FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

High-Quality Rendering of Interactive Particle Systems for Real-Time Applications

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Alexander Heinz
Matrikelnummer 01426648

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Johannes Unterguggenberger, BSc

Wien, 29. April 2019

Alexander Heinz	Michael Wimmer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

High-Quality Rendering of Interactive Particle Systems for Real-Time Applications

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Alexander Heinz
Registration Number 01426648

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. Johannes Unterguggenberger, BSc

Vienna, 29th April, 2019

Alexander Heinz	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Alexander Heinz
2620 Mollram, Jägerweg 5 Haus 3

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. April 2019

Alexander Heinz

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Programmierung und Verfassung dieser Bachelorarbeit unterstützt und motiviert haben. Ohne sie würde diese Arbeit nicht in dieser Form vorliegen.

Zuerst gebührt mein Dank Univ.Ass. Dipl.-Ing. Johannes Unterguggenberger BSc, der meine Bachelorarbeit betreut und begutachtet hat. Für die konstruktive Kritik und die zahlreichen vielseitigen Hilfestellungen möchte ich mich herzlich bedanken.

Ebenfalls möchte ich mich bei Lukas Gersthofer BSc bedanken, der mich während der praktischen Arbeit unterstützt hat. Bedanken möchte ich mich für die vielen Ideen, Fehlerfindungungen und der technischen Wissensübetragung.

Abschließend möchte ich mich bei meinen Eltern Nina Heinz und Christian Freiler bedanken, die mir mein Studium durch ihre Unterstützung ermöglicht haben.

Acknowledgements

In this section I would like to thank all those who supported and motivated me during the programming and writing of this bachelor thesis. Without them this work would not be as it is.

First, thanks to Univ.Ass. Dipl.-Ing. Johannes Unterguggenberger BSc, who supervised and examined my bachelor thesis. For the constructive criticism and the many versatile help I would like to thank you.

I would also like to thank Lukas Gersthofer BSc, who supported me during the practical work. I would like to thank you for the many ideas, error finding and the technical knowledge transfer.

Finally, I would like to thank my parents Nina Heinz and Christian Freiler, who made my education possible through their support.

Kurzfassung

Partikelsysteme sind weit verbreitet in Echtzeitanwendungen. In dieser Arbeit werden aktuelle Techniken zur Darstellung und Simulation von Partikelsystemen beschrieben und verglichen. In der Computergrafik werden Partikelsysteme verwendet, um Flüssigkeiten wie Wasser, gasähnliche Stoffe wie Feuer und Rauch oder Effekte wie Explosionen und Feuerwerke darstellen zu können. Im Fall von Wasser könnte man jeden einzelnen Partikel als ein oder mehrere Wassertropfen interpretieren. Bei einem Feuerwerk wären das die Funken und Feuer bzw. Rauch könnte durch Überlagerungen von größeren Partikeln simuliert werden. Im Kontext dieser Arbeit werden alle beschriebenen Vorgehensweisen entweder dem Rendern, also der Darstellung, oder der Simulation zugeordnet.

Um ein Partikel mit Hilfe einer Grafikkarte zu rendern gibt es viele verschiedene Techniken. Bei der wohl einfachsten Variante wird eine Textur auf ein Quadrat, welches immer zur Kamera gerichtet wird, gezeichnet. Dieses Verfahren wird auch *Billboarding* genannt und kann sehr einfach und effizient implementiert werden. Ein Nachteil ist, dass die Kanten des gezeichneten Quadrates sichtbar werden können, wenn sich Bereiche des Partikels mit anderen Gegenständen in der Szene überlappen und somit das volumetrische Erscheinungsbild verändert wird. Um dies zu verhindern kann die Transparenz des Quadrates in der Nähe anderer Gegenstände vermindert werden, um den harten Übergang etwas weicher zu machen. Diese Vorgehensweise wird auch *Soft Particles* genannt und wird verwendet, weil es die Darstellungsqualität der Partikel erhöht.

Als Simulation, welche den wesentlichen Kern dieser Arbeit bildet, versteht man die Annäherung an ein physikalisches Verhalten, welches durch Manipulation der Daten erreicht wird. Dabei geht es also um das Verändern, Erstellen und Löschen von Partikeln. Ein Regentropfen-Partikel würde als Beispiel am Himmel bei einer Wolke erstellt werden, dann hinunter fallen und am Boden wieder gelöscht werden. Es liegt Nahe, dass ein System aus sehr vielen Partikeln bestehen kann bzw. sollte, um das zu approximierende Phänomen zu erhalten. In Echtzeitanwendungen sollten alle Partikel im optimalen Fall mindestens 60 mal in der Sekunde simuliert werden können, jedoch ist dies nicht zwingend nötig bei der Art der Simulation, die dieser Arbeit zugrunde liegt. Darum ist es besonders wichtig, dass die Simulation möglichst effizient ausgeführt werden kann. In dieser Arbeit werden drei verschiedene Grundtechniken zur Simulation beschrieben und verglichen. Die zwei wesentlichen Unterschiede sind die Verwaltung des Speichers der Daten und die Art der Simulationsberechnung. Während bei der einen Technik die Daten im Hauptspeicher

gespeichert und die Berechnungen auf der CPU durchgeführt werden, werden diese zwei Prozesse bei den anderen beiden Techniken direkt auf die GPU ausgelagert. Am Ende eines Simulationsschrittes wird überprüft, ob das Partikel mit anderen Objekten kollidiert ist. Wenn dies der Fall ist, wird das Partikel dementsprechend reflektiert. Um durchsichtige Strukturen, wie Rauch, richtig darzustellen, müssen die Partikel nach ihrer Distanz zur Kamera sortiert und - beginnend mit dem weitest entfernten - gerendert werden. Um das zu erreichen, werden solche Partikel nach der Simulation entsprechend sortiert. Auf der CPU kann die Partikelliste ganz einfach mit klassischen Sortieralgorithmen sortiert werden, aber auf der GPU werden andere Algorithmen, wie der *Bitonic Merge Sort*, benötigt.

Abstract

Particle systems are widely used in real-time applications. This thesis presents and compares several state-of-the-art methods for rendering and simulating particle systems. In computer graphics, particle systems are used to represent fluids like water, gas-like substances like fire and smoke or effects like explosions and fireworks. In the case of water, each particle can be interpreted as one or more waterdrops. To simulate a firework, the particles can be seen as sparks, and for smoke or fire big particles can be used and blended over each other. In the context of this thesis all described techniques can be associated with either rendering or simulation.

For rendering a single particle using the graphics processing unit (GPU), several methods exist. Probably the easiest would be to draw a texture onto a quad that is always looking towards the camera. This is called *Billboarding* and can be implemented very easily and efficiently. One drawback is that the hard edges of the drawn quad can become visible, when parts of the particle overlap with other objects in the scene. In this case the volumetric appearance of the particle can easily be destroyed. In order to avoid this, the transparency can be adapted near the region of overlap in such a way that the hard edges are smoothed out. This method is called *Soft Particles* and is used, because it increases the picture quality of the particle.

The simulation, as the main topic of this work, can be interpreted as the approximation of a physical behavior, that can be achieved by manipulation of the data. Therefore it is about the changing, creation and deletion of particles. A raindrop-particle, as an example, is created in the sky in a cloud, then fall towards the ground, and at the ground the particle gets deleted. It is obvious that a system may contain a high number of particles in order to obtain the phenomenon to be approximated. In real-time applications, all particles should ideally be simulated at least 60 times per second, but this is not absolutely necessary in the type of simulation, that underlies this work. Therefore, an efficient method for simulation is very important. In this work three different basic methods for simulation are presented and compared. The two most important differences are the memory management and the way of computation of the simulation. While in one technique the particle data is stored in the main memory and the computation is done by the CPU, these two processes are outsourced to the GPU, when using one of the other methods. At the end of each simulation step the particles are checked against collision with other scene objects. In case of a collision the particle is reflected respectively. In

order to represent transparent substances like smoke properly the particles have to be sorted and - starting with the farthest away - rendered. The easiest way would be to sort the particles according to their distance to the camera. When simulation is done on the CPU this can be easily achieved using classical sort algorithms, but when storing the particle data on the GPU other algorithms, like the *Bitonic Merge Sort*, have to be used.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem statement . . . 1
1.3 Aim of the work . . . 2
1.4 Methodological approach . . . 2
1.5 Contributions . . . 2
1.6 Structure of the work . . . 3

2 State-of-the-art **5**
2.1 Basic concepts . . . 5
2.2 Particles . . . 8
2.3 Fire and Smoke . . . 11
2.4 GPU sorting . . . 12
2.5 Collision detection and handling . . . 14
2.6 Order-Independent Transparency . . . 15

3 Methodology **17**
3.1 Used technologies . . . 17
3.2 Rendering . . . 17
3.3 Simulation . . . 24
3.4 Sorting . . . 29
3.5 Collision detection and handling . . . 30

4 Results **31**
4.1 Visualizations . . . 31
4.2 Analysis and performance . . . 33

5 Conclusion and Future Work **37**

List of Figures **39**

List of Tables **41**

List of Algorithms **43**

Bibliography **45**

CHAPTER 1

Introduction

This bachelor thesis deals with the simulation and rendering of fire and smoke effects in real-time applications, using billboarding-based rendering of particle systems. In this chapter, at first the motivation for this work is given. After that, the general problem statement and the aim are defined. Then, the implemented methodological approach is presented. The last section of this chapter shows the structure of the following chapters of this thesis.

1.1 Motivation

It is the time, where people are looking for more and more realistic and interactive representations and visualizations of all known phenomena. A reason for this is the ever-increasing demand for knowledge and understanding and in some cases this is more and more available, due to the increasing strength of hardware. After closer examination, it can be seen that some phenomena consists of very many small dynamic structures. As examples, water, consisting of water drops, or the little sparks of a campfire can be stated. And even after going a few steps further to atom level, it is obvious that everything can be interpreted as molecules. A way to approximate all these small structures within a phenomenon would be to interpret them as one or more particle systems. Therefore the need for having efficient methods for simulating and rendering particles is big.

1.2 Problem statement

The problem statement was to compute an interactive fire beam in real-time, including interacting on scene obstacles. A flame, for example, could be assumed to consist of fire particles and smoke particles. The challenge is to model these particle systems in such a way, that they can approximate liquid and gas like substances like fire and smoke. Therefore each system has to pretend to be one big dynamic substance instead of many

single particles. In order to achieve this, a huge amount of particles can be needed. Another challenge is to process a huge amount of particles efficiently, ideally 60 times per second, in order for this technique to be suitable for real-time applications.

1.3 Aim of the work

The aim of this thesis is to describe the implementation of different techniques for rendering and simulating large particle systems in real-time. They are compared with each other in terms of speed and required resources, especially memory. While for the rendering part a proper visual appearance is the main goal, the important conditions for the simulation are an efficient way to update a huge amount of particles and dealing with obstacle collisions in the scene.

1.4 Methodological approach

The methodological approach of this work consists of several tasks, where each can be assigned to either the simulation or the rendering step. First, the main parts of the rendering procedure were implemented. After having a solution for that, simple CPU-based simulation was applied. This basic simulation was then implemented on the GPU, too. While the CPU simulation had performance issues for very large particle systems at this point, the GPU simulations had much less troubles with the same particle system. In parallel the sequence of render tasks per frame and the used framebuffer setup were rearranged multiple times, until they were suitable for additive fire particles and effects like *Soft Particles* and *Bloom*. The result is summarized in subsection 3.2.1. The last tasks were to implement and integrate the GPU sorting algorithm and the collision detection and handling. All simulation tasks were implemented on the CPU first, then analyzed and implemented to be executed on GPUs. During the implementation, the used resources, mainly the memory requirements, were analyzed and compared per simulation approach. In order to compare the speed, the communication and computation costs were considered. Also the measured rendered frames per second contributed to the comparative result.

1.5 Contributions

The main contributions of this thesis are:

- Different approaches to simulate and render interactive particle systems in real-time are described along with concrete guidelines how they can be implemented.
- Various optimizations using different tools of modern GPUs for the simulation of big particle systems are presented.

- Comparisons of all introduced approaches in terms of execution time, used memory and needed communication amount are stated.

1.6 Structure of the work

In this section, a short overview of the next parts of this thesis is given. In chapter 2 the list of related state-of-the-art literature for each task is described. In chapter 3 the implemented system is explained in more detail and chapter 4 explains the results of the comparisons and generated pictures. In chapter 5 the thesis is reflected and possible additional ideas for improvement as future work are given.

