A Physically Based Pipeline for Real-Time Simulation and Rendering of Realistic Fire and Smoke

Yiyang He

Bachelor's Thesis in Computer Science
at Stockholm University, Sweden, 2018

Stockholm University

A Physically Based Pipeline for Real-Time Simulation and Rendering of Realistic Fire and Smoke

Yiyang He

Bachelor's Thesis in Computer Science (15 ECTS credits)
Bachelor's Programme in Computer Science
Stockholm University year 2018
Supervisor was Christopher Peters, KTH EECS
Examiner was Mårten Björkman, KTH EECS

Department of Mathematics
Stockholm University
SE-106 91 Stockholm, Sweden

Abstract

With the rapidly growing computational power of modern computers, physically based rendering have found its way into real world applications. Real-time simulations and renderings of fire and smoke had become one major research interest in modern video game industry, and will continue being one important research direction in computer graphics.

To visually recreate realistic dynamic fire and smoke is a complicated problem. Furthermore, to solve the problem requires knowledge from various areas, ranged from computer graphics and image processing to computational physics and chemistry. Even though most of the areas are well-studied separately, when combined, new challenges will emerge. This thesis focuses on three aspects of the problem, dynamic, real-time and realism, to propose a solution in form of a GPGPU pipeline, along with its implementation. Three main areas with application in the problem are discussed in details: fluid simulation, volumetric radiance estimation and volumetric rendering. The weights are laid upon the first two areas. The results are evaluated around the three aspects, with graphical demonstrations and performance measurements.

Uniform grids are used with Finite Difference (FD) discretization scheme to simplify the computation. FD schemes are easy to implement in parallel, especially with ComputeShader, which is well supported in Unity engine. The whole implementation can easily be integrated into any real world applications in Unity or other game engines that support DirectX 11 or higher.

Sammanfattning

En fysiskt baserad rörledning för realtidssimulering och rendering av realistisk eld och rök

Med den snabbt växande beräkningskapaciteten hos moderna datorer, har fysiskt baserad rendering realiserats i verkliga applikationer. Realtidssimuleringar och rendering av eld och rök har blivit ett viktigt forskningsintresse i den moderna videospelsindustrin och kommer fortsätta att vara en viktig forskningsriktning inom datorgrafik.

Att visuellt återskapa realistisk och dynamisk eld och rök är ett komplicerat problem. För att lösa problemet krävs dessutom kunskap från olika områden, allt från datorgrafik och bildbehandling till beräkningsfysik och kemi. Trots att de flesta områdena har studerats grundligt var för sig, kommer nya utmaningar att uppstå när de kombineras. Denna uppsats fokuserar på tre aspekter av problemet, nämligen dynamiken, realtid och realism, för att föreslå en lösning i form av en GPGPU-rörledning tillsammans med dess implementation. Tre huvudområden med tillämpning på problemet diskuteras i detalj: fluidsimulering, volymetrisk strålningsestimering och volymetrisk rendering. Fokus läggs på de två första områdena. Resultaten utvärderas kring de tre aspekterna, med grafiska demonstrationer och prestationsmätningar.

Uniform grid används med Finite Difference (FD) diskretiseringsschema för att förenkla beräkningen. FD-scheman är lätta att genomföra parallellt, särskilt med ComputeShader, som är välskött i spelmotorn Unity. Hela implementeringen kan enkelt integreras i alla verkliga applikationer i Unity, eller andra spelmotorer, som stöder DirectX 11 eller högre.

Acknowledgements

I would like to thank my supervisor Christopher Peters at KTH for his continuous support and direction giving. I would also like to thank my thesis examiner Mårten Björkman at KTH for reading and commenting this thesis. I also want to thank my educational counsellor Caroline Nordquist for her support during my years at NADA. A special thanks to Lennart Edsberg at KTH for his help after the lecture on numerical analysis of PDEs.

