low-overhead API's is generally considered to be more difficult than other comparable API's, because additional work must be dealt with. But it's also more future proof because of multi-threading capabilities, which was the reason I chose a low overhead API. I chose Vulkan over Microsoft DirectX12 or Metal (from Apple Inc.) because it's cross-platform.

The application mostly uses Vulkan in a computing context, because ray-

tracing by itself is a computing task. This is evident by the well known fact that a lot of production rendering systems use software solutions, for example render farms. An interesting side note is that in 2018, Nadjib Mammeri et al. [20] actually published a paper that compares the compute capability of Vulkan drivers to other compute API's like CUDA or OpenCL. While they only measured performance on GPGPU and embedded hardware the results still give a indication as to how Vulkan generally compares to these compute API's. What they found was that Vulkan drivers in comparison have the ability to perform about 1.5 times faster than comparable CUDA and OpenCL implementations. They attribute this mostly to Vulkan's low level synchronisation mechanisms. On the other hand, they mention that performance portability is not guaranteed because of issues like driver implementation quality.

I use the meta-loader volk [21] which enables the application to look for a Vulkan driver at run-time. This made it unnecessary to have a Vulkan dependency at build-time, thus simplifying the building process. The application also supplies necessary vulkan headers to the linker. All in all, this means that the application can be build without an installed vulkan driver, though a vulkan driver is required to use the application.

4.1.2 GLSL and SPIR-V

Figure 8: SPIR-V Logo [5]

Shaders used by the application are written in the OpenGL Shading Language (GLSL). The programming language has a C/C++ style syntax and new versions are rolled out simultaneously with OpenGL updates [22]. Some semantics are Vulkan specific but most of the language can be used as with a OpenGL backend. However, to use these GLSL shaders with Vulkan they need to be converted to SPIR-V binaries. SPIR-V is a intermediate language for parallel computing and graphics and also usable in OpenCL. The current version of SPIR-V is 1.3 and was released on March 7th, 2018 to accompany the launch of Vulkan 1.1 [23]. SPIR-V binaries can be directly loaded by a

vulkan driver for execution.

To create SPIR-V binaries I used a program called glslangValidator, which is hosted on a github repository called glslang [24]. I included the repository as a sub-module in the application repository. It will be downloaded at build time and can be installed by passing a option to CMake. The main CMakeLists.txt contains several examples on how to automatically convert GLSL shaders to SPIR-V binaries using glslangValidator. Note that this is purely optional and only meant to aid the shader compilation process while developing. The README.md contains further information on this topic.

4.1.3 ImGui

For the GUI I wanted to use a library which is both lightweight and sophisticated. It also needed to integrate well with Vulkan. In the end I chose ImGui, which is a proven library used by many applications [25]. I use a C port of this implementation because of 4.1.4. ImGui is an immediate mode GUI implementation, which means that event processing is not based on callbacks but is directly controlled by the user. This makes handling the GUI very simple. The integration of this library is spread out in 4 source files, which contain everything GUI related. ImGui does not directly render the GUI, rather it outputs buffers which the application can choose to render on it's own accord.

4.1.4 Platforms, Programming Language, etc.

It was decided early on that the application should be as portable as possible. For this and other reasons, such as personal preference or compatibility with other programming languages, I used the C programming language to write the application. Mainly because of some necessary alignment specifications, I used the C standard revision ISO/IEC 9899:2011 (better known as C11) which superseded ISO/IEC 9899:1999 or C99. For building the application in a cross-platform capable way I decided to use the meta-builder CMake. At this time of writing building on Linux and Microsoft Windows platforms is possible. Building with Makefile(s) on Linux and VC++ project files (.vcxproj) on Windows was tested but other forms may be possible as well. Because on Windows it's common to use Visual Studio for programming, I want to note that the VC++ project files (or better: .sln solution files) can be loaded into Visual Studio. There are other things to be aware of but for those I want to recommend the README.md in the gitlab repository [1]. It provides further inside regarding dependencies, development, etc..

4.2 Outline

The outline of the implementation is just a high level sketch of the various components that make up the application. This is useful for learning the overarching logic behind the construction of the application. In addition, it provides understanding of how the components connect to each other. More specifically, we will begin with a top-level view of all the components that make up the implementation in section 4.2.1. Then we will take a closer look at the individual components by focusing on explaining some conventions regarding their source structure in section 4.2.2.

4.2.1 Overall Source Structure

The implementation is divided into 7 components which connect to each other in various degrees. This sort of compartimentation provides a good overview at a high level and makes finding specific components easy. It also makes extending the application with new functionality straight forward because you either extend a component or add one. Figure 9 shows these components and their relations in a UML style diagram. The components correspond to the directories found in the source/ folder at the root of the project folder.

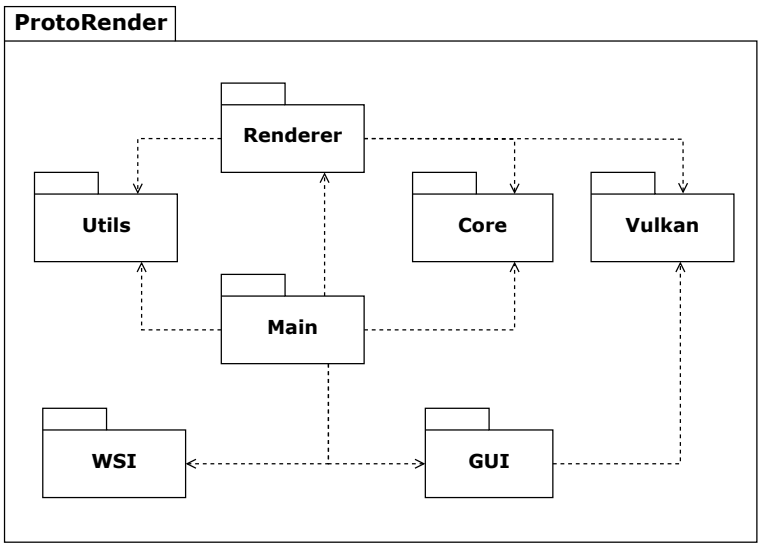


Figure 9: Implementation source structure.

Clearly observable is the central role of the *Main* component. It connects to 5 out of 7 components and can be considered as the controller of the application. This is also where the execution entry point is. The *Main* component thus contains the start-up procedure as well as both rendering

and input loops. The *Renderer* component contains procedures for the main viewport, which means it mostly fulfills rendering related requests. Similar to the *Renderer*, the *GUI* component contains procedures for the GUI including rendering. The remaining 4 parts do not interact between each other in terms of function calls. They are used by *Main*, *Renderer* and *GUI* and can be considered the base of the implementation.

4.2.2 Component Source Structure

The components that make up the application can be seen in figure 9. These components all share a common principle regarding their structural architecture. This principle is based on the C programming language, more specifically on it's key components and 2 design decision follow it.

Design decision 1:

> The first design decision is that functions, structs, enums, etc. of the components are to be contained in different directories. I found after a lot of try and error that this structure works well for C based applications. This is partly because C conceptually is rather simple and it is easy to divide a C program into it's logical units because there are not many to begin with. Combined with the second design decision it makes for a very clean structure.

Design decision 2:

> The second design decision is that all source files are named after what they contain and each file can only contain one elementary component. For example a file named SomeFunction.c contains only a implementation of SomeFunction(...) and nothing else except locally used helper. This one to one mapping of source file names and it's content makes it easy to find and extent components. On the down side it means that a lot of individual source files are needed, but I think it's worth it because compiling individual files in C is generally very fast.

The application of the first design decision can be seen for instance in figure 14 which shows the *Renderer* component of the application and is put in a high level context in figure 9. The *Renderer* directory contains directories which are named and contain source files based on design decision 1, such as

Structs and *Functions*. A special case is the *Resources* directory because it contains binaries instead of source files and can be regarded as an exception of design decision 1. An application of design decision 2 may be seen in figure 11 where source files are named based on the name of the function implementation they contain. It's important to mention that, for a function called PR_someFunction(...), the prefix will be neglected so that the corresponding files are named SomeFunction.h and SomeFunction.c.

4.3 Main

As mentioned in section 4.2.1, the *Main* component is a important part of the application. It contents can be seen in figure 10.

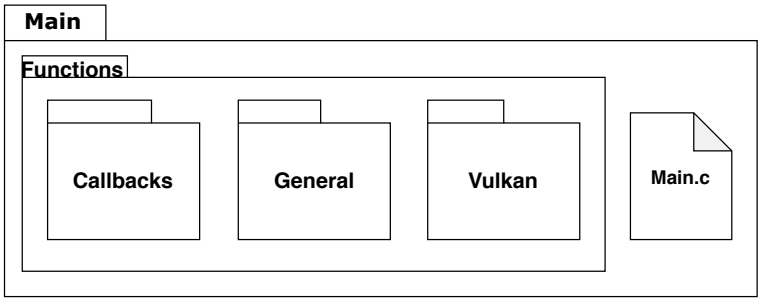


Figure 10: Main source structure.

Main.c holds the execution entry point. For this reason I decided to give it special meaning by placing it outside the *Functions* directory. The functions directory of *Main* is divided into 3 logical units. The first unit is comprised of callback functions which are executed by the Window System Integration (WSI), for more info on this see section 4.3.1. The second unit consists of general functions which are really the heart of the implementation. Those can be further studied in section 4.3.2. Last but not least, the Vulkan related functions of the *Main* component are explained in section 4.3.3.

4.3.1 Callbacks

The callbacks in figure 11 handle events which are triggered by the WSI. Some of those functions are dummies, which means that currently there will be no effect when they are called. Apart from that, callbacks like *MouseCb(...)* or *KeyboardCb(...)* are essential because they handle the respective input devices. They mostly interact with the GUI implementation by changing variables of the GUI structure. The GUI structure is briefly mentioned in

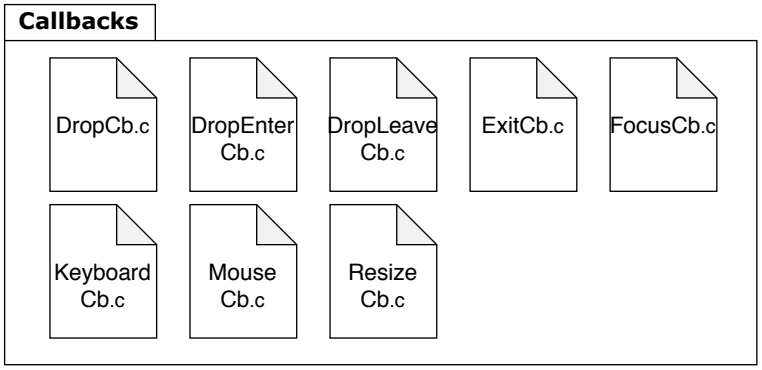


Figure 11: Main callback functions.

section 4.5. The *DropCb(...)* function is also quite important because it initiates the procedure for loading .spheres files. Other than those, the *ExitCb(...)* function is worthy of note since it triggers when a user presses the 'x' button of the window to close the application.

4.3.2 General

The general functions of the *Main* component of the implementation are perhaps the most important to consider. As figure 12 shows, there is a total of 9 functions. If there is a function called outside this directory, the stack trace will probably show that somewhere before there was one of these 9 functions called.

The first function to point out is *StartUp(...)*, which orchestrates the start-up procedure of the application. In the process it first looks for a valid Vulkan driver and initializes Vulkan accordingly. Then it creates representations of the devices for which there are Vulkan drivers. Next, it creates the window and related window resources by calling *CreateWindow(...)* and *CreateWindowResources(...)*. In the end it starts both the render loop and the input loop.
The *RunInputLoop(...)* function asks the WSI in constant intervals if there is new input from the user. If so, it will call the appropriate callback function from section 4.3.1. Because the application is multi-threaded, this loop is handled by a single thread which terminates when the user closes the application.