CHAPTER 2

State-of-the-art

In this chapter some state-of-the-art methods concerning the main topics of this thesis are presented. Therefore some needed basic concepts like billboarding, particle systems and bloom are discussed. Furthermore, particle simulation and rendering techniques are described. Then some papers with focus on rendering fire and smoke are given. The last three sections handle GPU sorting techniques, GPU collision handling and Order-Independent Transparency.

2.1 Basic concepts

2.1.1 Billboarding

A billboard is a polygon, usually a quadrilateral, oriented in space relative to the view direction [AMHH08]. In combination with textures, complex geometry or phenomena can be approximated very easily and efficiently. The orientation of a billboard can be represented by a normal and an up vector and the final position in space can be determined by the polygon's center, for example. According to [AMHH08], there are different types of billboards:

- Screen-Aligned Billboard
- World-Oriented Billboard
  - Viewplane-Oriented Billboard
  - Viewpoint-Oriented Billboard
- Axial Billboard

A **Screen-Aligned Billboard** is always parallel to the screen - the normal vector is pointing in the negative direction of the viewplane's normal vector and the up vector is the same as the camera's. That means, that the billboard is independent from the camera's rotation along the view direction (e.g. z-axis).

A **World-Oriented Billboard**'s up vector is not bound to the camera, but usually aligned with the world's up vector. The polygon is called a **Viewplane-Oriented Billboard**, if the polygon's surface is parallel to the viewplane. One rotation matrix can be applied to all billboards. When the normal vector of the billboard is pointing from the center to the camera's position (viewpoint), it is called a **Viewpoint-Oriented Billboard**. In this case all billboards have a different rotation matrix. In Figure 2.1 these two methods are illustrated.

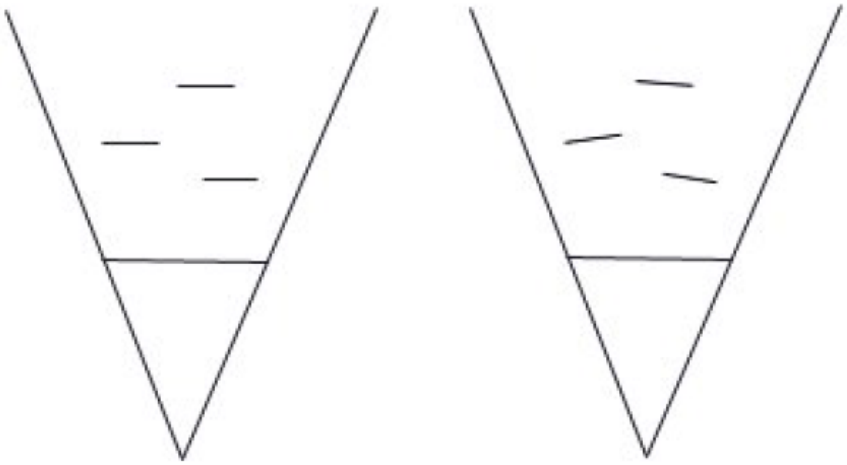


Figure 2.1: This figure shows the differences between viewplane- and viewpoint-oriented billboards. The left picture illustrates the viewplane-based approach, and on the right the viewpoint-based polygons can be seen. Adapted from [Lar10].

An **Axial Billboard**'s orientation is limited to the rotation along only one world space axis. The polygon aligns itself towards the camera as best as possible within this limitation. This sort of billboarding can be used for distant trees for example.

Billboards can be easily created on the fly from point data on the GPU using the geometry shader [Bas10].

2.1.2 Particle Systems

A particle system contains many different small objects, which are moved and animated in order to approximate some phenomena like fire, smoke or explosions [AMHH08]. Primarily, the purpose of a particle system is the simulation and not the rendering of particles. The main operations are creating, manipulating and deleting many of these

small objects in a dynamic manner. A single particle can be represented in many different ways like, for example, as point, line or billboard.

Often, a particle system needs emitters [Lar10]. An emitter is handling the creation of new particles and defines parameters like position, speed, distribution, direction, size and so on. Another important concept is the usage of external forces like wind or gravity in order to make the simulation more realistic.

Anderdahl et al. [AD14] add the terms of a demitter and the bounding volume to the definition of a particle system. A demitter or despawn is the opposite of an emitter or spawn and deletes a particle. A bounding volume can be seen as the border or the validity area of the system, if available.

2.1.3 Texture Animation

In order to make the appearence of a textured geometry more dynamic the texturing can be animated. This can be easily achieved by changing the texture to be sampled and/or the sampling coordinates from frame to frame [AMHH08]. Operations can be simple moving the coordinates around in the texture space or applying linear transformations (e.g. zoom, rotation, ...). Many effects like a waterfall and fire can be approximated with this approach.

When changing textures from frame to frame a texture atlas can be used. A texture atlas is a big texture that contains many smaller independent textures [Lar10]. They can be aligned in a grid or a line for example. For sampling, the original texture coordinates need to be converted to point at the appropriate subregion of the texture. Using a texture atlas decreases the number of texture switches and is therefore more efficient than binding all the small textures separately.

2.1.4 Image Processing and Filters

The images created when rendering geometry can be processed further before showing on the screen. Applying image processing methods on images after rendering is also known as post processing [AMHH08]. Such an image processing technique would be applying a gaussian filter in order to blur the image.

Another image processing technique to blur an image is to downsample an image. The downsampled image can then be blended over the original image [AMHH08].

The gaussian filter is a so called seperable filter kernel. Seperable means, that the filter can be applied in two seperate one-dimensional filter passes [AMHH08][PVV05]. This is much more efficient, because much less texel-fetches have to be done. As an example a 5 by 5 gaussian filter kernel would need 25 texel-fetches in one pass. Using the seperable gaussian filter would mean, that a 5 by 1 and a 1 by 5 filter need to be applied in two passes, leading to only 10 texel-fetches.

2.1.5 Bloom and Glow

Bloom is an effect, where very bright areas shine over neighboring pixels. To achieve such overexposed images the areas which should receive the effect need to be rendered into a seperate texture, which, subsequently, is blurred and added to the original image. These steps can easily be done with image processing [AMHH08]. The interesting areas can be determined by applying a bright-pass filter, where only pixels to be bloomed are not set to black. This makes the bloom effect react dynamicly on very bright areas of the scene.

Instead of a bright-pass filter, an additional texture, that identifies the interesting regions of geometry, can be used [Fer04a]. This makes the bloom effect more controlable over the scene, because only parts defined in the textures are overexposed. The composition of the final image can be seen in Figure 2.2.

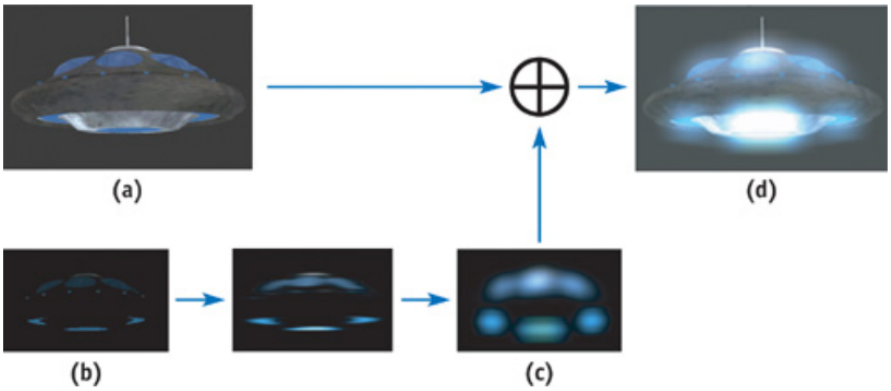


Figure 2.2: This figure shows the process to create the bloom/glow effect. (a) is the original scene image and (b) is the texture, that holds the parts which are to receive the effect. These parts were defined by an additional texture (not shown; often called Alpha Texture). (c) holds the blurred image from (b) and the final image (d) is composited as the sum of (a) and (c). Adapted from [Fer04a].

2.2 Particles

2.2.1 Soft Particles

A problem with billboard rendering of particles is, that the hard edges of the polygon can become visible when hitting scene geometry. Lorach [Lor07] introduces the idea of fading the fragments out in regions close to other geometry/other objects. An easy way to achieve this is to use the scene depth (e.g. z-buffer). Figure 2.3 shows, why using depth information helps to overcome this problem.

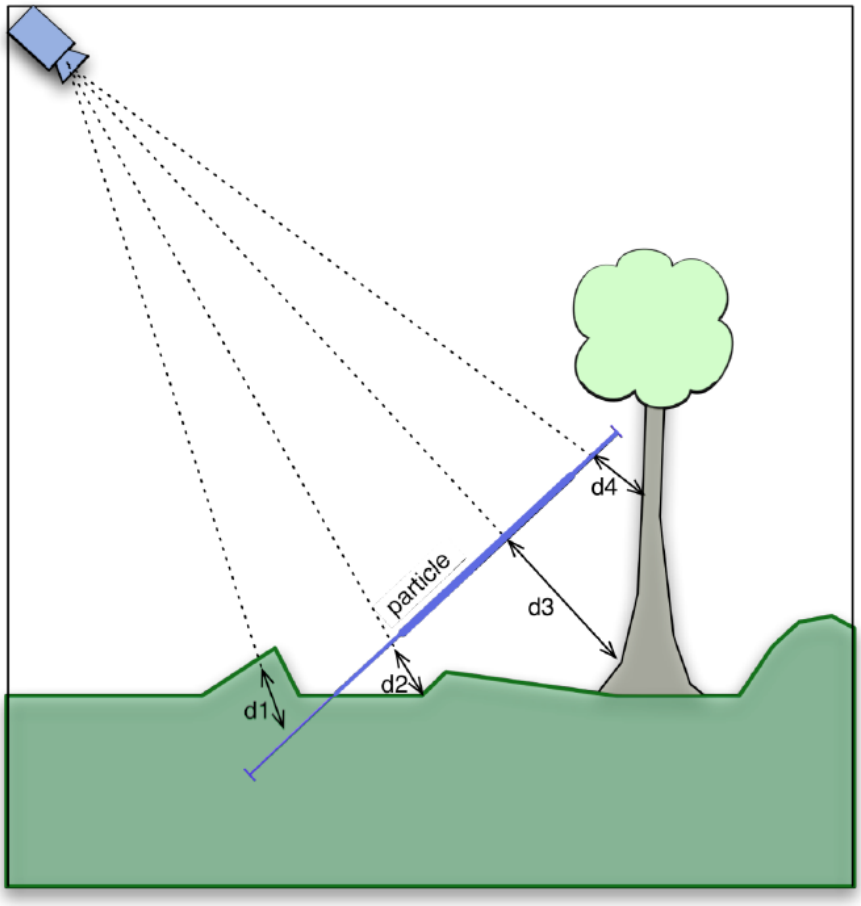


Figure 2.3: This figure demonstrates the idea of using depth information for removing hard edges. It can be seen, that d1 is behind the scene geometry/other objects and d2 is a little bit in front of the scene geometry/other objects. When rendering the particle the common way, a hard edge would be noticeable where other geometry is intersected by the bounded plane representing the particle. This can be solved by smoothly decreasing the alpha value of the particle in that regions. In general, that means the closer fragments are to the scene geometry, the more they have to be faded out. Adapted from [Lor07].

2.2.2 High-Speed, Off-Screen Particles

In [Ngu07] a method to improve the performance of particle rendering is described. The main problem is, that rendering alpha transparent particles needs a lot of blending operations and one way to lower the number is to render these into a smaller render target, leading to less fragments that need to be alpha blended. Therefore, all particles are rendered into an off-screen render target with lower resolution. To test the particles against the depth of the scene the z-buffer needs to be downsampled until it fits the resolution of the off-screen render target. However, using binary depth testing when

rendering the particles makes downsampling artifacts visible. Implementing *Soft Particles* from last section shrinks these artifacts, but there are still some noticeable blocky regions in the images. To overcome this, the authors introduce the *Mixed-Resolution Rendering* technique, that uses an edge image, the stencil buffer and a second particle render pass. Figure 2.4 shows a result.

