School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Self-illuminating Explosions

Szymon Spiesz

Submitted in accordance with the requirements for the degree of MSc High Performance Graphics and Games Engineering

2019/2020

The candidate confirms that the following have been submitted:

Items	Format	Recipient(s) and Date
Deliverable 1	Report	SSO 28/08/20
Deliverable 2	URL Software code, GitHub repository	Supervisor, Assessor 28/08/20

Type of project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) ____________________

Summary

Explosions are often featured in various applications to enhance user's experience. In case of entertainment purposes, simulations of explosions are approximations of real explosions because only visual effects are desired, and not the physical accuracy. However, very often simulations are based on simplified physical assumptions to achieve an effect similar to the real one.

Modelling and rendering of self-illuminating explosions is a complicated task which requires many aspects of computer graphics to work together. Volumetric rendering can quickly get very expensive which makes it hard to create visually convincing effects while maintaining reasonable time of computations.

In this paper I present a program for rendering self-illuminating explosions. The model of explosion is based on physical equations of fluid flow called Navier-Stokes equations. These equations are hard to solve when they are used for precise physics calculations, which is not the case here. Therefore, usage of Jos Stam's algorithms for solving Navier-Stokes equations for visual effects is sufficient. Part of the program responsible for self-illumination of the explosion is done with casting rays from the light source and depending on the density accumulated along the ray, picking the color from the transfer function.

Acknowledgements

I would like to thank my supervisor Hamish Carr for pointing me in the right direction. His support and valuable advices helped me to find the right solutions to the problems I have been encountering and finalize the project.
I would also like to thank my assessor He Wang for his opinion about my project and for the propositions on how to improve it.

Contents

1 Introduction **2**

1.1 Project Aim 2

1.2 Objectives 2

1.3 Deliverables 2

1.4 Ethical, legal, and social issues 2

2 Background Research **3**

2.1 Literature Survey 3

2.1.1 Illumination 3

2.1.2 Ray Tracing 3

2.1.2.1 Algorithm overview 4

2.1.2.2 Ray Casting 4

2.1.2.3 Umbra and Penumbra 5

2.1.2.4 Types of Surfaces 5

2.1.3 The Rendering Equation 7

2.1.3.1 Finite Element Methods 8

2.1.3.2 Monte Carlo Ray Tracing Methods 8

2.1.3.3 Hybrid Methods 9

2.1.4 Photon Mapping 9

2.1.4.1 Photon Map 9

2.1.4.2 Kd-Tree 10

2.1.4.3 Rendering an Image using Photon Map 11

2.1.5 Lighting Grid Hierarchy 12

2.1.5.1 Illumination 13

2.1.5.2 Shadows 14

2.1.6 Lightcuts 14

2.1.6.1 Light Tree 15

2.1.6.2 Finding an acceptable lightcut 16

2.1.6.3 Reconstruction cuts 16

2.1.7 Fluid Dynamics - explosion data generation 18

2.1.7.1 Computational Fluid Dynamics 19

2.1.7.2 Mathematical Model of the Fluid Flow 19

2.1.7.3 Conservation Laws 19

2.1.7.4 Navier-Stokes Equations 20

2.1.7.5 Fluid Dynamics For Games - Mathematical Model 20

2.1.7.6 Particle-Based Fluid Simulation for Interactive Applications - Mathematical Model . . . 24
2.2 Choice of methods . . . 27
2.2.1 Physics Based Model of Explosion . . . 27
2.2.2 Illumination of Model . . . 27

3 Software Requirements and System Design 28
3.1 Software Requirements . . . 28
3.2 System Design . . . 29
3.2.1 C++ . . . 29
3.2.2 OpenGL . . . 29
3.2.3 GLSL . . . 29
3.2.4 GLM . . . 30
3.2.5 QT . . . 30

4 Software Implementation 31
4.1 Model of Explosion . . . 31
4.1.1 Navier-Stokes Equations . . . 31
4.1.1.1 Simulation Grid . . . 31
4.1.1.2 Fluid . . . 31
4.1.1.3 Velocity and Density Fields . . . 32
4.1.1.4 External Forces . . . 32
4.1.1.5 Diffusion . . . 33
4.1.1.6 Linear Solver . . . 33
4.1.1.7 Boundary of Fluid . . . 33
4.1.1.8 Advection . . . 34
4.1.1.9 Project function . . . 35
4.2 Illumination of Explosion . . . 36
4.3 Determining which vertices are needed . . . 36
4.3.1 Casting ray from the light source . . . 38
4.3.1.1 Finding a Path . . . 39
4.3.2 Setting Color - Transfer Function . . . 41
4.3.2.1 Color Lookup Table . . . 41
4.3.2.2 Determining Colour . . . 42

5 Software Testing and Evaluation 43
5.1 Initial conditions . . . 43
5.1.1 Grid Size . . . 43
5.1.2 Fluid . . . 43
5.1.3 Velocity Field . . . 44
5.1.4 Density Field . . . 44

5.2 Parameter sensitivity 46
5.2.1 Time Step 46
5.2.2 Diffusion 47
5.2.3 Viscosity 48
5.2.4 Number of Iterations 49
5.3 Dependant Parameters 50
5.3.1 Grid Size and Density Field 51
5.3.2 Grid Size and Velocity Field 51
5.4 Physics of Explosion and Illumination Algorithm Efficiency 51
5.4.1 Efficiency of Physics Calculations 51
5.4.2 Efficiency of Color Calculations 52

6 Future Work and Conclusions 54
6.1 Future Work 54
6.1.1 Possible Improvements 54
6.1.1.1 Bigger Grids - Better Resolution 54
6.1.1.2 Fluid simulation 54
6.1.1.3 Picking the color 54
6.2 Conclusions 55

References 58

Appendices 60

A External Material 61

B Ethical Issues Addressed 62

List of Figures

2.1 An example of ray tracing method [7] 3
2.2 Ray Casting [7] 4
2.3 Area light casting shadows on a scene [7] 5
2.4 Light ray bouncing off a mirror surface 5
2.5 Light ray bouncing off a glossy surface 6
2.6 Light ray bouncing off a diffuse surface 6
2.7 Photons inside the sphere are used to approximate irradiance at the point of interest [9]. $L(\mathbf{x} \rightarrow \vec{\omega})$ is the reflected radiance from the point x in direction $\vec{\omega}$. 10
2.8 kd-tree decomposition for points (5, 5), (1, 7), (8, 1.5), (2.5, 2), (4, 8.5), (7, 8.5), (1, 6) [4] 11
2.9 Lighting of an explosion done with lighting grid hierarchy 12
2.10 Contribution of light from different grid levels for a single frame 13
2.11 First picture shows the exact solution with four point lights, the second shows approximated solution achieved by clustering lights 3 and 4. The third picture shows the difference (gray colour) between picture one and two. 15
2.12 An example of light tree and light cuts. The orange cut clusters lights 3 and 4, the blue cut clusters lights 1 and 2, and the purple cut clusters 1 with 2 and 3 with 4. Coloured regions marks the regions where effects given by exact and clustered solution are indistinguishable. 16
2.13 Effects achieved with lightcuts method. 17
2.14 Steps in the process of development of the computational fluid dynamics system. [8] 18
2.15 Navier-Stokes equations describing fluid flows: equation describing evolving velocity in a compact vector notation (top), equation describing density moving through the velocity field. [17] 20
2.16 Steps in solving change in density field. [17] 21
2.17 Grid in the middle shows moving cell's center forward in time according to the velocity field (left grid), which is a complicated solution. Grid on the right shows finding previous cell's center positions, which is a better solution. [17] 22
2.18 The Helmholtz-Hodge Decomposition states that the sum of a mass conserving field and gradient field results in a velocity field (top). Therefore, to get a mass conserving field, gradient field has to be subtracted from velocity field (bottom). [17] 23

2.19 Effects generated with fluid dynamics solver by Jos Stam [17] 24
2.20 Effects generated with method based on smoothed particle hydrodynamics [14] 26

3.1 Vertex Shader and Fragment Shader in the OpenGL rendering pipeline [16] 30

4.1 Class of a fluid containing fluid properties and functions needed for explosion data generation. 32
4.2 Function for solving linear systems 33
4.3 *setBoundary*() function prevents fluid spilling out of the box 34
4.4 *advect*() function moves the velocities along themselves and moves densities along velocities 35
4.5 *project*() function is responsible for making velocity vector field mass conserving 36
4.6 Grid of the vertices is created once at the beginning of the program. 37
4.7 Code responsible for determining which vertices need to be send to the vertex shader. It also creates new indices of vertices because the number of vertices will be smaller and therefore indices will have to be changed. . . 38
4.8 Functions needed for casting the ray. 38
4.9 Distance function 39
4.10 Function which determines which points is the closest to the destination. . 39
4.11 Function which goes from one point to another and returns the path between them. 40
4.12 Colors used for interpolation in transfer function. 41
4.13 Function which linearly interpolates given colors. 41
4.14 setting the color for all of the needed vertices in a particular time step . . . 42

5.1 initialization of a fluid: first variable is a time step, second is diffusion and the third is viscosity 43
5.2 velocity modelling - each cube to be modelled has specified velocity values for x, y or z axis 44
5.3 density modelling - 5x5 voxel field filled with initial density 45
5.4 result of simulation in a 27x27x27 grid with settings showed in code snippets in this section 46
5.5 Visual changes dependent on *timeStep* variable, pictures of the same frame, but with different time steps 47
5.6 Results produced by fluid simulation with different $diffusion$ values. The frames and all settings are the same except $diffusion$ variable. 48
5.7 Impact of *viscosity* value on results of the simulation. Pictures present the same frames of simulation with *viscosity* values changed. 49
5.8 Differences in results caused by changing the accuracy of linear solver. . . . 50

6.1 Example of explosion 1 56
6.2 Example of explosion 2 57

List of Tables

5.1 Efficiency of physics calculations depending on the number of iterations for a grid size 27^3. 52
5.2 efficiency of the program dependent on the number of grid cells with initial density greater than 0 52
5.3 efficiency results for different initial density values 53
5.4 Results for efficiency depending on the $diffusion$ 53
5.5 *viscosity* increases the stickiness of the fluid, which keeps it from spreading to a larger number of grid cells, making the simulation faster 53

Chapter 1

Introduction

1.1 Project Aim

The aim of the project is to implement a system, which given initial set of data (velocity field and density field values) and conditions (3D grid size, time step, viscosity, diffusion rate) will render a self-illuminating explosion in a 3D grid. The project is expected to include an algorithm for generation of data needed to model the explosion. It is also expected to implement a lighting technique for self-illumination of generated model.

1.2 Objectives

The project consists of two important parts which combined together will deliver the final effect. The first one is physics based simulation of explosion which will be used to generate volume of smoke produced by the explosion. The second part is a lighting technique which can be used when dealing with volume rendering to simulate temperature transition in close proximity of the explosion.

1.3 Deliverables

The expected deliverables are:

- The program for rendering self-illuminating explosions.
- A written report

1.4 Ethical, legal, and social issues

There is no ethical, legal or social issues related to this project.

Chapter 2

Background Research

2.1 Literature Survey

2.1.1 Illumination

In the next few subsections I am going to describe the methods used to handle lighting computations in computer graphics, which over the years researchers developed. I will also briefly explain concepts needed to understand how the described methods work.

2.1.2 Ray Tracing

Ray tracing is a rendering technique introduced by Turner Whitted in 1980 [23]. It is used to track the light rays through the scene and visualize a variety of optical effects such as scattering, reflection or refraction. It is a more expensive technique than ray casting which does not trace the rays recursively.

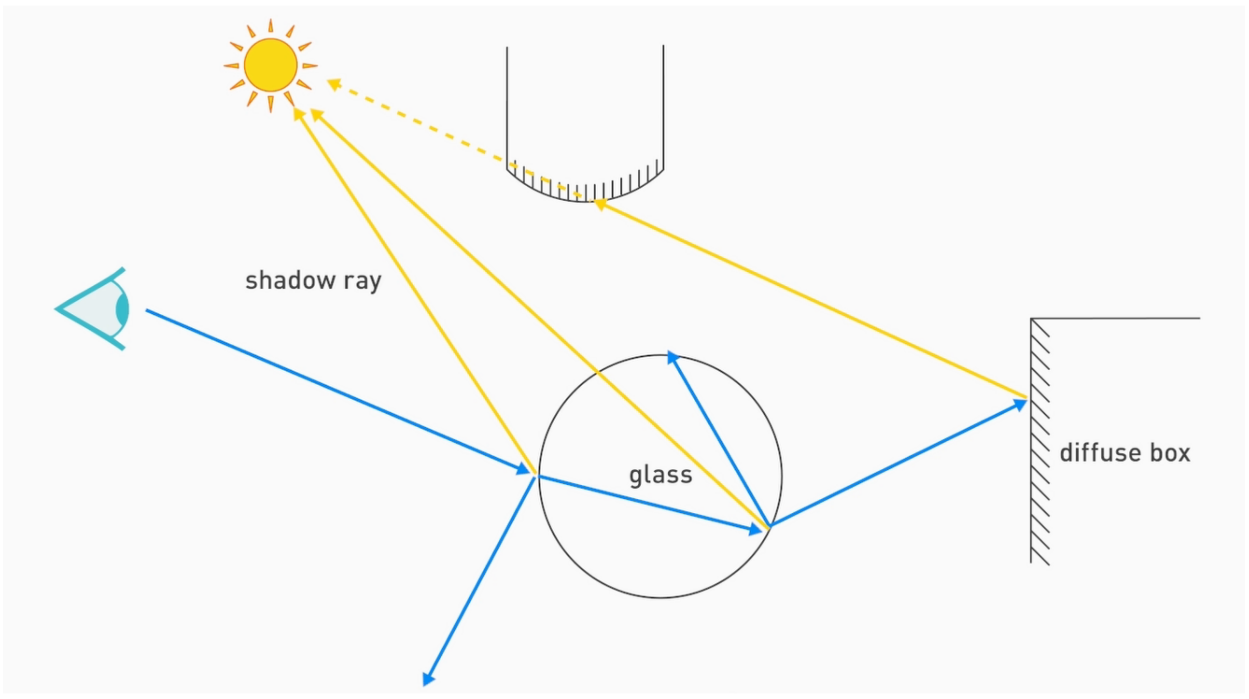


Figure 2.1: An example of ray tracing method [7]

2.1.2.1 Algorithm overview

In ray tracing technique, when the ray hits a point on a surface it can be bounced off and further traced to account for any other impacts the rest of the scene can have on this particular point. Furthermore when the ray intersects with an object in a scene it can create even more reflection and refraction rays bouncing off the point of intersection in different directions [7] or entering the object if it is transparent. This is done to improve the quality of the image because more rays mean that the final result will be closer to the real solution which is highly desired. The number of times the ray can bounce off the objects in a scene is usually given a limit which results in a faster computation time but worsens the quality.

2.1.2.2 Ray Casting

Ray casting is a method for finding an object along the path of the ray. When the ray hits the object, another ray can be cast from the intersection point to the light source to test if the point is in shadow or not. In case of the volume rendering, the rays can be cast through the volume and sampled along its path to acquire data varying across the volume [22]. This technique can also be used for collision detection [24] or to determine if a surface is visible from a particular point of view.

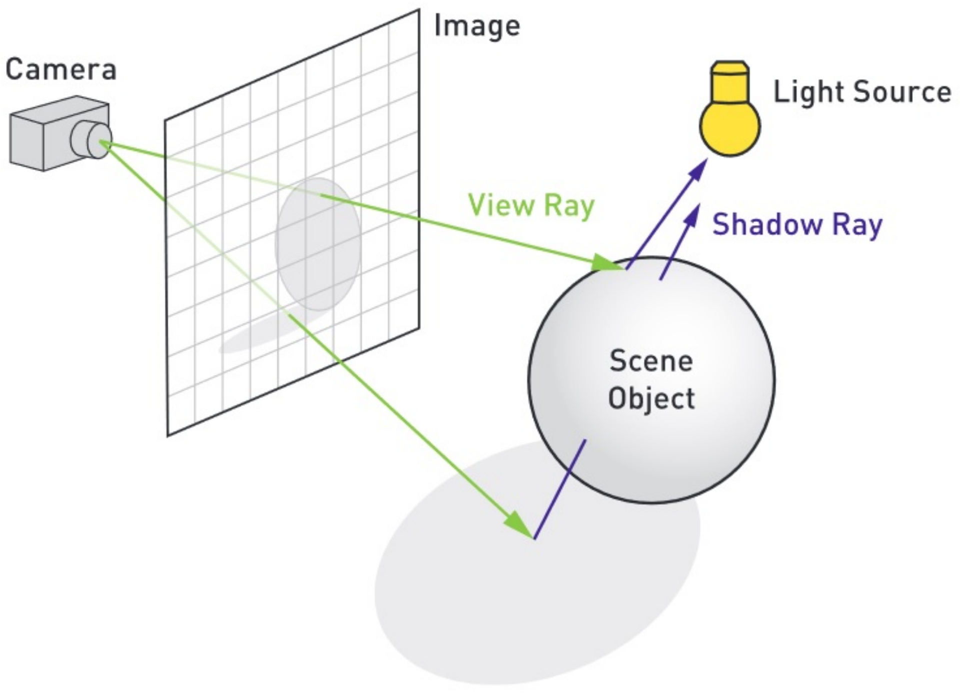


Figure 2.2: Ray Casting [7]

2.1.2.3 Umbra and Penumbra

Depending on a type of the light source, different effects can be produced. For example when the light source is not a point light, a point on the surface can be partially illuminated. This means that for this point, part of the light source is visible and the other part is not and casts shadows. Areas of points which are partially illuminated are called *penumbra* and areas which are fully occluded are called *umbra*.

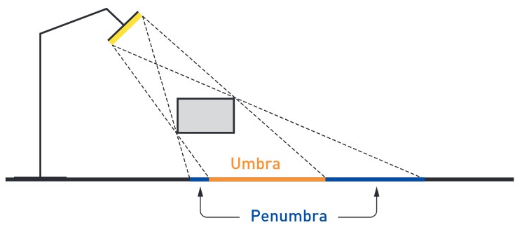


Figure 2.3: Area light casting shadows on a scene [7]

2.1.2.4 Types of Surfaces

When ray tracing was first introduced it considered all surfaces as perfectly shiny and smooth. But in reality there are different types of surfaces, which have various impacts on reflected rays. For a **mirror type surface** the incoming ray is reflected in a single direction, where the angle of reflection is equal to the angle of incidence.

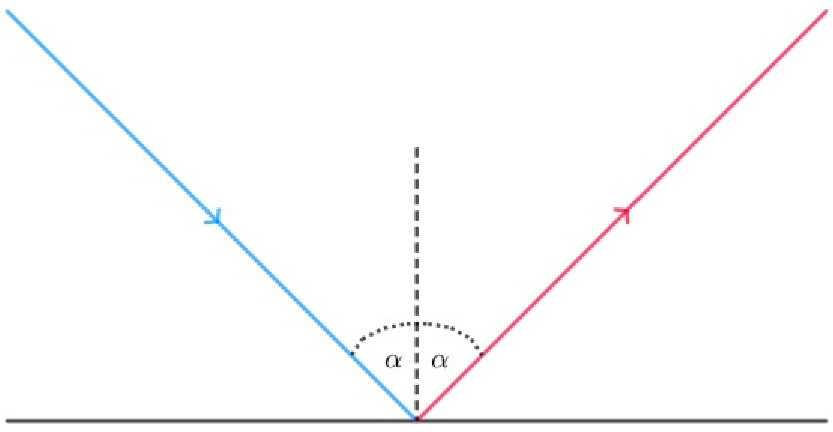


Figure 2.4: Light ray bouncing off a mirror surface

Another type of surface which impacts a reflection is a **glossy surface**. This type of a surface sends multiple rays around the direction of reflection which results in a glossy reflection.

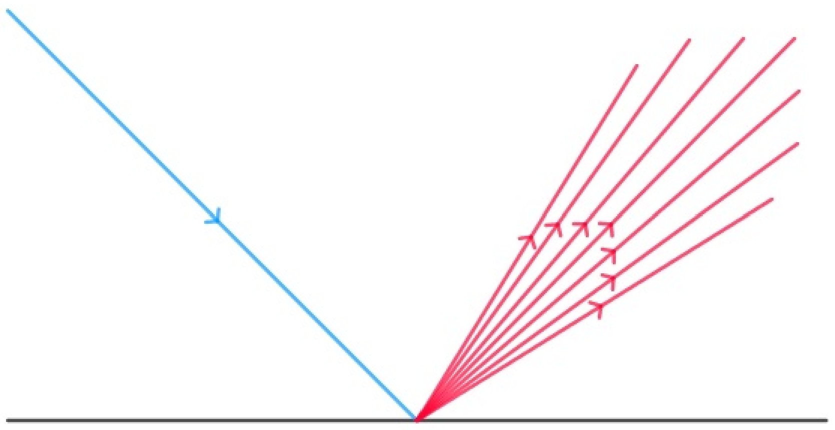


Figure 2.5: Light ray bouncing off a glossy surface

In case of the **diffuse or matt surface**, when the ray hits the point, the light gets scattered in all directions. One of the types of diffuse reflection is Lambertian reflectance [12] in which the light is reflected equally in all directions and the surface appears to be equally bright.

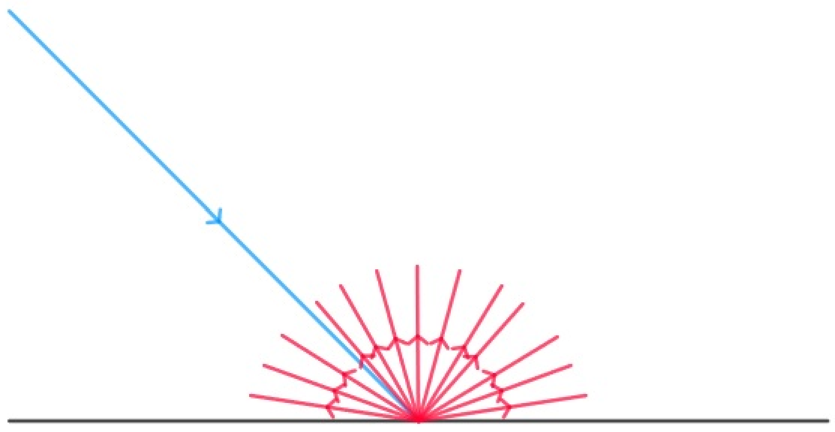


Figure 2.6: Light ray bouncing off a diffuse surface

Ray-tracing method is a realistic simulation of how the light is transported in reality. Thanks to that, this method is capable of producing images that have a realistic look.

However to improve the quality of an image, more rays have to be shot which will obviously increase the computational cost. Therefore this method will be especially useful in applications where long time of rendering is not a problem, such as film and television. For this method to be suited for real-time applications, where the scene has to be rendered many times per second, the hardware has to be of good quality and the method itself needs hacks and speed-ups.