The *RunRenderLoop(...)* is called by the *StartUp(...)* function and handles all rendering. Similar to *RunInputLoop(...)* it is run by a single thread that terminates on application exit. It calls functions from both GUI and Ren-

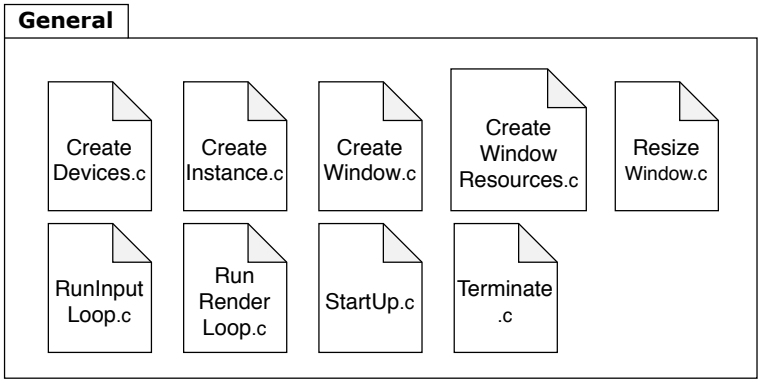


Figure 12: Main general functions.

derer components and sends compute tasks to the Vulkan device in order to present them afterwards. It also handles window resizing and updating the GUI. The statistics that are shown by the application are gathered at various places inside the loop.

The *ResizeWindow(...)* function is actually not as simple as it sounds. It recreates both GUI resources and the VkWindowLink in order to sustain a valid window for the *RunRenderLoop(...)* function to render into. Other than perhaps OpenGL, Vulkan does not offer help in this regard because of its low-level nature.

The *Terminate(...)* function is called by the *ExitCb(...)* function from section 4.3.1. Currently it only sets a variable which causes both *RunRenderLoop(..)* and *RunInputLoop(...)* to stop, thus terminating the application. It could be further extended to initiate procedures to for example save progress before termination.

4.3.3 Vulkan

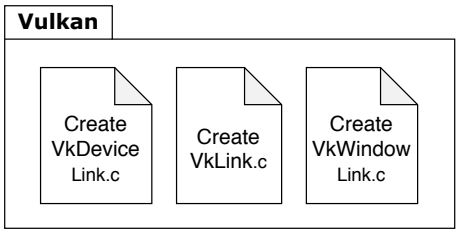


Figure 13: Main Vulkan functions

The Vulkan related functions in *Main* seen in figure 13 do not produce render

resources. They mostly prepare for the actual rendering by setting up Vulkan in various ways.

The *CreateVkDeviceLink(...)* function creates everything that is necessary to use a Vulkan capable device. For this, it sets up things like VkPhysicalDevice, VkDevice, etc. handles. It tells the Vulkan driver of the device which features should be enabled and if validation layers should be activated.

The *CreateVkLink(...)* function creates the VkInstance handle, which is a link to Vulkan on the host machine. This function also insures that a Vulkan loader is present.

The *CreateVkWindowLink(...)* function is primarily called by the *CreateWindowResources(...)* function mentioned in section 4.3.2. It creates everything that is necessary to use a window that was provided by the WSI in combination with Vulkan. For this it creates the VkSwapchainKHR handle etc.. Every window has its own VkWindowLink and the link needs to be reestablished when a window resize event occured.

4.4 Renderer

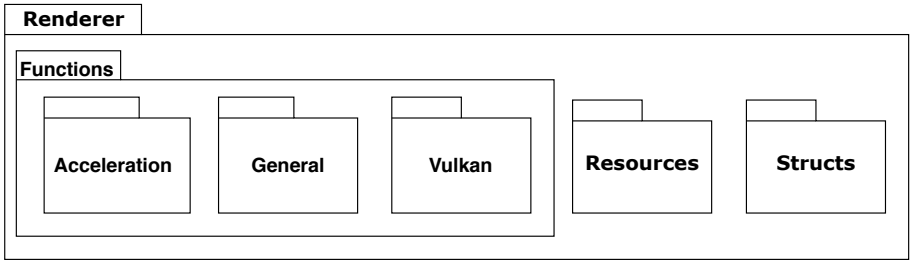


Figure 14: Renderer source structure.

The *Renderer* component is probably the most interesting bit of the implementation (apart from the shaders) in terms of the goal of this bachelor report and can be seen in figure 14. As always, the context of this implementation component can be seen in section 4.2.1. Let me first mention the least interesting thing, which is the *Resources* folder. Here reside the shader binaries that were generated as described in section 4.1.2. The *Renderer* component actually declares some structs of its own and I will briefly explain them in section 4.4.1. The *Acceleration* part contains functions which implement acceleration structures used by the shaders and can be further understood in section 4.4.2. The *General* part contains some helper-functions etc. which

are explained in section 4.4.3 and the *Vulkan* part implements most of the Vulkan procedures that are needed to render the sphere packings.

4.4.1 Structs

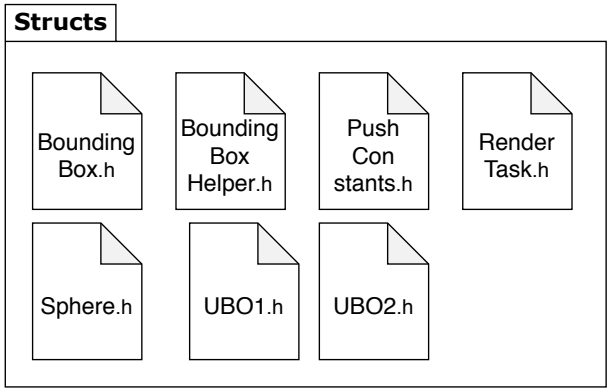


Figure 15: Renderer Structs.

Most of the structures that can be seen in figure 15 are used locally but some are used throughout the implementation. The bounding-box structures are mostly helper which are used for the creation of acceleration structures. For more information on this, see section 4.4.2. The Uniform Buffer Object (UBO) structures are used whenever a shader is created or updated. Almost all shaders rely heavily on UBO's for data which changes between frames. An alternative to UBO's are *PushConstants* and the corresponding structure is used whenever a shader uses PushConstants. PushConstants are rather limited on how much data they can hold, which is around 100 byte depending on the driver implementation. On the flip side, the Vulkan specification mentions that they might perform better, which is why I wanted to try them. I would avoid PushConstants in retrospect because its more work to set them up compared to UBO's and the performance increase is probably marginal.

The *Sphere* structure holds the definition of a sphere. Currently it only contains the position and the radius. The structure should be extended if there needs to be other attributes which might be unique to a single sphere, such as color or weight.

There is also the *RenderTask* structure which is very important and has many use-cases across the implementation. Basically a RenderTask structure contains the state of a single sphere packing while it's being rendered by the application. It also holds most of the Vulkan resources needed to render the sphere-packing.

4.4.2 Acceleration

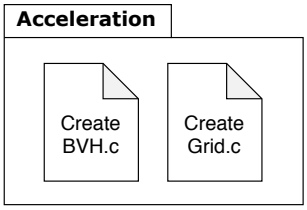


Figure 16: Renderer acceleration functions.

The approach for acceleration can be read in section 3.3. The 2 functions in figure 16 implement the creation of the acceleration structures and some other render resources which are needed in combination with these acceleration structures.

4.4.3 General

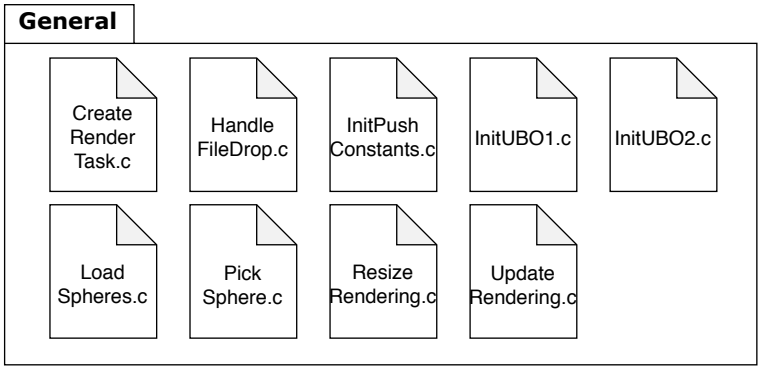


Figure 17: Renderer general functions.

Some functions in figure 17 are just for initialisation and not really worthy of note, for example *InitUBO1(...)*. However, there are other functions which control the creation of render resources and do some software rendering.

The *LoadSpheres(...)* function parses the file path that was transmitted by the *DropCb(...)* from section 4.3.1. It does so in 2 phases, first seeking the number of spheres in the sphere packing. The second phase will actually create the spheres in the form of Sphere structures. It does so by iterating over each line and allocating the spheres as it goes. This is not optimal since it may lead to slow load times on certain systems. Pre-allocation of the necessary resources would be a better solution, but for now the current

implementation suffices. For more info on the Sphere structure see section 4.4.1.

The *PickSphere(...)* function casts a ray through the scene to determine which sphere was selected. This function is executed by the *MouseCb(...)* function from 4.3.1. It does so purely in software, not touching the GPU. I consider this function experimental because there is no use-case for it yet.

4.4.4 Vulkan

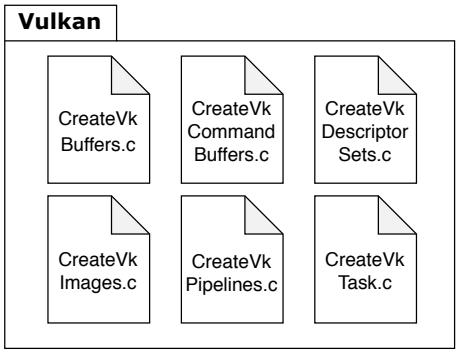


Figure 18: Renderer vulkan functions.

The execution of most functions from 18 is rooted in the function *CreateVkTask(...)*. This function provides a good overview of all resources that need to be created on the vulkan side, to finally create a *VkTask* handle which is needed by the *RenderTask* structure. Vulkan resources are heavily dependent on each other, which means that there is a set sequence in how these resources need to be created: First the function creates *VkBuffer* handles, then *VkImage* handles and after that *VkDescriptorSet* handles. Next the function will create*VkPipeline* handles and based on that *VkCommandBuffer* handles, which is the final step.

The *CreateVkBuffers(...)* function creates *VkBuffer* handles for sphere packing rendering. A *VkBuffer* handle is used to represent general data which is needed by shader programs. For example vertices or UBO's. Similarly the *CreateVkImages(...)* functions creates *VkImage* handles for texture data.

Descriptors created by *CreateVkDescriptorSets(...)* are used to package the data and make it usable.

In the end, pipelines are defined which load shaders and set other parameters, such as the viewport and depth/stencil operations. All these resources

need to be linked by a command buffer so that geometry can be drawn. This is done in *CreateVkCommandBuffers(...)*.

4.5 GUI

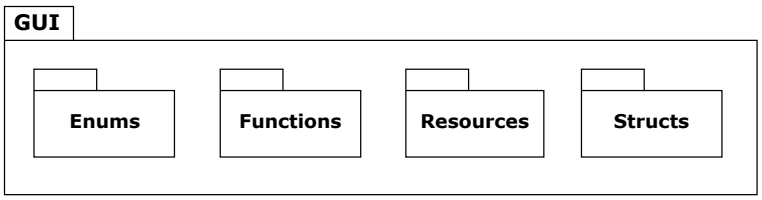


Figure 19: GUI source structure.

The GUI implementation is pretty straight forward and is based on ImGui as explained in section 4.1.3. The source structure that can be seen in figure 19 reflects this. A *Resources* directory contains shader binaries similar to the *Resources* directory in figure 14. I will refrain from explaining the *Enums* and *Structs* directories in detail because they are rather trivial in that it takes one look to understand what goes on. One thing to say though is, that the *GUI* structure in the corresponding directory contains important variables (mostly flags) which trigger GUI logic in the *GetImGuiFrame(...)* function. This specific function is further explained in the next section. The flags are mostly set via the callbacks mentioned in section 4.3.1. *Enums* contains definitions of GUI types that can be used to differentiate between GUI implementations and the application currently makes active use of it, even though there is only one GUI implementation based on ImGui. The reason for that is that in the future it might be desired to use other GUI libraries which can be easily achieved with the current implementation.

4.5.1 Functions

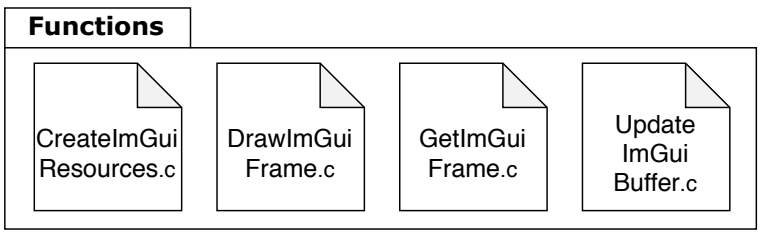


Figure 20: GUI functions.