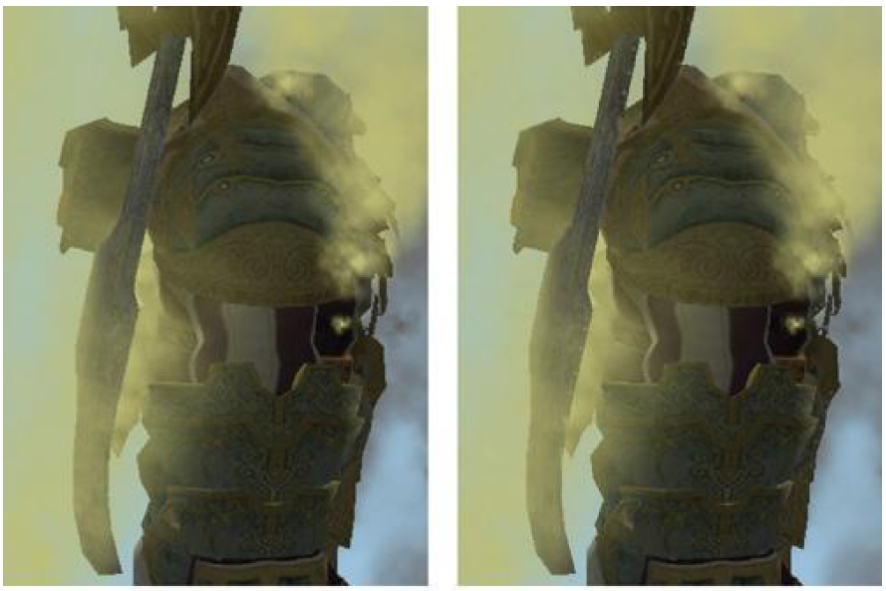


Figure 2.4: This figure shows the differences of particles rendered at full resolution (left image) and the *Mixed-Resolution Rendering* technique (right image). Adapted from [Ngu07].

2.2.3 GPU Simulation

McCool et al. [MDTP+04] describe a method for manipulating states of particles of a particle system using a stream program.

Another way to simulate particles on the GPU is storing particles as texels inside a texture. They can be updated by executing a pixel shader [KLRS04][Bas10].

Another approach for GPU-based particle simulation is to store particles as vertices inside a vertex buffer. An update step is done by executing a geometry shader, whose output is not passed to the pixel shader, but written into another vertex buffer using stream-out (DirectX's Stream-Out [JK18] is comparable to OpenGL's Transform Feedback [Wik18b]) [Bas10].

A geometry shader can output a dynamic amount of points or other geometries [Wik19b]. That is the reason why this shader stage can be used for the creation, manipulation and deletion of particles on the GPU [Bas10].

2.2.4 Comparison Between Particle Rendering Techniques in DirectX 11

Andersson and Johansson [JA17] are comparing one CPU-based solution using hardware instancing and one GPU-based approach using DirectX's Stream-Out technique. The results show, that the GPU version is much better than the CPU version, when talking about big non-interacting particle systems. On the other hand the CPU approach is better, when it comes to smaller particle systems with the need for collision detection or vector forces.

2.2.5 Particle Systems Using 3D Vector Fields with OpenGL Compute Shaders

Anderdahl et al. [AD14] are comparing two compute shader based particle simulation solutions using vector fields. A compute shader can perform operations on several data structures (e.g. buffers, textures) in parallel. They are a good tool/approach for particle simulation, because manipulating vertex buffer data entries is the only operation, that needs to be done for particle simulation. Compute shaders can perform these operations more efficiently than, e.g., Transform Feedback methods, because much less state changes need to be done (e.g. no vertex-, geometry-shader stage and no vertex array object needed) and only one buffer is needed. The authors test their solutions with different work group sizes for all compute shaders (one for creating vector fields and one for moving the particles) and visualize the results. The insight was a different execution time for different work group sizes. In addition the authors discover the assumption, that the computation time does not depend directly on the number of particles, but on the size of the data.

2.3 Fire and Smoke

2.3.1 Fire in the "Vulcan" Demo

In [Fer04b] the authors introduce a technique for rendering realistic fire and smoke coming out of a character. They use video-textured sprites (billboards with animated textures) for rendering. The authors use additive blending for fire particles, because the composition is independent from the order of the particles. A problem is, that the resulting colors get uncontrolable and the mixing with the smoke particles is very tricky. So they decide to first sort all particles (fire and smoke) each frame and then render them alpha blended over each other. The particles are rendered in a seperate render target, that has a lower resolution. This reduces the amount of needed blending operations and increases the performance. To improve the quality of the images a glow effect is implemented (see subsection 2.1.5[Fer04a]). Figure 2.5 gives an illustration of the rendering pipeline.

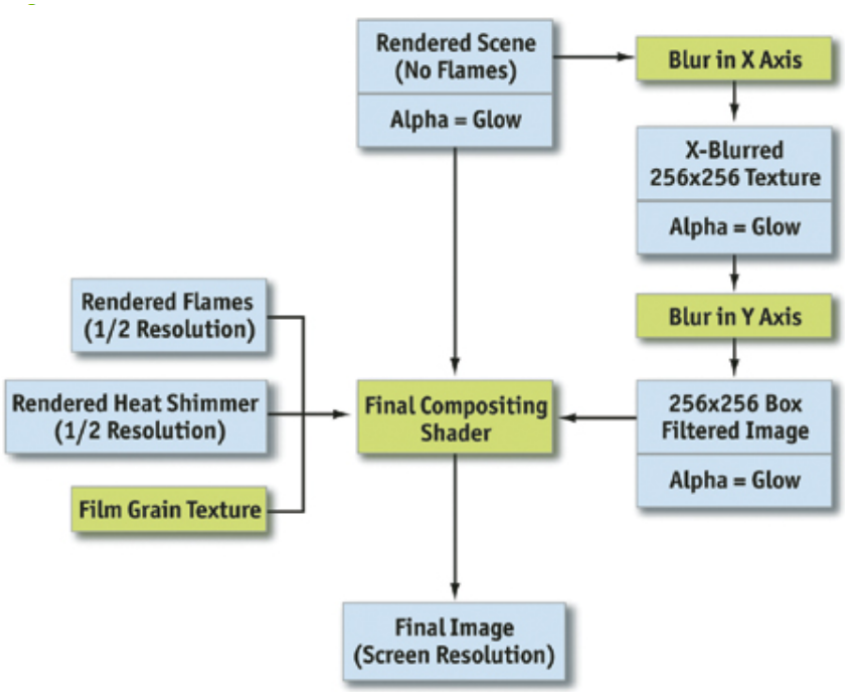


Figure 2.5: This figure shows the rendering process used in [Fer04b]. The glow stage can be seen on the right and the particles on the left. Adapted from [Fer04b].

2.3.2 Smoke rendering

When rendering smoke billboard particles additive blending cannot be used, because the undesired blending results are not typical for translucent materials. Therefore smoke particles need to be alpha blended [Bas10]. For the proper appearence of alpha blended geometry the fragments need to be composited in the right depth order, namely distant fragments first. To achieve this the smoke particles can be sorted before rendering. When simulating the particles on the GPU a GPU sorting algorithm need to be applied (see section 2.4).

Smoke can also be simulated with *Fluid Simulation* techniques, e.g. simulate smoke particles on the GPU, create a volume by rendering them into textures and render the smoke with volume ray casting/marching to the screen [Lar10].

2.4 GPU sorting

2.4.1 Odd-Even Merge Sort and Bitonic Merge Sort

Performing sort algorithms effieciently on GPUs needs algorithms for parallel architectures, for example the *Odd-Even Merge Sort* or the *Bitonic Merge Sort*. Both algorithms don't suffer or benefit from a special order or structure of the unsorted elements. Additionally

the worst-, best- and average-performance is in $O(log(n)^2)$ parallel time [con19a][Wik19d]. [PF05] provides pixel shader implementations for both.

The *Odd-Even Merge Sort* needs more passes to fully sort a list of elements. In general, a pass is a command (draw or dispatch) on the CPU side. Too many of these passes are bad for the performance. However, a strength of the Odd-Even network is, that the list is not getting less sorted after some passes. This leads to the mechanic of splitting the passes over frames. That means the list is not fully sorted after the first frames, but it gets better over time [Bas10][KLRS04].

The *Bitonic Merge Sort* needs fewer passes to fully sort a list of elements, but the sortedness does not increase after each pass. This is, because the groups are sorted alternating descending and ascending and therefore the algorithm cannot be splitted over frames [Wik19d][Bas10]. Figure 2.6 shows the sorting network.

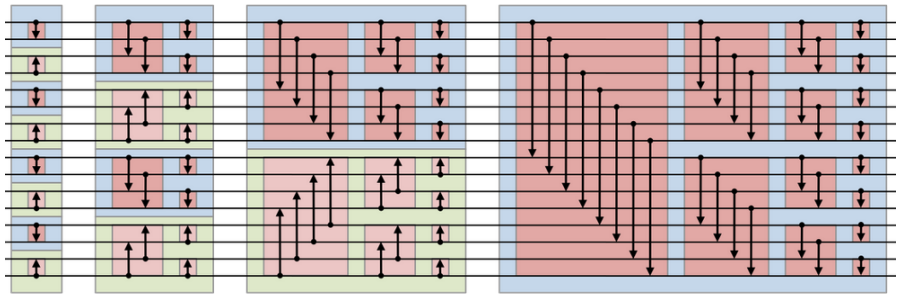


Figure 2.6: This figure shows the bitonic network for 16 elements. The black arrows define the comparisons and the colors (green, blue) are encoding the sort direction of this block in this pass. Adapted from [Wik19d].

An illustration of the *Bitonic Merge Sort* algorithm can be seen in Figure 2.7. The algorithm starts with $bunchSize = 2$. In this example a bunch is a group of elements and the list contains $n/bunchSize$ bunches. Within a bunch pairs of elements are ordered in the same direction. The order direction alternates over the bunches. First all direct neighbors are compared with each other, that means $jumpSize = 1$. $jumpSize$ defines the distance of one element to the element to be compared with within a bunch. The sort direction, that is depending on the current bunch, is stated with the gray dotted arrows in the figure. The elements, that need to be switched, are marked in red. In the figure, the list should be sorted in ascending order. When sorting needs to be descending, all sort directions need to be flipped. Then $bunchSize$ is doubled to 4. Now there are two invocations, the first, which compares each element with the second next neighbor ($jumpSize = 2$), and the second, which compares again each direct neighbors with each other ($jumpSize = 1$). Then $bunchSize$ is doubled to 8. That means, that there are three invocations, where the second and third have the same comparison schema as for $bunchSize = 4$. In the first invocation each element is compared with the neighbor, that is four fields further ($jumpSize = 4$). This results in 6 shader invocations for $n = 8$ elements.