2.1.3 The Rendering Equation

Rendering Equation introduced into computer graphics in 1986 by James T. Kajiya [11] is an integral equation which describes the outgoing radiance from a point on a surface. The equations states that the light reflected from a point of a surface is the sum of light emitted by this point and reflected light which is coming to this point by scattering from all other points at the scene. This equation is a more or less accurate approximation of what all rendering algorithms try to achieve, which is a physical phenomenon of scattering of light.
The rendering equation has a following form:

$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right] \tag{2.1}$$

In the above equation $I(x,x')$ is the intensity of light which passes from point x' to x, $g(x,x')$ is a term which determines if points x and x' are mutually visible. If they are the term is equal to $\frac{1}{r^2}$, where r is a distance from point x to x', if they are not the term $g(x,x')$ is 0. $\epsilon(x,x')$ is the emittance term which specifies the amount of energy emitted by point x' reaching point x, expressed in $\frac{joule}{m^2 sec}$ and is given by:

$$dE = \frac{1}{r^2}\epsilon(x,x')dtdxdx' \tag{2.2}$$

$\rho(x,x',x'')$ is a scattering term which tells the intensity of energy scattered from surface point x'' through x' to x. It is calculated by using formula:

$$dE = \frac{1}{r^2}\rho(x,x',x'')I(x'x'')dtdxdx'dx'' \tag{2.3}$$

The rendering equation is very expensive to compute, thus many approximation algorithms have been developed over the years. These algorithms are divided into three categories: finite element methods, Monte Carlo ray tracing methods and hybrid methods [9].

2.1.3.1 Finite Element Methods

Before introducing Rendering Equation to computer graphics, the physical phenomenon it simulates was studied in the field of radiative heat transfer from where the finite element methods were adopted. The idea behind these methods is to discretize the geometry of the scene into bunch of smaller patches which are the base for lighting computations. Rendering equation in this case is approximated as a linear system of equations which describes how the light is exchanged between patches.

Finite element methods, however have heavy restrictions on the complexity of the scene geometry, materials and light sources. It puts the limit to where these methods could be applied, but the algorithm is easy to implement and produces view independent effects, which may be desired in some cases. Researchers have been working on improving the solution over the years, trying to remove the restrictions, while making the algorithm more complex.

2.1.3.2 Monte Carlo Ray Tracing Methods

The idea behind these methods is to acquire samples randomly and based on them produce the result. Arthur Appel [1] introduced Monte Carlo methods to computer graphics when he proposed to use ray casting for rendering images. Over ten years later Turner Whitted[23] proposed to trace the rays to simulate reflection, refraction and shadows.

Unlike finite element methods, Monte Carlo ray tracing methods work for complex lights and surfaces. Researchers used combination of Monte Carlo methods and ray tracing to simulate visual effects such as motion blur, glossy reflections or global illumination. However, because of the statistical variance of unbiased Monte Carlo sampling, images generated by this technique contain high-frequency noise which worsens visual quality. Biased Monte Carlo ray tracing algorithms such as Photon Mapping or Irradiance Caching do not have that problem because of the introduced bias error.

2.1.3.3 Hybrid Methods

As the name indicates, there are solutions which combine two methods described above. By using mixed technique it is possible to reduce discretization artifacts. Visual quality of an image produced by finite elements method can be improved by additionally applying Monte Carlo ray tracing method which once more bounces illumination and removes many visible artifacts.

2.1.4 Photon Mapping

Photon mapping is a two-pass, biased rendering technique which handles global illumination in complex environments. It was introduced in 1995 by Henrik Wann Jensen [10]. This technique approximately solves the rendering equation and takes into account all kinds of reflections. Photon mapping consists of two steps: generation of photon map and rendering of an image.

2.1.4.1 Photon Map

Process of creating a photon map starts with emitting rays from different positions on all of the light sources. The number of positions depends on the size of the light source and the size of the scene. At each position a projection map is created, which is a 2 dimensional map containing information about the types of the surface around the light source position. The number of rays emitted in a certain direction in the projection map, depends on what kind of objects (diffuse, specular) are contained in this direction. Emitted rays carry the energy from the light source which is then stored in illumination map [2] when the ray hits an object. In photon mapping method there are two types of illumination map used: one is for storing irradiance which is not the part of the caustics and the other one for storing caustics only. Such partition is done because of the difference in precision needed in these two cases (caustics are visualized directly). When the ray hits the surface it gets absorbed or reflected (BRDF is used to generate the direction of reflection) which is decided by taking Russian roulette approach.

However, illumination maps can only be used for objects which surface can be described parametrically. Therefore, photon maps are used to represent irradiance on a surface. If a ray carrying the energy hits the surface which cannot be parameterized, informations like intersection point, normal, energy and bit flags (distinction between caustic and non-caustic photons) are stored in a structure called kd-tree. Limiting the storage requirements and the usage to only special objects makes photon maps usable.

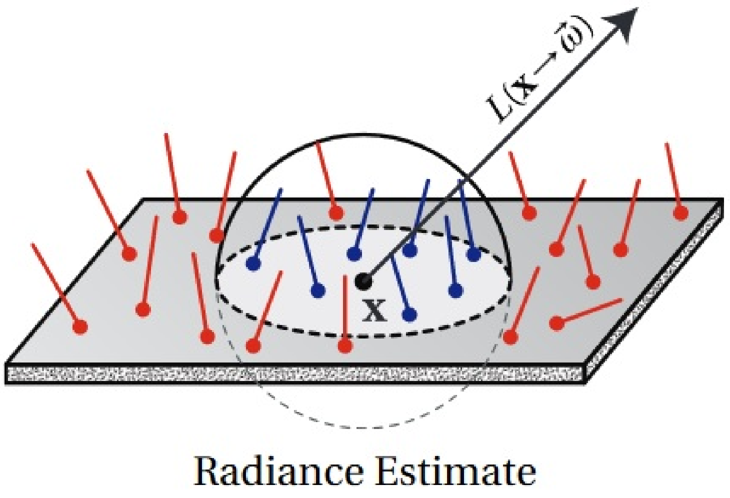


Figure 2.7: Photons inside the sphere are used to approximate irradiance at the point of interest [9]. $L(\mathbf{x} \to \vec{\omega})$ is the reflected radiance from the point x in direction $\vec{\omega}$.

To approximate irradiance at a certain point, the sphere around it is created. The size of a sphere depends on a number of photons affecting the area (max. density of photons) within the sphere or the radius of a sphere (max. radius). Accumulated energy of photons is divided by the projected surface area of the bounding sphere. Only photons for which the dot product of their normal and the normal at a point of interest is positive are taken into account. The formula for irradiance calculation at a point x:

$$E_x = \frac{\sum_{i=1}^{n} e_i}{\pi r^2} \tag{2.4}$$

E_x is the irradiance at the point x, $\sum_{i=1}^{n} e_i$ is a sum of energies of all valid photons, πr^2 is the area of the sphere projected on surface.

2.1.4.2 Kd-Tree

Kd-Tree is a binary search tree data structure [4]. Every leaf node in this tree is a k dimensional point. This data structure is used for space partitioning and multidimensional searching like nearest neighbour searching or range searches. In a kd-tree, each node, which is not a leaf, creates a hyperplane (subspace of dimension one less than ambient dimension - 1 dimensional lines are hyperplanes of 2 dimensional space) which divides the space into two parts called half-spaces.

Kd-tree is created by selecting axes one by one, which will divide the space. For a 2 dimensional tree, the root would be a hyperplane splitting space along x axis, children of the root along y axis, grandchildren of the root along x axis again and so on. The value at which the splitting hyperplane is located usually depends on the median of all values of particular axis sorted in ascending order.

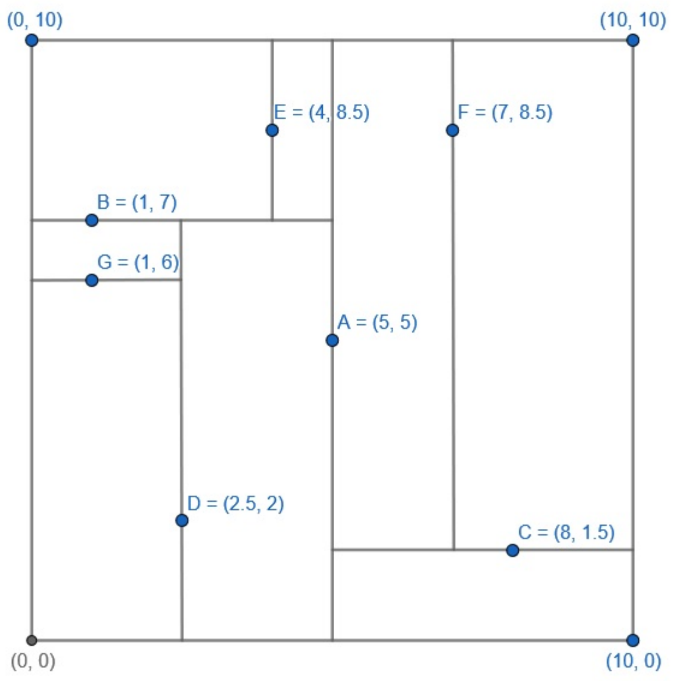


Figure 2.8: kd-tree decomposition for points (5, 5), (1, 7), (8, 1.5), (2.5, 2), (4, 8.5), (7, 8.5), (1, 6) [4]

2.1.4.3 Rendering an Image using Photon Map

Second part of the Photon Mapping method starts when photon tracing is done and all photons are stored. Rendering of a model is done with Monte Carlo ray tracing which reduces aliasing but produces unwanted noise. Radiance at the pixel is determined by shooting and tracing a ray, or multiple rays, from the eye through the pixel. When the ray hits the object, pixel radiance L_p is calculated as a sum of radiance emitted L_e and radiance reflected L_r.

$$L_p = L_e + L_r \tag{2.5}$$

As mentioned before, this method produces noise. However, the noise can be reduced by separating L_r into diffuse-like component and specular-like component because of the differences in directionality. While specular-like component is estimated by recursive Monte Carlo ray tracing, diffuse-like component is further divided into light coming from light sources, light coming from specular reflection and light coming from multiple diffuse reflections. Light coming from light sources is estimated by using stratified sampling (a way to minimize variance by separate sampling from each partition of subjects created from the whole population of subjects)[5] to minimize the approximation error. There can be two kinds of the light coming from specular reflection:

- caustics, which is taken directly from the caustics irradiance map

- multiple diffuse reflections, calculated by using irradiance gradient method with a change that secondary diffuse reflections are read directly from the irradiance maps.

The last part of diffuse-like component, light coming from multiple diffuse reflections, is calculated the same way as the second part of the light coming from specular reflection described above.

2.1.5 Lighting Grid Hierarchy

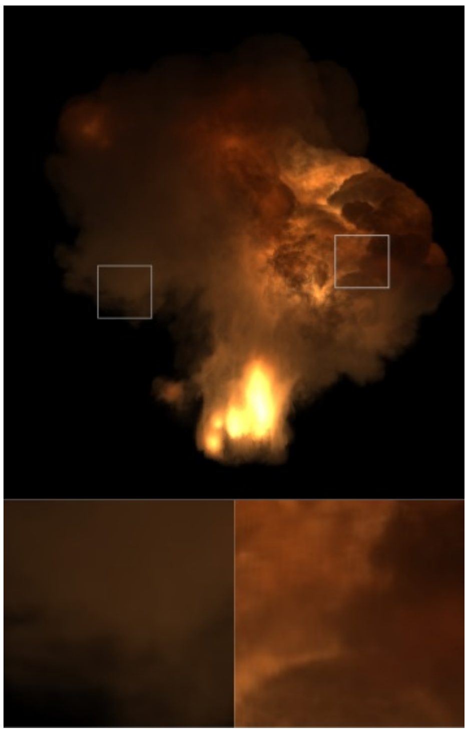


Figure 2.9: Lighting of an explosion done with lighting grid hierarchy

In a paper called Lighting Grid Hierarchy for Self-illuminating Explosions, Can Yuksel and Cem Yuksel proposed a solution to self-illumination problem [25]. The whole idea is based on creation of a hierarchy of grids of different resolutions which are used to approximate volumetric illumination. Each grid in hierarchy corresponds to the same volume and has half the resolution of the previous one. In lighting grid hierarchy structure each vertex is considered as a light source which approximates light from neighbouring point lights. Lighting at any point can be approximated by taking into account contributions from all hierarchies of the grid, no matter whether the point is outside or inside of the explosion volume.

2.1.5.1 Illumination

Illumination at a given point in space is a combination of incoming illumination from different distances (illumination is determined by approximating light from different grid resolutions). To quickly retrieve lights which are within a given radius of a point to be shaded, generated lights are kept in kd-trees. Lights with small illumination value are removed from lower hierarchy levels to speed up computations, however their impact is summed up and contributes to light in higher levels. Contributions from different grid resolutions are blended by using overlapping blending functions to achieve temporal coherence and avoid flickering.

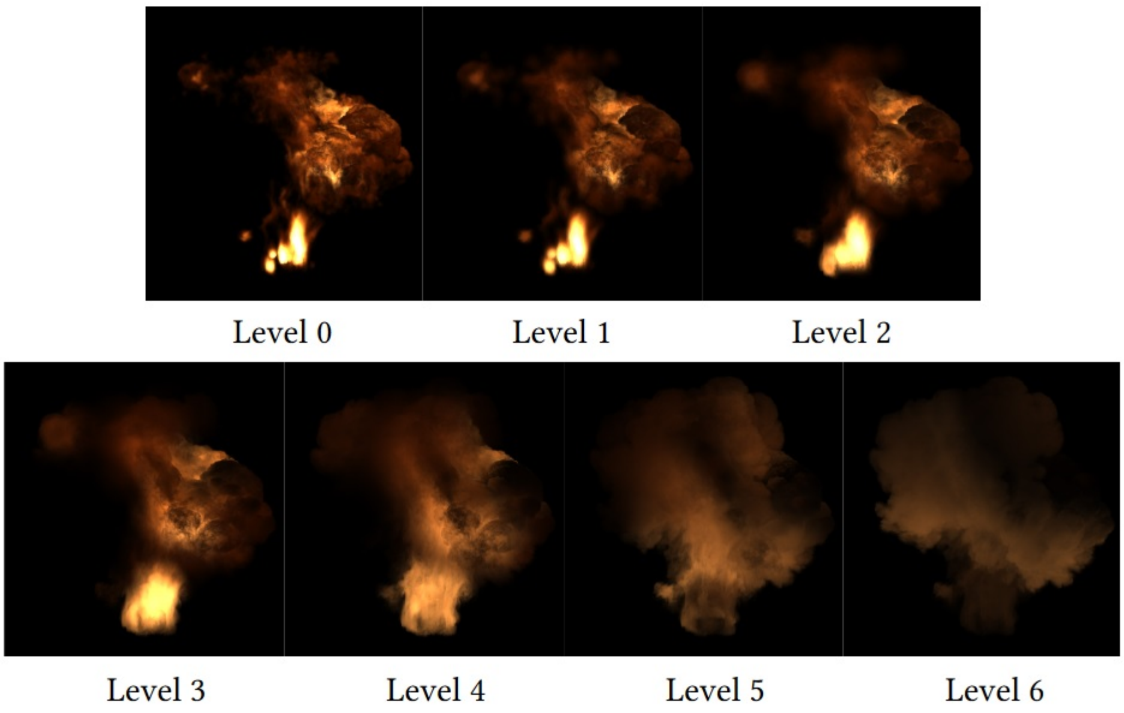


Figure 2.10: Contribution of light from different grid levels for a single frame

2.1.5.2 Shadows

The structure of the grid also allows to pre-compute volumetric shadows for faster computations during rendering. For each light, at each level, shadow map is created. In lower level grids where light sources influence limited regions, there is no need to generate shadow maps in full resolution. In case of higher levels influence radius increases but the number of light sources decreases, therefore size of shadow maps is usually smaller than in the case of lower grid levels. Usage of grid hierarchy properties to pre-compute volumetric shadows significantly speeds up the process of lighting computations.

Solution to the self-illuminating explosions problem proposed by Can Yuksel and Cem Yuksel presents visually convincing effects. As they showed, their method for self-illumination is much faster than path tracing and produces better results. Lighting Grid Hierarchy shows better effects than other method with similar approach for many point lights called lightcuts [20], because lightcuts does not avoid temporal flickering. Also, lightcuts do not limit the radius of influence of a point light which means that the approach for pre-computing shadows cannot be applied and as authors proved in a paper, the performance of the method depends on the pre-computation of volumetric shadows.

2.1.6 Lightcuts

Lightcuts method introduced in 2005 [20] at the ACM SIGGRAPH conference is another solution to many lights problem. This method has similar approach to many lights problem as Lighting Grid Hierarchy. Lightcuts method clusters lights to calculate impact of light once for a cluster and not many times for each light. When a cluster is created it is considered as a single light, but with stronger illumination. Error of the cluster will depend on the differences between lights within a cluster.

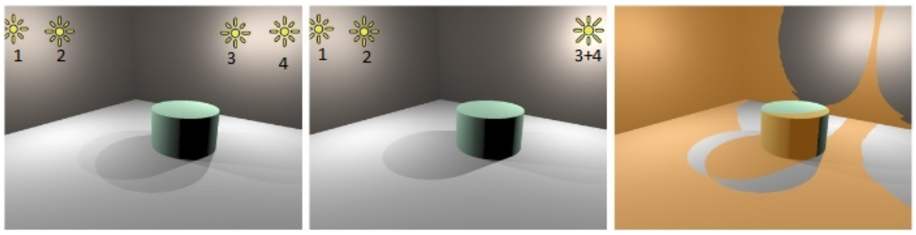


Figure 2.11: First picture shows the exact solution with four point lights, the second shows approximated solution achieved by clustering lights 3 and 4. The third picture shows the difference (gray colour) between picture one and two.

2.1.6.1 Light Tree

Light tree is a binary tree which stores individual point lights as leaves. Nodes which are not leaves are light clusters, which contain lights below them in a tree. Making a cut through a tree partitions lights into clusters, but it has to be done in a way that when walking any path from the root of the tree to any leaf node, the path will contain exactly one node from the cut. This approach makes each cut, partitioning the lights into clusters, valid. Despite the fact that each cut is valid does not mean it can be used. The problem arises because of the illumination approximation error, which may become too big and cause visible differences between using single point lights and when clustering them. To avoid visible artifacts, a certain threshold has to be set, so the cluster can be rejected when approximation illumination error is above that threshold.

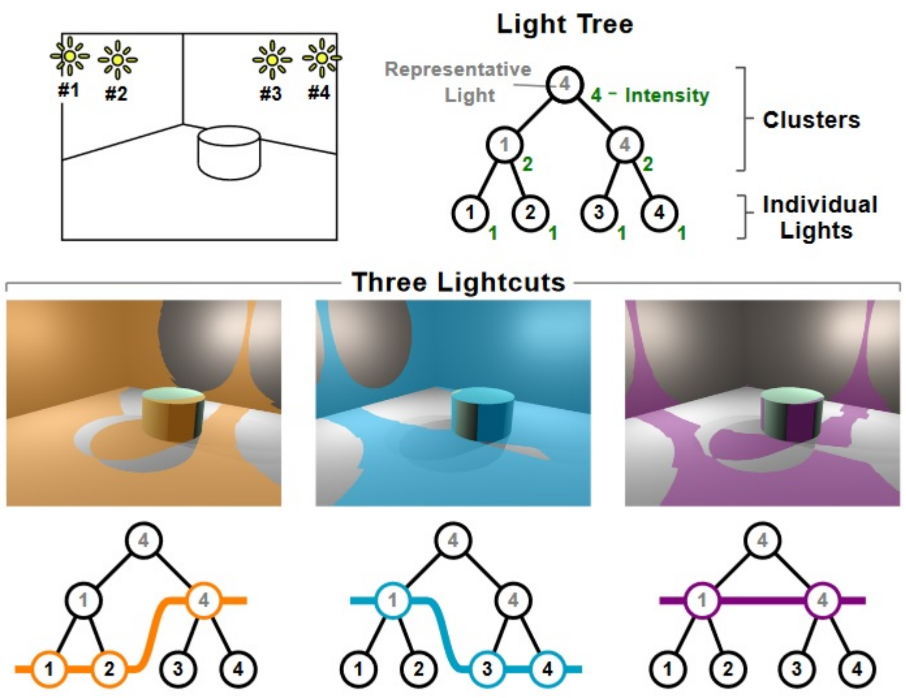


Figure 2.12: An example of light tree and light cuts. The orange cut clusters lights 3 and 4, the blue cut clusters lights 1 and 2, and the purple cut clusters 1 with 2 and 3 with 4. Coloured regions marks the regions where effects given by exact and clustered solution are indistinguishable.

2.1.6.2 Finding an acceptable lightcut

The process of choosing a valid lightcut starts with the cut at the root (or other very coarse cut), which is then refined until the error is below threshold. For each node its cluster estimate and an upper error bound are calculated. If error bound of the node is too big, the node is removed from the cut and replaced with its children. The process is repeated until the result is below the error threshold.

2.1.6.3 Reconstruction cuts

Reconstruction cuts is a way of providing a speed up to the shading process. The method relies on sparse computations of lightcuts over the image and interpolating illumination information to fully shade an image. However, simple interpolation could cause visible image artifacts (i.e. blurring surface effects which should be sharp). Thus, the authors propose a solution to this problem by determining the situation the point is at. Firstly, set of locations close to the point, where lightcuts were computed is needed. Then the process of traversal of the global light tree is performed which can have three outcomes. At each sample the node is occluded and is discarded; at all samples, illumination of a node is similar and interpolation is done; or refine down the tree until error is below threshold, approximate the illumination from the cluster and shoot shadow rays.

Lightcuts method clusters lights using binary trees, which are build with the stochastic binary decisions. This process produces different binary tree every time the algorithm is run, which creates temporal flickering problem. However, lightcuts is a scalable method which produces visually convincing effects and efficiently handles many lights problems. Authors also point out that their method can be improved by adding more illumination types, more type of light sources and better defined reconstruction cut rules.

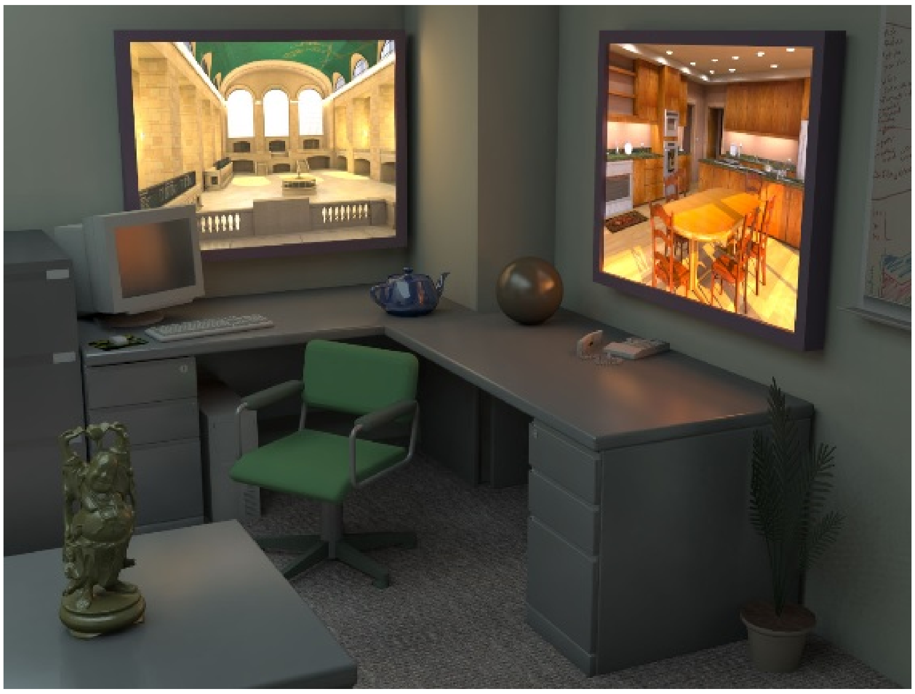


Figure 2.13: Effects achieved with lightcuts method.