The functions in figure 20 contain everything that is needed to render the GUI. The render-loop in section 4.3.2 only presents the resources which were created based on these 4 functions. The functions are pretty self explanatory by their name only, so a few words about them except for one will suffice. *CreateImGuiResources(...)* is executed first to lay the basis for GUI rendering. It only needs to be called once for every window size, meaning that in case of a window resize event, new resources need to be created. *DrawImGuiFrame(...)* and *UpdateImGuiBuffer(...)* are called each frame and update the *VkCommandBuffer* handles which in turn updates the GUI rendering.

The *GetImGuiFrame(...)* function defines the look and functionality of the GUI. To read more about the design see the next section.

4.5.2 Design

For the design I implemented a window that is split into 4 elements, as can be seen in figure 21. The red numbers in the figure correspond to a specific window element and explanations for these elements can be read beneath the figure.

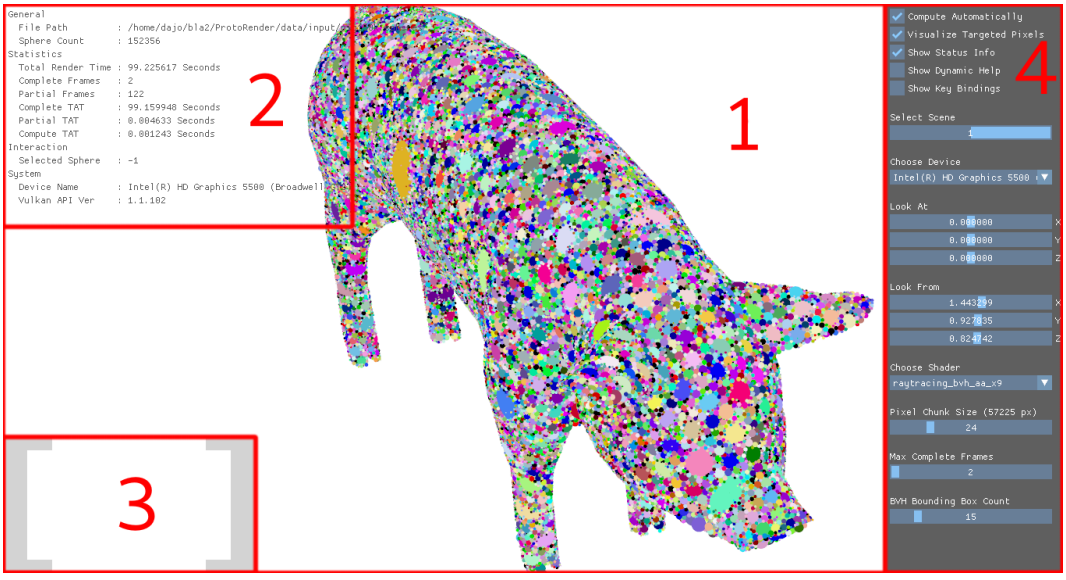


Figure 21: Final GUI design.

The following explanations provide a basic understanding of the purpose of the elements inside the main window. For more detailed insight, the application provides a option to enable a dynamic help mechanic which displays useful information based on where the cursor is hovering. To enable it, toggle

the 'Show Dynamic Help' button in the window on the right hand site (4 in figure 21).

1. The viewport which shows the rendered scene: The viewport spans the entire window and is rendered over by the GUI. Though that doesn't mean that the GUI obstructs the viewport. Each part of the GUI can be individually toggled so that you can hide the GUI completely if you want to. The viewport changes it size whenever the window resizes. This means that I found it necessary to automatically re-render sphere-packings in order to have it fit the updated viewport size. Beware of this because previous renders will be lost when resizing the window.

2. A window with transparent background that shows status info and other useful data: The window visibility can be enabled and disabled. It shows the file path of the currently visible sphere-packing and the total sphere count. More interesting is the part below that because it shows render statistics such as the Turn Around Time (TAT). The TAT information is divided into three parts. The 'Compute TAT' measures how long the Vulkan device takes to render a specific tile. The 'Partial TAT' adds the 'Compute TAT' and the amount of time it takes to do everything else related to the tile. By 'tile' mean the part of the frame that is being rendered. For more information on the tile based rendering approach, see section 3.2.3. Finally, the 'Complete TAT' shows how long it took to render a complete frame.

3. An indicator for the tile that is being computed: The indicator in figure 21 is in the default idle position. The indicator can be toggled and it's size is adjustable. This means that a frame could be rendered using one tile (so basically the entire frame is being rendered in one piece, which means that the tile has the same size as the window) or up to thousands of tiles, depending on the window size. Please note that each tile comes with a slight performance-overhead during rendering, so the fewer tiles are needed the better the performance is usually.

4. The main GUI window for changing settings: This is the heart of the GUI and allows the user to make all kinds of changes. The user can toggle the visibility of windows, enable dynamic help and choose the rendering device. It also features the scene selector where the user can switch between multiple sphere-packings. Using a drop-down menu the user is able to choose the shader and as a consequence, the look of the render. The pixel-chunk size slider allows the user to alter the number of pixels in a tile and visually changes the 3rd element of the GUI. At the bottom the user can find shader-specific options.

Section 8.3 provides a close-up look at the different GUI elements as well as the Metaballs-specific GUI-panel, which has not been mentioned yet. The Metaballs-Panel is shown when the user selects the *metaballs* shader. It enables the user to change weight, threshold and sphere-count parameters.

4.6 Core

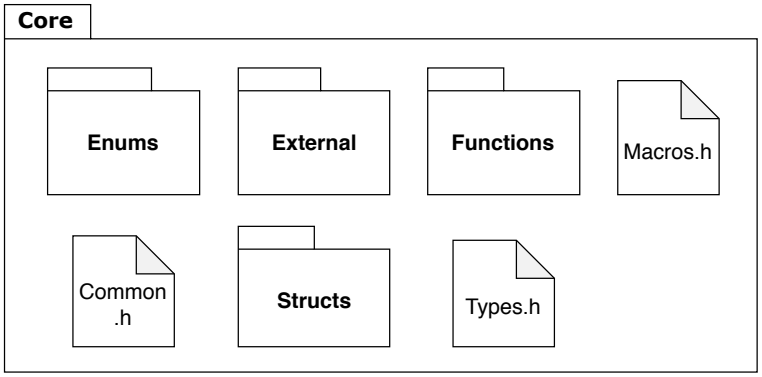


Figure 22: Core source structure.

The *Core* contains things that are used throughout the implementation which have a very basic and abstract nature. This will make more sense after reading section 4.6.1. Figure 22 also shows some features that extend on this principle. For example, it is the place where often used macros are defined (Macros.h). This centralizing approach allows to quickly change or extend macros. Combined with the Types.h file, it allows for fine control over popular used definitions and macros in the entire implementation. It is not needed to include these files by themself, rather I recommend to include Common.h which includes common components such as Macros.h and Types.h. Similar to the structures from the *Structs* directory, enumerations are defined in the *Enums* directory which are widely in use. One novelty is the *External* directory. It contains *include only* header files of external dependencies such as Volk. It's better to indirectly include a dependency by including a file which includes the dependency because it avoids redundancy. This redundancy can appear when definitions are needed to configure the include, which is the case for some of the external dependencies.

4.6.1 Structs

The structures seen in 23 are really the foundation of the application and represent a system independent layer. Hence, they abstract in a way that

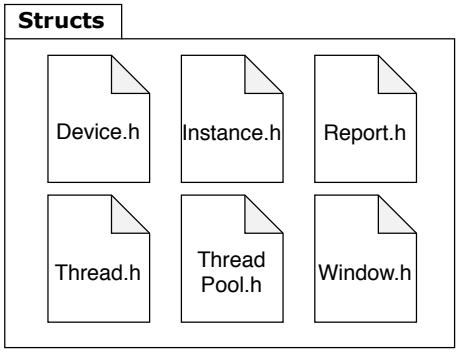


Figure 23: Core structs.

makes it possible to work with them with different operating system targets in mind. The *Device* struct contains information about devices such as GPU's or CPU's which can be used for rendering. The *Instance* contains the state of the program and its data. There is only one *Instance* structure at runtime and it essentially contains pointers to every other resource. The *Window* structure contains, not surprisingly, everything related to the window provided by the Window System Integration (WSI), ranging from Vulkan and general WSI data, to GUI data. None of these structures are global, meaning that everything must be passed to functions in one way or another to use them. In general, the application does not use any global variables or structures, because global data make programs harder to debug. In addition, using the code inside a shared library (which may be something to be considered in the future) becomes almost impossible when a lot of global data is used.

4.6.2 Functions

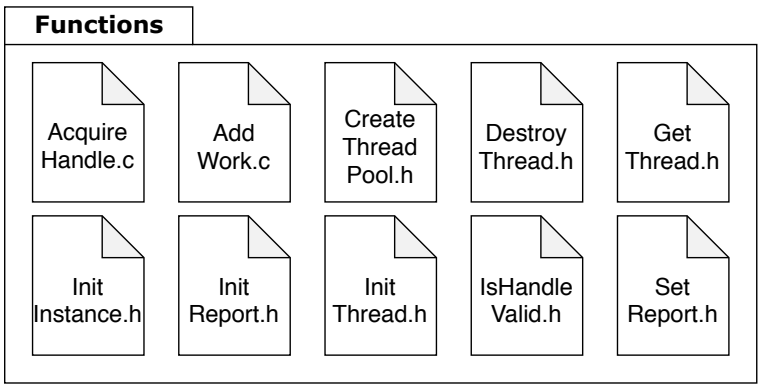


Figure 24: Core functions.

The *Core* comprises functions which implement memory allocation, threading and other fundamental tasks. *InitInstance(...)* is quite important, since it is called at the beginning of execution and configures some variables used throughout the program. The *AcquireHandle(...)* function allocates memory from the thread, as explained in section 3.4. It does so by deciding if a new chunk of a specific type of data should be allocated or if the current chunk is enough. Then it returns a pointer to a memory address inside the chunk.

4.7 WSI

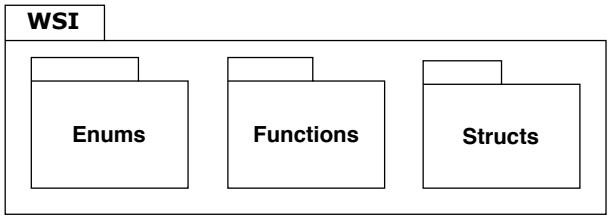


Figure 25: WSI source structure.

The Window System Integration (WSI) executes the callbacks in section 4.3.1. It can be seen as an important layer that allows the abstraction of platform details. The WSI source structure is quite simple, as can be seen in figure 25.

4.7.1 Functions

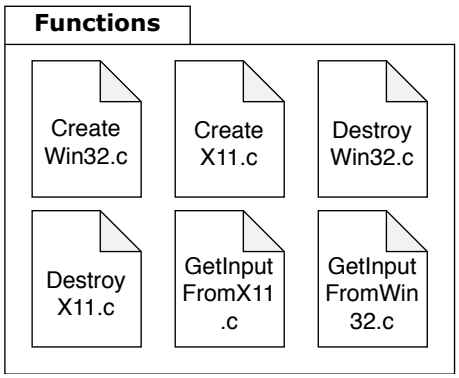


Figure 26: WSI functions.

The functions in 26 are pretty self explanatory. They handle everything related to the corresponding structures. The purpose of the WSI is basically

to provide windows for rendering and to relay user input, such as mouse movement, clicks, and file drops. The *Create* and *Destroy* functions spawn and destroy windows from the operating system. Those windows can then, by association, be queried for input in the *GetInput* functions. The input depending on its nature is then directed to the corresponding callback function in 4.3.1. Input querying is currently done by a single thread which does so in 12 ms intervals. Based on section 4.1.4, these functions need to be implemented twice since they are OS dependent. Because I do not deem it necessary to show the aforementioned structures in a figure, here a little explanation. The two corresponding structures are called *Win32* and *X11*. Naturally they can be found in the *Structs* directory seen in 25. These structures contain OS related WSI resources where *Win32* covers the Window platform and *X11* the Linux platform.

4.8 Vulkan

The *Vulkan* directory has the same structure as seen in figure 25. The directory contains helper functions and structures which are useful when dealing with Vulkan. The enumerations in the *Enums* directory are each used once in 2 functions, so I will refrain from explaining them because they are really more of a detail and their purpose can be examined in the source code.