Contents

Chapter 1

Introduction

Visually reproducing realistic fire and smoke is a complicated task with no lack of specific studies, most of them are focused on deep knowledge of a few subareas of the task, for example within fluid dynamics [1][2][3] or rendering of fire [4] or smoke [5][6], but rarely combined. Due to the complexity of the task, amount the studies, pre-computations are often used and real-time solutions are rarely seem, even less results can be found where the visual aspects of fire and smoke are treated as a combined problem. The main contribution of this thesis deal with this combined complexity, and provide a collection of underlying theories and carefully selected methods for designing and implementing solutions of such task. Figure 1.1 and 1.2 show by systematic study and careful implementation of this multi-disciplinary problem, the solution is able to produce a very wide range of interesting combustion phenomena with quality.

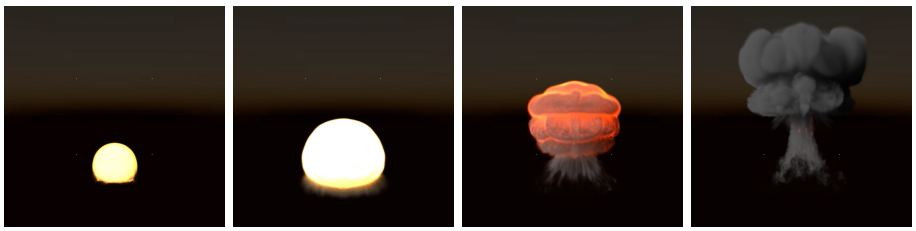


Figure 1.1: A nuclear explosion generated and rendered by the implementation of the pipeline in this thesis.

1.1 Problem Statement

Large scale fire or smoke can often be found in scenes of movies and video games, which add a certain "kick" to these scenes. On the hand a small scale fire or smoke, for example the smoke trail from a cigarette, can create create dramatic atmosphere for the scene. Combustion phenomena are frequently seem

in the real-world, from the fire above the stove to a tragic gas explosion. The scenes with fire and smoke feels more realistic, because it reflexes the natural phenomena of the real-world.

How to recreate such real-world phenomena using physically based renderings, is the central question in this thesis. To start with, observations should be made and questions should be asked. Questions like what makes fire and smoke appear as the way they look? Here, we list some of the questions as a start point of the research. The goal is to recreate these visually stunning phenomena. With the goal in mind, a implementation will be done to test to what standard the goal can be reached.

The are three questions can be stated around three keywords: dynamic, real-time and realistic:

1. What are the difficulties in dynamically recreate these phenomena in real-time?

2. What are the factors that have the most impact on the visual appearance of these phenomena?

3. To what standard is it possible to recreate these phenomena using the available theories and computer hardware today?

The first question is about the dynamic and real-time aspect of the problem, concerning about the difficulties of real-time dynamic simulations of the problem. Also about the quality performance trade off. The second question is about the concern of definition of visual realism for fire, smoke and explosion, also about how to recreate the realism. The final question shows the concern about the limitation of solution methods and hardware available today, and asking about how far can the limits be pushed.

1.2 Hypothesis

The first question can be answered by provide discussions about the algorithm selection in order to solve the problem efficiently, along with rough performance measurements in the Evaluation section. Due to time limits, the third question can only be answered by providing rendering results, and make some of them comparable, and leave the judgment to the readers. Regarding the second question, more precise hypothesis should be made to provide an evaluation framework to measure the realism of the result, around the three keywords: focus, dynamic, real-time and realism.

The keyword "dynamic" mainly concerns with the flexibility and dynamics of the fluid simulation. Dynamical simulation can be interpreted in two ways, the first way can be seems as the dynamical nature of the set of parameters, which includes camera position, user interaction and physical properties etc. The other way to interpret it can be that, the simulations should not require pre-computations, for example pre-generated fluid data or view-dependent illumination. With these interpretation the result should be dynamic.

The keyword real-time is often used and easiest to understand, although it can sometimes have different meanings and be measured differently. There are two common measurements when dealing with total frame generation time, either frames per second(FPS) or milliseconds(ms). It is commonly accepted that if a sequence of images is shown above 30FPS, humans will start to perceive motion, this FPS is equivalent to a total rendering time of 35ms. This will be the definition used here to define real-time performance.