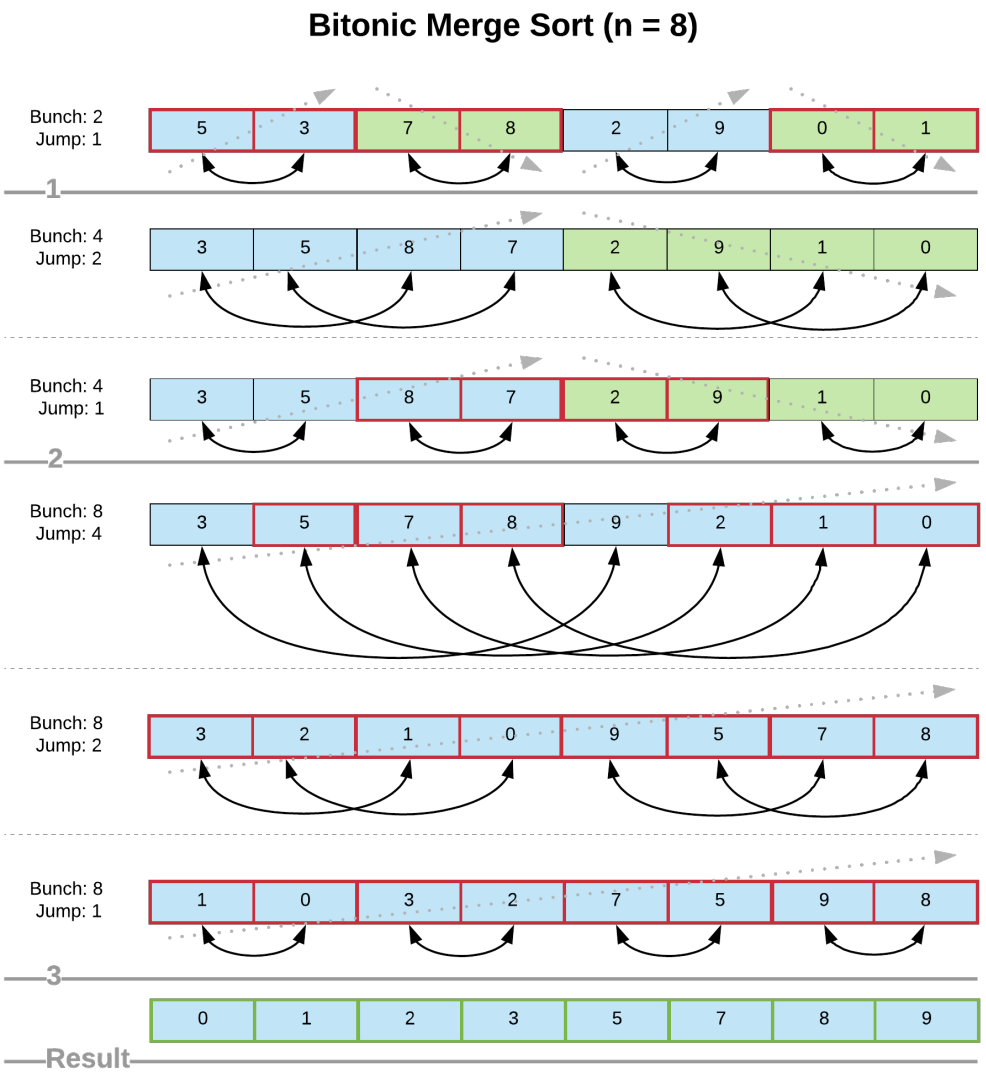


Figure 2.7: In this figure, an example for the bitonic merge sort algorithm in ascending order for $n = 8$ elements is given, where Bunch is *bunchSize* and Jump is *jumpSize*.

2.5 Collision detection and handling

2.5.1 Hardware-based simulation and collision detection for large particle systems

Kolb et al. [KLRS04] introduce a GPU-based simulation for big particle systems with detecting collisions on surrounding objects. The collision detection method is image-based,

that means for each collider several depth maps are created. The depth maps approximate the border of the object and each depth map contains the distance to the original collider object per pixel, the corresponding normal vector and the transformation matrix from collider object space to depth map space. During the particle system simulation procedure the depth maps are accessed in the shader and the collision detection can be computed (e.g. set velocity to zero or simple reflect on normal vector). In Figure 2.8 some results for colliding particles are shown.

Figure 2.8: This figure shows some results for big particle systems colliding with surrounding objects. Adapted from [KLRS04].

2.6 Order-Independent Transparency

An *Order-Independent Transparency* method's property is to automatically accumulate all transparent geometry parts in the right order independent from the rendering order. Such algorithms operate at fragment level, insuring correct composition over all fragments. During rendering, every fragment need to be stored somehow, because all pixels are contributing to the result. A simple approach would be to store all fragments in a buffer and sort each of them afterwards. The problem with that is a very tricky efficient implementation on the GPU. Another solution would be to store a fixed-size amount of values per pixel, but this is also very unefficient, because the fragment count can differ for all pixels. Maule et al. [MCTB12] introduce a memory-efficient technique for order-independent rendering of transparent geometries, using a method called *Dynamic Fragment Buffer*.

CHAPTER 3

Methodology

3.1 Used technologies

This section gives a short overview of the used technologies. The programming language C++ and the graphics library OpenGL [Rod18][Wik18a] were used for creating this application and resulting images. Because compute shaders were used in important parts of this work, OpenGL 4.3 is a minimum requirement [Wik19a].

3.2 Rendering

In this chapter, the rendering of billboard-based particles is described in detail. First, an overview of the sequence of render tasks and used framebuffer setup is presented. Subsequently, the layout of the buffer that should be drawn is introduced. In the next section an overview of the used *Billboarding* technique and shader programs is given. Then the progress, that makes an animated billboard out of a texture atlas is shown. Next, the bloom and the blurring processes are illustrated and at the end the *Soft Particles* effect is described.

3.2.1 Overview

In this section, the main tasks of a single render frame are described. A simplified depiction of the whole process can be seen in Figure 3.1. Rendering fire and smoke particles to the same scene is not trivial and therefore several textures and render targets are used.

During the *Render scene* task, the entities, a skybox and the floor are rendered. In parallel, an additional texture is filled with the corresponding scene depth values, because the *Soft Particles* effect needs information about the scene depth and the current depth-stencil render target cannot be used for reading, when currently assigned to a framebuffer.

During *Render smoke* alpha transparent smoke particles are rendered onto the scene. This results in very dark and dense smoke. Particles are not rendering their depth to the current depth-stencil render target, but their fragments are tested against the depth values from the scene.

After that, the fire particles are rendered during the *Render fire* step. The fragments are additively blended over each other, because this increases the intensities and lets the fire glow a little bit. The render target is cleared to fully transparent black at the beginning of each frame. This is done, because for additive blending the intensities are summed up per fragment. When rendering fire onto the scene like it is done for smoke, all the intensities would be almost one, because the intensities of the underlying scene fragments would contribute to the result.

The bloom is part of the general particle render process. In order to determine, which fragments need to be bloomed afterwards, the colors to be bloomed are rendered into the *Bloom Texture* as second render target during the *Render smoke* and *Render fire* stages. In case of smoke the bloom texture fragments are black, because the bloom effect would not be noticeable at all. In general, bloom increases the glow even more in the case of fire. Therefore, the fragment color with different weights per color channel is rendered. The weight of the red channel is the most dominant, because this lets the fire glow more in red, what makes the fire more realistic.

During *Blur textures* the *Bloom Texture* is blurred, because in general this is part of the bloom effect. The whole bloom procedure is described in detail in subsection 3.2.5.

In order to merge all created textures to the final image, first the scene and fire textures are alpha blended over each other. This is done, because it preserves the intensity and transparent parts of the fire particles. The next step is to add the blurred bloom texture fragment-wise, because this is the last step of the bloom effect. The result is drawn to the screen.

3.2.2 Layout of buffers

A very simple way to render a particle would be to store a quad inside a GPU buffer and prepare a transformation matrix for each particle. Then iterate over all particles, upload the current transformation matrix to the shader and render the quad. There are several performance issues with that approach. A 4 by 4 transformation matrix has to be stored and uploaded for each particle per frame, where only a 3-component vector for the position and some single float values for rotation and scaling are needed in the simple case. Besides that, a single draw call is executed per particle, what is also a big waste of resources. A smarter version would be to store only necessary data into a buffer, create the billboard on the GPU using the geometry shader and execute only one draw call for all particles. With that approach, the buffer is a list of particle data, where for each particle a 3-component vector for the position and a 2-component vector for rotation and scaling are stored. A single value for rotation is enough, because a billboard can only be rotated along the local z-axis. Otherwise it would not look towards the camera

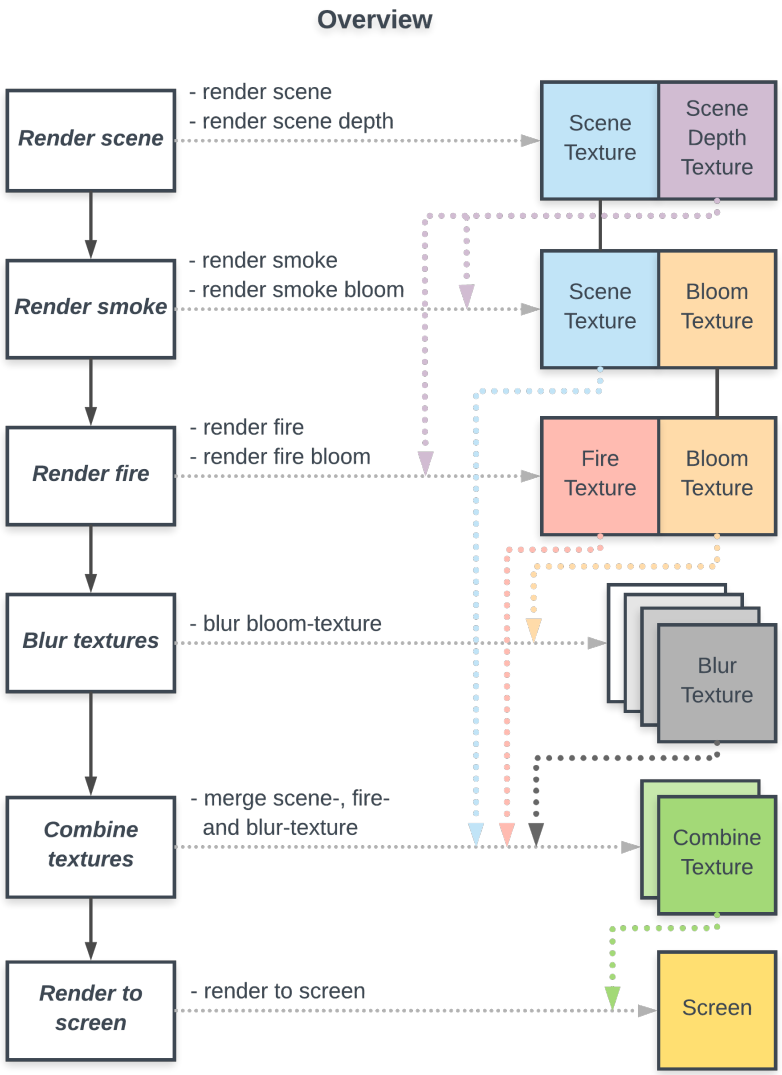


Figure 3.1: This figure shows all the steps of our algorithm. At the left, the separate compute/render steps are depicted. On the right, the required resources are shown which are read by the steps on the left or written into, respectively.

anymore. For this thesis, an extended version of the described buffer was used for the GPU simulation techniques in section 3.3. For each particle, the following data is stored:

1. 4-component vector for position; w value is used for distance to camera

2. 2-component vector for elapsed time and maximal life time

3. 3-component vector for rotation and scaling; 2 values for scaling, where the actual scaling is the interpolation of these two values according to the elapsed time

4. 3-component vector for current velocity of the particle

Some of the values like the two life time values and the velocity are only used for simulation, but the same buffer is used for rendering and simulation.

3.2.3 Billboarding and rendering pipeline

Consider having a buffer with a list of particle data stored as discussed in the previous section. Knowing the amount of stored particles we can draw as much points as particles after binding the proper shader program. First the vertex shader is executed. This shader passes all components to the next stage. Then the second stage the geometry shader is executed. The main purpose is to create the billboard and set all preparations for the rasterizer. In order to create the billboard the passed world position is transformed into view space using the view matrix. Up to now we have only one single point in view space instead of a quad in clip space. A view space quad can now easily be created by adding the four corner points of a quad to the view space position. The quad can be scaled by using the size for the computation of the relative corner points. In order to apply also the rotation to this quad a z-rotation matrix is built inside the geometry shader and each corner point is transformed accordingly, before adding it to the view space position. After that the projection matrix is applied to the quad and the primitive is emitted. The computation of the left top corner point in clip space can be seen in Equation 3.1,

$$\begin{aligned} pos_{vs} &= m_{view} * pos_{ws} \\ lt_{vs} &= (m_{model} * vec4(-s, s, 0, 0)) + pos_{vs} \\ lt_{cs} &= m_{proj} * lt_{vs} \end{aligned} \tag{3.1}$$

where pos_{ws} is the input position in world space, m_{view} the view matrix and pos_{vs} the transformed position in view space. In the second line m_{model} describes the z-rotation matrix and s is the size of the particle. In the last line m_{proj} is the projection matrix and lt_{cs} holds the left top corner point in clip space. In order to get the other three corner points of the quad, only the sign of each s inside the vector has to be modified. As output mode of the geometry shader a triangle strip is used. It is not hard to see that the generated quad is always facing the camera, because it creates the quad in front of the camera in view space. But as simple as this approach is, it brings some performance issues. The quad and the model matrix for the z-rotation have to be generated new for each particle in each frame. In the next section the fragment shader and the needed preparations are described.

3.2.4 Animated textures

Having the properly generated billboard, each fragment needs to be illuminated. Because of the dynamic appearance of fire and smoke, it can be beneficial to not use static textures but animate them. Such an animated texturing can be achieved by using a texture atlas (see Figure 3.2). This atlas is built up like a grid. While the particle is alive, that means the elapsed time is lower than the max life time, the life progress can be computed.