4.8.1 Structs

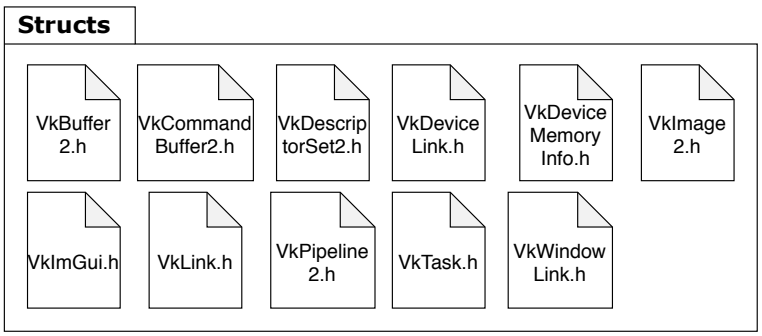


Figure 27: Vulkan helper structures.

Most of the structures seen in figure 27 are just wrapper around the corresponding Vulkan structures. For example, *VkBuffer2* wraps the opaque structure *VkBuffer* and so on. I have done this because some structures are frequently used in combination with each other and it's handy to have them in one place. A *VkBuffer*, for example, does not contain the buffer memory

on its own, for this a separate *VkDeviceMemory* structure is used. So having these 2 structures combined in one structure is very convenient. Most of the other structures follow the same principle.

4.8.2 Functions

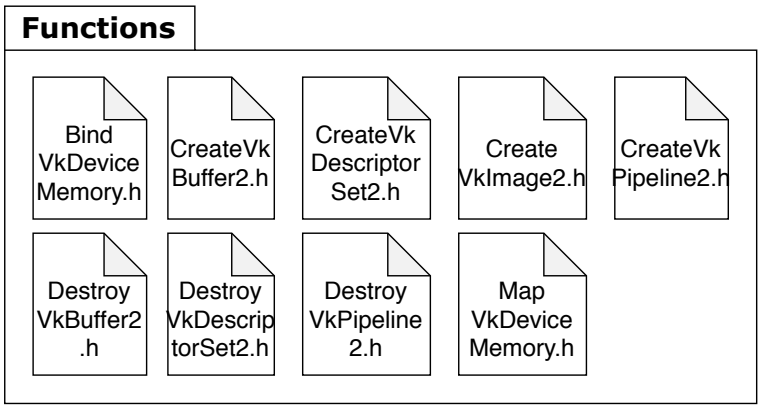


Figure 28: Vulkan functions.

The functions 28 are not very complex and quite similar in purpose, so I won't mention every single one. They are mainly helpful because they abstract some of the explicit procedures that occur quite frequently. For example *CreateVkImage2(...)* creates an image while also binding and mapping the related memory. Additionally it can create the view and sampler of the image. The function name already hinds that it creates a *VkImage2* structure, which is a wrapper of *VkImage* from the Vulkan API.

4.9 Shaders

The shaders have been named according to their function and can be easily identified 29. The *imgui* shaders implement the drawing routine for the GUI.

They are included into the program by the *CreateVkResources(...)* function 20. More precisely: The function creates and passes a pipeline object to the driver which in turn loads and executes the shaders.

The *present* shaders are also quite substantial since they draw images created by the compute (.comp) shaders to the screen. They present, so to say, the computation output. The output from shaders such as *raytracing_grid.comp* is stored in GPU-local memory to achieve maximum access speeds. Based on the double-buffer principle, the program allocates two such memory locations

and alternates between them. The *present* shaders look up the data (color information in the RGBA format) from these memory locations and draw them to the screen. The *present.frag* shader is also responsible for drawing the tile-visualization, which can be seen in figure 19.

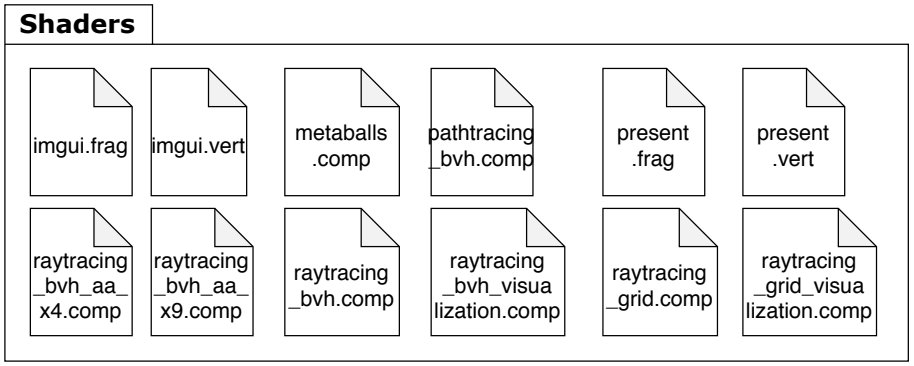


Figure 29: Shaders used by the program.

The only time that the rasterization pipeline is touched, except for GUI drawing, is when the *present* shaders use a rectangle (made up of 2 triangles) to map image data onto. The image data was generated beforehand by a specific compute shader. The rectangle covers up the entire viewport.

Each compute shader implements the tiled rendering routine 3.2.3 so that it's consistent across each use-case, even regarding the metaball shaders. Please note that when looking at the actual shader-directory, you will notice additional metaball shaders with suffixes ranging from 1 to 3. These shaders can be tested by changing the name to *metaballs.comp* and recompiling the application.

The different metaball shaders are all based on the approach laid out in section 3.2.2. The result section 5 covers two of these shaders in detail, the others are merely experiments. All metaball shaders are to varying degrees based on shadertoy-shaders. The source-code references the sources accordingly.

4.10 Mentionable Difficulties

One interesting problem I encountered was that with Vulkan, because of its low level nature, inconsistencies between devices make developing cross-platform a real challenge. For example, Vulkan driver memory layouts of GPU's from different manufactures vary in such a way, that the programmer (under circumstances) has to compensate for these differences. I recommend

the Vulkan Hardware Database [26] for a thorough view of Vulkan capable devices and their differences, ranging from extensions and formats to memory.

Another difficulty regarding portability was actually compiler related. I developed on Linux using GCC. Once I wanted to port the application over to Windows, I had to ask myself which compiler I should use on Windows. An obvious choice would have been MinGW or Cygwin, but those would have resulted in big dependencies. I figured, since most users will probably have Visual Studio installed in one way or another, that the MSVC compiler which comes with Visual Studio is a good choice. What I did not expect was, that the MSVC implementation of the C11 standard is actually not complete. This resulted in some rewrites regarding non-MSVC compliant code sections, such as dynamic arrays. The MSVC compiler also gives some extra warnings which I did not have with GCC, so I mostly ignored them which is quite common for secondary build procedures.

I also have to admit that I underestimated the difficulties of programming a cross-platform application without any big dependencies such as *Qt* or *Gtk* while using a low-level programming language. The development platform was Linux, so the application will perform best on Linux-based systems. I tested four different windows-machines and it ran fine most of the time, but time-constraints did not allow me to fix quirks that I encountered which were hard to debug. The most serious bug I encountered was process-termination on window-resizing. I suspect this has something to do with Vulkan in combination with Windows since I never encountered this on Linux. Thus, cross-platform is more of a experimental feature. In retrospect I would recommend avoiding the low-level route if its not required. It also depends on experience and time constraints.

Last but not least I want to mention that its probably better not to overdo architecture planning. For example, I have a lot of abstraction in places where its not really necessary at this time. A dangerous line of thought occurs when one thinks that something may be necessary in the future. This just adds overhead and slows down initial development. However, I'm fine with the current architecture of the application because in the overall approach, extensibility is pretty important.

[4]From wikimedia.org/Vulkan_API_logo (public domain)
[5]From wikimedia.org/SPIR_logo_2014 (public domain)

5 Results

Here I present the resulting functionality of the application. First, I will focus on the visual fidelity of the implemented rendering algorithms in section 5.1 by looking at some renders and commenting on them. I will also look at the performance using multiple test cases and systems. I will try to draw many implementation parallels, so that this section and section 4 supplement each other as much as possible. In the case of the Metaball-shaders I will actually talk quite a bit about the implementation because its easier to follow explanations when pictures are involved.

5.1 Rendering

5.1.1 Simple ray-tracing

I implemented 3 different types of simple ray-tracing. These types differ only in the 'sharpness' or 'smoothness' of the image they produce. The dragon in figure 30 does not feature any Anti-Aliasing (AA) thus the sphere edges appear to be 'jagged'. The counter example can be seen in figure 31. Here a shader applies AA and the sphere edges appear smooth. However, in it's current implementation this comes with a significant performance drop of factor 2.

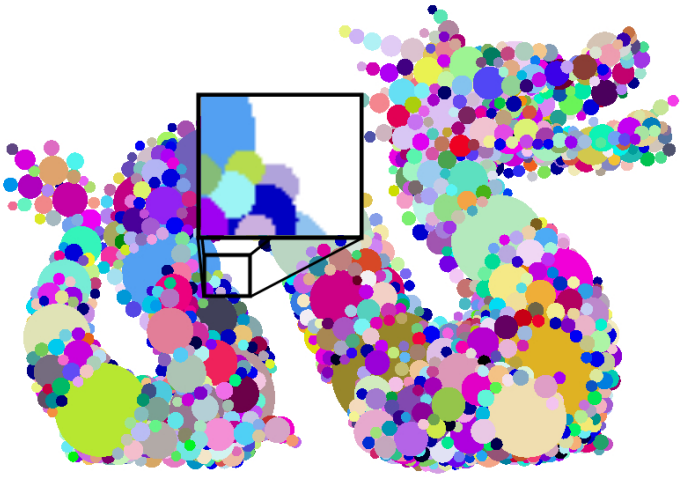


Figure 30: Dragon, 2000 spheres, no AA.

The good thing is that each of these implementations works with BVH acceleration structures. Reasonable performance can be achieved by playing with

the settings regarding the BVH depth. The first algorithm which produces output as seen in figure 30 also works with the grid acceleration structure, which allows performance comparisons.

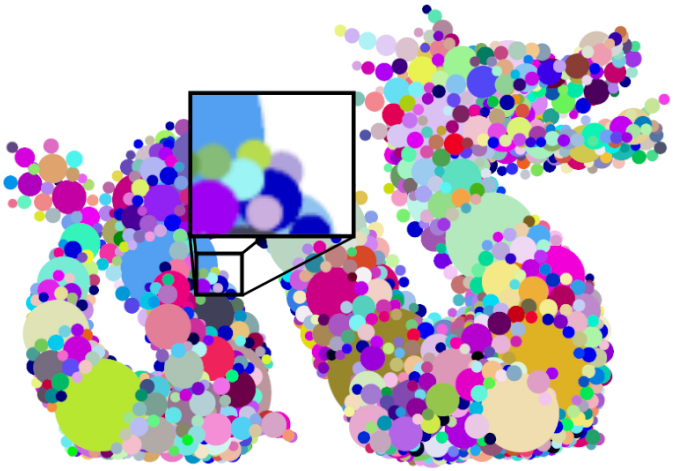


Figure 31: Dragon, 2000 spheres, 9xAA.

Another thing to note is that I encountered visual inconsistencies between different Vulkan driver versions. I used the mesa linux driver (running on a Intel(R) HD Graphics 5500) as the reference Vulkan driver. While the shaders producing figure 30 run fine on all tested systems and drivers, the shaders producing figure 31 caused artifacts on Windows using NVIDIA's proprietary Vulkan driver. As of now I don't know what the cause for this is, because the validation layers do not hint on any likely causes.

5.1.2 Path-tracing

The path-tracing algorithm is by far the slowest, purely ray-tracing based algorithm for rendering that the application supports, but its also the most visually appealing. The algorithm adds ambient occlusion which makes the image look more believable. The quality of the produced image mostly depends on the number of samples the algorithm uses to approximate the ground truth. Figure 32 shows a render with only two samples, thus producing a noisy image. If one is to look closely, he or she would also spot lines running across the image forming up a grid pattern. These are caused by the tiled rendering process and vanish with higher sample count. For example 33 does not show any lines.