2.1.7 Fluid Dynamics - explosion data generation

In this section I will describe the concept of computational fluid dynamics. I will also present methods which are used to simulate fluid-like substances in computer graphics.

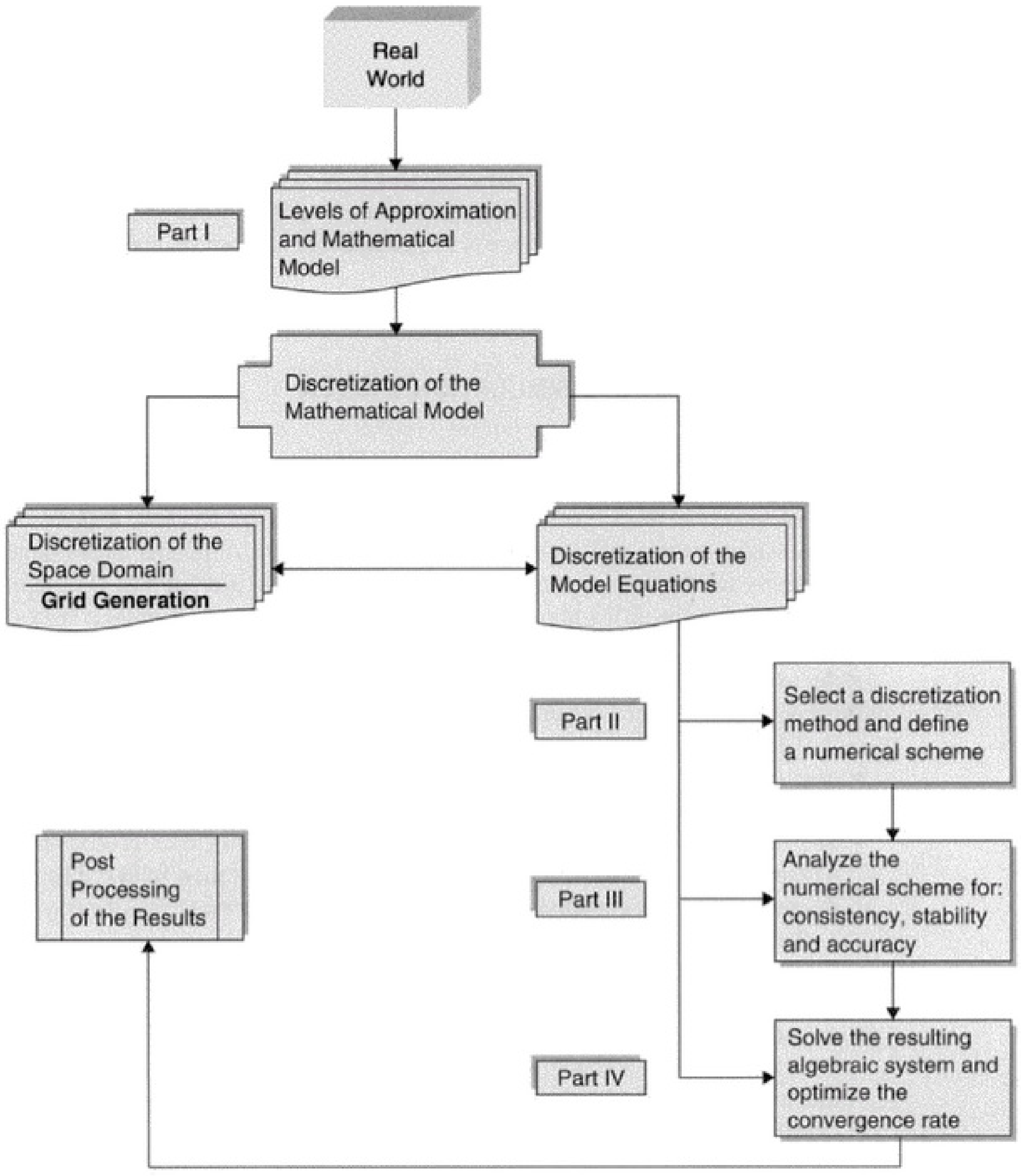

Real World
Part I
Levels of Approximation and Mathematical Model
Discretization of the Mathematical Model
Discretization of the Space Domain
Grid Generation
Discretization of the Model Equations
Part II
Select a discretization method and define a numerical scheme
Post Processing of the Results
Part III
Analyze the numerical scheme for: consistency, stability and accuracy
Part IV
Solve the resulting algebraic system and optimize the convergence rate

Figure 2.14: Steps in the process of development of the computational fluid dynamics system. [8]

2.1.7.1 Computational Fluid Dynamics

Computational Fluid Dynamics methods are used to numerically simulate fluid flows. Such system of fluid flows, when visualized can produce the movements of fluid, which look very similar to the real ones. The similarity level depends on the mathematical model used for approximation. Later those effects can be used in computer games, animations and movies [8].

2.1.7.2 Mathematical Model of the Fluid Flow

The final effect of the simulation will be very dependent on the level of approximation. What needs to be established is a type of physical forces to be simulated, which can be for example combustion, multiphase and multispecies flows which may have additional effects such as in chemical reactions or liquid drops. In order to achieve phenomena like that a model of physical laws has to be built. Because of the fact that theoretical models are used to simulate those phenomena, the produced effects will not be the same as in the real world.

2.1.7.3 Conservation Laws

The concept of conservation is fundamental for the computational fluid dynamics. Without a conservation, numerical CFD simulations would lose mass or energy and would be useless and not reliable. Therefore, it is important that after discretization of equations describing the phenomenon, properties like mass or energy will be conserved at the discrete level.
Conservation law states that for a particular quantity, variation of its amount inside a certain domain is equal to the variation of amount of this quantity leaving and entering the domain, acknowledging additional inputs from the sources generating this quantity. The motion of fluid flows is described by the conservation of three basic properties: mass, momentum and energy. Conservation of these properties completely determines the behaviour of the fluid evolution system without any additional dynamical law. The only additional information which can change the evolution of a system is the nature of the fluid which can be for example incompressible, condensable or viscoelastic. The set of equations which describe viscous fluids are called Navier-Stokes Equations.

2.1.7.4 Navier-Stokes Equations

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Figure 2.15: Navier-Stokes equations describing fluid flows: equation describing evolving velocity in a compact vector notation (top), equation describing density moving through the velocity field. [17]

Navier-Stokes equations provide a precise mathematical model which describes the motion of viscous fluids. Without the number of simplifications applied to them they are very hard to solve. Therefore, many approximative solutions have been developed over the years which focused either on physical precision or computation speed and quality of visual effects. Navier-Stokes equations can be used to model visual effects such as water flow, air flow, swirling smoke or to create simulations of physical phenomena needed in engineering or scientific fields.

2.1.7.5 Fluid Dynamics For Games - Mathematical Model

In a paper called Real-Time Fluid Dynamics for Games [17], Jos Stam presented set of algorithms which produce visual effects of fluid flows. Those algorithms provide approximative solution to physical equations of fluid flow called Navier-Stokes Equations.

Changes in Density Field

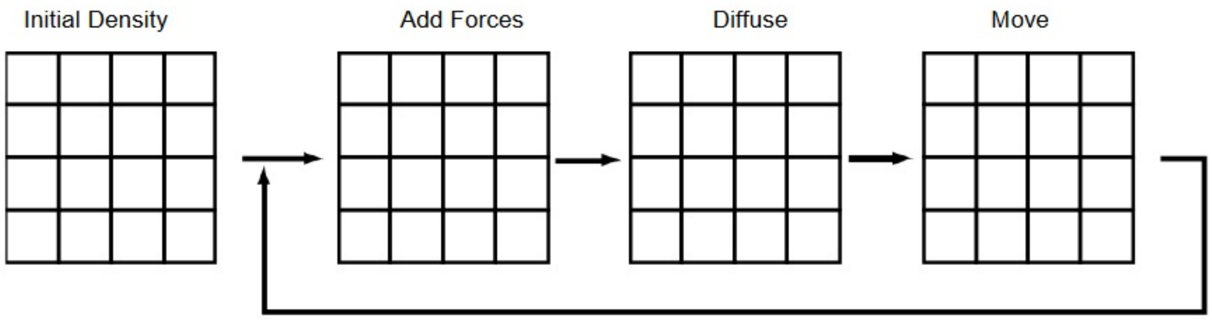


Figure 2.16: Steps in solving change in density field. [17]

Equation below describes the changes in a density field which depend on three terms. The first one $-(u \cdot \nabla)\rho$ says that density follows velocity field, the second, $\kappa\nabla^2\rho$ determines diffusion rate of density depending on the diffusion variable and S accounts for increase of density produced by some sources.

$$\frac{\partial\rho}{\partial t} = -(u \cdot \nabla)\rho + \kappa\nabla^2\rho + S \tag{2.6}$$

The first aspect to consider when determining changes in density field, represented as a 2D or 3D grid, is increase of density, which is rather trivial task to do. Another thing to consider is diffusion which at first does not seem complicated. Basic idea for diffusion is to, for the particular grid cell, exchange the density with neighbouring grid cells. However, diffusion rate depends on a given variable which when set for a large value produces negative results which defies the purpose of simulation. Therefore to make simulation stable, iterative solver called Gauss-Seidel relaxation is used. The last aspect to account for is density following velocity field. This is done by checking cell's center position over velocity field in previous time step and linearly interpolating density value. Tracing cell's center positions not forward, but back in time is done because tracing them forward would require solving linear system depending on velocity, which would complicate solving process significantly.

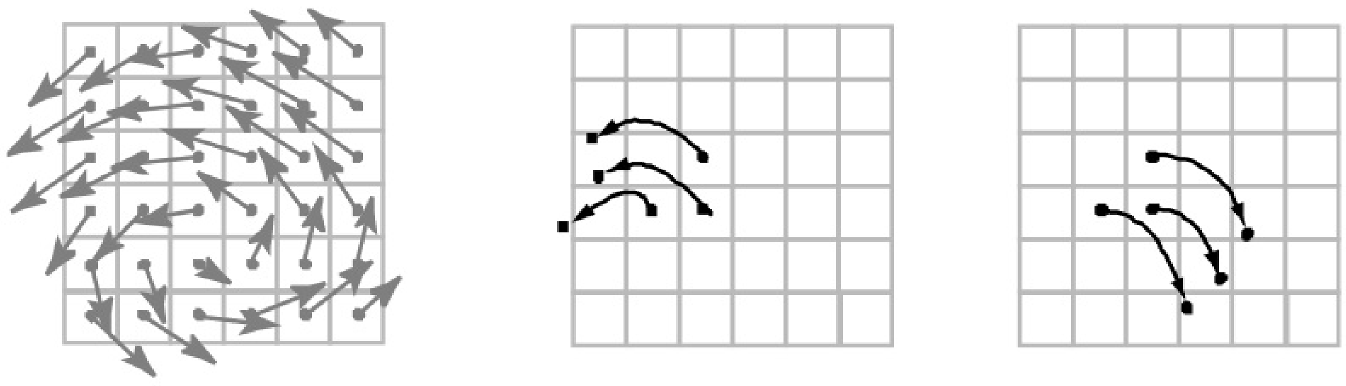


Figure 2.17: Grid in the middle shows moving cell's center forward in time according to the velocity field (left grid), which is a complicated solution. Grid on the right shows finding previous cell's center positions, which is a better solution. [17]

Changes in Velocity Field

The formula below describes how the velocity field changes over time. The change in velocity field depends on the three terms on the right hand side of the equation: self-advection, viscous diffusion and addition of forces.

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u + v\nabla^2 u + f \tag{2.7}$$

$-(u \cdot \nabla)u$ is a self-advection term which basically means that velocity moves along itself, $v\nabla^2 u$ accounts for diffusion of velocity depending on viscosity variable and f stands for addition of forces.

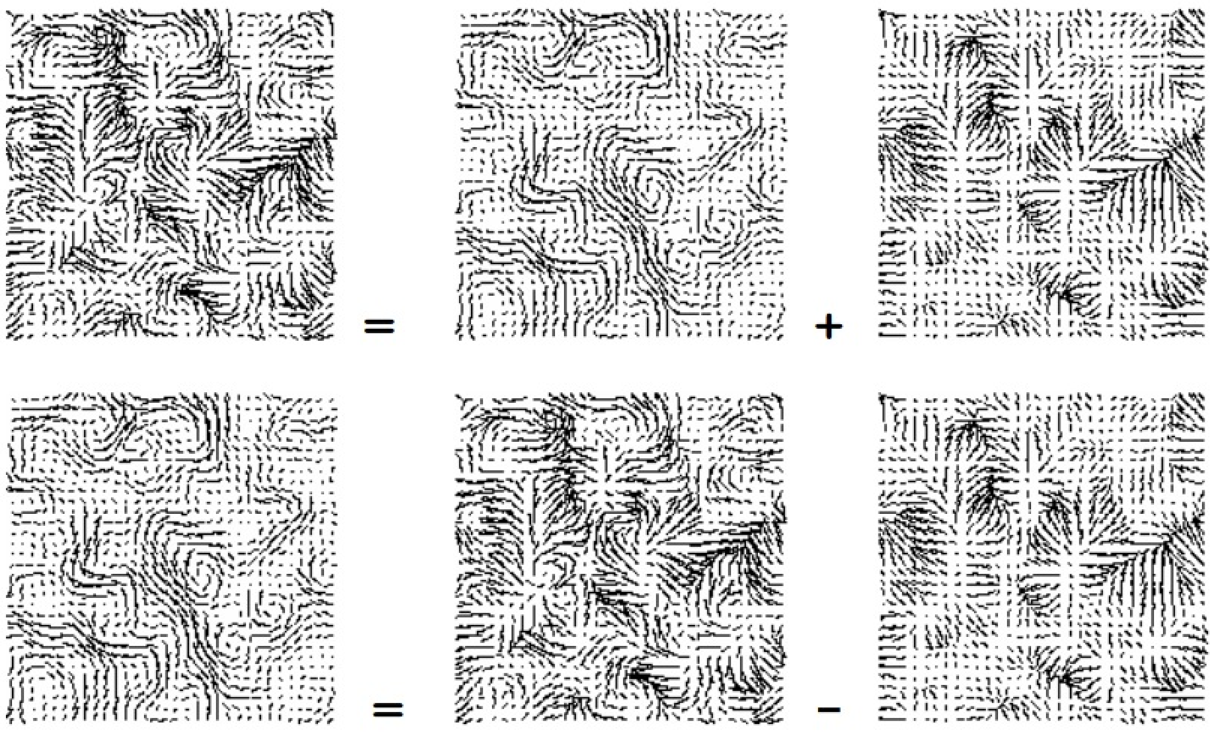


Figure 2.18: The Helmholtz-Hodge Decomposition states that the sum of a mass conserving field and gradient field results in a velocity field (top). Therefore, to get a mass conserving field, gradient field has to be subtracted from velocity field (bottom). [17]

In case of the velocity, addition of forces is as easy as it is in case of the diffusion. Much more complicated task is to fulfil the requirement of mass conservation. In this case of incompressible fluid, mass-conserving field with zero divergence needs to be obtained. As shown in figure above it can be achieved by using the Helmholtz-Hodge decomposition theorem which says that any vector field is a sum of mass conserving field and gradient field [21]. Gradient field can be computed by solving the linear system called Poisson equation. But since this system is sparse, Gauss-Seidel relaxation can be used again to solve it efficiently. Vector field can then be projected onto mass conserving field to meet the requirement of mass conservation for incompressible fluid.

Boundary of the Box

It is assumed that a fluid is contained in a solid box and no fluid can escape through the walls. This implies that for the vertical wall, the horizontal component of velocity on that wall should be zero and for the horizontal wall, the vertical component of velocity on that wall should be zero. In a method presented by Jos Stam, vector fields of given quantities are considered as continuous.

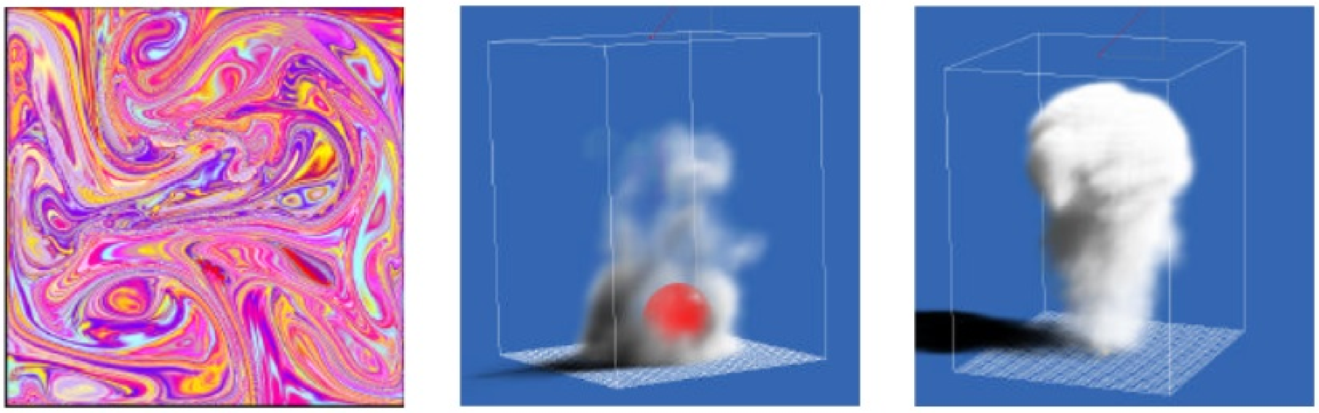


Figure 2.19: Effects generated with fluid dynamics solver by Jos Stam [17]

2.1.7.6 Particle-Based Fluid Simulation for Interactive Applications - Mathematical Model

In 2003 Matthias Müller, David Charypar and Markus Gross presented the particle-based method for fluid simulation [14]. This approach to computational fluid dynamics assumes generation of many particles and tracing their position and velocity. Each generated particle has mass, density and influences nearby area.
Proposed method is based on Smoothed Particle Hydrodynamics [13] which is an interpolation method used in particle systems. Smoothed Particle Hydrodynamics requires field quantities to be defined only at discrete particle locations, so they can be evalueted anywhere in space. So called, symmetrical smoothing kernels are used to distribute given quantities in a nearby are of each particle. Interpolation of scalar quantity at given location is done by:

$$A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h) \tag{2.8}$$

where j iterates over all particles, m_j is the mass of particle j, r_j is the position of particle j, ρ_j is the density of particle j and A_j is the field quantity at the position r_j. $W(r, h)$ is a smoothing kernel function where h is a radius of kernel's support.

Mass of the particle is constant at all times, but the density changes and therefore has to be calculated at every time step. Density $\rho_S(r)$ at location r is calculated by using the following formula:

$$\rho_S(r) = \sum_j m_j W(r - r_j, h) \tag{2.9}$$

Gradient of scalar quantity A is calculated by:

$$\nabla A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(r - r_j, h) \tag{2.10}$$

Laplacian of scalar quantity A is calculated by:

$$\nabla^2 A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(r - r_j, h) \tag{2.11}$$

Every particle has constant mass, so mass conservation is guaranteed. Another aspect to look at is momentum conservation described by Navier-Stokes equation:

$$\rho(\frac{\partial v}{\partial t} + v \cdot \nabla v) = -\nabla p + \rho g + \mu \nabla^2 v \tag{2.12}$$

in which from $(\frac{\partial v}{\partial t} + v \cdot \nabla v)$ only the substantial derivative $\frac{D_v}{D_t}$ is needed because the derivative of the velocity field is the derivative of the particle velocity and the convective term $v \cdot \nabla v$ is not needed for particle systems.

The next thing to consider is pressure which is not symmetric when Smoothed Particle Hydrodynamics is applied to the pressure term from Navier-Stokes equation. Assuming there are two particles a and b and when pressure is calculated for a, it is calculated based on the pressure of b, because for the particle a the gradient of the kernel is zero at its center. This produces non-symmetric pressure forces when two particles which have different pressures at their locations interact with each other. Therefore, to make them symmetrical, instead of:

$$f_i^{pressure} = -\nabla p(r_i) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(r_i - r_j, h) \tag{2.13}$$

the simple solution is to use the arithmetic mean of the pressures of interacting particles:

$$f_i^{pressure} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(r_i - r_j, h) \tag{2.14}$$

Viscosity term, similarly to pressure term, produces non-symmetric forces. It happens because velocity field differs at every particle, but this problem can be fixed by using velocity differences instead of absolute velocities:

$$f_i^{viscosity} = \mu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W(r_i - r_j, h) \tag{2.15}$$

Proposed fluid simulation method also accounts for the phenomenon of surface tension which describes how particles attract each other. Additional external forces such as gravity, collision and other caused by user interaction are included in the simulation by applying them straight to particles without the use of smoothed particle hydrodynamics. Smoothing kernel functions used in the simulation, influence stability, speed and accuracy of the smoothed particle hydrodynamics method. Therefore, it is important to design smoothing kernels depending on the intent of use, which can be for example computations of distance, pressure forces or impact of viscosity on the velocity field.

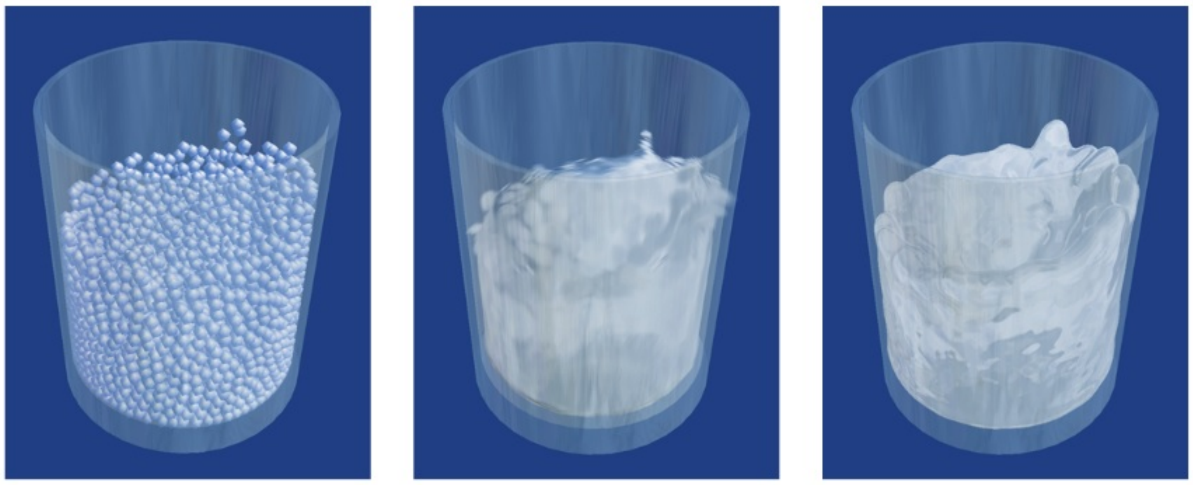


Figure 2.20: Effects generated with method based on smoothed particle hydrodynamics [14]

2.2 Choice of methods

In this section I will write about methods I have chose to generate self-illuminating explosions.