The third term "realism" is most difficult to define. Not only because different people will have different understanding of the term, but also because there are enormous factors affecting realism of the complicated problem. The definitions and measurements should be carefully selected and divided for each results generated by different parts of the pipeline. The definitions should also allow themselves to be verified to a certain degree with the results in the Evaluation section. Since the main problem in this thesis is complex in the way that it is a result of a combined system, the realism of the result depends naturally on each sub-problems, ie. the three main aspects in the solution: fluid dynamics, visual properties of smoke and of fire.

There are many factors affecting the realism of fluid simulation, however it can be mainly considered as details of fluid density and details of fluid motion. The simulations in the questions here start as a combustion process, thus the effects of combustion on the fluids are also important. With these factors in mind, the realism of fluid simulation can be defined as simulation details of fluid motion, fluid density and combustion. A high resolution advection scheme should provide the details for both motion and density, since such scheme usually implies higher-order advection and low-dissipation. However when simulating fluid motion, there are several other factors that play an important role. For example when simulating incompressible fluids, where the large portion of the rotational field of fluid velocity is dissipated, during the projection step, thus the accuracy of the pressure solver and methods to restore vorticity becomes important. Forces and other effects caused by a combustion process also affect the realism of the simulation, thus they should also be simulated.

The realism of visual smoke largely depend on the illuminations caused by external light source and emission of fire. To achieve realism, the light transportation between the elements should be simulated for the the smoke media. Global illumination of volumetric data is a way to compute such simulation, however globally simulating the transportation of light emitted by external light sources and fire is extremely difficult. The problem will however reduce significantly when only the one distant light is to be simulated globally. This will be assumed in this thesis, and only consider the illumination by fire locally.

One of the most important factor on the realism of fire is its color. Fire itself is a non-trivial media caused by complicated physics. Moreover the light waves emitted is then received by human which have a complicated way to interpret these information. To generate realistic fire, all these aspects should be taken in consideration and then simulated. The most significant effect of fire color is its relation to fire temperature, Figure 3.8a shows the well-known fire color gradient (red to blue) along the curve inside Planckian Locus.

As mentioned in Section 3 in [7], visual accuracy of such simulation problems usually implies physical accuracy. This intuition can also be verified by the application in computer graphics of the Vorticity Confinement method, originated in computational engineering [8]. This thesis should also take discussions on the underlying physics models, related with the hypothesis above, which will guide the evaluations. The definitions and requirements are summarized in a graph in Figure 1 in Appendices.

1.3 Limitations

The problem discussed in this thesis when all details are considered will be extremely complicated, thus huge simplifications will be made to satisfy the most of the hypothesis above. Below lists some of the limitations made in order to actually obtain a solution. The aspects will not be discussed in this thesis include but not limited to, static or dynamic boundary of complex shapes, chemical aspect of combustion, Compressible or super-sonic fluid dynamics(for accurate simulation of explosion), full spectrum simulation of fire, along with many other aspects not listed here. On the evaluation side, although statistical visual perception studies can be conducted to measure how well the hypothesis are satisfied, due to the lack of time and resources, only simple evaluation methods will be used, which includes demostrating figures and measure performances.

1.4 Implementation Environment

GPGPU stands for general purpose computing on GPU, which is popularly used in the scientific computation community. Aside from the popular API: CUDA introduced by Nvidia, ComputeShader is a part of DirectCompute introduced with DirectX 11 by Microsoft. ComputeShader is well suited for computations on uniform grids.

The issues of the book GPU Gems published by Nvidia, discuss the subjects of parallel computation with application in computer graphics, the programs in the book are mostly written in CUDA. [3] [9].