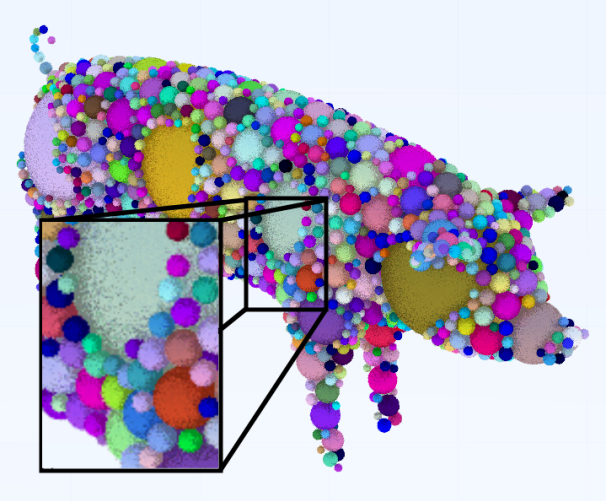


Figure 32: Pig, 2000 spheres, 2 samples.

One other difference from the algorithms from section 5.1.1 is that the path-tracing algorithm does not support the Grid acceleration structure. However, it supports the BVH acceleration structure which speeds up rendering tremendously. I implemented the algorithm because I wanted a visual benchmark which produces some fine looking images. For very quick rendering I recommend to use simpler algorithms like the aforementioned ones.

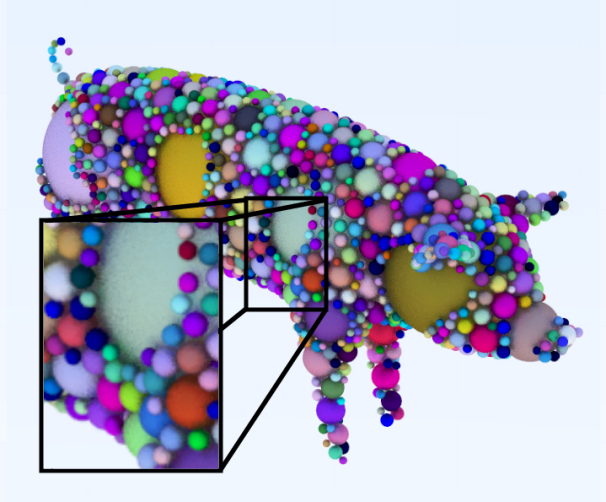


Figure 33: Pig, 2000 spheres, 48 samples.

5.1.3 Visualizer

To make sure that the acceleration structures are valid, I implemented two shaders that visualize the structures calculated by the acceleration algorithms. More info about these can be found in section 4.4.2. Combined with the dynamic nature of the implementation, these visualizations also hold educational value because they show the space partitioning and how it changes depending on the parameters.

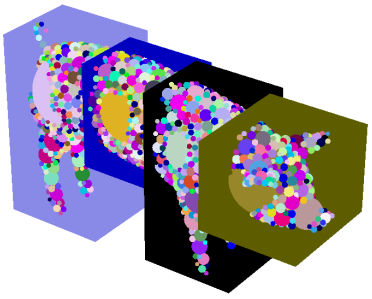


Figure 34: BVH Visualizer, box count: 7

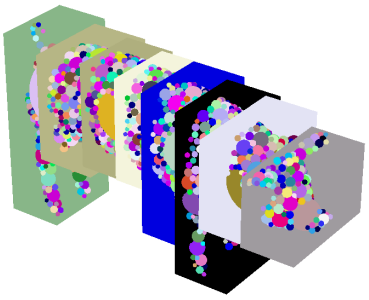


Figure 35: BVH Visualizer, box count: 15

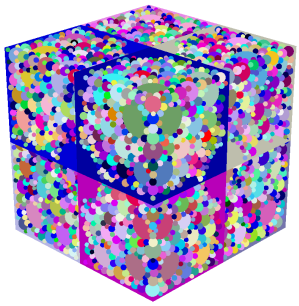


Figure 36: Grid visualizer, cuts: 1

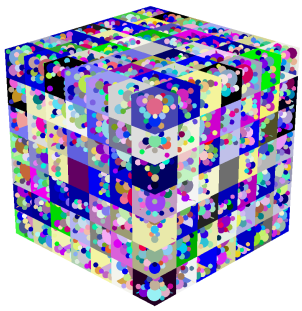


Figure 37: Grid visualizer, cuts: 6

Both visualizers render bounding boxes that contain at least one sphere. Currently there are some overlapping issues from certain perspectives, but for its purpose the implementation suffices. The visualizers can be activated by choosing the corresponding shaders in the shader selection.

As can be seen from figure 34 to 37 the bounding boxes behave as expected depending on the algorithm. Please note that both acceleration algorithms work with any kind of sphere-packing. Something else to note is that the BVH visualizer only renders leaf-boxes.

5.1.4 First Metaballs Shader

The outcome of rendering Metaballs mostly depends on the various variables that can be set in the algorithm. As already mentioned 3.2.2, the implementation allows the user to change the threshold which defines when the charge-sum is big enough to cause the pixel to be colored. I therefore want to show the differences between the rendered images depending on the threshold.

The first figure 38 shows a render-outcome using a Metaball-Threshold of 10. This is the maximum threshold that the user can define and it renders the spheres mostly like the other ray-tracing techniques, that is, you can tell apart each sphere.

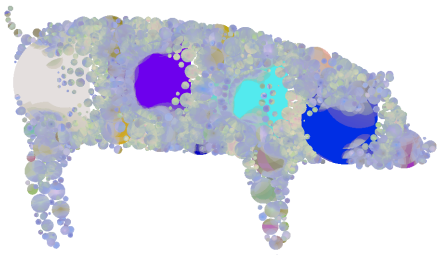


Figure 38: First Metaballs Shader, Threshold: 10, AABB count: 15

The next figure 39 shows a render-outcome using a much lower threshold. As can be seen, it's harder to tell apart each sphere because they merge at specific points. Also, because the threshold is now much lower and thus causing more charge-sums to pass beyond the threshold, the model appears to be thicker.

As a side note, I found that altering the AABB count also alters the Metaball behavior. While a lot of AABB's cause the model to slim down, less AABB's cause it to appear thicker. This is most definitely because more AABB's also means that less charges are accumulated per step, thus causing fewer charge-sums to reach the threshold. The AABB-Metaball relation can be seen when comparing figure 40 and 42. Even though both images where created using two different algorithms to color the pixel it's clearly visible that in the second render, featuring a much lower AABB count, the model has a higher volume then in the first render.

Figure 40 shows a great increase of volume compared to the last two figures. A lot of spheres appear to be merged and the model looks a good bit smoother. The coloring of the surface is based on a mixing algorithm which

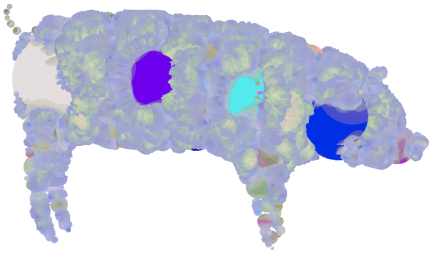


Figure 39: First Metaballs Shader, Metaballs Threshold: 3

Figure 40: First Metaballs Shader, Metaballs Threshold: 1

tries to mix the colors of the different contributing spheres depending on the contribution amount. The algorithm does not perform any interpolation between color values so that's why it looks like there may be clipping issues, when in fact its just the coloring algorithm.

To research the algorithm a little further, I wanted to see what the contribution distribution looks like for the metaball-surface. For this, I changed the algorithm which colors the pixel when the threshold is reached. The new algorithm colors the pixel in the color of the sphere which contributed the most to the charge-sum. Not surprisingly, figure 41 shows an image not to different from the images in section 5.1.1. This means that the each sphere has the highest contribution in there own radius, which is exactly what we want. After that it gets interesting though.

Figure 41: First Metaballs Shader, Threshold: 10, AABB count: 1

Figure 42: First Metaballs Shader, Threshold: 1, AABB count: 1

Figure 42 shows that spheres with relatively immense radius contribute the most to much of the metaball-surface. To see this, compare the colors of the big spheres in figure 41 to the color of the surface of figure 42. The problematic is caused by the different sphere sizes when working with sphere-

packings. The big spheres are packed tightly by the smaller spheres which means that each ray, which is close to a smaller sphere, will also gather a rather large charge from the big spheres. Now because the big spheres have a wide radius, this charge will be greater than most charges gathered from the small spheres, even in the immediate vicinity of these smaller spheres. Only spheres that are far enough from the big spheres or are placed in a secluded area will be rendered individually. This characteristic causes the surface to lose it's features depending on the threshold.

5.1.5 Second Metaballs Shader

I implemented another metaballs shader, which creates images using almost the same technique as 5.1.4. The main difference rendering wise lies in the coloring algorithm which adds lighting and material simulation. The second Metaballs-Shader is also the default one when using ProtoRender. It can be considered as the improved version of the first metaballs shader.

The main points made in section 5.1.4 apply to this shader as well. However, instead of implicitly controlling the volume with the AABB count, the shader uses user-defined weight parameters to scale sphere-contributions depending on sphere-size ranges. The GUI interface can be seen in figure 61.

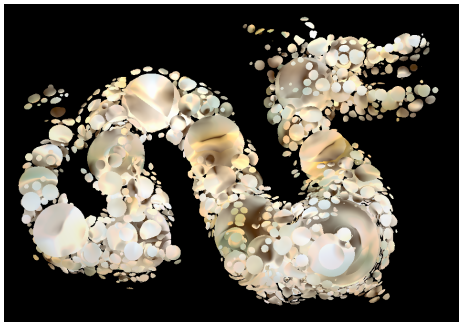


Figure 43: Second Metaballs Shader, Spheres: 1700, Threshold: 0.92

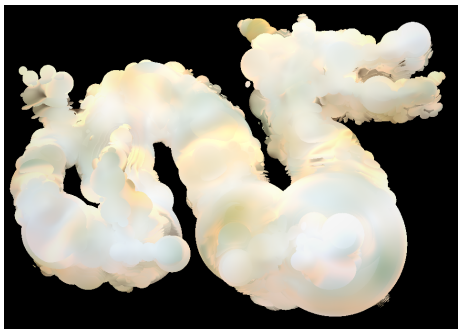


Figure 44: Second Metaballs Shader, Spheres: 1700, Threshold: 1.26

Comparing figure 43 and 44 shows us the same behavior we have already seen in the first metaballs shader, but with better color calculation. The background is black because of the otherwise low contrast but can be changed easily. Despite the improved picture quality, the shader still has some drawbacks though. It can take quite some time to find good weight-values. Also there is some behavior where the shader forms disc-shaped plateaus instead

of merging the spheres. This is noticeable in figure 44. These artifacts can be mostly compensated by finding good weight values.

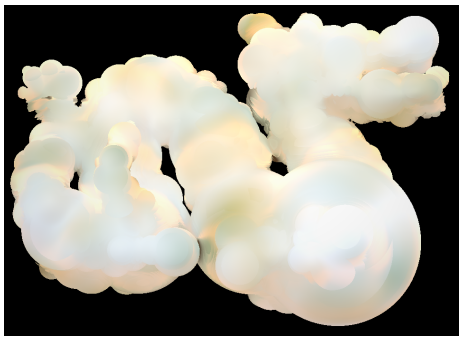


Figure 45: Second Metaballs Shader, Spheres: 1700, Threshold: 1.39

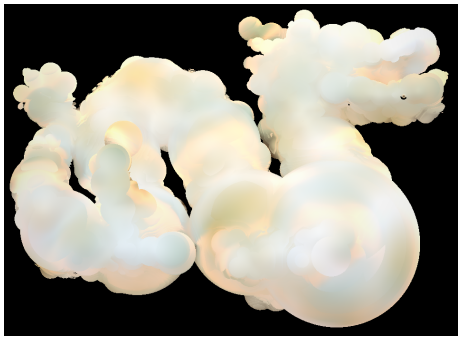


Figure 46: Second Metaballs Shader, Spheres: 2100, Threshold: 1.39

Comparing figure 45 and 46 shows that using a higher sphere count can result in a more accurate representation of the model. On the flip side, a higher sphere count also means drastically lower performance and a higher chance of bad spots.

Additional pictures can be seen in section 8.4.

5.1.6 Miscellaneous

While working on the different techniques mentioned in previous sections, I also came up with other ways to color spheres. One technique can be seen in figure 47. These techniques can be activated by defining specific constants in some shaders.