2.2.1 Physics Based Model of Explosion

As described in previous sections, methods used to simulate fluids can be based on tracking of particles or discretization of space into a grid and calculations based on voxels of the grid. Particle based simulations tend to be less accurate but maintain relatively quicker time of computations [6]. Also, managing the fluid as a fixed points in a grid is easier than managing the bunch of particles which create unstructured shape. For the generation of data needed to model the explosion, I have chose to use the method proposed by Jos Stam [17] and its interpretation by Mike Ash [3] sacrificing computations speed but achieving better numerical accuracy.

2.2.2 Illumination of Model

The illumination of explosion model is done with algorithm which casts rays from the light source cell to other grid cells. Only grid cells with density greater than zero are considered in color computations. The reason for that is, the density of a grid cell is used as an alpha component of the color to be assigned to this cell and if the density is equal to zero the cell is not visible. Color of the grid cell to be set depends on the density cumulated from grid cells which lay along the ray shot from the light source to the destination cell. Accumulated density is then used in a transfer function to pick the color.

Chapter 3

Software Requirements and System Design

In this chapter I will describe software requirements related to the project. This will include main steps needed for the project to work and produce the desired result. I will also discuss the design of different parts of the project.

3.1 Software Requirements

The first thing required is generation of a grid inside of which the explosion will happen. It will require data structures needed to store all of the informations about the grid. This includes positions of vertices and indices of vertices. Indices are needed to divide the grid into voxels and avoid data redundancy by using the same vertices for different voxels.

Another thing required are data structures needed to hold the data generated by fluid simulation. They need to be constructed in a way the data is fast and easily accessible in order to be read or changed. The next thing required is implementation of physically based fluid simulation. This will require the set of functions which will approximately solve the Navier-Stokes equations. Fluid simulation will generate the data needed to simulate the explosion. Generated data has to then be visualised by rendering the explosion on the screen. To do that a window used for rendering has to be created. For the process of rendering, vertex shader as well as fragment shader will be needed. Vertex shader and fragment shader data has to be then organized and loaded into buffers.

Another aspect that needs to be taken care of is self-illumination of the explosion. Source of the light has to be created and located inside of the explosion volume. It has to make an impact on the appearance of the explosion by interacting with the data in a grid, generated by the fluid simulation. The next thing needed is an algorithm describing the interaction of light source and explosion data. Results of that interaction has to be processed, copied to a buffer and sent to the fragment shader.

3.2 System Design

This section describes how certain parts of the system were designed. Each subsection includes short description of the tool needed to make the system work as expected.

3.2.1 C++

The project uses C++ as the main programming language. C++ is a general-purpose programming language created by Bjarne Stroustrup. It was released in 1985 as an extension to the C language and since then many features were added in updates released over the years. C++, unlike other programming languages like Java or C, does not have a garbage collector which automates the process of memory management. C++ gives the control over memory to programmers who can manage memory allocation and deallocation. C++ was designed to be efficient and flexible, which was found useful in computer system software, embedded systems and systems with resource constraints like video games, desktop applications, servers[19][18].

3.2.2 OpenGL

OpenGL (Open Graphics Library) is a language independent, platform independent application programming interface. It is used for rendering 2D or 3D scenes, usually utilizing possibilities given by graphics processing unit (GPU) [16]. OpenGL allows to produce high quality graphical images by specifying objects, properties and operations which influence them. The set of commands the OpenGL offers, gives many possibilities in the context of computer graphics. OpenGL can be seen as a pipeline which controls the drawing operations, some of which are programmable.

3.2.3 GLSL

GLSL (OpenGL Shading Language) is a high-level shading language used to write functions called vertex shader and fragment shader. Vertex shader specifies operations modifying attributes of vertices that happen to every vertex during processing. Fragment shader is responsible for determining what color values are assigned to a fragment produced in the rasterization stage. Vertex shader and fragment shader have to be compiled and attached to a shader program which is then linked and can be used to process data.

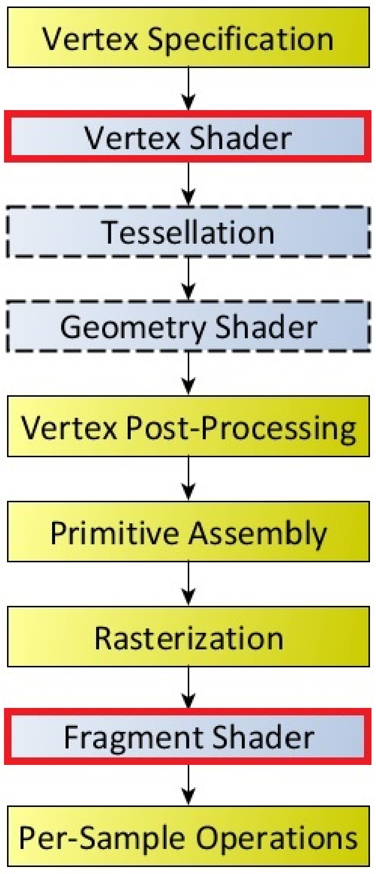


Figure 3.1: Vertex Shader and Fragment Shader in the OpenGL rendering pipeline [16]

3.2.4 GLM

GLM (OpenGL Mathematics) is a C++ mathematics library used in graphics software. This written in C++98, header only library provides very useful mathematical operations which are very often needed when creating graphics software. GLM is used in this project mainly for creating matrices needed for MVP (Model, View, Projection) transformations.

3.2.5 QT

Qt is a cross platform widget toolkit used to create software applications and graphical user interfaces (GUI) for many desktop, mobile and embedded platforms. Besides GUI programs, command line applications or applications requiring consoles only like servers can also be created with Qt. In case of this project Qt is used to create GUI and utilize functionalities of Qt OpenGL module [15].

Chapter 4

Software Implementation

In this chapter I will show how fluid simulation and illumination parts of the project were implemented. I will show code snippets of main parts of the program and describe what they do.

4.1 Model of Explosion

4.1.1 Navier-Stokes Equations

Solution of Navier-Stokes equations needed to simulate the fluid flow is presented as a set of functions. Each of those functions takes care of different part of the Navier-Stokes equations, but when run in a proper order during one time step, they produce the desired effect. The functions involved in approximative solution of NS equations are: $addDensity()$, $addVelocity()$, $setBoundary()$, $linearSolve()$, $project()$, $advect()$, $diffuse()$ and $fluidTimeStep()$.

4.1.1.1 Simulation Grid

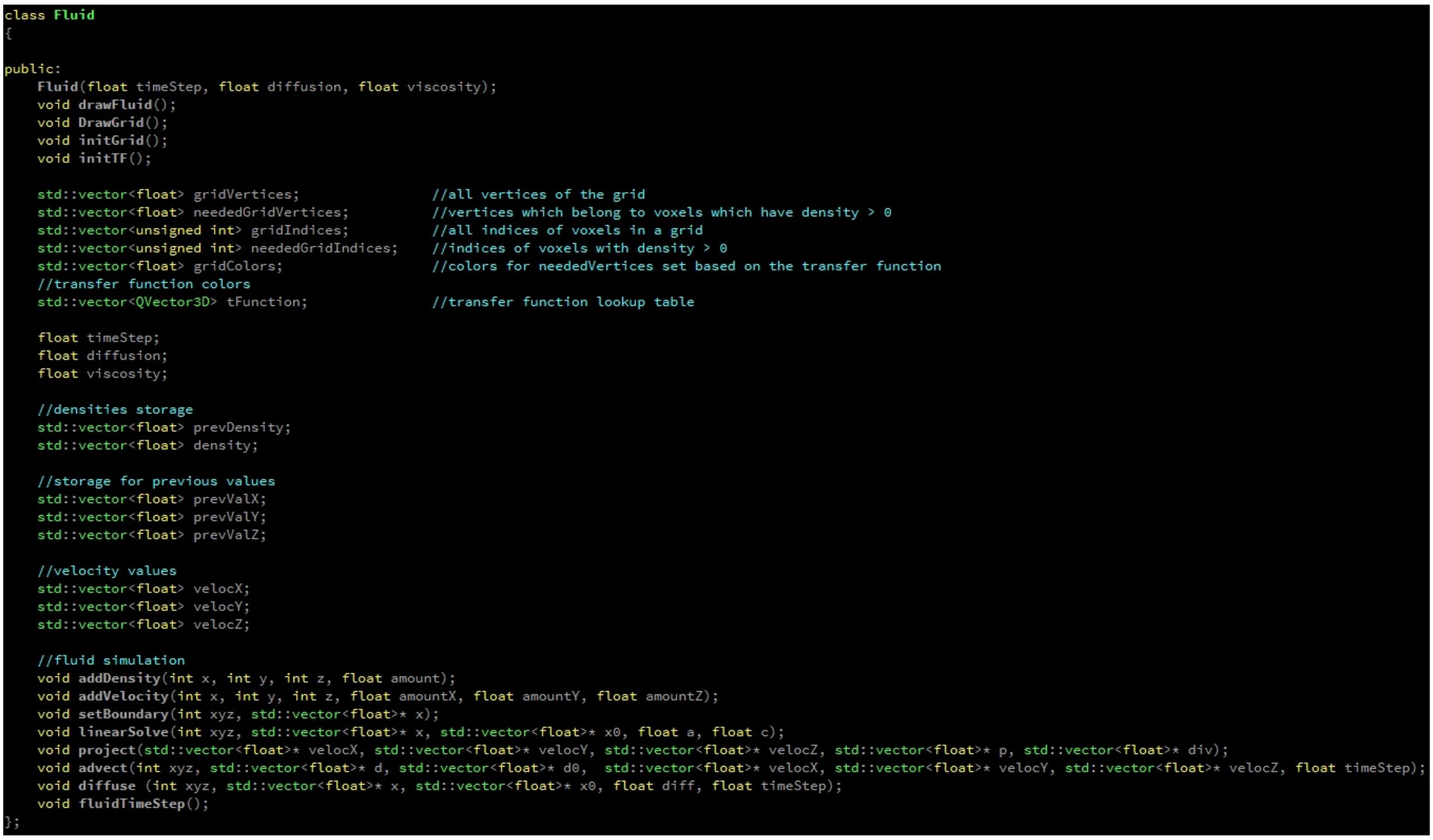
The simulation in this project is done in a 3 dimensional grid of size N and it is stored in a one dimensional array in order to save memory. Accessing the index of a particular voxel in a 3D grid represented as one dimensional array is done with simple index function:

$$index = x + y * N + z * N * N \tag{4.1}$$

where x, y and z represent coordinates of a voxel in a 3d grid and N is the size of the grid.

4.1.1.2 Fluid

Fluid is represented as a class with functions needed for simulation. Fluid's constructor takes input values for the length of the time step, diffusion and viscosity. Diffusion specifies how fast given quantity spreads in the fluid and viscosity specifies the thickness of the fluid. Fluid class also holds informations about grid, density field, velocity field or transfer function colors.



```
class Fluid
{

public:
    Fluid(float timeStep, float diffusion, float viscosity);
    void drawFluid();
    void DrawGrid();
    void initGrid();
    void initTF();

    std::vector<float> gridVertices;                //all vertices of the grid
    std::vector<float> neededGridVertices;          //vertices which belong to voxels which have density > 0
    std::vector<unsigned int> gridIndices;          //all indices of voxels in a grid
    std::vector<unsigned int> neededGridIndices;    //indices of voxels with density > 0
    std::vector<float> gridColors;                  //colors for neededVertices set based on the transfer function
    //transfer function colors
    std::vector<QVector3D> tFunction;               //transfer function lookup table

    float timeStep;
    float diffusion;
    float viscosity;

    //densities storage
    std::vector<float> prevDensity;
    std::vector<float> density;

    //storage for previous values
    std::vector<float> prevValX;
    std::vector<float> prevValY;
    std::vector<float> prevValZ;

    //velocity values
    std::vector<float> velocX;
    std::vector<float> velocY;
    std::vector<float> velocZ;

    //fluid simulation
    void addDensity(int x, int y, int z, float amount);
    void addVelocity(int x, int y, int z, float amountX, float amountY, float amountZ);
    void setBoundary(int xyz, std::vector<float>* x);
    void linearSolve(int xyz, std::vector<float>* x, std::vector<float>* x0, float a, float c);
    void project(std::vector<float>* velocX, std::vector<float>* velocY, std::vector<float>* velocZ, std::vector<float>* p, std::vector<float>* div);
    void advect(int xyz, std::vector<float>* d, std::vector<float>* d0,  std::vector<float>* velocX, std::vector<float>* velocY, std::vector<float>* velocZ, float timeStep);
    void diffuse (int xyz, std::vector<float>* x, std::vector<float>* x0, float diff, float timeStep);
    void fluidTimeStep();
};
```

Figure 4.1: Class of a fluid containing fluid properties and functions needed for explosion data generation.

4.1.1.3 Velocity and Density Fields

3D velocity field is represented as three one dimensional arrays, one for each dimension and values of density field are stored in one dimensional array. Moreover, there are additional arrays for storing old values of velocities and densities which are needed for calculations.

4.1.1.4 External Forces

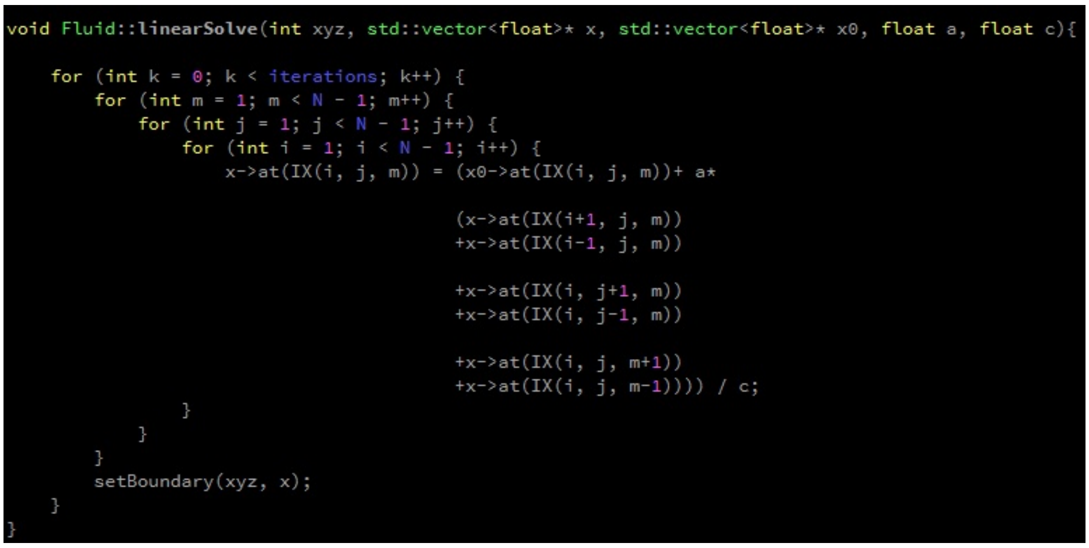
To account for the other changes to the density and velocity fields, than these caused by diffusion or advection, two functions are used. Sources may change the values of density and velocity which is done by using *addDensity*() and *addVelocity*() functions which take the values and coordinates of the voxel which values are to be changed.

4.1.1.5 Diffusion

Diffuse() function is used in both cases: diffusion of density and diffusion of velocities of the fluid. Velocity and density of a voxel diffuse depending on the values of the neighbouring voxels. In case of a 3d grid each voxel has 6 neighbours (actual simulation grid is one voxel bigger on each side, so every voxel can have the same number of neighbours which makes calculations less complicated) which it trades its values with. This function calculates the diffusion rate variable and passes the data to another function called *linearSolver*() for further calculations.

4.1.1.6 Linear Solver

linearSolve() function is a Gauss-Seidel relaxation - iterative method used for solving linear systems. It takes an array of values, either densities or velocities, and calculates the transfer of given quantity between a voxel and its 6 neighbours. The accuracy of the value produced by the linear solver depends on the number of iterations it does. For bigger grids, increasing the number of iterations will significantly slow down the computations. Also, after each iteration of the linear solver, the function *setBoundary*() is called to prevent the fluid going through the boundary of the grid.

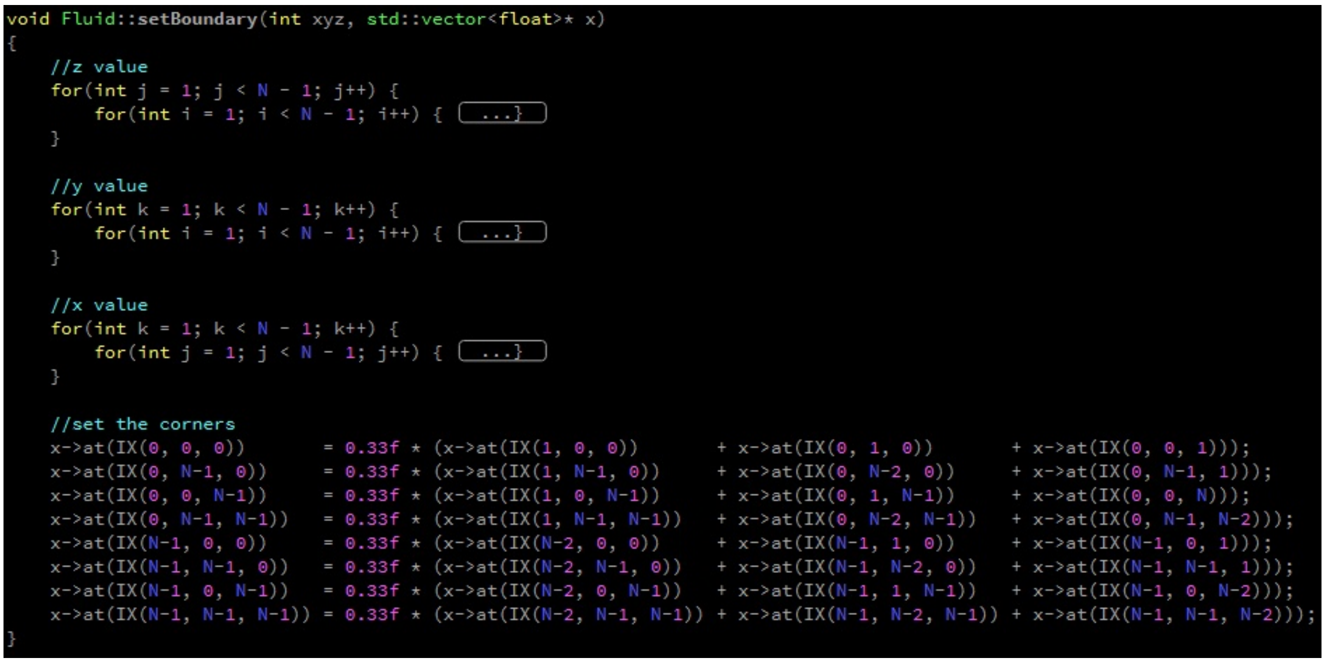


```
void Fluid::linearSolve(int xyz, std::vector<float>* x, std::vector<float>* x0, float a, float c){

    for (int k = 0; k < iterations; k++) {
        for (int m = 1; m < N - 1; m++) {
            for (int j = 1; j < N - 1; j++) {
                for (int i = 1; i < N - 1; i++) {
                    x->at(IX(i, j, m)) = (x0->at(IX(i, j, m))+ a*

                                         (x->at(IX(i+1, j, m))
                                         +x->at(IX(i-1, j, m))

                                         +x->at(IX(i, j+1, m))
                                         +x->at(IX(i, j-1, m))

                                         +x->at(IX(i, j, m+1))
                                         +x->at(IX(i, j, m-1)))) / c;
                }
            }
        }
        setBoundary(xyz, x);
    }
}
```

Figure 4.2: Function for solving linear systems

4.1.1.7 Boundary of Fluid

As mentioned before *setBoundary*() function makes the fluid stay inside the box and it applies to both velocity and density. It is done by passing velocity field or density field data which is to be kept inside of the box and the value of *xyz* parameter, which determines what kind of data is passed to the function. Based on the *xyz* parameter, boundary cells of velocity field or density field are set to be mirrors of their neighbours. It means that they perfectly counter the values their neighbours have which results in a fluid remaining inside of the box. In case of the corner cells value is set to the average of the neighbouring cells values.

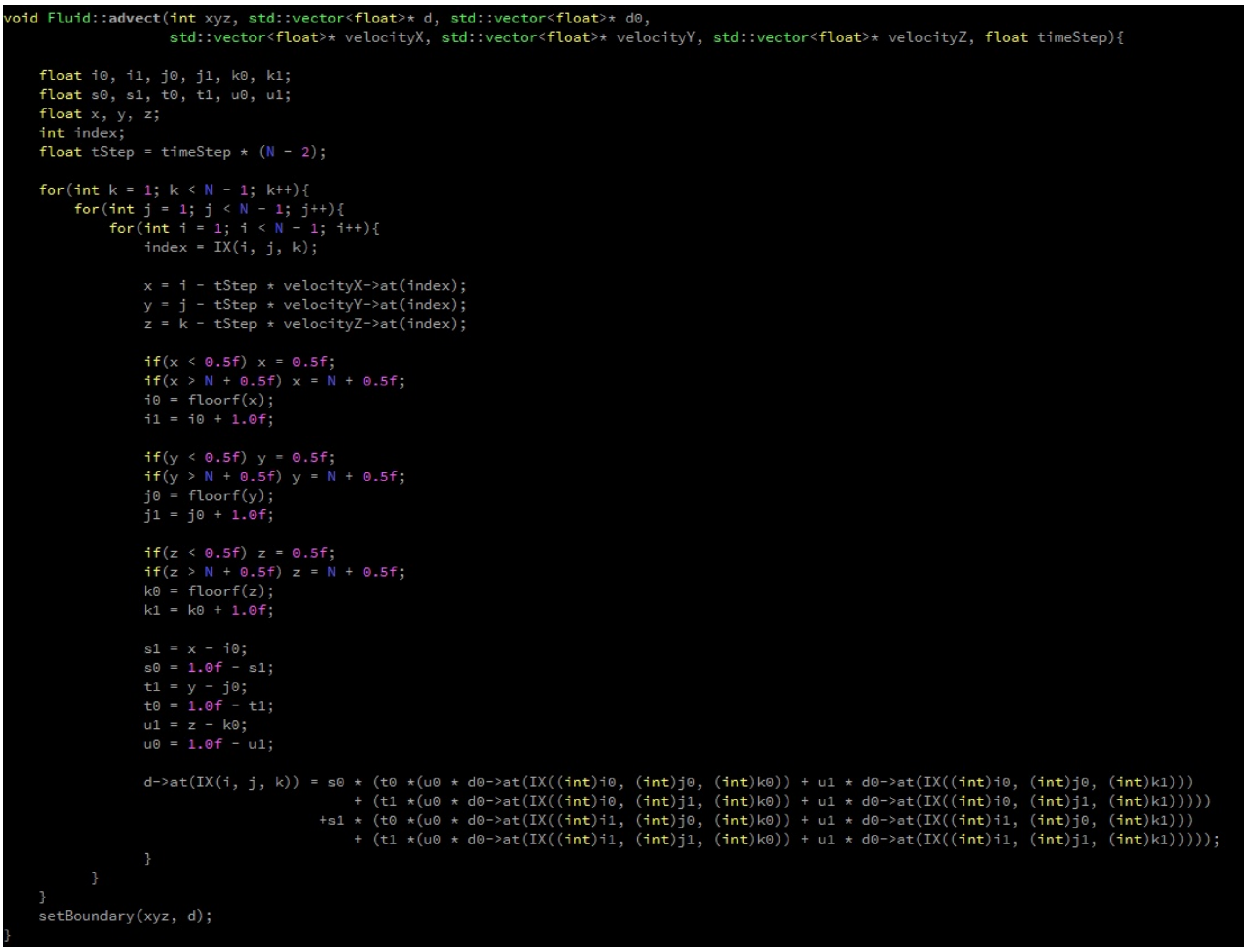


```
void Fluid::setBoundary(int xyz, std::vector<float>* x)
{
    //z value
    for(int j = 1; j < N - 1; j++) {
        for(int i = 1; i < N - 1; i++) { ...}
    }

    //y value
    for(int k = 1; k < N - 1; k++) {
        for(int i = 1; i < N - 1; i++) { ...}
    }

    //x value
    for(int k = 1; k < N - 1; k++) {
        for(int j = 1; j < N - 1; j++) { ...}
    }

    //set the corners
    x->at(IX(0, 0, 0))       = 0.33f * (x->at(IX(1, 0, 0))       + x->at(IX(0, 1, 0))       + x->at(IX(0, 0, 1)));
    x->at(IX(0, N-1, 0))     = 0.33f * (x->at(IX(1, N-1, 0))     + x->at(IX(0, N-2, 0))     + x->at(IX(0, N-1, 1)));
    x->at(IX(0, 0, N-1))     = 0.33f * (x->at(IX(1, 0, N-1))     + x->at(IX(0, 1, N-1))     + x->at(IX(0, 0, N)));
    x->at(IX(0, N-1, N-1))   = 0.33f * (x->at(IX(1, N-1, N-1))   + x->at(IX(0, N-2, N-1))   + x->at(IX(0, N-1, N-2)));
    x->at(IX(N-1, 0, 0))     = 0.33f * (x->at(IX(N-2, 0, 0))     + x->at(IX(N-1, 1, 0))     + x->at(IX(N-1, 0, 1)));
    x->at(IX(N-1, N-1, 0))   = 0.33f * (x->at(IX(N-2, N-1, 0))   + x->at(IX(N-1, N-2, 0))   + x->at(IX(N-1, N-1, 1)));
    x->at(IX(N-1, 0, N-1))   = 0.33f * (x->at(IX(N-2, 0, N-1))   + x->at(IX(N-1, 1, N-1))   + x->at(IX(N-1, 0, N-2)));
    x->at(IX(N-1, N-1, N-1)) = 0.33f * (x->at(IX(N-2, N-1, N-1)) + x->at(IX(N-1, N-2, N-1)) + x->at(IX(N-1, N-1, N-2)));
}
```

Figure 4.3: *setBoundary*() function prevents fluid spilling out of the box

4.1.1.8 Advection

advect() function is responsible for moving density along the velocity field and moving velocities along themselves. It is done by assuming that density is presented as particles, which are traced back from grid cell centers over velocity field to their old positions. Therefore it is known from where particles came after one time step. The values they carry with themselves can be then interpolated, based on the values of their neighbours.