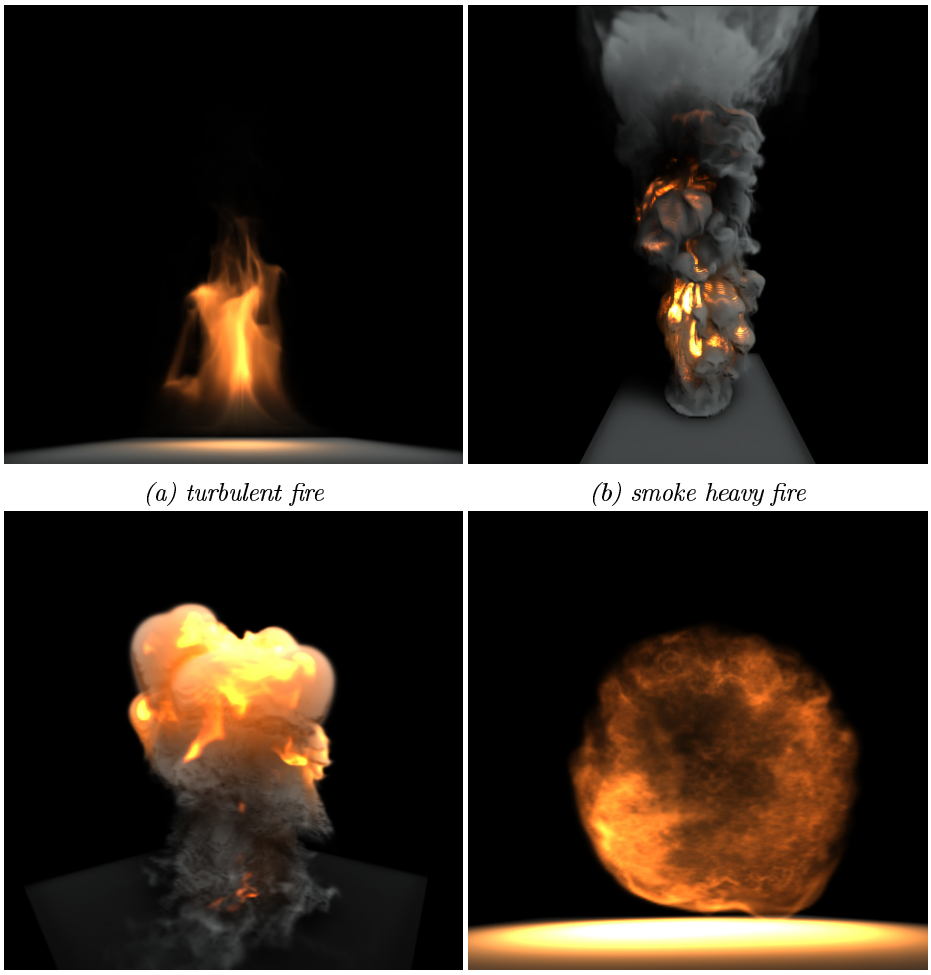
Unity is a game engine that was very popular amount the mobile game developers. However in the new and upgraded version of Unity 5, they emphasised heavily on physically based shading and baked global illumination features, which allowed the engine to be more competitive on the next-generation games. ComputeShader is well supported in Unity, which made the engine popular for implementing computer simulations. The implementations of this thesis be done in Unity, most programs are written in ComputeShader. The graphics pipeline in Unity will be adapted and modified to cope with the GPGPU pipeline here. More specifically, the final image generated by the simulation will be displayed as an overlay image, blended with the image of rendered scene, on the near plan of the camera. This will be realized with OpenGL codes.

1.5 Contribution

The main contributions of this thesis consist of the following:

- An overall pipeline for fire and smoke with a clear overview of the simulation and rendering process.
- A re-implementation and some analysis of a fluid solver [1][2][3][8][9][10][11][12][13], along with a simple extension for combustion modeling and its implementation for rough approximation of combustion processes.
- A re-implementation and some optimization in ComputeShader of the PDE based volumetric global illumination method [5], along with its application in real-time smoke rendering.
- A method and implementation for local hard shadow approximation, to compensate the high frequency lose when solving the lighting PDE with low resolution, using gradient based lighting technique.
- A practical approximation and re-implementation in ComputeShader for real-time physically based fire color reproduction, along with an algorithm as well as its application in real-time fire rendering.
- A re-implementation of a Poisson solver [11][5], and its application on local multiple scattering of light emitted by fire and other light sources, the same solver is also used in the fluid solver.
- A re-implementation of fast direct volume rendering method (section 1.2 in [14]) assuming the simple emission-absorption model, and demonstrated when combined with the grid based simulations, is applicable in real-time realistic fire and smoke rendering.