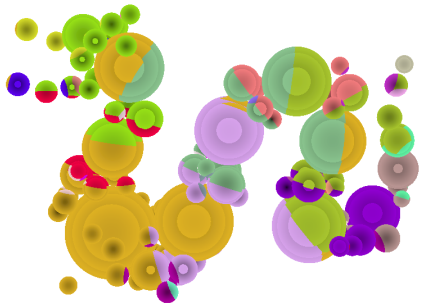


Figure 47: Miscellaneous shader

5.2 Performance

To measure the performance I used different kind of systems ranging from low-end to high-end rigs 1. I tested both Linux (Arch Linux) and Windows (Windows 10) implementations. The tests were performed using 1 integrated (Intel HD Graphics 5500) and 4 external GPU's. The Vulkan drivers of these devices are all based on different API versions 2. There aren't any major differences between these versions except some added extensions.

I prepared different test-cases for each of the implemented algorithms. For the ray-tracing algorithm I tested using sphere counts ranging from 5.000 to 300.000 spheres. The tests for the path-tracing algorithm span sphere-counts from 2000 and up to 100.000 spheres. I set the viewport resolution at 1000x1000 pixel across all tests to make sure that the data is comparable. This is a good middle-ground resolution taking in mind common laptop displays as well as desktop monitors. I also used the same camera position ($x = 3, y = 3, z = 3$) for all measurements which means that the calculations are mostly the same in that regard (different camera angles can cause different amounts of intersections).

GPU	CPU	RAM	OS
Intel HD Graphics 5500	Intel i3 5005U	8 GB	Arch Linux
NVIDIA GTX 1070	Intel Core i5-7600K	16 GB	Windows 10
AMD R9 390	AMD Ryzen 9 3900X	64 GB	Arch Linux
NVIDIA GTX 1080 TI	Intel Core i7-6700K	32 GB	Windows 10
NVIDIA RTX 2080 TI	Intel Core i7-7800X	64 GB	Windows 10

Table 1: Test-Systems

GPU	Vulkan API version	release date
Intel HD Graphics 5500	v1.1.106	Apr 2019
NVIDIA GTX 1070	v1.1.121	Sep 2019
AMD Radeon R9 390	v1.1.113	Jul 2019
NVIDIA GTX 1080 TI	v1.1.109	May 2019
NVIDIA RTX 2080 TI	v1.1.99	Feb 2019

Table 2: Tested Vulkan API versions

For a detailed documentation of the tests and precise values, see PERFORMANCE.txt in the root directory of ProtoRender [1].

Example: The performance-test scene in figure 48 shows most of the dragon model and is a good approximation of a common render scenario. The weird rotation is caused by the default loading process. I did not change the default point of view, which made replication of the test a lot simpler and less error prone. The figure was rendered using the ray-casting algorithm, to test the path-tracing algorithm I used a similar scene.

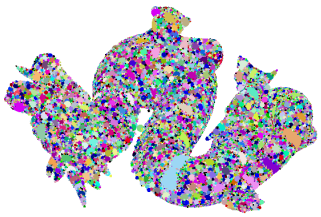


Figure 48: Perfomance-test scene, 50.000 spheres.

Regarding graph tables:

> All performance graph tables follow the same layout. The x-axis defines the used acceleration algorithm. When no acceleration algorithm was used it will say 'No acceleration'. The y-axis represents the seconds it took to render the performance-test scene. Each color at the bottom of the graph-tables represents a specific sphere count. These colors map to the columns which represent individual tests.

5.2.1 Ray-tracing

First of, I tested how fast brute ray-tracing runs. This means that there is no acceleration algorithm in play to discard unnecessary operations. The result is that it's still somewhat interactive at a sphere count of 50.000 using a reasonably fast GPU, for example a GTX 1070 (figure 49).

Further (not documented) tests show, that the algorithm becomes unusable at a sphere count close to and beyond 100.000 (meaning render times longer than 10 seconds) even with fast hardware. This is not surprising, since the

GPU has to iterate relatively complex mathematics over 50.000 spheres about 1.000.000 times each frame. The GPU will execute these operations on many cores (e.g. there are 1920 cores on a GTX 1070) using many threads, but its still quite an undertaking. Taking all this into consideration, it's fairly remarkable what hardware nowadays is capable of.

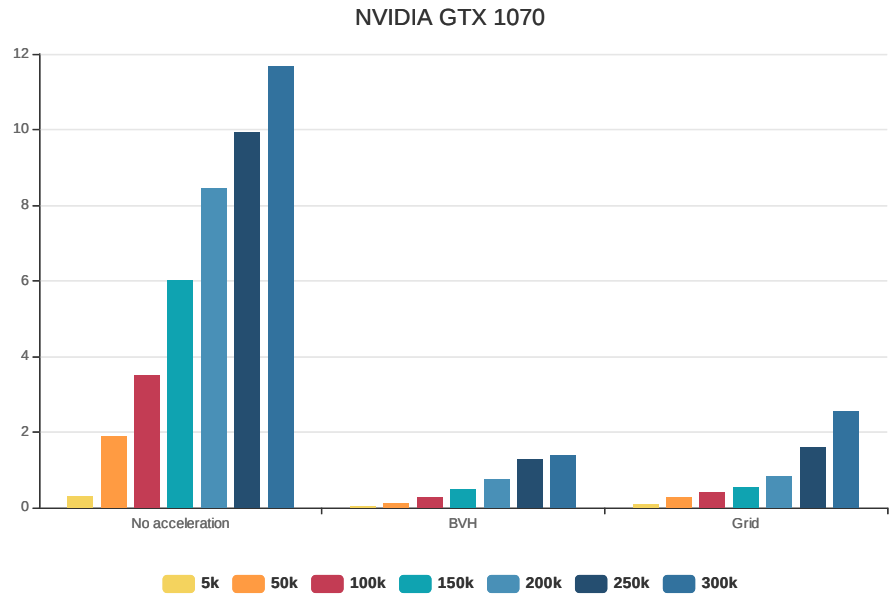


Figure 49: Ray-tracing performance using NVIDIA GTX 1070

It is also interesting to note that while there is about a 20% second performance difference between 49 and 50 using the no acceleration approach, the performance difference between 50 and 51 is even bigger. When comparing certain sphere counts (e.g. 150.000) these differences can be larger than a factor of 1.5. I didn't expect that because although the RTX 2080 TI has more stream processors, the GTX 1080 TI has a higher base clock and only a slightly slower boost clock. This indicates to me that the program benefits from parallel processing even more so then from fast throughput. Hardware architecture changes could also have a hand in this, keeping in mind that the RTX is one of the first consumer grade GPU's that features hardware accelerated ray-tracing (even though the program doesn't make use of it directly).

A render-time speed up of about factor 10 can be measured when rendering using a BVH compared to the brute method. One frame takes about 130 milliseconds on a GTX 1070 with a sphere count of 50000. A RTX 2080 TI renders the scene almost twice as fast, taking only 78 milliseconds. On

low-end hardware, such as the integrated graphics of the Intel i3 processor, the render still finishes in under one second.

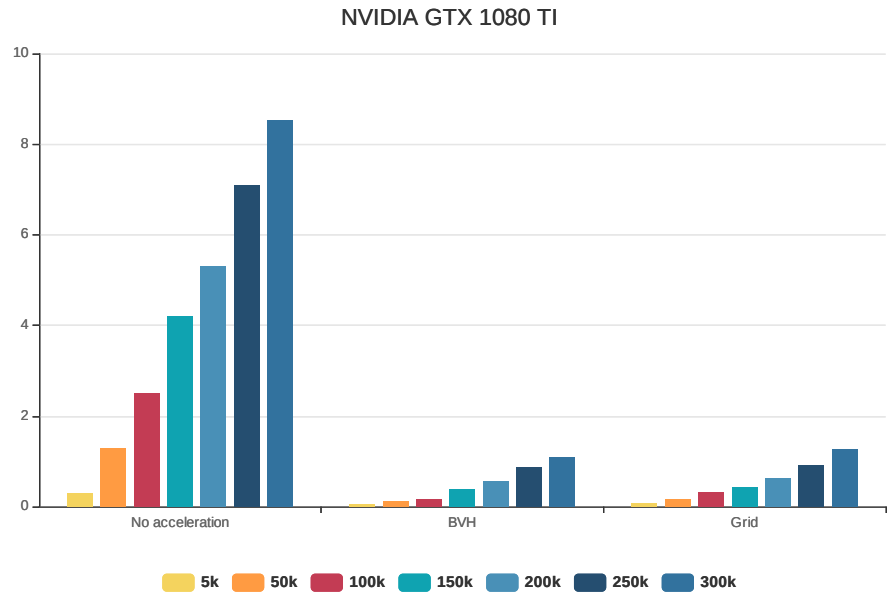


Figure 50: Ray-tracing performance using NVIDIA GTX 1080 TI

The BVH and Grid acceleration algorithms perform quite similar across the bank. However, comparing for example the results of figure 50, there is a tendency to discover that the BVH algorithm performs slightly better. Comparing these measurements, the BVH algorithm is between 10 and 300 milliseconds faster in most cases, than it's counterpart. This is quite interesting and may be explained by the tight fitting nature of the AABB's used in the BVH algorithm. Later tests with a higher sphere count show similar results. It is also interesting to note that NVIDIA cards perform best across all performance tests. NVIDIA and Vulkan have a infamous reputation where it is said, that NVIDIA cards do not perform as well using Vulkan as the AMD counterparts. This is partly caused by differences in architecture and the fact, that Vulkan is based on AMD's Mantle API. The tests do not support these speculations, even though they are not completely meaningful in that regard, since the AMD R9 390 has generally lower specs compared to the other NVIDIA cards.

It is save to say that the performance almost linearly worsens/improves related to the sphere count. Using the GTX 1080 TI 50 as reference, the performance slows down about 200 milliseconds when increasing the sphere count by 50000. This sounds pretty bad but depending on the hardware,

interactive frames can still be achieved even with a high sphere count. For low/mid-range hardware tests, go to section 8.1.

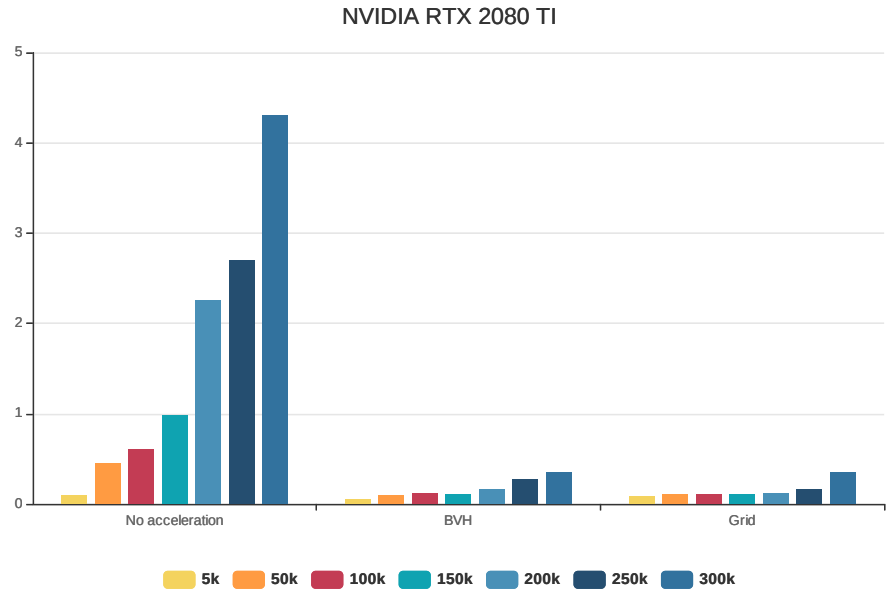


Figure 51: Ray-tracing performance using NVIDIA RTX 2080 TI

5.2.2 Path-tracing

Path-tracing works by accumulating multiple samples of pseudo random generated outcomes. This algorithm principle has great impact on the performance. The tests are all based on a sample count of 16 and render the pig sphere-packing 33. I chose this sample count because at this point, most of the pixels are close to or have converged to the expected result. Apart from these differences, the tests follow the setup method previously mentioned 5.2.