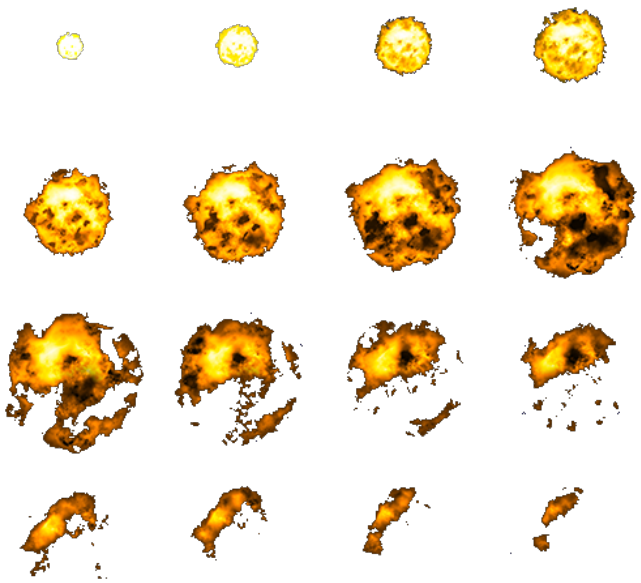


Figure 3.2: This figure shows an example of a fire texture atlas, arranged in a 4 by 4 regular grid. This atlas was used to animate the fire particles. In general the layout of the atlas doesn't matter, the subtextures could also be aligned in one row or column. It can be easily seen, that this atlas delivers many changes regarding size and volume, what makes the appearance of the fire more dynamic.

According to this value the current independent texture of the atlas can be determined. The next independent texture in the grid can be used to smoothly interpolate between two subimages. Therefore, in the geometry shader the texture positions for the current and the next atlas cell can be assigned to each corner point and passed to the fragment shader. The next values, that can be computed according the life progress value, are the fade-in and fade-out alpha values. The smoke particles, for example, are very transparent at the beginning and smoothly get more visible over time. For a nice optical effect, we suggest to fade out the fire particles in order to see the visible smoke at the end of the beam. This means also these computed alpha multipliers are passed to the fragment shader. Finally the fragment shader is executed. Thanks to the pre-calculated values in the geometry shader, it only has to sample the texture atlas and set all output values

accordingly. The color computation of the particle fragment can be seen in Equation 3.2,

$$\begin{aligned} c_1 &= sample(t_{atlas}, uv_1) \\ c_2 &= sample(t_{atlas}, uv_2) \\ c_{out} &= t_{fade} * ((1 - a_p) * c_1 + a_p * c_2) \\ c_{bloom} &= b_{weights} * c_{out} \end{aligned} \tag{3.2}$$

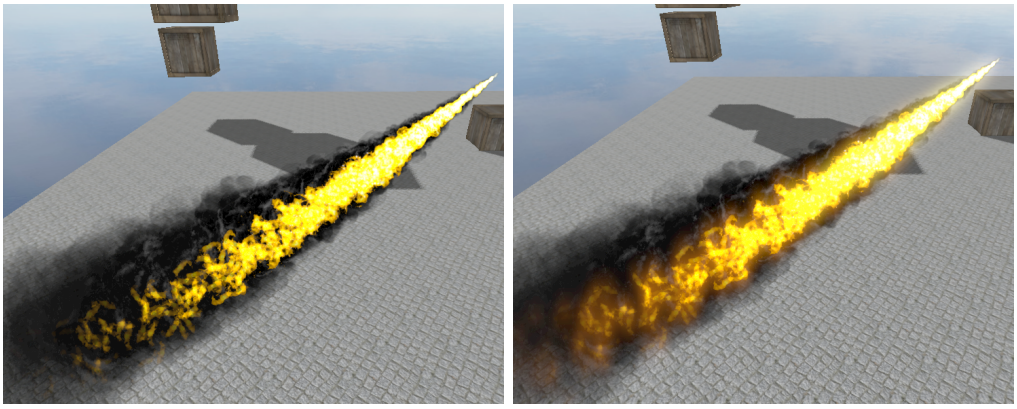
where uv_i are the two texture atlas coordinates and c_i hold the sampled atlas colors, for $i \in [1, 2]$. t_{atlas} is the bound texture atlas, t_{fade} the alpha multiplier and $a_p \in [0, 1]$ the interpolation factor between the adjacent atlas cells. The fragment color is stored in c_{out}. c_{bloom} holds the color rendered to the bloom render target and is set to the fragment color weighted by $b_{weights}$.

3.2.5 Bloom

Up to now particles are stored in one buffer and can be efficiently rendered using animated billboards. This part gives a detailed description, how the appearance of the fire particles can be improved using a simple bloom effect (see subsection 2.1.5, [AMHH08], [Fer04a]). In general, this effect consists of three parts, namely rendering into a separate texture, blurring it and adding the result to the scene. First, the parts of the scene, that should be bloomed, have to be rendered into one texture. This happens during the fragment shader of the particle rendering shader program, where the input for this texture is a weighted color of the fragment color itself (last line of Equation 3.2). In the case of smoke these weights can be zero, because blooming the low intensity fragment colors of the used atlas wouldn't be noticeable at all. On the other hand blooming increases the quality of the visualizations of fire, because this approximates the glowing parts of real fire. The differences of disabling and enabling bloom can be seen in Figure 3.3. After rendering into the texture the data has to be blurred in a separate stage. We chose gaussian kernels of size 11 with sigma 2.0 for blurring, but it depends on the scene and the required distortion.

The blur can be reinforced by using smaller textures as render targets. Due to the fact, that a gaussian kernel is separable (discussed in subsection 2.1.4[AMHH08][PVV05]) and the bloom texture is blurred twice in this project, the whole procedure consists of four steps:

1. horizontal blur with third of screen dimensions
2. vertical blur with third of screen dimensions
3. horizontal blur with sixth of screen dimensions
4. vertical blur with sixth of screen dimensions



(a) Scene without bloom. (b) Bloomed fire and smoke.

Figure 3.3: This figure shows the visual differences of disabled and enabled bloom. It can be easily seen, that the bloom increases the quality of the fire.

The blur is applied twice, because it reinforces the blurring even more. This approach is more efficient than using an accordingly bigger kernel, because it allows using much smaller kernels applied on much less fragments, due to the shrunken textures, to get the same result. When only using the second blur artifacts would be noticeable. To finalize the bloom effect the blurred texture is added fragment per fragment to the combined scene and fire texture.

3.2.6 Soft Particles

Up to now interactive images can be created using the techniques mentioned in the previous sections. But there is an issue with the used billboarding rendering technique. The quad can become visible, when parts of a particle overlap with other scene objects and this might destroy the visual appearance of fire and smoke. This section shows a method to counteract this visual issue, called *Soft Particles* [Lor07]. An introduction to *Soft Particles* has been given in subsection 2.2.1. This method smooths out these hard edges in overlapping regions using the depth buffer. The original depth buffer, that is used for depth testing and writing, cannot be used efficiently for that, because in order to use this depth texture it must not be bound as the current depth-stencil render target. Instead of unbounding it from that target before rendering and bounding it again to that target afterwards, it is easier to create another independent texture for that. This texture stores 32 bit float values and is bound as the second render target, when rendering scene geometry. During the fragment shader the generated depth texture can be used, in order to decrease the overall alpha value for fragments, that are very close to the scene. The computation of this alpha value can be seen in Equation 3.3,

$$a_{soft} = abs(d_{scene} - d_{particle}) * s_{soft} \tag{3.3}$$

where d_{scene} is the obtained scene depth value, $d_{particle}$ the length of the particle position in view space, s_{soft} a scaling value and a_{soft} the resulting alpha value, that is multiplied to the alpha channel of the fragment color. The value of s_{soft} is very important for the result, but it totally depends on the scene. In this project s_{soft} is set to 2. Illustrations of the effect of *Soft Particles* can be seen in Figure 3.4.

(a) Hard edges of quad can be seen. (b) Scene with *Soft Particles*.

Figure 3.4: This figure shows the visual differences of *Soft Particles* disabled and enabled. It can be easily seen, that hard edges of the billboard quad can become visible without this effect.

3.3 Simulation

In the previous chapter several techniques for rendering particles have been presented. The goal for this chapter is to give an introduction into the implemented simulation types, that are more or less suitable for very big particle systems. The comparison of the results is presented in chapter 4. First, the CPU-based technique will be described. Second, the first GPU-based technique will be described, which is using transform feedback. An improved version of this technique is described as the third variant. It makes use of compute shaders. The next section deals with sorting the GPU particle buffers. In section 3.5 the collision detection and handling of particles and scene obstacles are described.

3.3.1 Introduction

During the simulation of particles the physical behavior of the underlying phenomenon is approximated. This can be achieved by manipulation of the particle data during the update procedure of the engine. First the components of a particle, which are needed for simulation, are discussed. It is obvious that a simple particle needs a position. Due to the fact that a particle can move around, an easy way to model this would be to add a velocity vector to the data, that keeps track of the current moving direction and

speed. Also active particles may die or a new particle can get created. That means each particle has to keep track how long it has been living, in order to determine, when to end being active. So we added the elapsed life time and the max life time. After defining the components of the particles the simulation process can be discussed.

3.3.2 CPU-based simulation

Consider having particle objects stored in an array or a dynamic list data structure. The first thing during the update method is to determine, how many new particles have to be created in this frame. This amount of new particles is added to the data structure. Then iterate over the data structure and do a simulation step for each active particle. This simulation step is performing the following steps:

1. Increase elapsed life time
2. Change position according velocity
3. Change velocity regarding global forces, like gravity or wind

Particles, where the elapsed life time is greater than the max life time, need to be removed from the data structure, because these particles are no longer active. The advantage of having the particle data stored on CPU side is the practicality, because all methods that can be applied to this data structure or the particle object itself can be executed easily. The big disadvantage is the fact, that some components of each particle, especially the position, needs to be uploaded to the GPU shader in order to render the particle. This is not suitable for large particle systems, because the amount of data, that needs to be transferred would worsen the performance very much. This leads to the next section.

3.3.3 GPU-based simulation - Transform feedback

In this section a technique for particle simulation is described, that is able to read, process and then write back a huge amount of particle data stored in a GPU buffer. Consider having a GPU buffer and a particle list on the CPU. The list can easily be uploaded to that GPU buffer and rendered. Another important detail is, that in our implementation the GPU buffers have one additional particle stored, namely the emitter. The emitter cannot be updated or rendered and its task is to emit new particles during its update step.

The next lines will explain the functionality of transform feedback, that is the core part of this approach. With transform feedback the result of a vertex or geometry shader can be written into another buffer [Wik18b]. That means a second buffer is needed as target of the written geometry. To use transform feedback for particle simulation the vertex shader is invoked with that input particle data buffer and the output of the geometry shader is written into another buffer. In the next frame, this buffer will be the input and the other buffer the target. So the buffers are *Ping-Ponging* every frame. In our

implementation it is important to disable the rasterizer, because we don't want to render anything in this case, but just perform the simulation.

The vertex shader is just passing the input components further to the geometry shader [Wik19c]. The procedure of the geometry shader can be seen in algorithm 3.1. This algorithm is executed for each particle in parallel. In line 2, the current particle is obtained and in line 3, it is checked if the current particle is the emitter. In this thesis, an emitter is a particle that never dies and is responsible for spawning new particles. Therefore the emitter is written into the output buffer without changes, because all buffers need exactly one emitter. The function $WriteParticle$ executes the output of a particle to the output buffer. In lines 5 to 8, new particles are created using the $CreateNewParticle$ function and written into the output buffer. If the current particle is not the emitter, then it is updated in line 10. $GetUpdatedParticle$ takes the current particle, manipulates and returns it. If it is not active anymore (determined by $IsAlive$), then it won't be written to the output buffer. Otherwise the collision handling is done in line 12 and in line 13 the collision handled particle is written to the output buffer. $GetCollisionHandledParticle$ takes the updated particle, checks it against all colliders, manipulates it if necessary and returns it.