Due to the lack of time and the scale of the problem, peudocodes for most of the re-implementation are not presented, instead the theoretical solutions are discussed, some of the peudocodes as well as the ComputeShader codes (in the Listings after the Appendix) are provided. A great deal of graphical results will be demonstrated for evaluation purposes. All figures described without the "(Image source:)" mark, are generated by the (re-)implementation of the pipeline proposed in this thesis.



(a) turbulent fire *(b) smoke heavy fire*

(c) large scale explosion *(d) aero explosion*

Figure 1.2: A wide variety of combustion phenomena produced by the implementation of the pipeline in this thesis.

Chapter 2

Background

In computational physics Monte-Carlo methods is very general and thus widely used. However simulations using this method is usually too expansive to run in real-time in large scale. In such situation, grid based method will be a better fit, not only does it is easy to implement in parallel, but it also presents a straight forward structure for the descretization.

2.1 Grid Based Fluid Simulation

Joe Stam proposed the first unconditionally stable method for fast parallel simulation of the incompressible Navier-Stokes equations [1], this was a great process in the field of computer graphics, for visual simulation of nature phenomena. Figure 2.1 shows some of the early results of simulation and rendering of fluids in computer graphics [1]. Later Fedkiw et al. also discussed other important factors on visual simulation of smoke, i.e. buoyancy, gravity forces and the treatments of other physical quantities like temperature etc [2]. Vorticity Confinement was first introduced by John Steinhoff [8] and applied on engineering problems, it can be used to reduce vorticity dissipation caused by Helmholtz decomposition used to solve the equations, see Figure 2.2. An unconditionally stable modified MacCormack method was proposed by Selle et.al. [10], this method is fast and stable for descretizations using finite difference scheme.

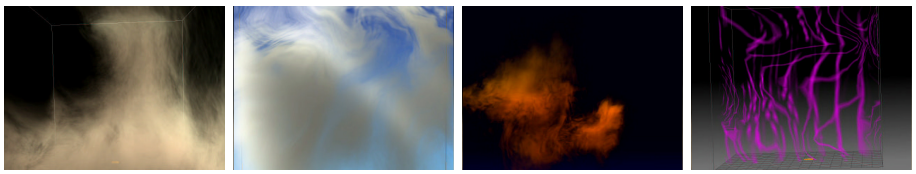


Figure 2.1: Some early results of cloud rendering. (Image source: [1].)

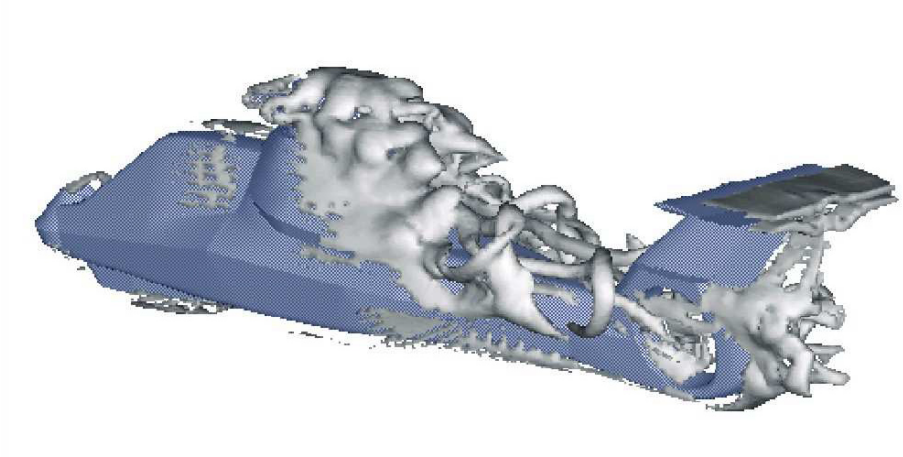


Figure 2.2: "Computed Vorticity Isosurfaces for the Comanche Fuselage with Rotating Shanks." (Image source: [8].)