It is quite easy to see that there aren't any big differences between figure 52 and figure 53 regarding not accelerated path-tracing. Some 10 to 20 seconds differences mainly at higher sphere counts, but that was to be expected. Again, what caught my attention was the RTX 2080 TI 54, where even sphere-packings containing 100.000 spheres have finished rendering 16 samples in under 1 minute. You might have noticed that I neglected showing the tests using 75.000 and 100.000 spheres in figure 52 and 53 in the not accelerated section. That's because it took so long to render these sphere-packings that it would have dominated the chart, making it difficult to tell

apart the other tests with lower sphere counts.

The RTX 2080 TI runs the tests regarding not accelerated path-tracing around 3 to 5 times faster than the GTX 1080 TI. This goes in line with observations in section 5.2.1.

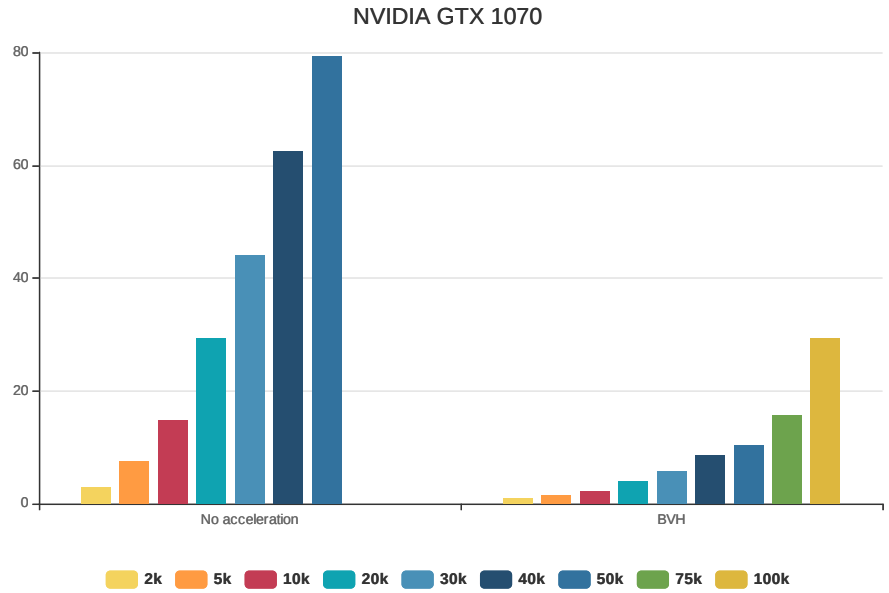


Figure 52: Path-tracing performance using NVIDIA GTX 1070, 16 samples

A huge performance increase can be observed when comparing non accelerated and BVH accelerated path-tracing. The performance increase measures factor 4 to 10, which is similar to section 5.2.1. This means that it becomes feasible to render sphere-packings with a sphere count higher than 50.000 even on mid-range cards, such as 52.

Because bouncing rays are elemental to path-tracing, segmentation via a BVH comes with a loss of shadow, ambient occlusion and color bleeding quality. That's because rays can only hit spheres which are inside the bounding box, otherwise the performance gain using a BVH wouldn't be as drastic. Luckily, at a relatively low bounding box count these quality trait offs are not noticeable, meaning that one can find sweet spots between image quality and performance. For the tests, I tried to find the bounding box count which gives the best performance. Even with obvious quality trait offs, the image quality was still far superior than 5.2.1.

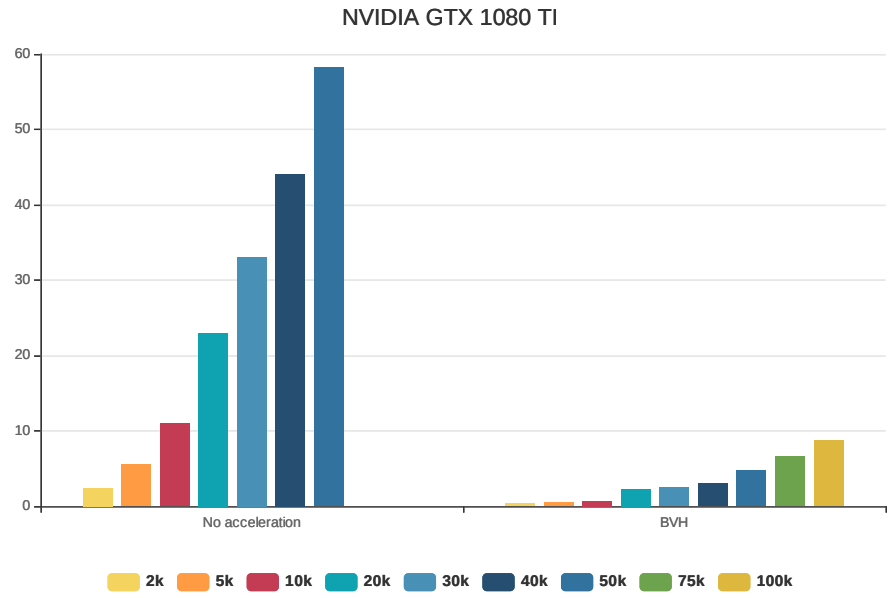


Figure 53: Path-tracing performance using NVIDIA GTX 1080 TI, 16 samples

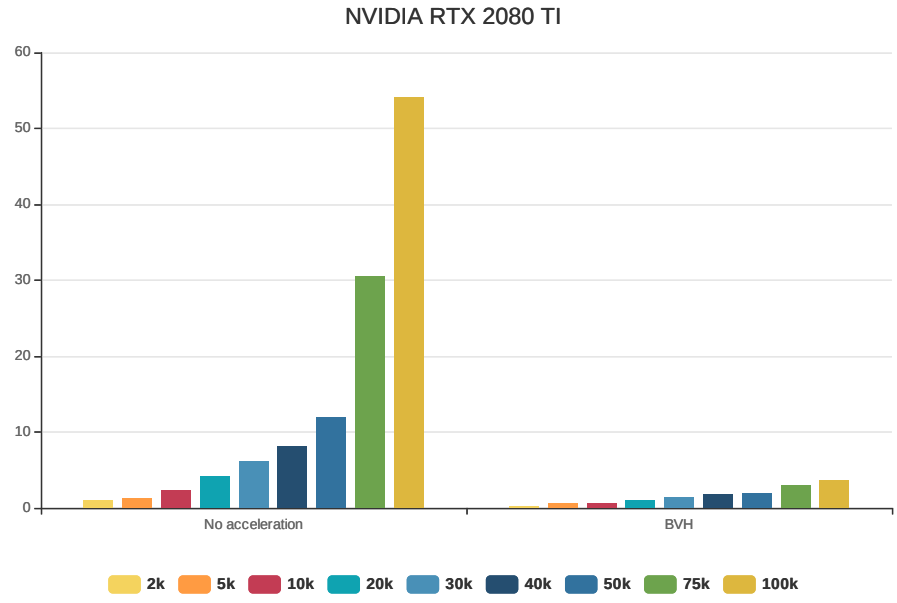


Figure 54: Path-tracing performance using NVIDIA RTX 2080 TI, 16 samples

Using a RTX 2080 TI 54, rendering 100.000 spheres takes only 3.609 seconds, which is more than twice as fast as with a GTX 1080 TI 53. What's also

interesting about the RTX 2080 TI is that there is a lot less difference in BVH accelerated render performance using different sphere counts. It scales much better than the GTX 1080 TI even though they are quite similar specification wise. Additional tests using low to medium-range rigs can be found in the appendix 8.2.

5.2.3 Metaballs

I used the dragon sphere-packing for testing Metaballs performance. The test-scene looks similar to figure 45 and covers about 3/4 of the viewport.

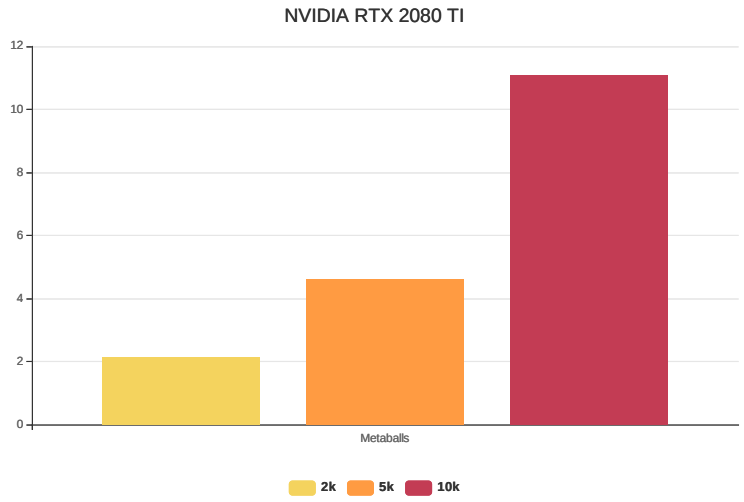


Figure 55: Metaballs performance using NVIDIA RTX 2080 TI

As can be seen in figure 55, the sphere count has massive impact on the performance. Rendering a frame takes between 2 and 2.5 times longer when doubling the sphere count, which is something I expected. In reality, I think it is quite unlikely that someone would need to use this shader on more than 2000 spheres since the output won't change much at that point. In my experience, around 2000 spheres suffice to create a good approximation of the model. Thanks to the possibility to dynamically change the sphere-count when using the metaballs-shader, it is also possible to configure the parameters using a low sphere count and switching to a higher sphere count once all parameters are set.

An additional performance test can be seen in section 8.4. The test shows the same behavior as the first test.

6 Conclusion

In conclusion, I want to say that the task set out in section 1.2 was achieved: A renderer with multiple rendering and acceleration techniques was implemented. The rendering methods are both useful for quick visualizations or for creating presentable images. The performance in the first case is good enough for interactive rendering while the latter case takes longer. However, combining these two methods (e.g. use a fast shader to position the camera and then use a slower, but more realistic shader to make the final render) yields a good user-experience as well as good looking images. Experiments with Metaballs show interesting results and may prove to be a good starting point for future work.

The application is interactive in that it let's the user decide on a lot of details, even regarding the rendering approach. The user can choose the tile size, thus changing the way the application interacts with the GPU. In addition, the user is able to change parameters concerning acceleration activated shaders, with which the complexity (depth or divisions) of these structures can be altered. There are other features (e.g. GPU selection on multi-GPU systems or the dynamic sample count), which all contribute to fulfilling the approach that was set out at the beginning, which said that the application should be extensible and customizable.

The software is based on technology that makes it future proof for the next years to come. This is important, because in the ever so fast world of computer science, relying on established technology is a good way to makes sure that dependencies won't break the program. Even though time constraints based on the 12 CP workload resulted in some features being scrapped (for example further optimization or better Metaballs), I think the implementation in its current state shows good results and can be expanded on in a lot of possible ways. Related works may be faster performance wise, but the approach laid out in this report is feature-rich and simple to work with.

7 Future Work

Future work can be of various nature:

Most of the rendering algorithms can be further optimized, especially meta-balls and path-tracing.

The application could also be ported to macOS. General implementations like memory allocation could be extended like it's mentioned in section 3.4.

The acceleration structures could be improved upon. More specifically, section 3.3.2 talks about further enhancements to the grid acceleration structure. The limitations of the BVH approach 3.3.1 could also be dealt with.

Another thing which could benefit from additional work is the GUI. Some ideas regarding this have been mentioned in section 3.5.

Lastly, it is probably a good idea fix some cross-platform related issues, for more info on that see section 4.10.