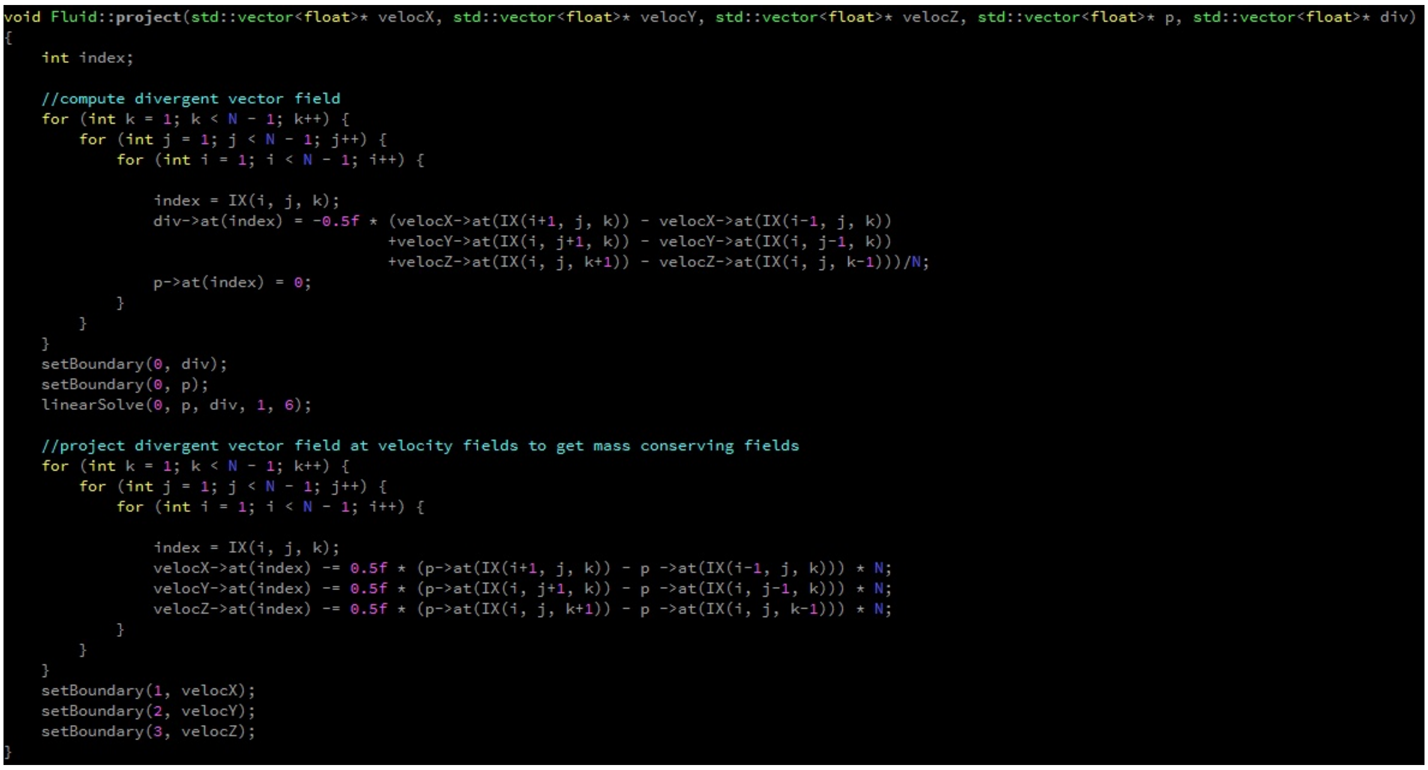


```
void Fluid::advect(int xyz, std::vector<float>* d, std::vector<float>* d0,
                   std::vector<float>* velocityX, std::vector<float>* velocityY, std::vector<float>* velocityZ, float timeStep){

    float i0, i1, j0, j1, k0, k1;
    float s0, s1, t0, t1, u0, u1;
    float x, y, z;
    int index;
    float tStep = timeStep * (N - 2);

    for(int k = 1; k < N - 1; k++){
        for(int j = 1; j < N - 1; j++){
            for(int i = 1; i < N - 1; i++){
                index = IX(i, j, k);

                x = i - tStep * velocityX->at(index);
                y = j - tStep * velocityY->at(index);
                z = k - tStep * velocityZ->at(index);

                if(x < 0.5f) x = 0.5f;
                if(x > N + 0.5f) x = N + 0.5f;
                i0 = floorf(x);
                i1 = i0 + 1.0f;

                if(y < 0.5f) y = 0.5f;
                if(y > N + 0.5f) y = N + 0.5f;
                j0 = floorf(y);
                j1 = j0 + 1.0f;

                if(z < 0.5f) z = 0.5f;
                if(z > N + 0.5f) z = N + 0.5f;
                k0 = floorf(z);
                k1 = k0 + 1.0f;

                s1 = x - i0;
                s0 = 1.0f - s1;
                t1 = y - j0;
                t0 = 1.0f - t1;
                u1 = z - k0;
                u0 = 1.0f - u1;

                d->at(IX(i, j, k)) = s0 * (t0 *(u0 * d0->at(IX((int)i0, (int)j0, (int)k0)) + u1 * d0->at(IX((int)i0, (int)j0, (int)k1)))
                                        + (t1 *(u0 * d0->at(IX((int)i0, (int)j1, (int)k0)) + u1 * d0->at(IX((int)i0, (int)j1, (int)k1)))))
                                    +s1 * (t0 *(u0 * d0->at(IX((int)i1, (int)j0, (int)k0)) + u1 * d0->at(IX((int)i1, (int)j0, (int)k1)))
                                        + (t1 *(u0 * d0->at(IX((int)i1, (int)j1, (int)k0)) + u1 * d0->at(IX((int)i1, (int)j1, (int)k1)))));
                }
            }
    }
    setBoundary(xyz, d);
}
```

Figure 4.4: *advect*() function moves the velocities along themselves and moves densities along velocities

4.1.1.9 Project function

project() function deals with requirement of mass conservation for incompressible fluid. It means that the amount of the fluid inflow into a cell has to be equal to the outflow. Since this simulation deals with incompressible fluid, the vector fields with no divergence are needed, but *diffuse*() and *advect*() functions make the vector fields divergent. The problem is solved by applying Helmholtz-Hodge decomposition and reusing Gauss-Seidel relaxation solver. First, the divergent vector field is calculated and then it is projected at velocity field to make it mass conserving.



```cpp
void Fluid::project(std::vector<float>* velocX, std::vector<float>* velocY, std::vector<float>* velocZ, std::vector<float>* p, std::vector<float>* div)
{
    int index;

    //compute divergent vector field
    for (int k = 1; k < N - 1; k++) {
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {

                index = IX(i, j, k);
                div->at(index) = -0.5f * (velocX->at(IX(i+1, j, k)) - velocX->at(IX(i-1, j, k))
                                        +velocY->at(IX(i, j+1, k)) - velocY->at(IX(i, j-1, k))
                                        +velocZ->at(IX(i, j, k+1)) - velocZ->at(IX(i, j, k-1)))/N;
                p->at(index) = 0;
            }
        }
    }
    setBoundary(0, div);
    setBoundary(0, p);
    linearSolve(0, p, div, 1, 6);

    //project divergent vector field at velocity fields to get mass conserving fields
    for (int k = 1; k < N - 1; k++) {
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {

                index = IX(i, j, k);
                velocX->at(index) -= 0.5f * (p->at(IX(i+1, j, k)) - p ->at(IX(i-1, j, k))) * N;
                velocY->at(index) -= 0.5f * (p->at(IX(i, j+1, k)) - p ->at(IX(i, j-1, k))) * N;
                velocZ->at(index) -= 0.5f * (p->at(IX(i, j, k+1)) - p ->at(IX(i, j, k-1))) * N;
            }
        }
    }
    setBoundary(1, velocX);
    setBoundary(2, velocY);
    setBoundary(3, velocZ);
}
```

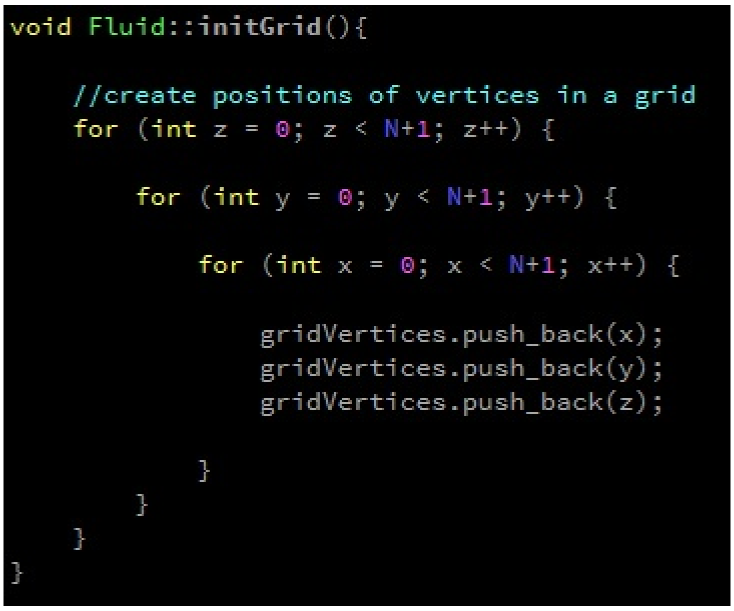
Figure 4.5: *project*() function is responsible for making velocity vector field mass conserving

4.2 Illumination of Explosion

In this section I will describe how the problem of self-illumination was solved in this project. I will describe the steps in colour computations and present the code snippets.

4.3 Determining which vertices are needed

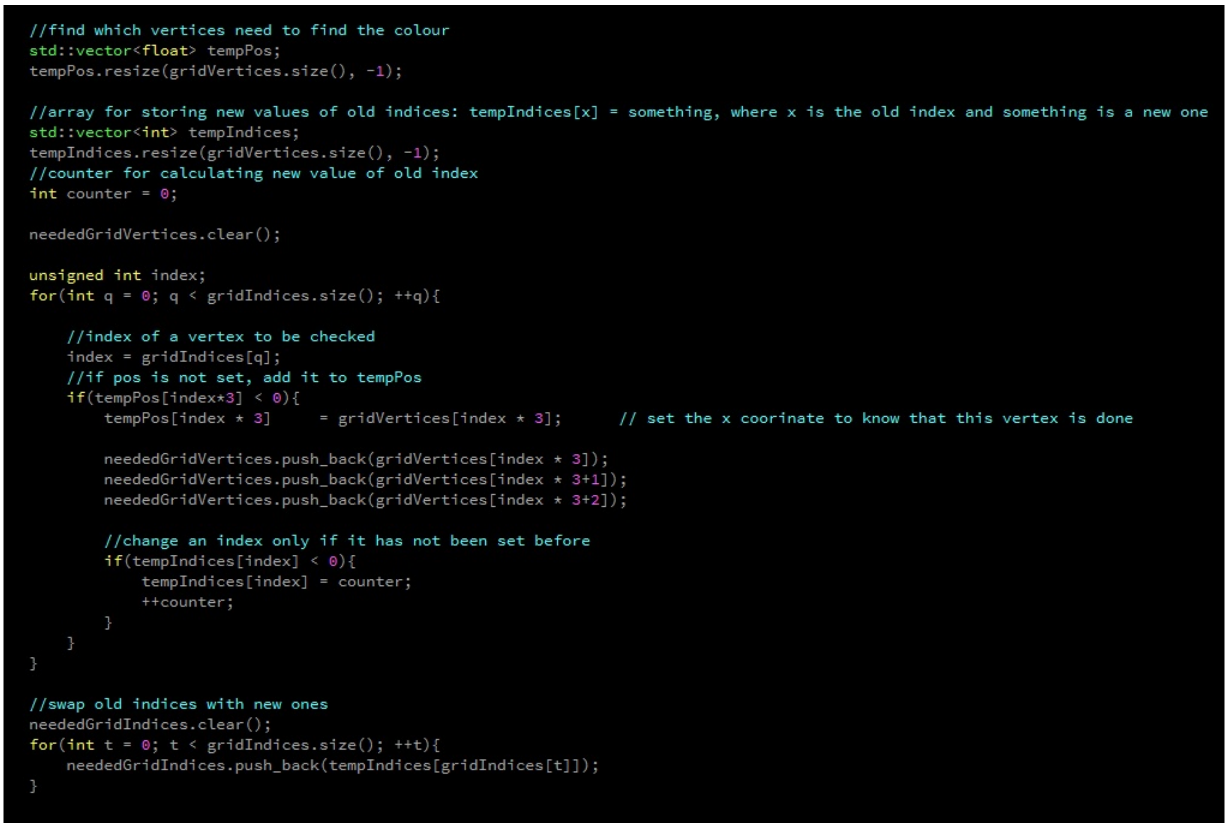
Except the simulation grid of size N, the program also creates the grid of positions of vertices of size $N + 1$. Those vertices are used to create quads and cubes to be displayed on the screen.



```
void Fluid::initGrid(){

    //create positions of vertices in a grid
    for (int z = 0; z < N+1; z++) {

        for (int y = 0; y < N+1; y++) {

            for (int x = 0; x < N+1; x++) {

                gridVertices.push_back(x);
                gridVertices.push_back(y);
                gridVertices.push_back(z);

            }
        }
    }
}
```

Figure 4.6: Grid of the vertices is created once at the beginning of the program.

Initially the cubes were build with vertices which were not indexed, so the number of vertices sent to the vertex shader was much bigger than it should be. For example, cube number 2 was created with four vertices which were used in cube number 1, so they were already in an array ready to be copied to a buffer. In order to remove data redundancy problem and improve efficiency, indexing of vertices was needed, so the cubes could be built with indices of vertices instead of vertices themselves.

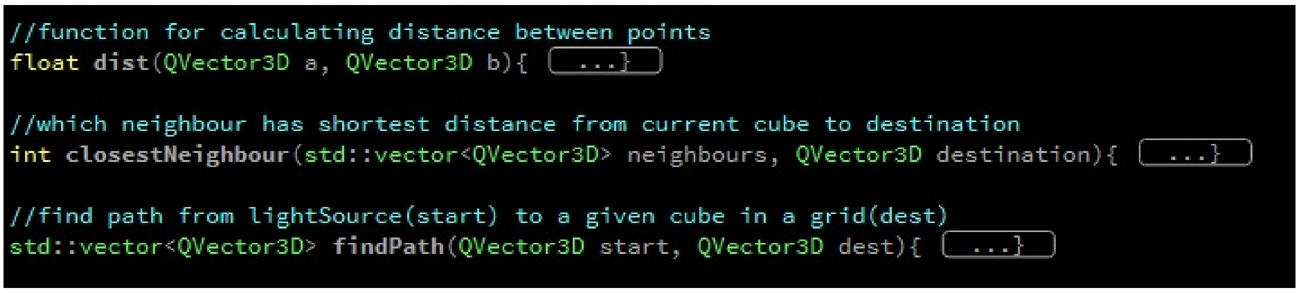


```
//find which vertices need to find the colour
std::vector<float> tempPos;
tempPos.resize(gridVertices.size(), -1);

//array for storing new values of old indices: tempIndices[x] = something, where x is the old index and something is a new one
std::vector<int> tempIndices;
tempIndices.resize(gridVertices.size(), -1);
//counter for calculating new value of old index
int counter = 0;

neededGridVertices.clear();

unsigned int index;
for(int q = 0; q < gridIndices.size(); ++q){

    //index of a vertex to be checked
    index = gridIndices[q];
    //if pos is not set, add it to tempPos
    if(tempPos[index*3] < 0){
        tempPos[index * 3]     = gridVertices[index * 3];      // set the x coorinate to know that this vertex is done

        neededGridVertices.push_back(gridVertices[index * 3]);
        neededGridVertices.push_back(gridVertices[index * 3+1]);
        neededGridVertices.push_back(gridVertices[index * 3+2]);

        //change an index only if it has not been set before
        if(tempIndices[index] < 0){
            tempIndices[index] = counter;
            ++counter;
        }
    }
}

//swap old indices with new ones
neededGridIndices.clear();
for(int t = 0; t < gridIndices.size(); ++t){
    neededGridIndices.push_back(tempIndices[gridIndices[t]]);
}
```

Figure 4.7: Code responsible for determining which vertices need to be send to the vertex shader. It also creates new indices of vertices because the number of vertices will be smaller and therefore indices will have to be changed.

4.3.1 Casting ray from the light source

After determining which vertices are needed and creating quads and cubes from them, the colour has to be computed for each vertex. Color computation starts from shooting the ray from light source to the destination point. The ray which is cast accumulates densities on its path, until it reaches the destination, which is one of the needed vertices for which the color is to be set. After that, the color from the transfer function is fetched, depending on the amount of density accumulated on the path.

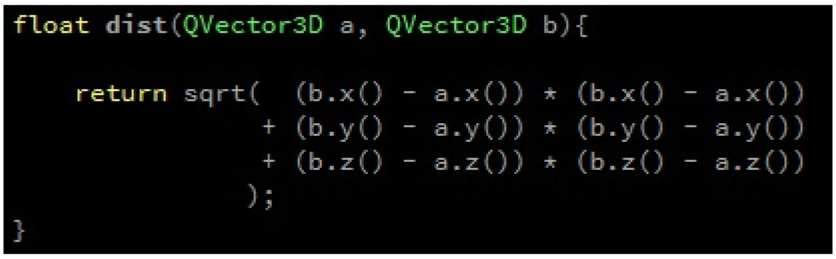


```
//function for calculating distance between points
float dist(QVector3D a, QVector3D b){ ...}

//which neighbour has shortest distance from current cube to destination
int closestNeighbour(std::vector<QVector3D> neighbours, QVector3D destination){ ...}

//find path from lightSource(start) to a given cube in a grid(dest)
std::vector<QVector3D> findPath(QVector3D start, QVector3D dest){ ...}
```

Figure 4.8: Functions needed for casting the ray.