2.2 Poisson Equations

Efficient solvers for solving large linear equations, is important for real-time simulation of physical phenomena, modeled by differential equations. Linear equation system solving is a well-studied problem, with a wide set of efficient methods. Straight forward methods like Gauss-Seidel method are easy to understand and can be easily implemented in parallel. The method was used by Zhang et.al. in [5], when solving the diffusion equation, to simulate multiple scattering of light. A similar parallel method, so called Red-Black Successive Over-Relaxation is used to solve the linear equation system formed in the simulation, details can be found in [11]. Multi-resolution analysis based methods or so called Multigrid methods, are the state of the art methods for solving these kind of equations [15], some results are shown in Figure 2.3

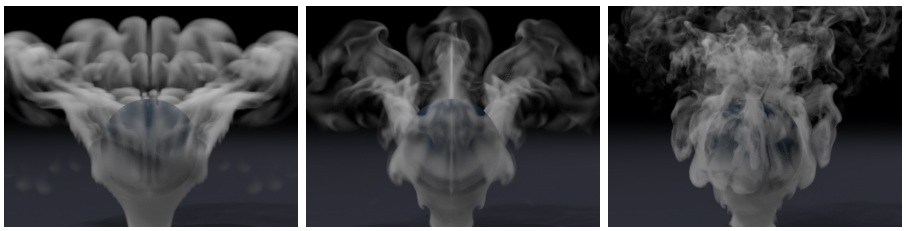


Figure 2.3: Smoke simulated and rendered under various resolution, low from left and high to the right using the Poisson solver proposed in [15]. (Image source: [15].)

2.3 Grid Based Radiance Estimation

The "two step approach" was proposed by the pioneer in computer graphics Kajiya [6], to speed up the otherwise expensive step when solving RTE, see Figure 2.4. Instead of using methods like Monte Carlo estimation, the "two step approach" uses computation grids in the first step to approximate light interaction between each grid cell. The data is then filtered and integrated during the second step. Since the most expansive computations occur in the first step, the grid approximation accelerates computation and made real-time simulation and rendering possible.

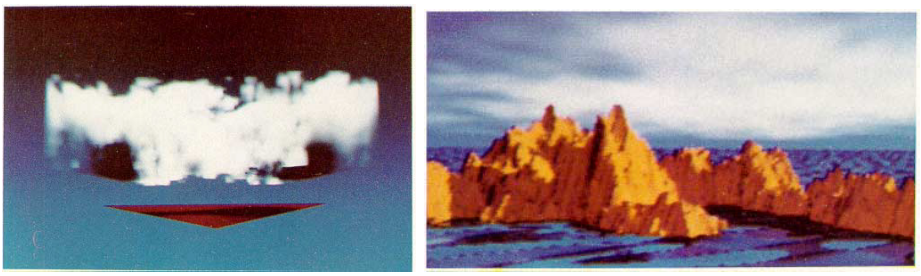
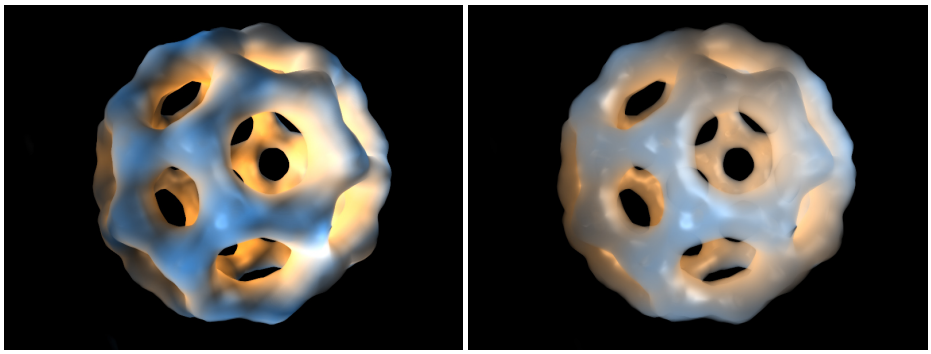


Figure 2.4: Ray traced cloud densities in [6] casts global shadows. (Image source: [6].)