Algorithm 3.1: Update particles using transform feedback

```
1  function GeometryShaderSimulationMain(dt, n_spawn, params_spawn, g,
     params_colliders)
       input : dt ∈ ℝ, dt > 0, dt is elapsed time since last frame
               n_spawn ∈ ℕ, n_spawn is amount of new particles to spawn
               params_spawn is list of spawn parameters
               g ∈ ℝ^3, g is gravity
               params_colliders is list of colliders
2      p_current = GetCurrentParticle();
3      if IsEmitter(p_current) then
4          WriteParticle(p_current);
5          for i = 1..n_spawn do
6              p_new = CreateNewParticle(params_spawn);
7              WriteParticle(p_new);
8          end
9      else
10         p_updated = GetUpdatedParticle(p_current, dt, g);
11         if IsAlive(p_updated) then
12             p_collisionhandled =
                GetCollisionHandledParticle(p_updated, params_colliders);
13             WriteParticle(p_collisionhandled);
```

In order to keep track of the current amount of active particles the number of written particles needs to be transferred from the GPU to the CPU. This is necessary, because the

CPU, which is executing the shader program, needs to specify the number of primitives that should be updated in the next frame. To achieve this an OpenGL query object, that listens to the generated primitives of the bound transform feedback target buffer, is used [Wik19c].

The goal was to find a method, that can efficiently simulate a huge amount of particles, using the parallelization strength of the GPU. With the transform feedback approach this goal is reached, but it brings also some disadvantages. One downside is the needed memory, because two buffers, one as input and one as output, have to be used. That means, that the particle data is stored twice on the GPU and that is a huge amount of additional needed storage in case of very large particle systems. Another disadvantage is, that the practicality of the CPU-based approach is gone. All functions, that can be applied on the data structure, like sorting, have to be implemented new, in order to be suitable for GPU buffers. A sorting algorithm will be described in section 3.4.

3.3.4 GPU-based simulation - Compute Shader

In this section a second GPU-based approach is described, which uses compute shaders. A simple way to use a compute shader for particle simulation is to implement it exactly like the transform feedback approach from the previous section.

In general, a compute shader can manipulate images and buffers, that are bound to the program [Wik19a]. There is no need for an additional target image or buffer, because it can read and write from and to the same image or buffer in one execution step. When executing the shader the amount of instances, that should run in parallel, has to be stated. The index of the current executed instance can be queried inside the shader code.

In order to implement the transform feedback algorithm using a compute shader, the input particle buffer, the target buffer and an additional counter buffer are bound. Instead of getting the particle count using a query object, the actual amount has to be computed in the shader code. This means every particle, that is written into the target buffer, increases the value in the counter buffer by 1. The value stored inside this buffer is the amount of instances for the execution in the next frame. It is obvious, that no matter which GPU technique is used for this algorithm, the needed memory is identical. An improved compute shader version would be an algorithm, that only needs one bound particle buffer instead of one input and one target buffer.

In algorithm 3.2 the improved algorithm can be seen. In line 2 the instance index is queried and in line 3 it is checked, if the particle at this index is the emitter. If yes, nothing needs to be done, because the emitter particle should stay unaltered at the same position in the buffer. If it is not the emitter, in line 5 it is checked, if the current index holds a valid particle. An invalid particle data would be an empty or died particle and is determined by the $IsValid$ function. If the data is a valid particle, then it is updated in line 6. When it is still alive after updating the collision handling is executed in line 9 and the particle is written back to the same position in the buffer. If the buffer holds an invalid particle at this position, this index in the buffer could be used for a new particle,

Algorithm 3.2: Update particles using compute shader

```
1  function ComputeShaderSimulationMain(dt, n_spawn, params_spawn, g,
     params_colliders, p_arr)
       input : dt ∈ ℝ, dt > 0, dt is elapsed time since last frame
               n_spawn ∈ ℕ, n_spawn is amount of new particles to spawn
               params_spawn is list of spawn parameters
               g ∈ ℝ³, g is gravity
               params_colliders is list of colliders
               p_arr is the buffer of particle data
2      i = GetCurrentIndex();
3      if !IsEmitter(p_arr[i]) then
4          alive = false;
5          if IsValid(p_arr[i]) then
6              p_updated = GetUpdatedParticle(p_arr[i], dt, g);
7              if IsAlive(p_updated) then
8                  alive = true;
9                  p_arr[i] = GetCollisionHandledParticle(p_updated, params_colliders);
10         if !alive then
11             n_spawn = n_spawn − 1;
12             if n_spawn > 0 then
13                 p_arr[i] = CreateNewParticle(params_spawn);
14             else
15                 p_arr[i] = CreateEmptyParticle();
```

that would need to be created and stored. This is the case, when the particle died or the data has not been valid at all. In line 11 the global value in the bound counter buffer is decreased. In this case the counter buffer doesn't hold the count of particles, but the amount of particles to spawn. When no new particles need to be created, the particle data at this position is set to empty in line 15 using the *CreateEmptyParticle*.

The advantage of this approach is, that it doesn't need to store all particles in two buffers, because the compute shader will only manipulate one buffer. A condition for this to work properly is to execute the program with the maximal count of particles, instead of the current particle count. This is, because the compute shader will place the new particles at random positions all over the buffer. On the other hand the particles during the transform feedback method are arranged on the left without holes. This is also illustrated in Figure 3.5. Note, that the indices buffer need to be as big, as the next power of 2 of the maximal particle count, because this is required by our sorting algorithm.

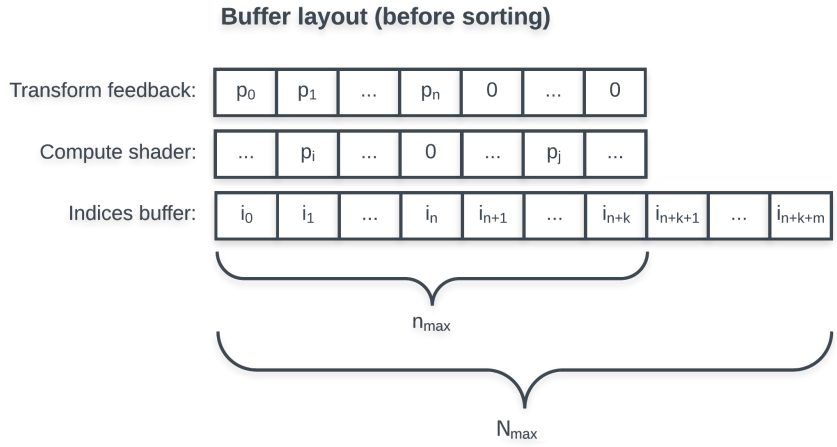


Figure 3.5: In this figure, an illustration of the layout of different GPU particle buffers can be seen. n is the current particle count and $n + k = n_{max}$ the maximum amount of particles that can be stored in this buffer. $n + k + m = N_{max}$ is the next power of 2 of n_{max}. It can be seen that transform feedback arranges the particles from left to right. The compute shader approach accesses the buffer at random positions and therefore the particles are stored at random indices in the buffer.

3.4 Sorting

Up to now particles can be rendered and simulated efficiently, but there are some visual issues with smoke particles. Those are rendered with alpha blending enabled and this requires the smoke fragments to be rendered from back to front, otherwise, it could give a different, incorrect result every frame. There are several techniques to do this, but in our implementation we are sorting descending the smoke particles regarding their distance to the camera. In case of the CPU approach, this can be easily achieved by applying standard sort algorithms, like *Quick Sort* [con19b]. When having one of the presented GPU approaches, a parallel sort algorithm suitable for GPU buffers has to be implemented. Therefore, we developed the *Bitonic Merge Sort* (see section 2.4), that sorts the indices in the elements buffers.

In order to implement this algorithm on the GPU using a compute shader, the tasks need to be split into CPU and GPU part. The comparison operations can be executed in parallel. In Figure 2.7 they are illustrated with the black arrows which, in each case, refer to the elements to be compared with each other. Therefore, each row of the figure is one shader execution.

One disadvantage of this algorithm is that it can only be applied to particle buffers, whose size is a power of two. To extend the algorithm to be able to sort an arbitrary

amount of particles, the indices buffer has to be extended to the next power of two number and inside the compute shader some sort values need to be treated separately. The sort values of particles, that are beyond the maximal data buffer position, are set to a very low value, in order to sort these indices to the end of the buffer.

3.5 Collision detection and handling

Another step in our particle simulation is the handling of collisions with other objects. Particles should not penetrate the scene entities. In general they should rather be reflected than only avoiding penetration. In our implementation, for simplicity, we decided to use scene obstacles with global axis-aligned bounding boxes of different sizes. Therefore a particle can be seen as inside a cube, if the particle position is between all faces, which are global axes, of the cube.

Due to the fact, that there are not that many scene obstacles, they can be stored and updated on the CPU. That means, that the information of the colliders needs to be uploaded to the GPU every frame. The collision detection is executed after the manipulation of position and velocity and only if the particle is still alive. Each particle is checked against all bounding boxes. When the position is inside a box, that means the particle is really penetrating the entity, the following things are done:

1. Change velocity according to the reflected and a random direction
2. Reset position to the exact position of impact
3. Decrease speed

The exact position of the impact can easily be determined, as it is the position clamped to the nearest face of the cube. This has to be done, because otherwise some particles would get stuck inside the cube. When collision has been detected the speed, encoded in the velocity, is decreased and a random direction is computed, that is pointing somewhere out of the hemisphere of the reflecting direction. The changed velocity is a linear combination of the reflected and the random direction with the length of the decreased speed. The reduction of speed approximates the loss of kinetic energy after collision and the usage of the random direction should simulate displacement of collisions between particles themselves during collision.

CHAPTER 4

Results

In this chapter the results are given and discussed. Also the implemented simulation methods are analyzed and compared with each other.

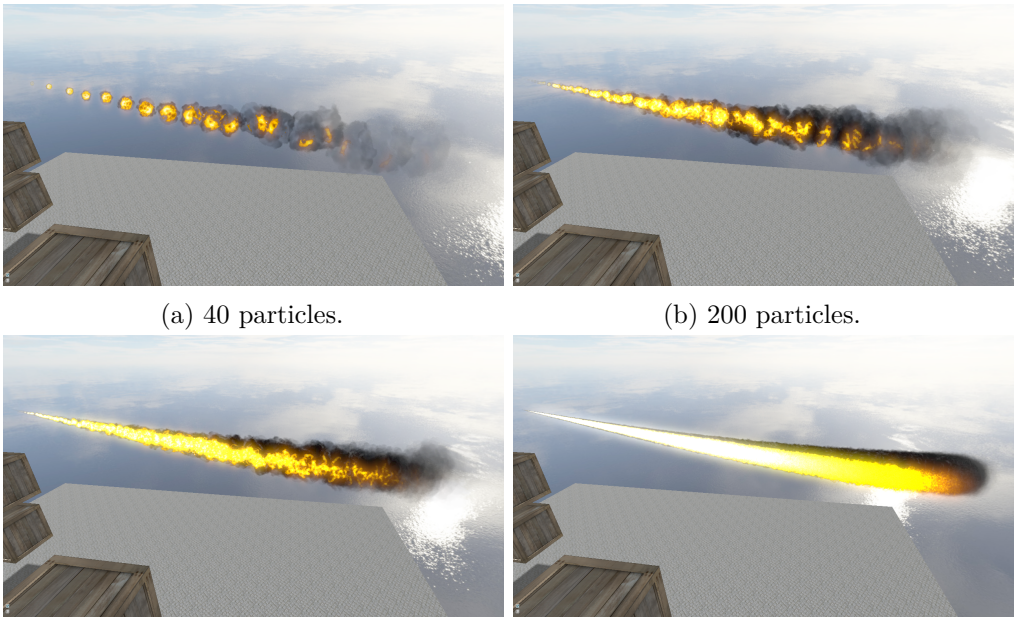
4.1 Visualizations

In order to determine which simulation approach to use for a specific task, 4 different sized fire and smoke systems were tested. The amount of particles that should be spawned per second n_{pps} and the maximal life time t_{max} in seconds determine the maximal number of particles n_{max}. Fire and smoke particles are processed in separate particle systems, but of the same size. Table 4.1 shows the different configurations.