4.3.1.1 Finding a Path

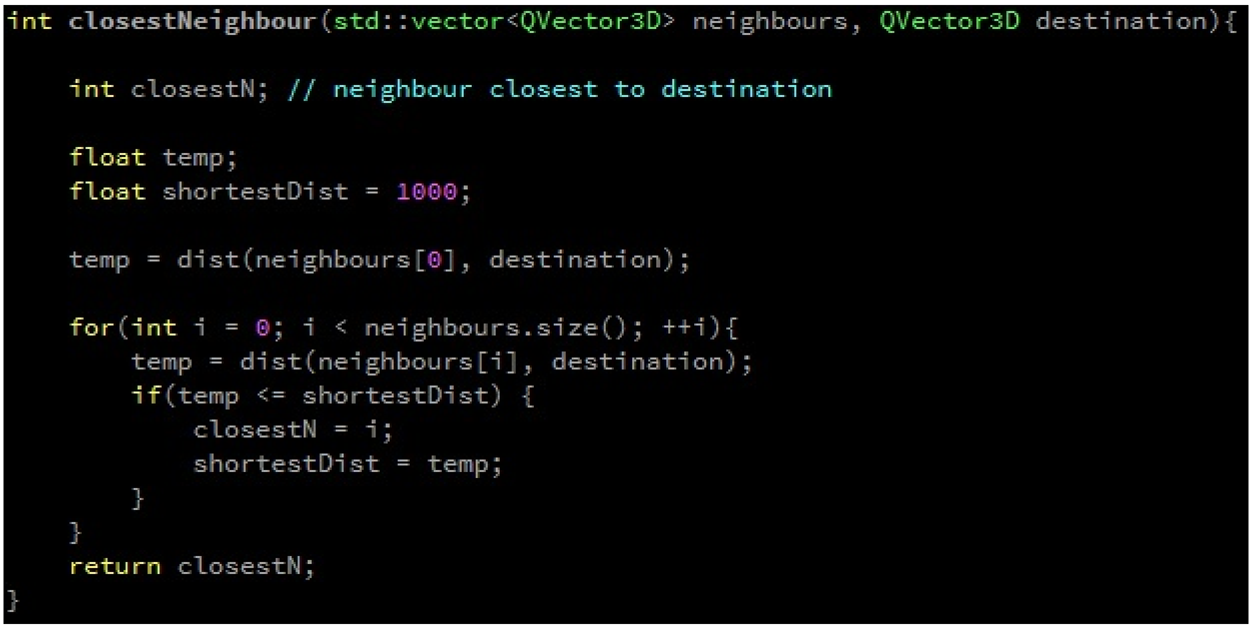
The process of finding a path consists of a few steps, presented as separate functions. The first function is called *dist*() and given two coordinates returns distance between them.



```
float dist(QVector3D a, QVector3D b){

    return sqrt(  (b.x() - a.x()) * (b.x() - a.x())
                + (b.y() - a.y()) * (b.y() - a.y())
                + (b.z() - a.z()) * (b.z() - a.z())
               );
}
```

Figure 4.9: Distance function

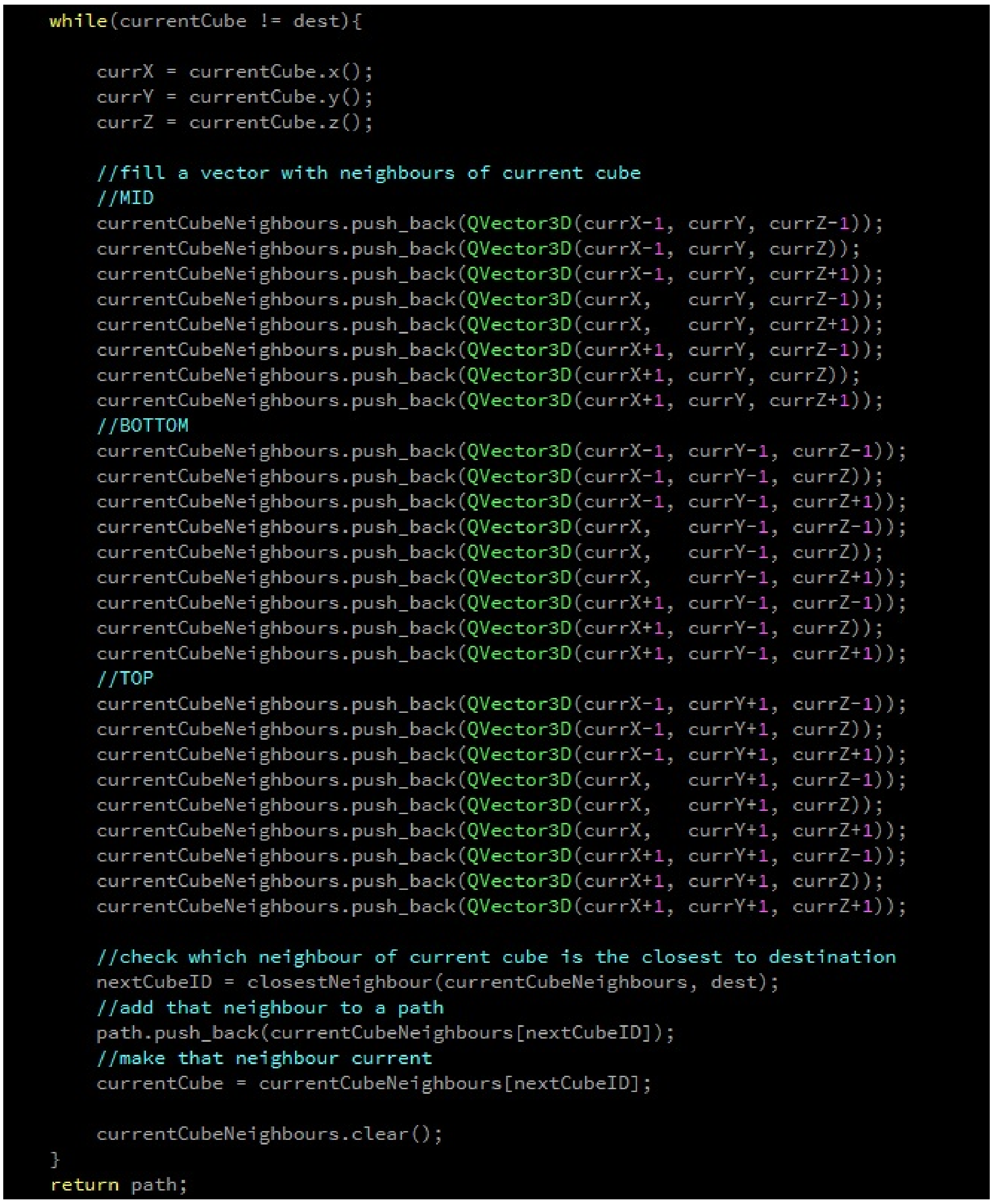
Another function involved in process of finding a path is called *closestNeighbour*. When given array of coordinates of points and destination point, the function calculates distances from each point in given array by using *dist*() function. Then, it determines which point is the closest to the destination and returns it.



```
int closestNeighbour(std::vector<QVector3D> neighbours, QVector3D destination){

    int closestN; // neighbour closest to destination

    float temp;
    float shortestDist = 1000;

    temp = dist(neighbours[0], destination);

    for(int i = 0; i < neighbours.size(); ++i){
        temp = dist(neighbours[i], destination);
        if(temp <= shortestDist) {
            closestN = i;
            shortestDist = temp;
        }
    }
    return closestN;
}
```

Figure 4.10: Function which determines which points is the closest to the destination.

The last function is called *findPath* and needs two previous functions to work. The function creates the list of neighbours for a given cube and uses *closesNeighbour*() function to determine which neighbour is closest to the destination point. Closest neighbour then becomes the current cube to be checked and list of its 26 neighbours [(3*3*3) - 1] is created. The function loops and adds cubes to the path array until the destination point is reached. It then returns the list of cubes on the path so their densities can be found and used for color setting.



```
while(currentCube != dest){

    currX = currentCube.x();
    currY = currentCube.y();
    currZ = currentCube.z();

    //fill a vector with neighbours of current cube
    //MID
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY, currZ+1));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY, currZ+1));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY, currZ+1));
    //BOTTOM
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY-1, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY-1, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY-1, currZ+1));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY-1, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY-1, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY-1, currZ+1));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY-1, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY-1, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY-1, currZ+1));
    //TOP
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY+1, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY+1, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX-1, currY+1, currZ+1));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY+1, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY+1, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX,   currY+1, currZ+1));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY+1, currZ-1));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY+1, currZ));
    currentCubeNeighbours.push_back(QVector3D(currX+1, currY+1, currZ+1));

    //check which neighbour of current cube is the closest to destination
    nextCubeID = closestNeighbour(currentCubeNeighbours, dest);
    //add that neighbour to a path
    path.push_back(currentCubeNeighbours[nextCubeID]);
    //make that neighbour current
    currentCube = currentCubeNeighbours[nextCubeID];

    currentCubeNeighbours.clear();
}
return path;
```

Figure 4.11: Function which goes from one point to another and returns the path between them.

4.3.2 Setting Color - Transfer Function

In this subsection I will show the process of picking the colour from the transfer function. I will also show how the transfer function is created and provide relevant code snippets.

4.3.2.1 Color Lookup Table

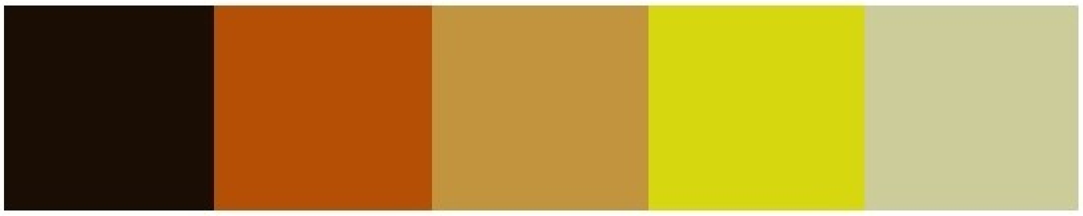
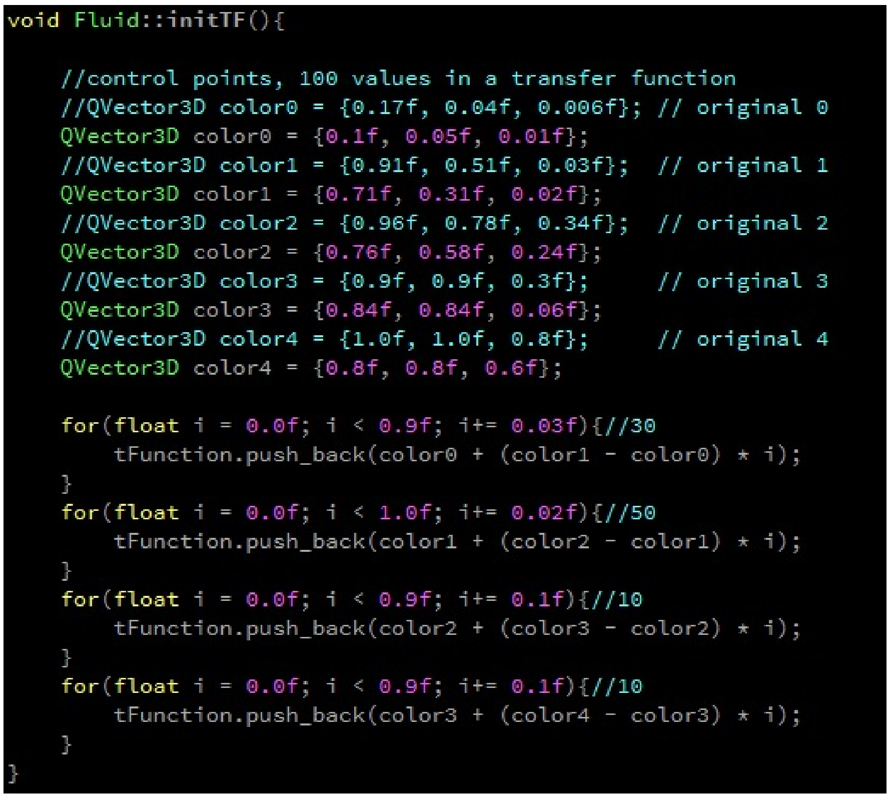


Figure 4.12: Colors used for interpolation in transfer function.

Color lookup table is created once at the beginning of the program and used later for fetching colors. The first thing needed for the transfer function are colors which will be placed at control points of the function. The colours will be then linearly interpolated and stored in an array for later use.

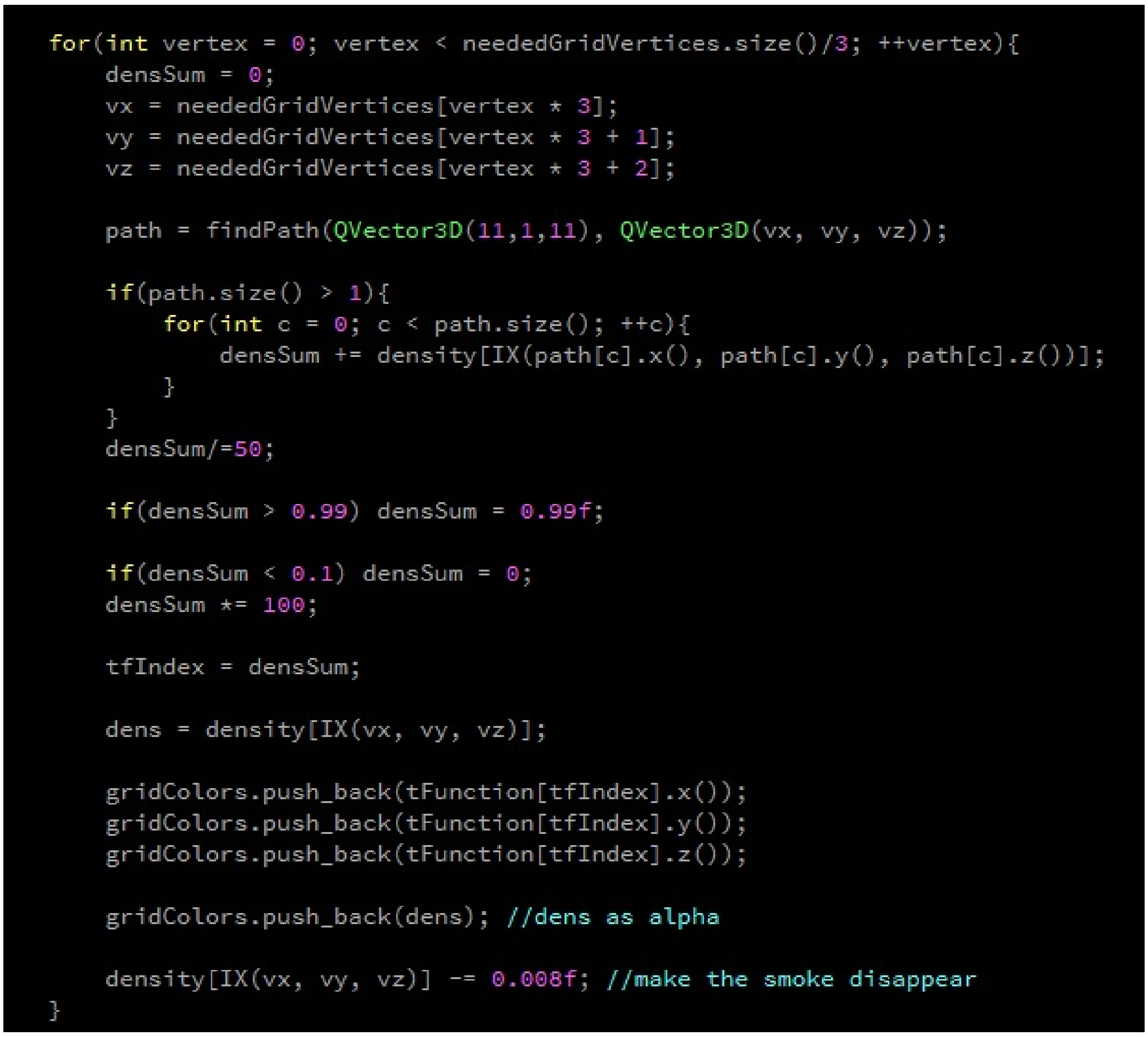


```
void Fluid::initTF(){

    //control points, 100 values in a transfer function
    //QVector3D color0 = {0.17f, 0.04f, 0.006f}; // original 0
    QVector3D color0 = {0.1f, 0.05f, 0.01f};
    //QVector3D color1 = {0.91f, 0.51f, 0.03f};  // original 1
    QVector3D color1 = {0.71f, 0.31f, 0.02f};
    //QVector3D color2 = {0.96f, 0.78f, 0.34f};  // original 2
    QVector3D color2 = {0.76f, 0.58f, 0.24f};
    //QVector3D color3 = {0.9f, 0.9f, 0.3f};     // original 3
    QVector3D color3 = {0.84f, 0.84f, 0.06f};
    //QVector3D color4 = {1.0f, 1.0f, 0.8f};     // original 4
    QVector3D color4 = {0.8f, 0.8f, 0.6f};

    for(float i = 0.0f; i < 0.9f; i+= 0.03f){//30
        tFunction.push_back(color0 + (color1 - color0) * i);
    }
    for(float i = 0.0f; i < 1.0f; i+= 0.02f){//50
        tFunction.push_back(color1 + (color2 - color1) * i);
    }
    for(float i = 0.0f; i < 0.9f; i+= 0.1f){//10
        tFunction.push_back(color2 + (color3 - color2) * i);
    }
    for(float i = 0.0f; i < 0.9f; i+= 0.1f){//10
        tFunction.push_back(color3 + (color4 - color3) * i);
    }
}
```

Figure 4.13: Function which linearly interpolates given colors.

4.3.2.2 Determining Colour

Process of finding the color of the vertex starts with finding the path from the light source, which is located inside of the explosion volume, to this vertex. When $findPath()$ function returns the array of grid cells which are on the path from light source to a vertex, the densities of all grid cells on the path are summed together. After that it is checked if the summed density is bigger than maximal or minimal allowed. Index for the transfer function is then created and color is fetched from the lookup table. The color and density (as alpha channel) is then sent to the array which contents will be later copied to a buffer.



```
for(int vertex = 0; vertex < neededGridVertices.size()/3; ++vertex){
    densSum = 0;
    vx = neededGridVertices[vertex * 3];
    vy = neededGridVertices[vertex * 3 + 1];
    vz = neededGridVertices[vertex * 3 + 2];

    path = findPath(QVector3D(11,1,11), QVector3D(vx, vy, vz));

    if(path.size() > 1){
        for(int c = 0; c < path.size(); ++c){
            densSum += density[IX(path[c].x(), path[c].y(), path[c].z())];
        }
    }
    densSum/=50;

    if(densSum > 0.99) densSum = 0.99f;

    if(densSum < 0.1) densSum = 0;
    densSum *= 100;

    tfIndex = densSum;

    dens = density[IX(vx, vy, vz)];

    gridColors.push_back(tFunction[tfIndex].x());
    gridColors.push_back(tFunction[tfIndex].y());
    gridColors.push_back(tFunction[tfIndex].z());

    gridColors.push_back(dens); //dens as alpha

    density[IX(vx, vy, vz)] -= 0.008f; //make the smoke disappear
}
```

Figure 4.14: setting the color for all of the needed vertices in a particular time step

Chapter 5

Software Testing and Evaluation

This chapter will be about the settings in the program and what influence they have on the simulation. I will present some of the examples of explosions and their initial conditions. Performance of the fluid simulation and illumination parts of the program will also be discussed.

5.1 Initial conditions

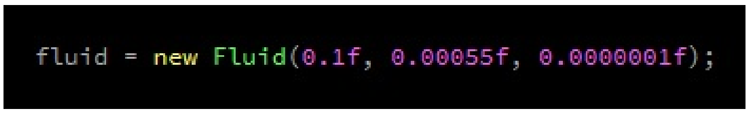
There is a few initial conditions/inputs that have to be provided to the program. The grid and the fluid have to be created and their structures have to be filled with some data. This section will describe what variables and other data have to be provided as an input.

5.1.1 Grid Size

The size of the grid represented by variable N in the program is a first variable to be considered when modelling an explosion. N is the number of voxels of a grid along one axis, so for the three dimensional grid N is cubed. This implicates that the number of the voxels in the grid grows very quickly, while greatly influencing the efficiency of the simulation.

5.1.2 Fluid

When fluid is initialized it takes three variables which describe it: *timeStep*, $diffusion$ and *viscosity*. *timeStep* can be of arbitrary value and the simulation will not blow up, $diffusion$ influences diffusion rate of densities and velocities, and *viscosity* is like stickiness, which describes how thick the fluid is. Without these three variables the fluid cannot be created, so they have to be set for the simulation to work.

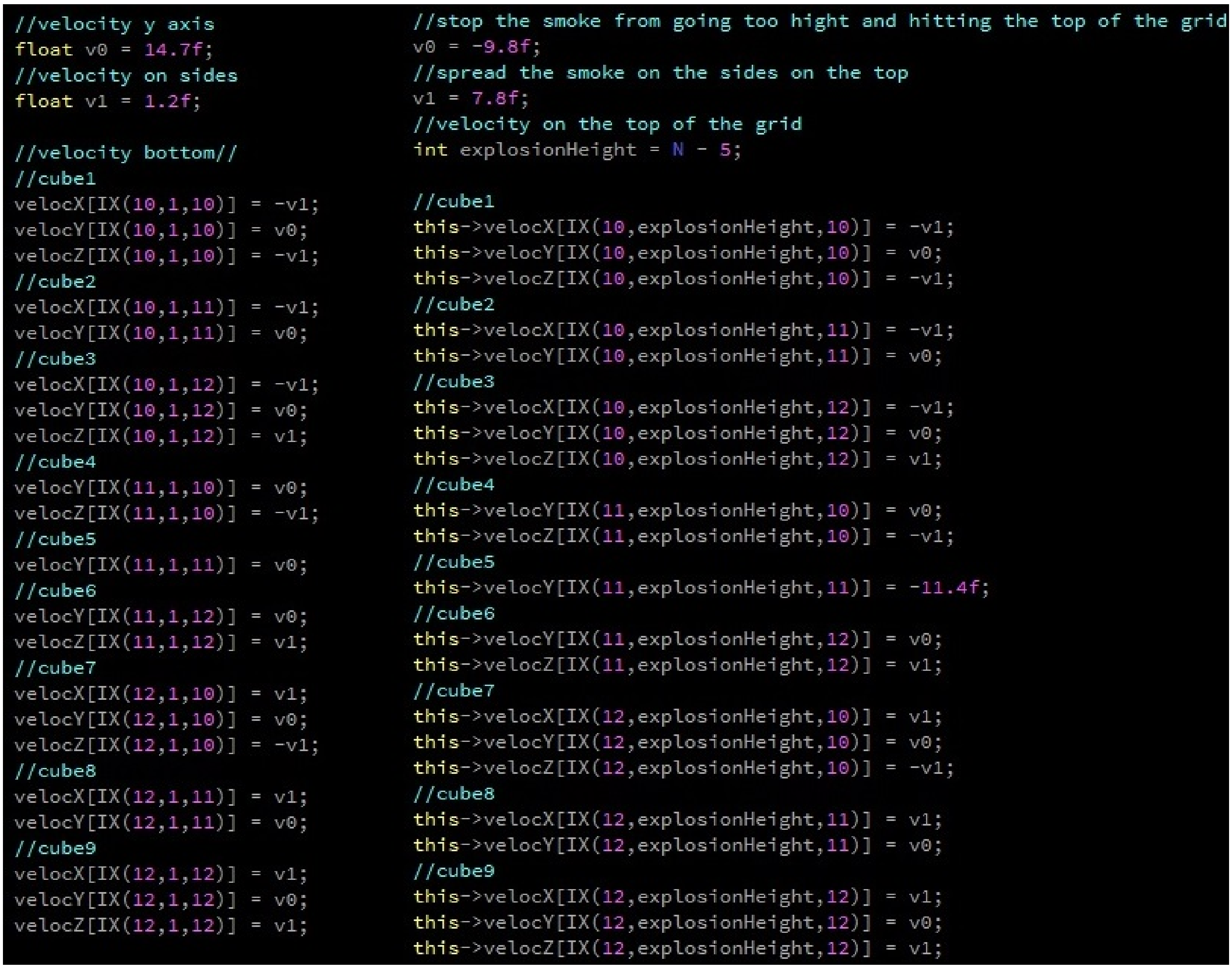


```
fluid = new Fluid(0.1f, 0.00055f, 0.0000001f);
```

Figure 5.1: initialization of a fluid: first variable is a time step, second is diffusion and the third is viscosity

5.1.3 Velocity Field

Velocity field obviously has to be of the same size as the the simulation grid. Initial velocities have to be set in exact voxels found by index function $IX()$ and depending on the desired effect, values for x, y and z have to be provided. Despite the fact that initial velocity field is filled with zeros, the fluid moves because of the diffusion.