2.3.1 Distant Light

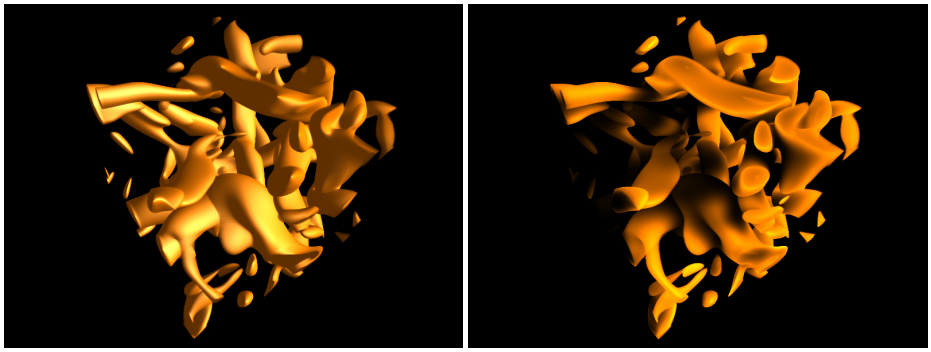
Global distant light shadow is important for realism, Zhang & Ma [5] applied the "two step approach", and introduced a PDE based method to efficiently simulate volumetric global light propagation. They obtained good global lighting and soft shadows with relatively low resolution grid, which allow them to simulate global propagation very fast, their results can be seem in Figure 2.7 and 2.6. A simple diffusion process is used to approximates the multiple scattering term in the equation. In very high or very low density smoke, good approximation can be achieved [6]. The diffusion step can be solved very fast with a multi-grid Poisson solver.

Figure 2.5: illuminated smoke. (Image source: [5].)



(a) no multiple scattering *(b) high multiple scattering*

Figure 2.6: Comparison of with and without multiple scattering [5].



(a) Phong shading *(b) global lighting*

Figure 2.7: Comparison of direct and global illumination. (Image source: [5].)

2.3.2 Fire Simulation and Rendering

Combustion dynamics play a important role in the simulation thus a simple combustion model should be used to capture the most basic properties of combustion, for example the various conditions for gas combustion, heat and soot releases. A simple trick can be used to imitate the effect of expanding volume caused by heat, this was mentioned in the article by Feldman et al [12], see Figure 2.10.

Nguyen et. al. discussed how to animate fire from both the dynamical and rendering aspect using level-set method [4], see Figure 2.9. Pegoraro & Parker described in deep about the underlaying theory in physics and chemistry, behind the rendering of realistic fire [7]. Again, when the smoke density is very high or very low, a simple method can be used to approximate the local multiple-scattering of fire emission, as for the simulation of distant light.

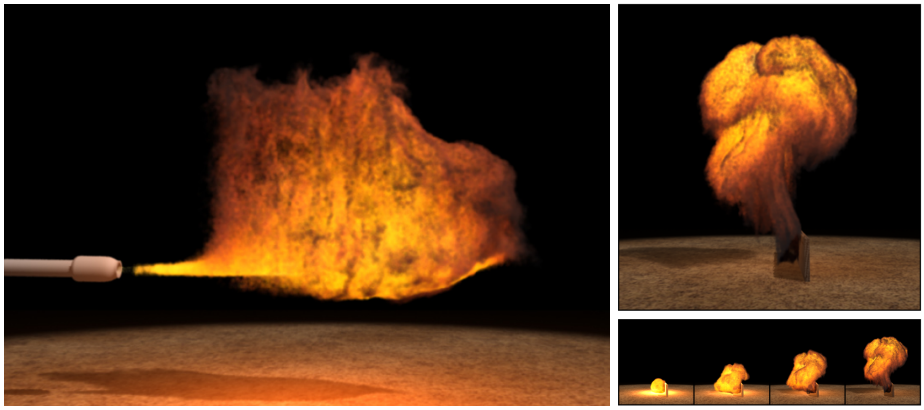


Figure 2.8: "Modeling the adaptation of the visual system allows for a faithful reproduction of the colors observed when looking at a flame under different lighting conditions." (Image source: [7].)