Table 4.1: Sizes of particle systems

$\mathbf{n_{pps}}$ **fire**	$\mathbf{t_{max}}$ **fire**	$\mathbf{n_{max}}$ **fire**	$\mathbf{n_{max}}$ **fire & smoke**
5	4	20	40
25	4	100	200
200	4	800	1600
20000	4	80000	160000

Figure 4.1 shows images of fire beams rendered with particle systems of different sizes. It can be seen, that the fire beam with maximal 1600 particles (Figure 4.1c) simultaneously approximates a real flame thrower better than the systems with maximal 40 particles (Figure 4.1a). This is the case, because the small system has holes. Therefore the system fails to pretend to be one big dynamic structure. In addition the glow is weak, because too little fragments are blended additively over each other. On the other hand the intensities for the systems with 160000 particles (Figure 4.1d) are very high during the



(a) 40 particles. (b) 200 particles.

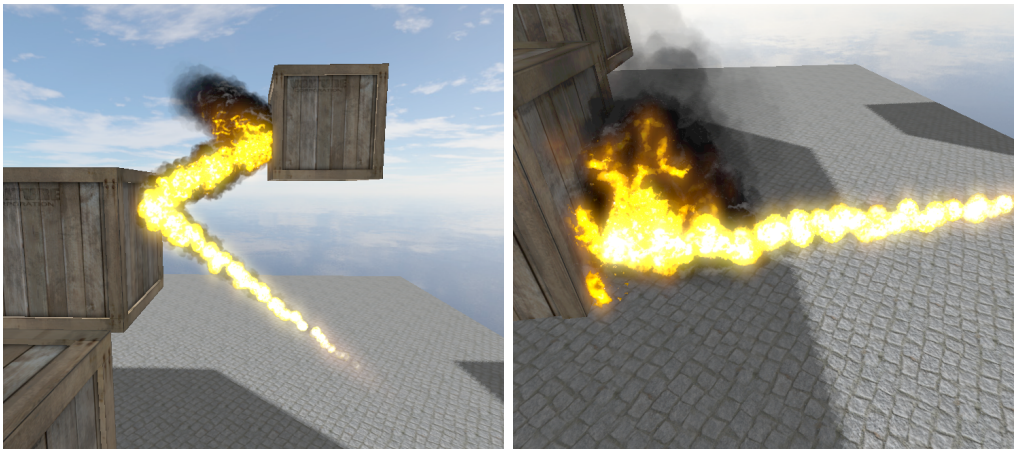
(c) 1600 particles. (d) 160000 particles.

Figure 4.1: This figure shows 4 particle systems with different amount of particles.

first half of the lifetime, that means almost white fire fragments. In addition, the overall shape appears very flatly. One reason for this is, that the high amount of particles is approximating the shape, defined by the simple velocity vector, without holes. Another reason is the very small size of the particles, leading to the loss of dynamism of the used texture atlas. Therefore very small and very big systems are not suitable for fire particles, at least not in the way, how we have implemented it.

For the collision handling, two parameters are modifiable in order to interactively determine a combination that delivers the most realistic collision behavior. These parameters are the randomness $c_{rand} \in [0, 1]$ and the slowdown $c_{slow} \in [0, 1]$. c_{rand} gives the interpolation value, that is used for the linear combination of the reflected direction. c_{slow} is a multiplicative number, where a value of 0.5 would reduce the speed to 50%. In Figure 4.2 results for different settings of these parameters are given.

As mentioned in the previous chapters, the fire particles are rendered after the smoke particles. This causes the fire to shine through the smoke every time. Rendering the smoke after the fire would have the effect that the smoke would be visible from every angle. But also the smoke would cover most parts of the fire, what would decrease the glow. So rendering fire after smoke seems to be a better solution. The issue can be seen in Figure 4.3, where the camera is in front of the beam. Actually the smoke particles should cover the glowing fire from this perspective, but they don't. A solution would be to store fire and smoke particles in the same data structure and order the whole buffer, like it is done for the smoke particles. The problem with this approach is, that applying



(a) Simple reflection. (b) Random reflection with speed reduction.

Figure 4.2: This figure illustrates two modes of collision handling.

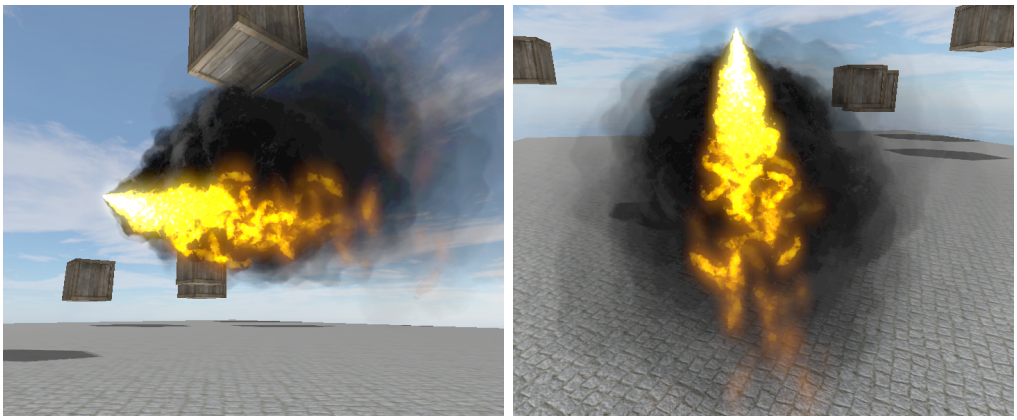


Figure 4.3: These figures illustrate the issue, that fire particles are always shining through the smoke.

different blending methods on different elements in the same buffer during one draw call is not possible.

4.2 Analysis and performance

In this section the advantages and disadvantages of all introduced methods are discussed and the obtained performance numbers are analyzed. The environment parameters for the benchmark and the used hardware is stated in Table 4.2.

First the analysis of the used resources is given. Consider having one GPU particle data buffer, where n_{max} is the maximal particle count and N_{max} is the next power of 2, that

Table 4.2: Benchmark parameters and hardware specification

Type	Value
CPU	AMD Ryzen 2600X
GPU	Nvidia GeForce GTX 1070
RAM	16GB DDR4
Display Dimension	1280x720
Number of Colliders	6

is greater or equals to n_{max}. The following calculations are applicable for the buffer setup used in our environment. For a single CPU particle, 11 float values are stored in the main memory, where only 8 of them need to be uploaded to the GPU fo rendering. For a single GPU particle 11 float values are stored and therefore $n_{max} * 11$ for the whole buffer. An indices buffer, that is needed for sorting, has N_{max} unsigned int values. When analyzing the biggest system ($n_{max} = 160000$, $N_{max} = 262144$), one GPU particle data buffer would take ~6,7 Mebibyte and one indices buffer 1 Mebibyte. Therefore the optimized compute shader approach takes ~7,7 Mebibyte. For transform feedback and the unoptimized compute shader approach one indices buffer and two particle buffers are needed, resulting in ~14,4 Mebibyte in total. The CPU approach takes ~6,7 Mebibyte in the main memory and ~4,9 Mebibyte on the GPU, resulting in ~11,6 Mebibyte in total.

The communication costs between CPU and GPU are analyzed. Each render frame consists of only one draw call for each particle system. So the communication costs are equal for all approaches for the rendering. On the other hand there are differences for the simulation. During the CPU approach simulation frame the active particle data needs to be uploaded to the GPU (~2,4 Mebibyte for the biggest system), what is obviously very slow. Another big drawback is, that the transferred data depends on the particle size. A particle can consist of many attributes, for example position, velocity, transformation, color, life progress, texture coordinates and so on. Each attribute increases the communication costs and therefore the particle size is critical for the CPU solution. On the other hand a GPU simulation frame consists of one draw or compute call, uploading the list of colliders and writing/reading the information, that is applied to the whole system, as uniforms or buffer values.

After the analysis the implemented methods are compared according speed, that means the elapsed time measured in milliseconds. Figure 4.4 shows the benchmarks of the four introduced simulation techniques without particle sorting applied on the biggest systems. It can be seen, that the CPU approach seems to be faster for small systems, that means less than ~4000 particles. When comparing the GPU approaches at first glance there seem to be no differences in speed. After tracking the elapsed time for each simulation function it can be seen, that transform feedback is slower than both computershader versions. In addition there were some breaks with the particle count, when using transform feedback, namely the full 160000 particles were not reached constantly. This is,

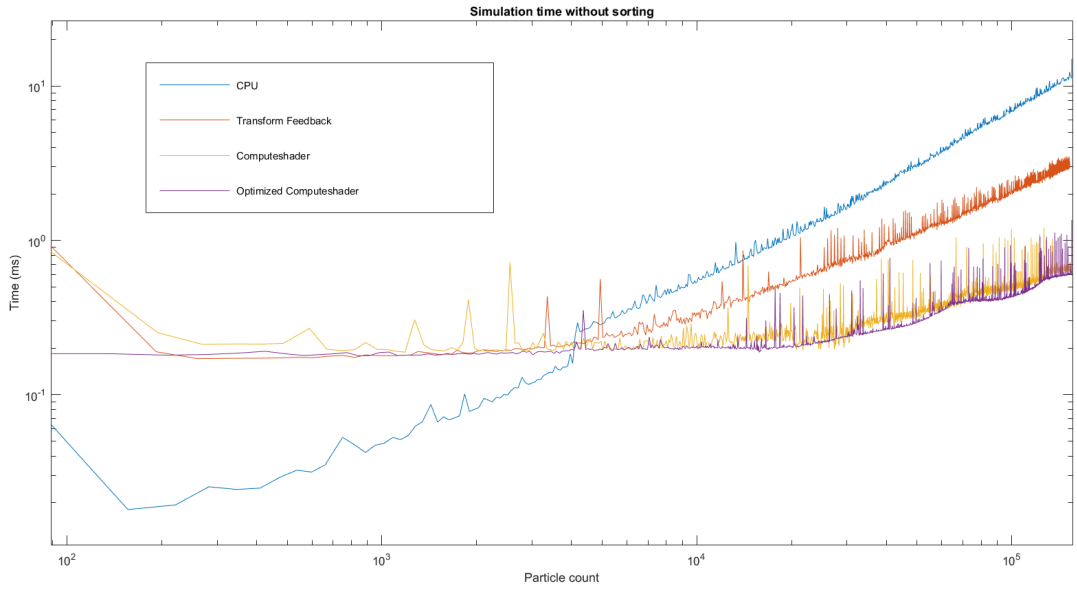


Figure 4.4

because the geometry shader's maximum output of new primitives is limited [Wik19b] and therefore the overall particle count depends on the current FPS. Furthermore the optimized computeshader seems to be a little bit faster than the simple computeshader approach with two buffers. That means, that the implemented optimized computeshader is the most efficient solution in terms of used resources and speed for big systems in our test application.

Figure 4.5 shows the elapsed times of simulation using the optimized computershader and the rendering (*Render fire*, *Render Smoke* and the whole process from Figure 3.1). This benchmark was tracked with the camera near in front of the beam, leading to particles covering the whole screen and a maximum of blended fragments. It can be seen, that rendering smoke particles needs more time than rendering fire particles. This can be explained with the fact, that alpha blending is more expensive than additive blending.

Figure 4.6 shows the speeds of the implemented GPU sort algorithm and the used CPU library. The sort algorithms were not applied on the biggest systems, because the GPU sort algorithm has got real performance issues with that amount of particles. It can be observed, that the CPU solution is much more efficient. In addition it seems to be, that the GPU sort is a little bit faster, when not using the transform feedback approach. This could be, because of the different data layout of the buffers, leading to much more unnecessary data movements.