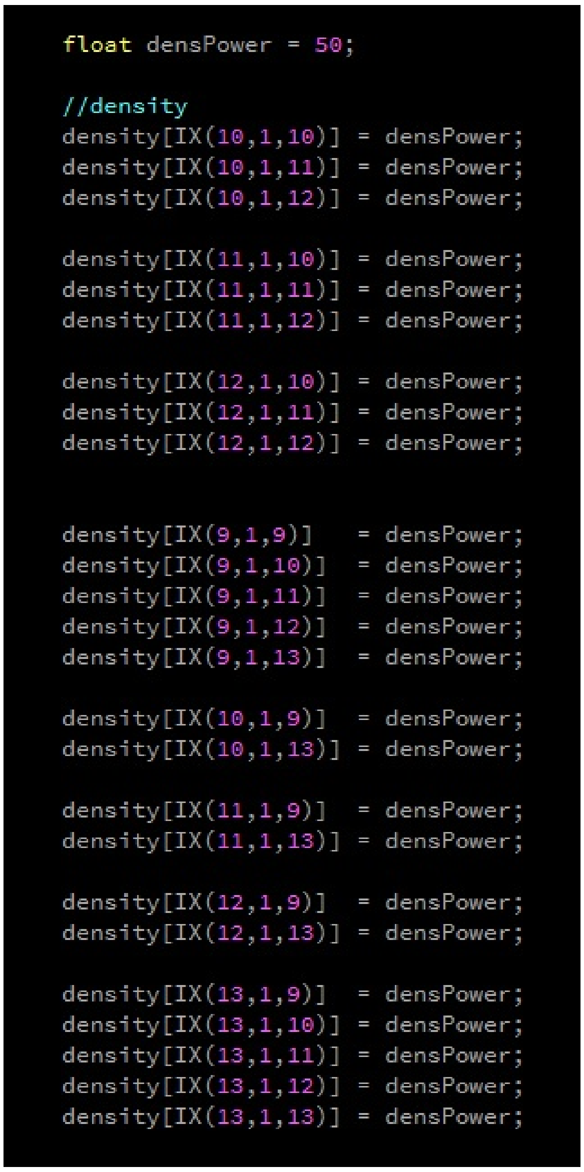
```
//velocity y axis
float v0 = 14.7f;
//velocity on sides
float v1 = 1.2f;

//velocity bottom//
//cube1
velocX[IX(10,1,10)] = -v1;
velocY[IX(10,1,10)] = v0;
velocZ[IX(10,1,10)] = -v1;
//cube2
velocX[IX(10,1,11)] = -v1;
velocY[IX(10,1,11)] = v0;
//cube3
velocX[IX(10,1,12)] = -v1;
velocY[IX(10,1,12)] = v0;
velocZ[IX(10,1,12)] = v1;
//cube4
velocY[IX(11,1,10)] = v0;
velocZ[IX(11,1,10)] = -v1;
//cube5
velocY[IX(11,1,11)] = v0;
//cube6
velocY[IX(11,1,12)] = v0;
velocZ[IX(11,1,12)] = v1;
//cube7
velocX[IX(12,1,10)] = v1;
velocY[IX(12,1,10)] = v0;
velocZ[IX(12,1,10)] = -v1;
//cube8
velocX[IX(12,1,11)] = v1;
velocY[IX(12,1,11)] = v0;
//cube9
velocX[IX(12,1,12)] = v1;
velocY[IX(12,1,12)] = v0;
velocZ[IX(12,1,12)] = v1;
```

```
//stop the smoke from going too hight and hitting the top of the grid
v0 = -9.8f;
//spread the smoke on the sides on the top
v1 = 7.8f;
//velocity on the top of the grid
int explosionHeight = N - 5;

//cube1
this->velocX[IX(10,explosionHeight,10)] = -v1;
this->velocY[IX(10,explosionHeight,10)] = v0;
this->velocZ[IX(10,explosionHeight,10)] = -v1;
//cube2
this->velocX[IX(10,explosionHeight,11)] = -v1;
this->velocY[IX(10,explosionHeight,11)] = v0;
//cube3
this->velocX[IX(10,explosionHeight,12)] = -v1;
this->velocY[IX(10,explosionHeight,12)] = v0;
this->velocZ[IX(10,explosionHeight,12)] = v1;
//cube4
this->velocY[IX(11,explosionHeight,10)] = v0;
this->velocZ[IX(11,explosionHeight,10)] = -v1;
//cube5
this->velocY[IX(11,explosionHeight,11)] = -11.4f;
//cube6
this->velocY[IX(11,explosionHeight,12)] = v0;
this->velocZ[IX(11,explosionHeight,12)] = v1;
//cube7
this->velocX[IX(12,explosionHeight,10)] = v1;
this->velocY[IX(12,explosionHeight,10)] = v0;
this->velocZ[IX(12,explosionHeight,10)] = -v1;
//cube8
this->velocX[IX(12,explosionHeight,11)] = v1;
this->velocY[IX(12,explosionHeight,11)] = v0;
//cube9
this->velocX[IX(12,explosionHeight,12)] = v1;
this->velocY[IX(12,explosionHeight,12)] = v0;
this->velocZ[IX(12,explosionHeight,12)] = v1;
```

Figure 5.2: velocity modelling - each cube to be modelled has specified velocity values for x, y or z axis

5.1.4 Density Field

Density field similarly to the velocity field has to be of the same size as the simulation grid. Unlike velocity field, density field is represented as one array, because for each grid cell there is one density value not dependent on dimensions. Value of the density is used as alpha channel in the color data which is sent to the fragment shader. Therefore, to see something on the screen densities have to be set.



```
float densPower = 50;

//density
density[IX(10,1,10)] = densPower;
density[IX(10,1,11)] = densPower;
density[IX(10,1,12)] = densPower;

density[IX(11,1,10)] = densPower;
density[IX(11,1,11)] = densPower;
density[IX(11,1,12)] = densPower;

density[IX(12,1,10)] = densPower;
density[IX(12,1,11)] = densPower;
density[IX(12,1,12)] = densPower;


density[IX(9,1,9)]   = densPower;
density[IX(9,1,10)]  = densPower;
density[IX(9,1,11)]  = densPower;
density[IX(9,1,12)]  = densPower;
density[IX(9,1,13)]  = densPower;

density[IX(10,1,9)]  = densPower;
density[IX(10,1,13)] = densPower;

density[IX(11,1,9)]  = densPower;
density[IX(11,1,13)] = densPower;

density[IX(12,1,9)]  = densPower;
density[IX(12,1,13)] = densPower;

density[IX(13,1,9)]  = densPower;
density[IX(13,1,10)] = densPower;
density[IX(13,1,11)] = densPower;
density[IX(13,1,12)] = densPower;
density[IX(13,1,13)] = densPower;
```

Figure 5.3: density modelling - 5x5 voxel field filled with initial density

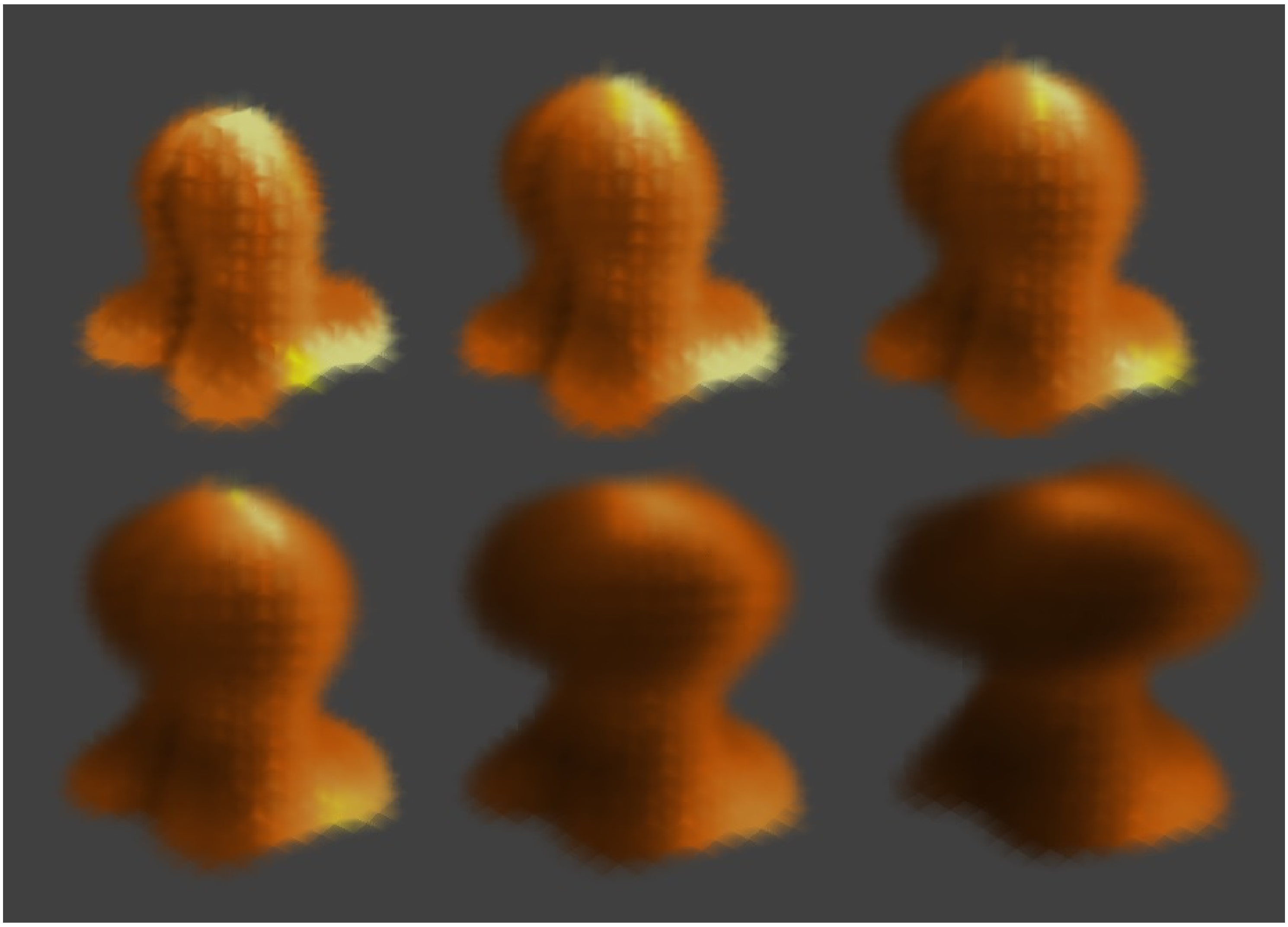


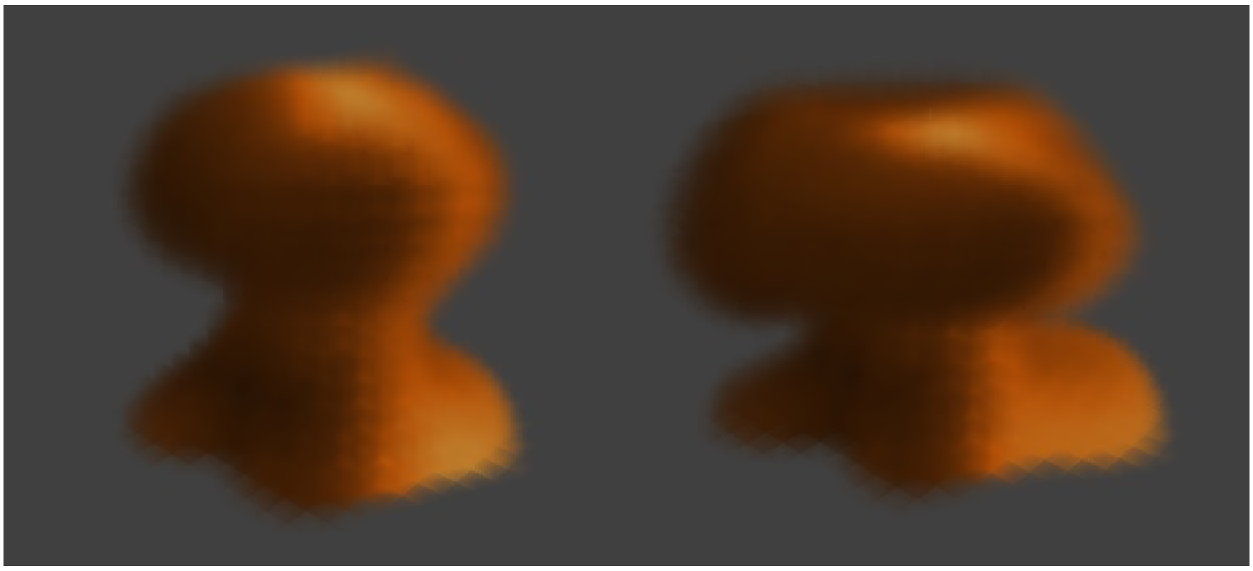
Figure 5.4: result of simulation in a 27x27x27 grid with settings showed in code snippets in this section

5.2 Parameter sensitivity

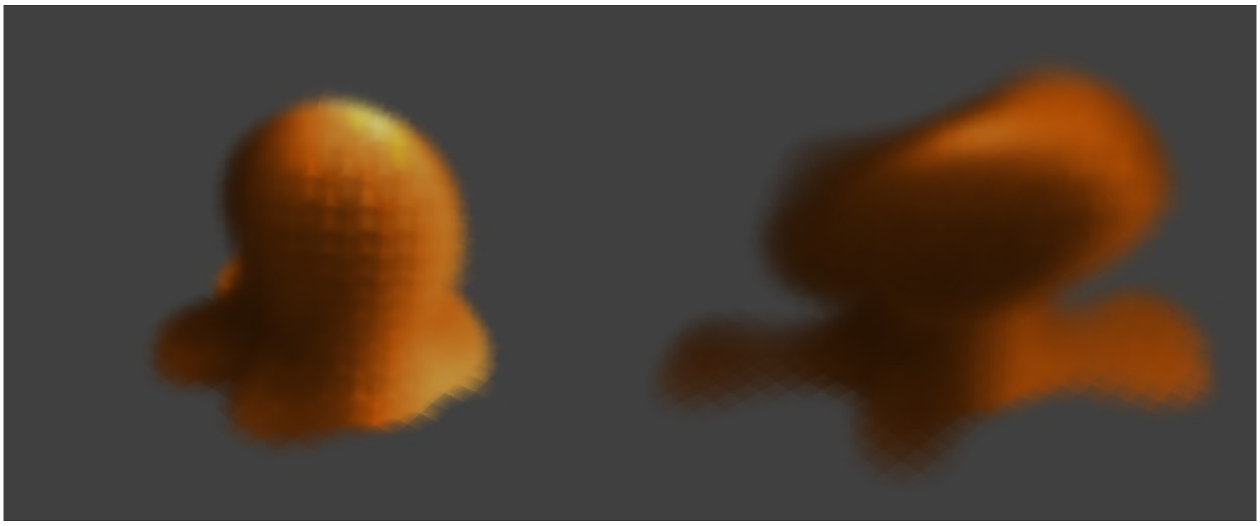
In this section I will describe how properties of fluid influence the simulation. Examples of how they change the visual effects of the simulation will also be presented.

5.2.1 Time Step

Time step is a very sensitive variable. As shown in previous section it was set to be 0.1. If it is changed even by only 0.01 the simulation looks noticeably different. Therefore, its value has to be carefully picked when simulating the explosion.



(a) $timeStep = 0.1$ (b) $timeStep = 0.11$

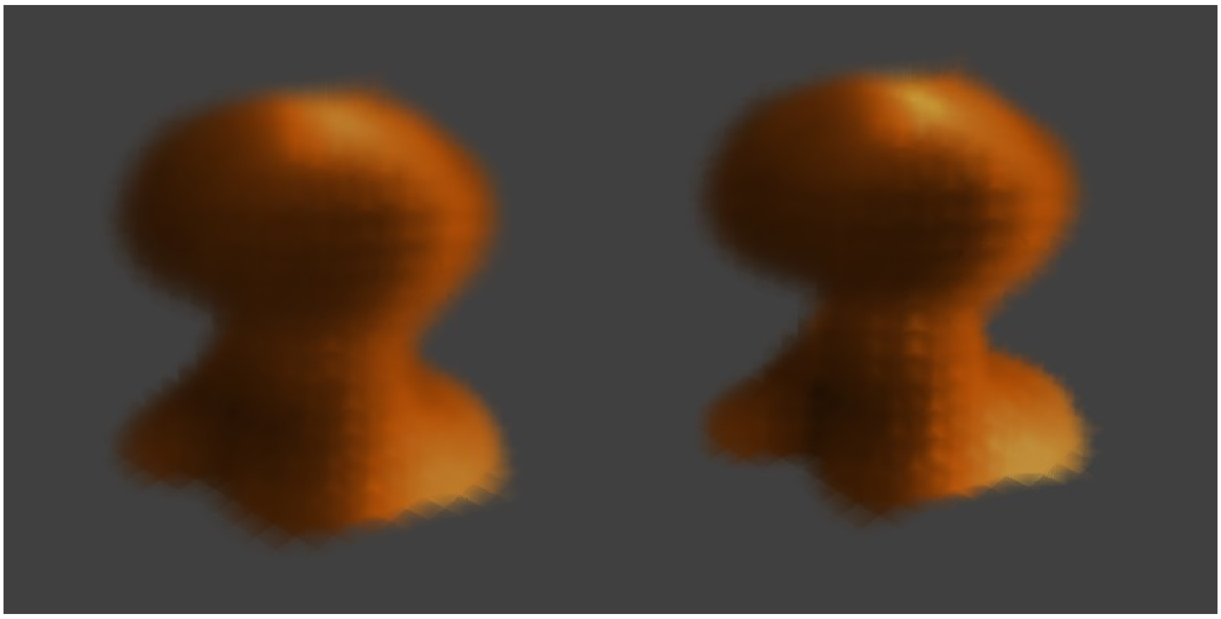


(c) $timeStep = 0.09$ (d) $timeStep = 0.15$

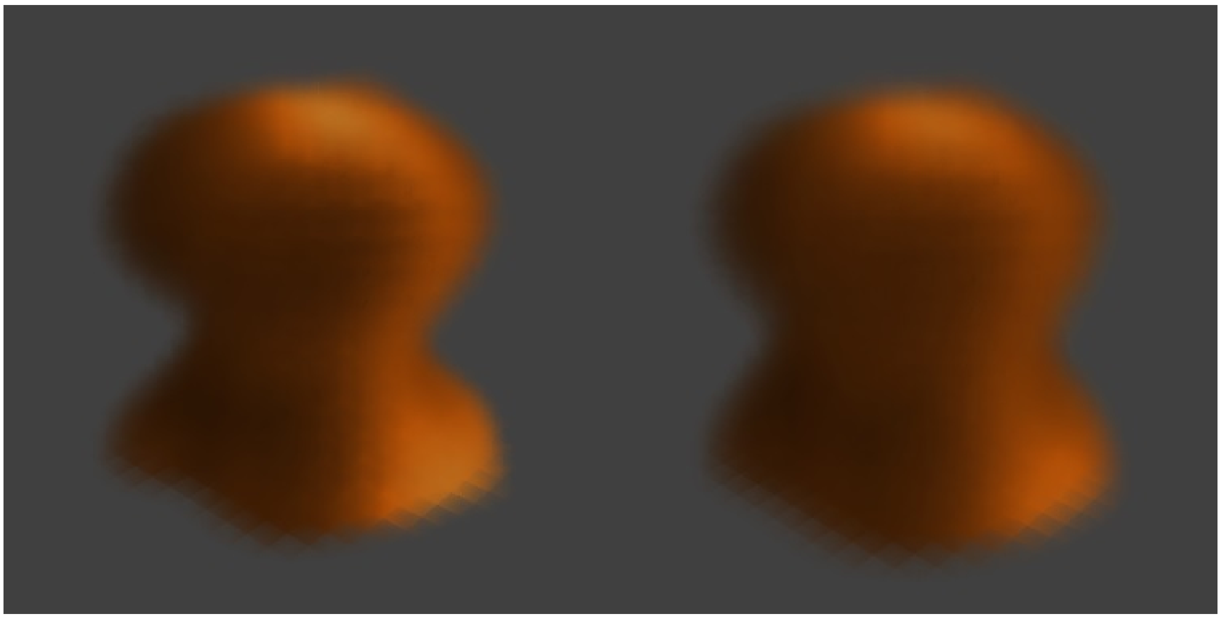
Figure 5.5: Visual changes dependent on *timeStep* variable, pictures of the same frame, but with different time steps

5.2.2 Diffusion

$diffusion$ variable, initially set to 0.00055, is used to calculate diffusion rate in a $diffuse()$ function. Depending on that, grid cells exchange their densities and velocities with their neighbours. Resulting changes in simulation caused by changes of $diffusion$ are presented below.



(a) $diffusion = 0.00055$ (b) $diffusion = 0.0003$

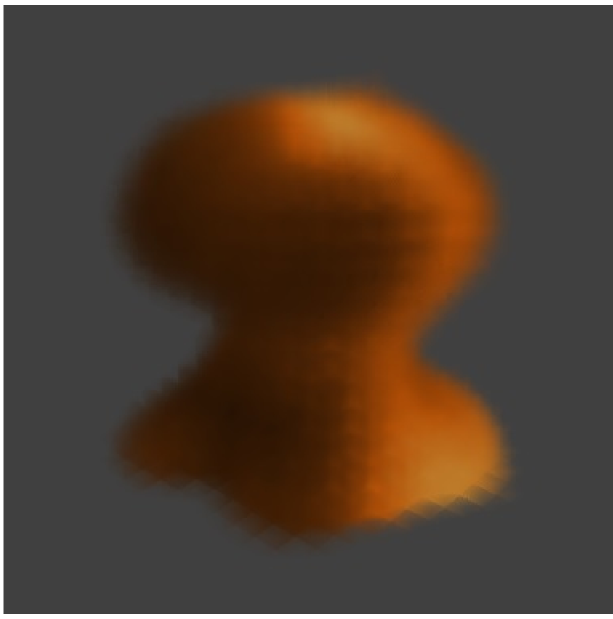


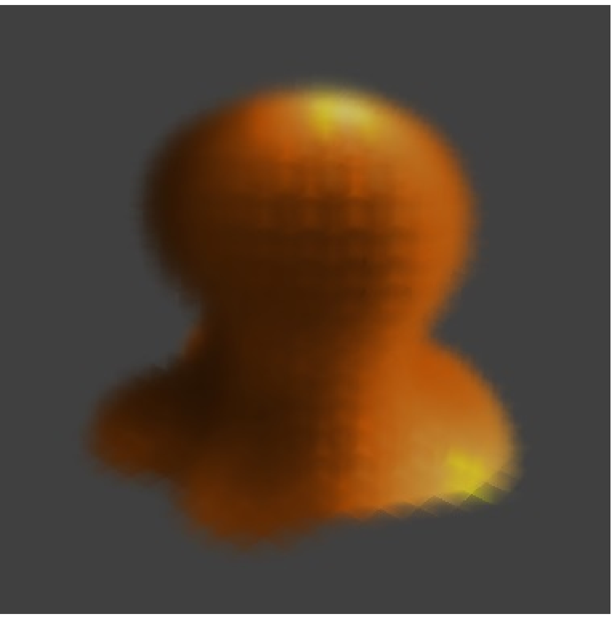
(c) $diffusion = 0.0009$ (d) $diffusion = 0.0015$

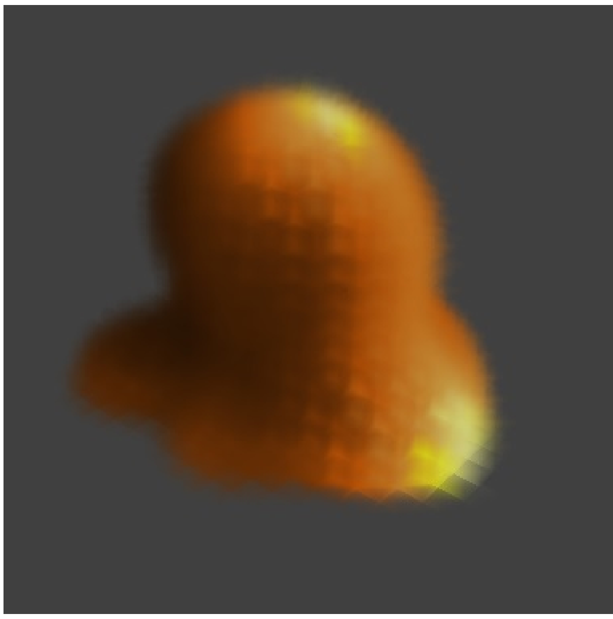
Figure 5.6: Results produced by fluid simulation with different $diffusion$ values. The frames and all settings are the same except $diffusion$ variable.