8 Appendix

8.1 Ray-tracing Performance

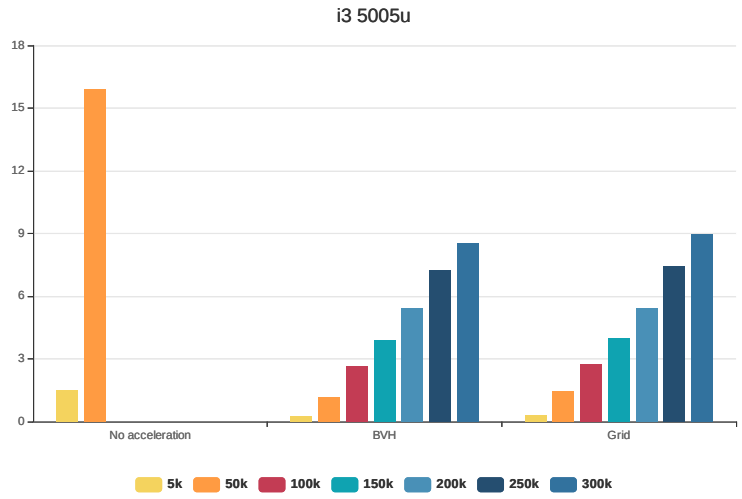


Figure 56: Ray-tracing performance using i3 5005u

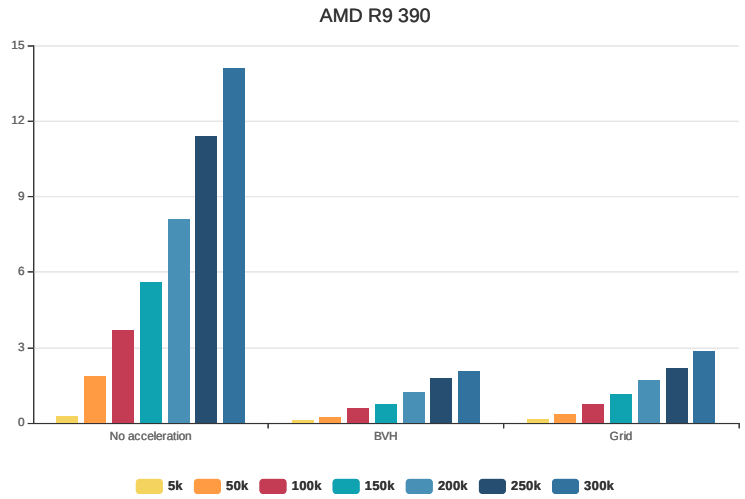


Figure 57: Ray-tracing performance using AMD R9 390

8.2 Path-tracing Performance

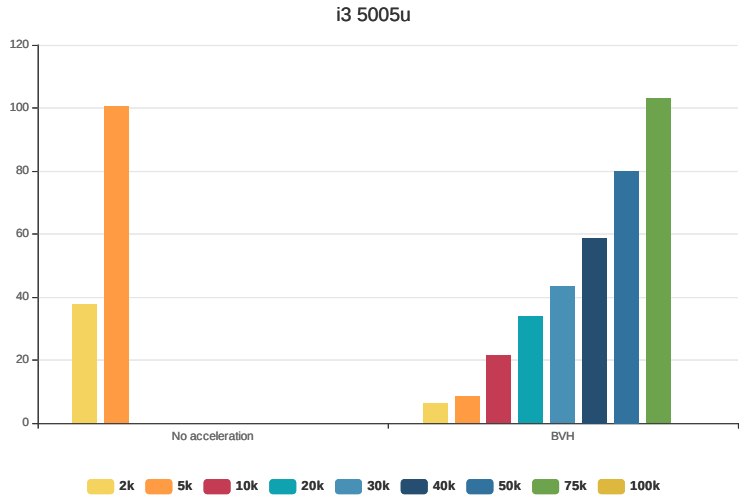


Figure 58: Path-tracing performance using i3 5005u, 16 samples

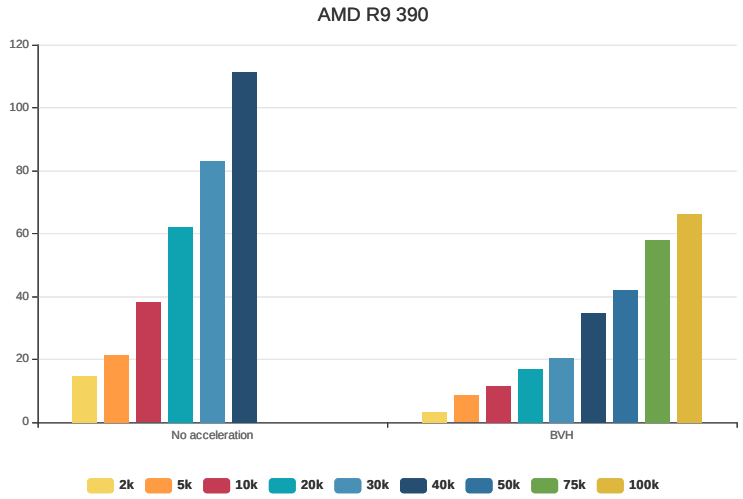


Figure 59: Path-tracing performance using AMD R9 390, 16 samples

8.3 GUI close-ups

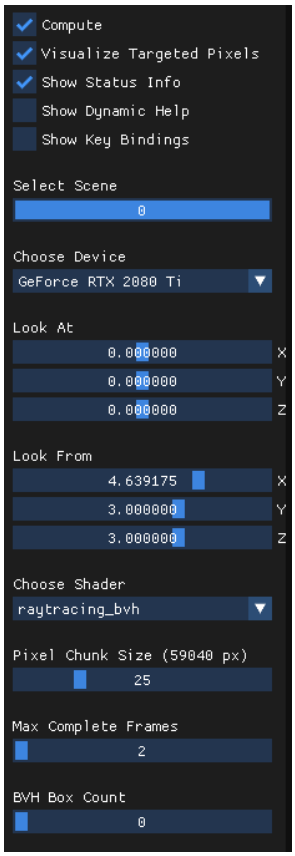


Figure 60: Main-Panel

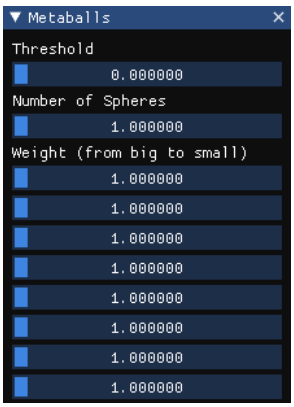


Figure 61: Metaballs-Panel

```
General
  File Path          : /home/dajo/ProtoRender/data/input/dragon50k.spheres
  Sphere Count       : 50000
Statistics
  Total Render Time  : 2.118115 Seconds
  Complete Frames    : 2
  Partial Frames     : 40
  Complete TAT       : 1.051743 Seconds
  Partial TAT        : 0.006159 Seconds
  Compute TAT        : 0.002198 Seconds
Interaction
  Selected Sphere    : -1
System
  Device Name        : Intel(R) HD Graphics 5500 (Broadwell GT2)
  Vulkan API Ver     : 1.1.102
```

Figure 62: Status-Panel

8.4 Metaballs

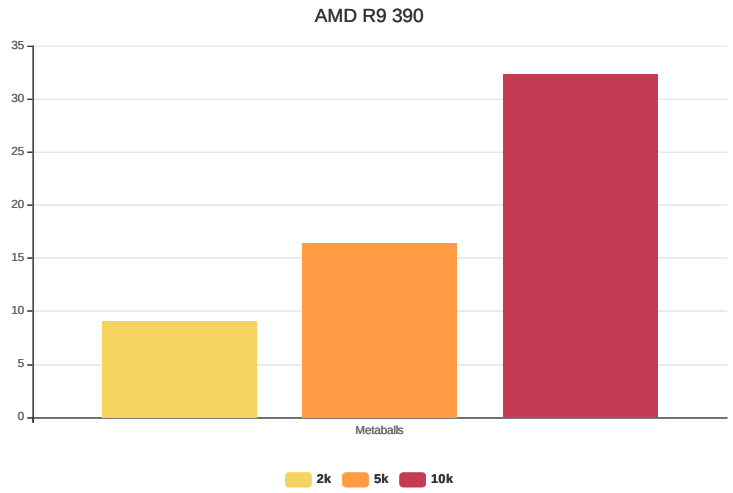


Figure 63: Metaballs performance using AMD R9 390

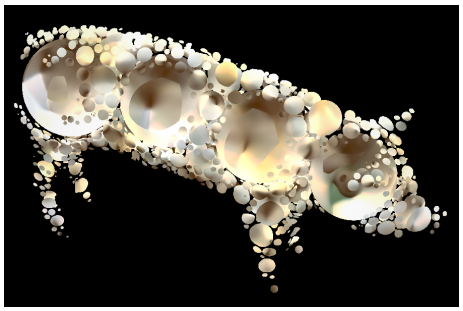


Figure 64: Second Metaballs Shader, Spheres: 1800, Threshold: 0.93

Figure 65: Second Metaballs Shader, Spheres: 1800, Threshold: 1.29

List of Figures

List of Tables

Listings

References

[1] Dajo Frey. (2019) Protorender. [Online]. Available: https://gitlab.informatik.uni-bremen.de/s_2otc2b/ProtoRender

[2] Rene Weller, Gabriel Zachmann. (2010) Protosphere: A gpu-assisted prototype guidedsphere packing algorithm for arbitrary objects. [Online]. Available: http://www.natural-interaction.de/papers/siggraph_asia2010/ProtoSphereSiggraph.pdf

[3] ——. (2009) A unified approach for physically-based simulations and haptic rendering. [Online]. Available: http://www.natural-interaction.de/papers/siggraph09/IST-SiggraphGames.pdf

[4] Scratchapixel. An overview of the rasterization algorithm. [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm

[5] Nvidia News Center. (2019) From rasterization to full real-time path tracing: The evolution of graphical rendering techniques. [Online]. Available: https://news.developer.nvidia.com/from-rasterization-to-full-real-time-path-tracing/

[6] Kevin Beason. (2014) smallpt: Global illumination in 99 lines of c++. [Online]. Available: https://www.kevinbeason.com/smallpt/

[7] Marc Levoy, Turner Whitted. (1985) The use of points as a display primitive. [Online]. Available: https://graphics.stanford.edu/papers/points/

[8] Lee Westover. (1989) Interactive volume rendering. [Online]. Available: https://dl.acm.org/citation.cfm?id=329138

[9] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, Markus Gross. (2000) Surfels: surface elements as rendering primitives. [Online]. Available: https://dl.acm.org/citation.cfm?id=344936

[10] Turner Whitted. (1980) An improved illumination model for shaded display. [Online]. Available: https://dl.acm.org/citation.cfm?id=358882

[11] Arthure Appel. (1968) Some techniques for shading machine renderings of solids. [Online]. Available: https://dl.acm.org/citation.cfm?id=1468082

[12] James T. Kajiya. (1986) The rendering equation. [Online]. Available: https://dl.acm.org/citation.cfm?id=15902

[13] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, Philipp Slusallek. (2007) Realtime ray tracing on gpu with bvh-based packet traversal. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4342598

[14] James F. Blinn. (1982) A generalisation of algebraic surface drawing. [Online]. Available: https://ohiostate.pressbooks.pub/app/uploads/sites/45/2017/09/blinn-blobby.pdf

[15] Agata Opalach, Steve Maddock. (1995) An overview of implicit surfaces. [Online]. Available: https://www.researchgate.net/profile/Steve_Maddock/publication/2615486_An_Overview_of_Implicit_Surfaces/links/0fcfd50e6f958ad10e000000/An-Overview-of-Implicit-Surfaces.pdf

[16] Johannes Meng, Marios Papas, Ralf Habel, Carsten Dachsbacher, Steve Marschner, Markus Gross, Wojciech Jarosz. (2015) Multi-scale modeling and rendering of granular materials. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.699.5457&rep=rep1&type=pdf

[17] Christian Sigg, Tim Weyrich, Mario Botsch, Markus Gross. (2006) Gpu-based ray-casting of quadratic surfaces. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.3208&rep=rep1&type=pdf

[18] Ryan Geiss. (2000) Metaballs (also known as: Blobs). [Online]. Available: http://www.geisswerks.com/ryan/BLOBS/blobs.html

[19] Khronos Group Inc. (2019) Vulkan overview - the khronos group inc. [Online]. Available: https://www.khronos.org/vulkan/

[20] Nadjib Mammeri, Ben Juurlink. (2018) Vcomputebench: A vulkan benchmark suite for gpgpu on mobile and embedded gpus. [Online]. Available: https://ieeexplore.ieee.org/document/8573477

[21] zeux. (2019) volk - meta loader for vulkan api. [Online]. Available: https://github.com/zeux/volk

[22] Khronos Group Inc. (2019) Core language (glsl). [Online]. Available: https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)

[23] ——. (2019) Spir overview. [Online]. Available: https://www.khronos.org/spir/

[24] ——. (2019) Khronos-reference front end for glsl/essl, partial front end for hlsl, and a spir-v generator. [Online]. Available: https://github.com/KhronosGroup/glslang

[25] ocornut. (2019) Dear imgui: Bloat-free immediate mode graphical user interface for c++ with minimal dependencies. [Online]. Available: https://github.com/ocornut/imgui

[26] Sascha Willems. (2019) Vulkan hardware database by sascha willems. [Online]. Available: http://vulkan.gpuinfo.org/listdevices.php