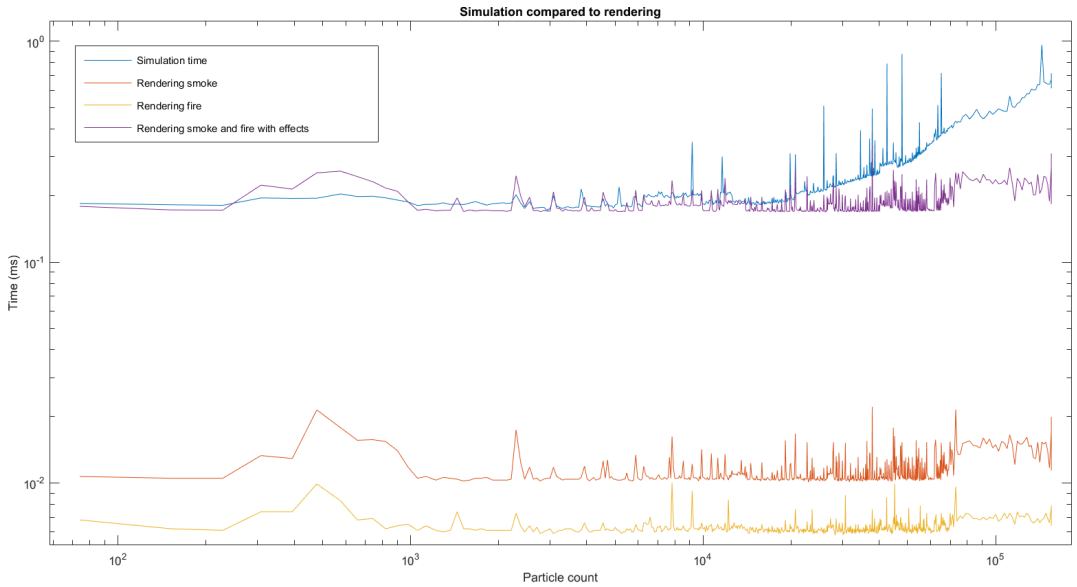


Figure 4.5

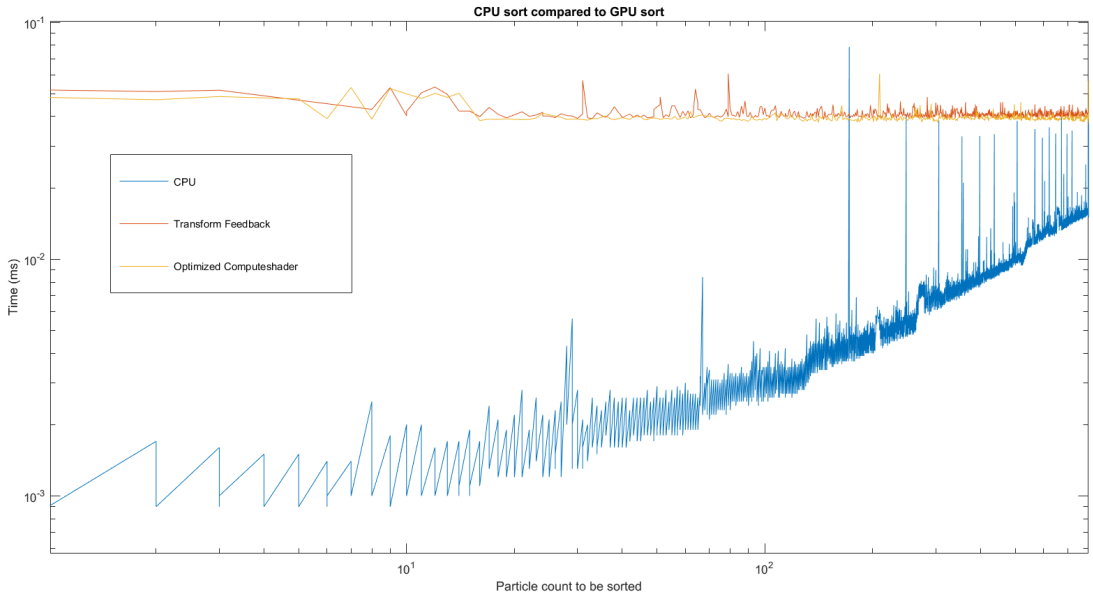


Figure 4.6

CHAPTER 5

Conclusion and Future Work

The aim of this thesis was to investigate different approaches for rendering and simulating large particle systems in real-time. The problem statement was to implement simulation and rendering of interactive particle systems consisting of various numbers of particles that should approximate the fire and smoke parts of a fire beam ideally with 60 frames per second. The most important condition for the simulation stage, namely an efficient way to update a huge amount of particles, can be measured. The collision handling environment was based on less than ten global axis-aligned cubes as colliders and therefore the implementation was easy and the influence on the simulation speed was small. A big drawback of the implemented rendering solution was the shine through of the fire parts, because of the seperate rendering processes of smoke and fire particle systems. Another drawback was, that the proper representation of transparent smoke particles need a way to sort the smoke fragments before rendering. To approximate this a GPU sort algorithm has been implemented, that sorts whole particles according the camera distance. The main issue with this algorithm is its high performance costs which become very noticeable for larger particle systems.

The introduced simulation methods all come with strengths and weaknesses. When a small particle system is needed the CPU approach is the best choice. Transform feedback can be used for bigger systems, because it is fast and a lower version of OpenGL is required [Wik18b]. For very big systems compute shaders are the best choice.

As future work we would like to find solutions to overcome the drawbacks. The shine through problem can be solved with a technique, that composes particle systems with different rendering states. In order to compose the fire and smoke particles with different blending equations *Order-Independent Transparency* (see section 2.6, [MCTB12]) would be a suitable method. This would also allow for omitting a GPU sorting algorithm, because the fragments can be sorted and individually composed at the same time. To improve the rendering of the particles, an approach like *High-Speed, Off-Screen Particles*, described in subsection 2.2.2 and in [Ngu07] could be used. To achieve collision detection and handling

with more complex geometries, the method described in subsection 2.5.1[KLRS04] can be implemented with e.g. 3D-Texture as depth maps.

List of Figures

2.1 This figure shows the differences between viewplane- and viewpoint-oriented billboards. The left picture illustrates the viewplane-based approach, and on the right the viewpoint-based polygons can be seen. Adapted from [Lar10]. 6

2.2 This figure shows the process to create the bloom/glow effect. (a) is the original scene image and (b) is the texture, that holds the parts which are to receive the effect. These parts were defined by an additional texture (not shown; often called Alpha Texture). (c) holds the blurred image from (b) and the final image (d) is composited as the sum of (a) and (c). Adapted from [Fer04a]. 8

2.3 This figure demonstrates the idea of using depth information for removing hard edges. It can be seen, that d1 is behind the scene geometry/other objects and d2 is a little bit in front of the scene geometry/other objects. When rendering the particle the common way, a hard edge would be noticeable where other geometry is intersected by the bounded plane representing the particle. This can be solved by smoothly decreasing the alpha value of the particle in that regions. In general, that means the closer fragments are to the scene geometry, the more they have to be faded out. Adapted from [Lor07]. 9

2.4 This figure shows the differences of particles rendered at full resolution (left image) and the *Mixed-Resolution Rendering* technique (right image). Adapted from [Ngu07]. 10

2.5 This figure shows the rendering process used in [Fer04b]. The glow stage can be seen on the right and the particles on the left. Adapted from [Fer04b]. 12

2.6 This figure shows the bitonic network for 16 elements. The black arrows define the comparisons and the colors (green, blue) are encoding the sort direction of this block in this pass. Adapted from [Wik19d]. 13

2.7 In this figure, an example for the bitonic merge sort algorithm in ascending order for $n = 8$ elements is given, where Bunch is *bunchSize* and Jump is *jumpSize*. 14

2.8 This figure shows some results for big particle systems colliding with surrounding objects. Adapted from [KLRS04]. 15

3.1 This figure shows all the steps of our algorithm. At the left, the separate compute/render steps are depicted. On the right, the required resources are shown which are read by the steps on the left or written into, respectively. 19
3.2 This figure shows an example of a fire texture atlas, arranged in a 4 by 4 regular grid. This atlas was used to animate the fire particles. In general the layout of the atlas doesn't matter, the subtextures could also be aligned in one row or column. It can be easily seen, that this atlas delivers many changes regarding size and volume, what makes the appearance of the fire more dynamic. 21
3.3 This figure shows the visual differences of disabled and enabled bloom. It can be easily seen, that the bloom increases the quality of the fire. 23
3.4 This figure shows the visual differences of *Soft Particles* disabled and enabled. It can be easily seen, that hard edges of the billboard quad can become visible without this effect. 24
3.5 In this figure, an illustration of the layout of different GPU particle buffers can be seen. n is the current particle count and $n + k = n_{max}$ the maximum amount of particles that can be stored in this buffer. $n + k + m = N_{max}$ is the next power of 2 of n_{max}. It can be seen that transform feedback arranges the particles from left to right. The compute shader approach accesses the buffer at random positions and therefore the particles are stored at random indices in the buffer. 29

4.1 This figure shows 4 particle systems with different amount of particles. . . 32
4.2 This figure illustrates two modes of collision handling. 33
4.3 These figures illustrate the issue, that fire particles are always shining through the smoke. 33
4.4 35
4.5 36
4.6 36

List of Tables

4.1 Sizes of particle systems 31
4.2 Benchmark parameters and hardware specification 34

List of Algorithms

3.1 Update particles using transform feedback 26
3.2 Update particles using compute shader 28

Bibliography

[AD14] Johan Anderdahl and Alice Darner. Particle systems using 3d vector fields with opengl compute shaders, 2014. Thesis, Faculty of Computing, Blekinge Institute of Technology.

[AMHH08] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2008.

[Bas10] CJ Bass. No fire without smoke: smoke rendering and light interaction for real-time computer graphics. 2010. Unpublished Thesis. Coventry: Coventry University.

[con19a] Wikipedia contributors. Batcher odd–even mergesort — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Batcher_odd-even_mergesort&oldid=886463978`, 2019. [Online; accessed 29-April-2019].

[con19b] Wikipedia contributors. Quicksort — Wikipedia, the free encyclopedia. `https://de.wikipedia.org/w/index.php?title=Quicksort&oldid=187811914`, 2019. [Online; accessed 29-April-2019].

[Fer04a] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 21. Real-Time Glow. Pearson Higher Education, 2004.

[Fer04b] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 6. Fire in the "Vulcan" Demo. Pearson Higher Education, 2004.

[JA17] Simon Johansson and Robin Andersson. Comparison between particle rendering techniques in directx 11. 2017. Thesis, Faculty of Technology and Society, Malmö University.

[JK18] Michael Satran John Kennedy. Stream-output stage. `https://docs.microsoft.com/en-us/windows/desktop/direct3d11/d3d10-graphics-programming-guide-output-stream-stage`, 2018. [Online; accessed 29-April-2019].

[KLRS04] Andreas Kolb, Lutz Latta, and Christof Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131. ACM, 2004.

[Lar10] Robert Larsson. Interactive real-time smoke rendering. 2010. Thesis, Chalmers University of Technology.

[Lor07] Tristan Lorach. Soft particles. *NVIDIA DirectX*, 10, 2007.

[MCTB12] Marilena Maule, Joao LD Comba, Rafael Torchelsen, and Rui Bastos. Memory-efficient order-independent transparency with dynamic fragment buffer. In *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images*, pages 134–141. IEEE, 2012.

[MDTP+04] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Transactions on Graphics (TOG)*, 23(3):787–795, 2004.

[Ngu07] Hubert Nguyen. *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 23. High-Speed, Off-Screen Particles. Addison-Wesley Professional, first edition, 2007.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 46. Improved GPU Sorting. Addison-Wesley Professional, 2005.

[PVV05] Tuan Q Pham and Lucas J Van Vliet. Separable bilateral filtering for fast video preprocessing. In *2005 IEEE International Conference on Multimedia and Expo*, pages 4–pp. IEEE, 2005.

[Rod18] Jorge Rodríguez. docs.gl - opengl api documentation. `http://docs.gl/`, 2018. [Online; accessed 29-April-2019].

[Wik18a] OpenGL Wiki. Main page — opengl wiki,. `http://www.khronos.org/opengl/wiki_opengl/index.php?title=Main_Page&oldid=14430`, 2018. [Online; accessed 29-April-2019].

[Wik18b] OpenGL Wiki. Transform feedback — opengl wiki,. `http://www.khronos.org/opengl/wiki_opengl/index.php?title=Transform_Feedback&oldid=14435`, 2018. [Online; accessed 29-April-2019].

[Wik19a] OpenGL Wiki. Compute shader — opengl wiki,. `http://www.khronos.org/opengl/wiki_opengl/index.php?title=Compute_Shader&oldid=14536`, 2019. [Online; accessed 29-April-2019].

[Wik19b] OpenGL Wiki. Geometry shader — opengl wiki,. http://www.khronos.org/opengl/wiki_opengl/index.php?title=Geometry_Shader&oldid=14512, 2019. [Online; accessed 29-April-2019].

[Wik19c] OpenGL Wiki. Query object — opengl wiki,. http://www.khronos.org/opengl/wiki_opengl/index.php?title=Query_Object&oldid=14509, 2019. [Online; accessed 29-April-2019].

[Wik19d] Wikipedia contributors. Bitonic sorter — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Bitonic_sorter&oldid=894063148, 2019. [Online; accessed 29-April-2019].