5.2.3 Viscosity

viscosity describes how thick, deformable the fluid is. Increase of value will result in higher stickiness of the fluid while smaller *viscosity* value will make the fluid thin. Initially it is set to be 0.0000001, so the fluid is not very thick. Results produced by simulation with increased *viscosity* value are shown below.


(a) $viscosity = 0.0000001$


(b) $viscosity = 0.0003$


(c) $viscosity = 0.00055$

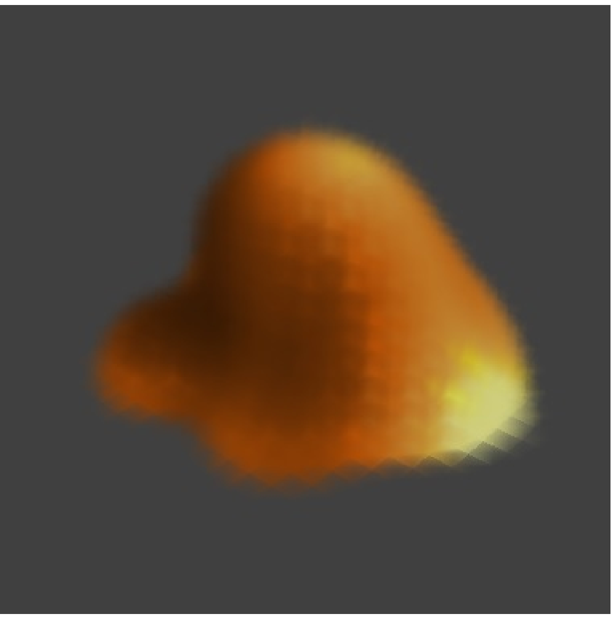
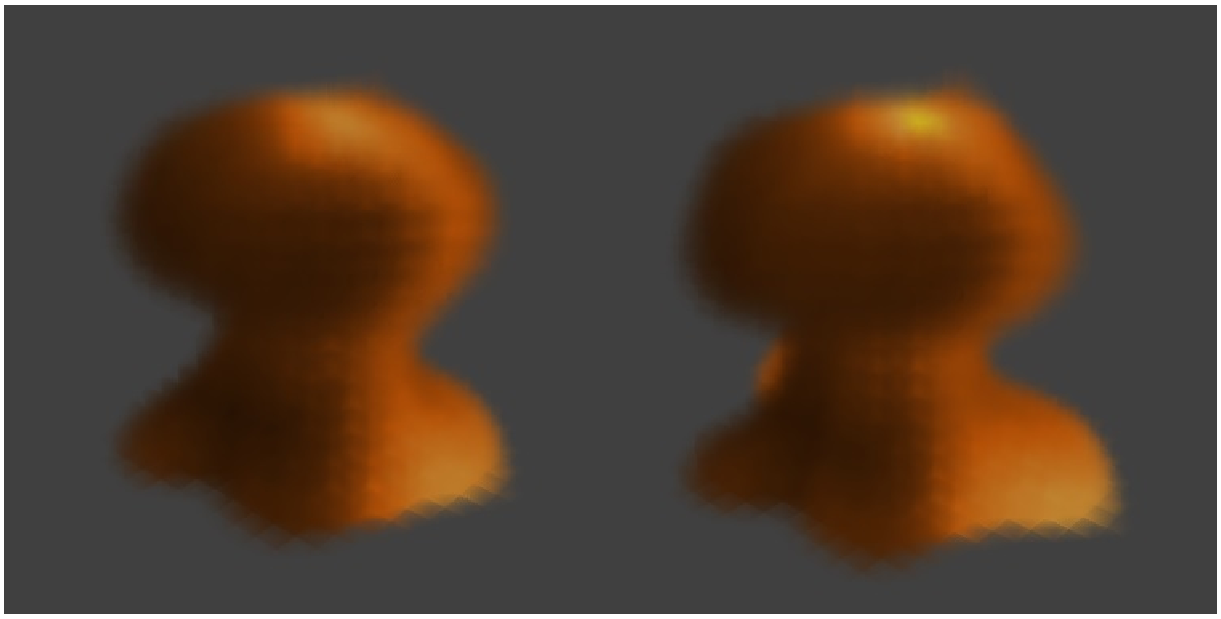

(d) $viscosity = 0.001$

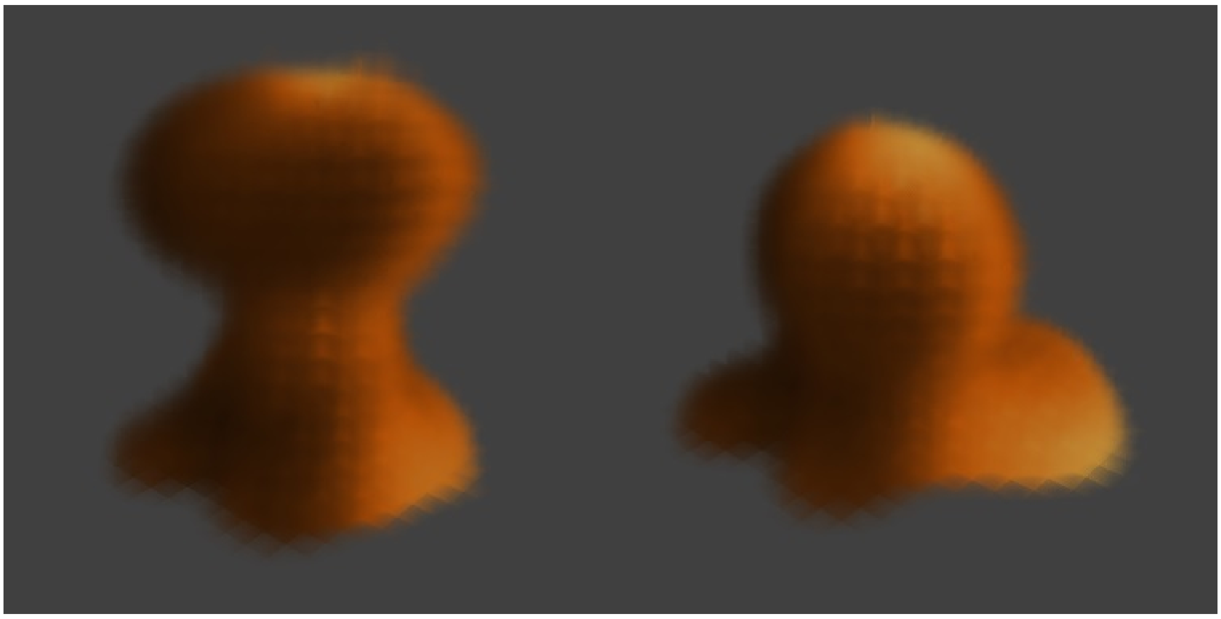
Figure 5.7: Impact of *viscosity* value on results of the simulation. Pictures present the same frames of simulation with *viscosity* values changed.

5.2.4 Number of Iterations

The number of iterations influences the the accuracy of results returned by $linearSolve()$ function. The more iterations is done the better results are given, but each iteration increases the computations time by going through additional $(N-1)^3$ grid cells. Therefore, there is a trade-off between computation time and numeral accuracy of the results.



(a) *iterations* = 4 (b) *iterations* = 3



(c) *iterations* = 5 (d) *iterations* = 7

Figure 5.8: Differences in results caused by changing the accuracy of linear solver.

5.3 Dependant Parameters

Some of the parameters used in the program, rely on how other parameters have been set. This mainly refers to the size of the grid and initial modelling of vector fields because the resolution of the explosion depends on the size of the grid. This section will discuss how those parameters are dependant.

5.3.1 Grid Size and Density Field

When simulation grid is created it is of the same size as the density field. Initial state of the density fields depends on how it was modelled. However, if the size of both of them changes, the positions of densities would have to be changed. If, for example, the size of the grid is smaller than it was before, it is possible that densities in certain grid cells will now be outside of the grid and trying to access those values by their index would be inappropriate. When making the explosion with better resolution, obviously more grid cells would have to have initial densities set.

5.3.2 Grid Size and Velocity Field

In case of the velocity field which is, as the density field, of the same size as the simulation grid. When initial velocity is modelled, indices of grid cells, for which the velocities are set, obviously have to be inside of the grid. But when the size of the grid is changed there can be the same problem as with density field. Furthermore, when dealing with velocities and different sizes of grids, the chosen forces for each grid cell have to be adjusted for the grid size.

5.4 Physics of Explosion and Illumination Algorithm Efficiency

This section is about influence of parts of the program on its efficiency. The efficiency results will be presented in the subsections below.

5.4.1 Efficiency of Physics Calculations

The program was modified to omit color calculations in order to measure efficiency of physics calculations. Buffers with vertices, indices and colors were removed as well as function for creating grid, its indices and colors. The program, when run only to perform numerical physics calculations works in 26 to 33 frames per second for 4 iterations in linear solver. Changes of fluid variables ($timeStep$, $diffusion$ and $viscosity$) do not influence the time of computations. The only thing, except obviously grid size, that has impact on calculations is *iterations* variable which increases the accuracy of linear solver, thus extends the calculations time.

Number of iterations	FPS min	FPS max
3	26	41
4	26	33
5	15	28
7	18	22
10	14	16
15	9	12

Table 5.1: Efficiency of physics calculations depending on the number of iterations for a grid size 27^3.

5.4.2 Efficiency of Color Calculations

The process of color calculations involves finding a path from the light source to a given point. This process is computationally expensive, because it involves looping through neighbours of each grid cell on the path and calculating their distance to the destination. In case of color calculations, performance depends on the number of grid cells with initial density greater than 0 and density value, because the greater the density value is the slower it decreases and spreads to more grid cells. It also depends on $diffusion$ and $viscosity$ of a fluid.

initial grid cells with density > 0	FPS min	FPS max
25	6	20
50	4	17
75	3	14
100	3	13
125	3	10

Table 5.2: efficiency of the program dependent on the number of grid cells with initial density greater than 0

Initial density value	FPS min	FPS max
20	8	18
30	7	18
50	6	17
75	4	14
100	4	14

Table 5.3: efficiency results for different initial density values

diffusion	FPS min	FPS max
0.0003	6	18
0.00055	5	17
0.0009	5	15
0.0015	5	12
0.003	4	12

Table 5.4: Results for efficiency depending on the $diffusion$

viscosity	FPS min	FPS max
0.0000001	6	15
0.0003	6	17
0.00055	7	17
0.001	9	18
0.1	15	20

Table 5.5: *viscosity* increases the stickiness of the fluid, which keeps it from spreading to a larger number of grid cells, making the simulation faster

Chapter 6

Future Work and Conclusions

6.1 Future Work

This chapter will be about what can be done to make the project better. I will present the ideas for improving different parts of the project.

6.1.1 Possible Improvements

6.1.1.1 Bigger Grids - Better Resolution

As shown in the previous chapters, the program works in a real time. Depending on the settings, the results are different, but usually about 10-20 frames per second is rendered. The number of rendered frames per second can be increased by making the calculations offline and displaying the final result later. Therefore, explosions could be generated in bigger grids and could be rendered smoothly which would significantly improve visual effects.

6.1.1.2 Fluid simulation

The fluid simulation part produces satisfying results of about 30 frames per second. But this is only when physics calculations are run without the color picking part of program. Physics calculations can be improved by using conjugate gradient solver which better handles convergence than Gauss-Seidel relaxation.

6.1.1.3 Picking the color

To improve the process of color calculations, the simulation of combustion can be done. This would require more computation time and different transfer function for picking the color depending on the temperature but it would improve the final look of explosion. The other approach for illumination of the explosion would be to create approximative solution of rendering equation in a volume rendering.
Another possible improvement is creation of a scene. On this scene the shadows from the density could be cast, however this would require implementation of a method for shadows calculations and more computation time but would produce better visual effects.

6.2 Conclusions

The process of understanding the physics involved in this project was a challenging task and the amount of scientific articles about simulating fluids read was overwhelming. Implementation of a fluid simulation for two dimensions and rendering effects took a lot of time but was a good start. However, converting it then to three dimensions was way easier. Also, removing data redundancy by creating indices for vertices in a grid and rendering only needed grid cells was a tricky task but it resulted in a better performance. A lot of work has been done, but the project can be improved in many ways regarding both explosion data generation and self-illumination parts.

More Examples of Explosions

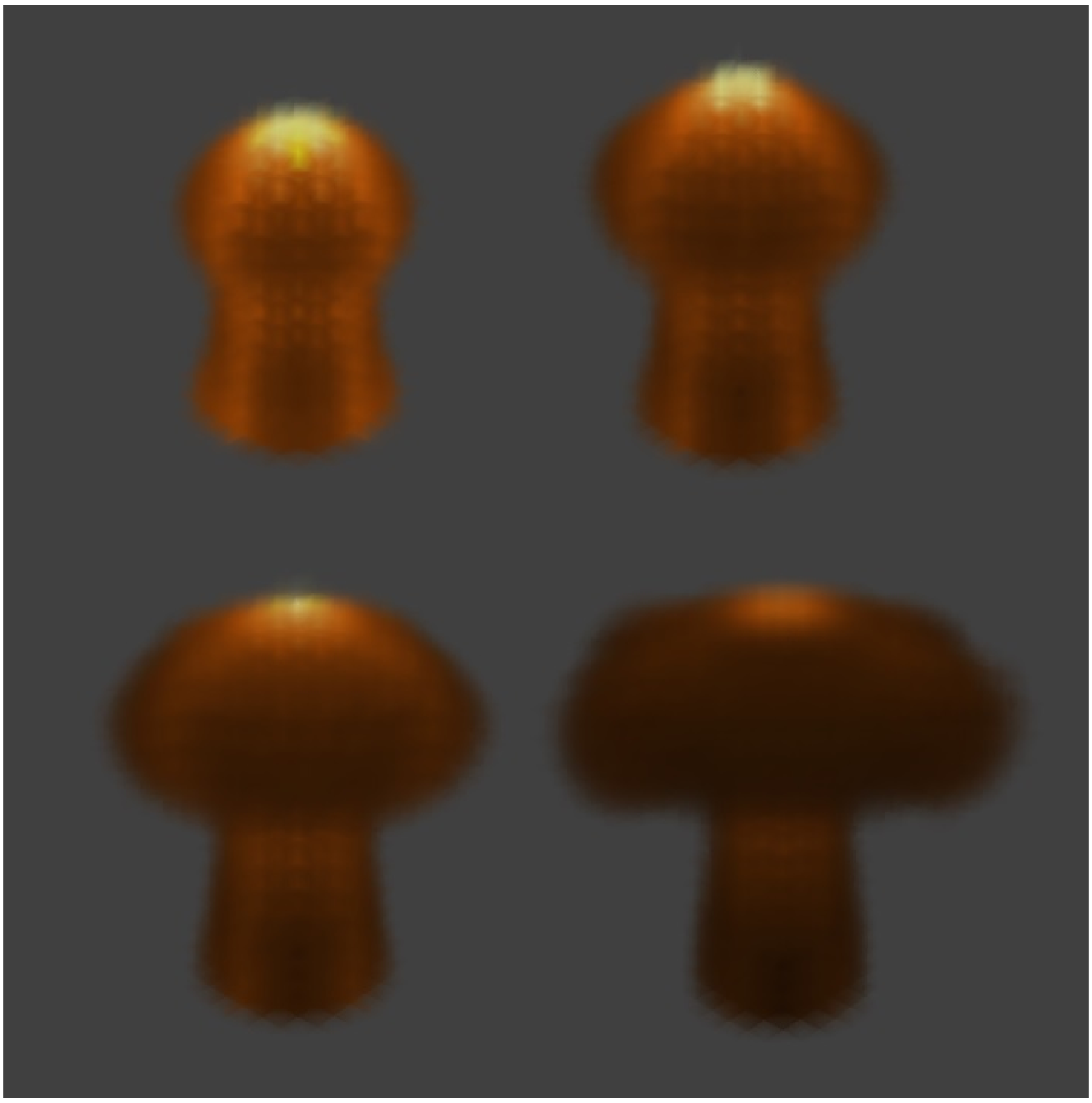


Figure 6.1: Example of explosion 1

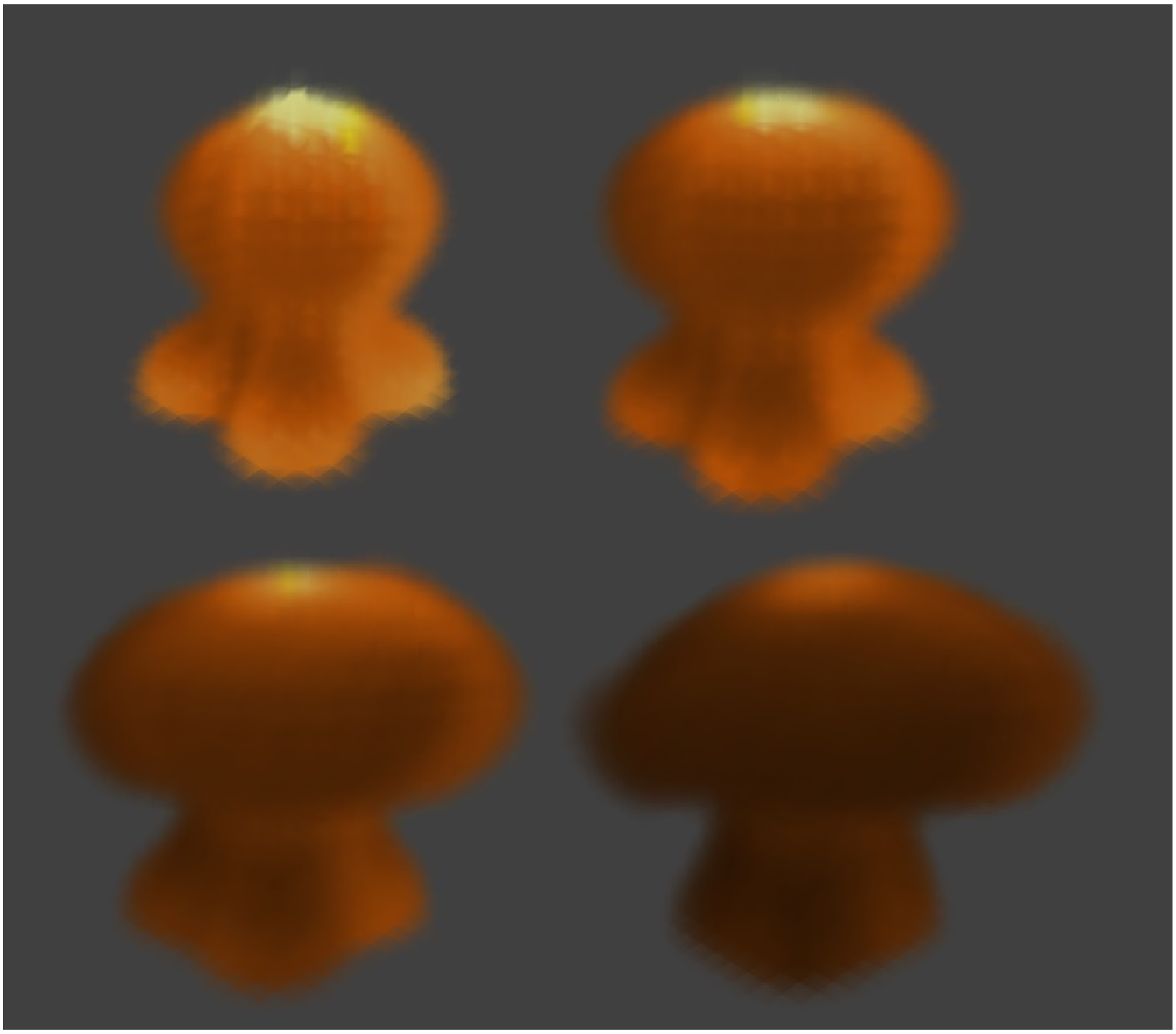


Figure 6.2: Example of explosion 2

References

[1] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 37–45, New York, NY, USA, 1968. Association for Computing Machinery.

[2] J. Arvo. Backward ray tracing. In *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, pages 259–263, 1986.

[3] M. Ash. Simulation and visualization of a 3d fluid. Master's thesis, Université d'Orléans in Orléans, Orlean, France, 2005.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[5] Z. Botev and A. Ridder. Variance reduction. *Wiley StatsRef: Statistics Reference Online*, pages 1–6, 2014.

[6] C. Braley and A. Sandu. Fluid simulation for computer graphics: A tutorial in grid based and particle based methods. *URL: http://www. colinbraley. com/Pubs/FluidSimColinBraley. pdf*, 1, 2009.

[7] E. Haines and T. Akenine-Möller, editors. *Ray Tracing Gems*. Apress, 2019.

[8] C. Hirsch. *Numerical Computation of Internal and External Flows: The Fundamentals of Computational Fluid Dynamics*. Elsevier Science and Technology, Oxford, 2007.

[9] W. Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.

[10] H. W. Jensen and N. J. Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, 1995.

[11] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, Aug. 1986.

[12] S. J. Koppal. *Lambertian Reflectance*, pages 441–443. Springer US, Boston, MA, 2014.

[13] L. Lucy. A numerical approach to the testing of the fission hypothesis. *aj*, 82:1013–1024, Dec. 1977.

[14] M. Müller, D. Charypar, and M. H. Gross. Particle-based fluid simulation for interactive applications. In *Symposium on Computer animation*, pages 154–159, 2003.

[15] The Qt Company. *Qt Documentation.*

[16] M. Segal, K. Akeley, C. Frazier, J. Leech, and P. Brown. The opengl graphics system: A specification.

[17] J. Stam. Real-time fluid dynamics for games.

[18] B. Stroustrup. C++ applications.

[19] B. Stroustrup. Lecture: The essence of c++. [online], May 2014.

[20] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Lightcuts: a scalable approach to illumination. In *ACM SIGGRAPH 2005 Papers*, pages 1098–1107. 2005.

[21] H. Wang. Fluids: Liquids and gases, lecture notes, animation and simulation module. [online], March 2020.

[22] D. Weiskopf. *GPU-based interactive visualization techniques.* Springer, 2007.

[23] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[24] S. Woop, J. Schmittler, and P. Slusallek. Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (TOG)*, 24(3):434–444, 2005.

[25] C. Yuksel and C. Yuksel. Lighting grid hierarchy for self-illuminating explosions. *ACM Trans. Graph.*, 36(4):110–1, 2017.

Appendices

Appendix A

External Material

The following external materials were used:

- QT Creator and its libraries - `https://www.qt.io/`

- OpenGL - `https://www.opengl.org/`

- GLM library for matrices - `https://glm.g-truc.net/0.9.9/index.html`

- Jos Stam's solution of Navier-Stokes Equations - `https://pdfs.semanticscholar.org/847f/819a4ea14bd789aca8bc88e85e906cfc657c.pdf`

- parts of Mike Ash's interpretation of Jos Stam's method `https://www.mikeash.com/pyblog/fluid-simulation-for-dummies.html`

The source code for the project is available at GitHub repository:

`https://github.com/szymonspiesz/MasterProjectSourceCode`

Appendix B

Ethical Issues Addressed

No ethical issues addressed.