Figure 2.9: (a): Fire simulated with increasing gaseous release; (b): Campfire rendered through stochastic sampling. (Image source: [4].)

2.4 Volume Ray Tracing

The estimated per-voxel radiance stored in cell centers, can be treated as emission, which allow the the integration of the final volume, assuming a simple absorption-emission model, using ray tracing and pre-multiplied alpha-blending, which is extremely fast. Other approaches include stochastic ray tracing which are more accurate but expansive [16]. Deterministic ray tracing can be very simple and fast see Chapter 1 in [14]. Pegoraro et. al. discussed in deep about the spectral properties of fire and the theory to render it [7], see Figure 2.10.

Figure 2.10: "Modeling the adaptation of the visual system allows for a faithful reproduction of the colors observed when looking at a flame under different lighting conditions." (Image source: [7].)

Chapter 3

Theory and Implementation

In this chapter theoretical aspects of the underlying physics models, along with possible implementation strategies are discussed.

3.1 Discretization

The vector or scalar fields $\boldsymbol{v}$ in the simulation can be constrained to be within the interval of $[0, 1]^3$, then the volume can be discretized using finite difference scheme, into uniform grids where values are stored in center of each grid cells, formally:

$$\boldsymbol{v}(\boldsymbol{x}) \rightarrow \boldsymbol{V}(\boldsymbol{p}) = \boldsymbol{V}_{i,j,k} = \boldsymbol{V}(i \cdot \boldsymbol{d}_i, j \cdot \boldsymbol{d}_j, z \cdot \boldsymbol{d}_k)$$

where $\boldsymbol{p}$ is the nearest point of $N \cdot \boldsymbol{x}$ with coordinate (i, j, k) in the 3D lattice with size $[0, N]^3$, with integer spacing $\boldsymbol{d} = (d_i, d_j, d_k)$, i, j, k are integers and $(i, j, k) \in [0, N]^3$, N is the spatial resolution of the grid. Figure 3.1 shows a graphical representation of the discretization.

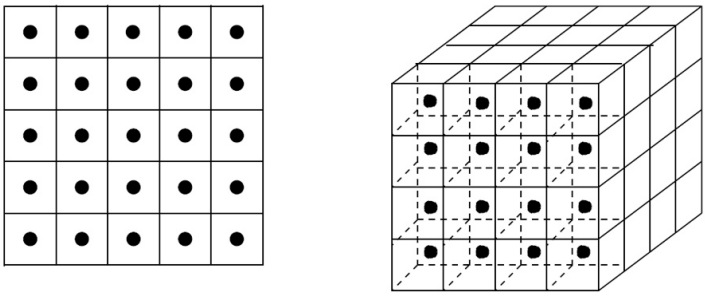


Figure 3.1: Graphical representation uniform grid. (Image source: [1].)

3.2 The Pipeline

The main problem faced here is the simulation and rendering complex combined phenomena of fire and smoke in real time. Due to this complexity, the problem is broken down into three parts and each part are solved one after another with the modules. The design consits of problem description definition, analysis and discussion of the underlying theory and carefully chosen algorithms.

The pipeline will be divided naturally into three modules, for each part of the problem, i.e. fluid simulation, physically based radiance estimation and rendering. The first module simulates the dynamics of fire and smoke, along with their physical properties. Most of the simulations involving solving the incompressible Navier-Stokes equations numerically. A simple combustion model will also be introduced to simulate the basic dynamics of a combustion process. Second and third module deal mostly with the rendering part of the problem, mainly involving solving the Radiative Transfer Equation(RTE). A critical scenario for the solutions, is the trade-off between capturing the visually important features and maintaining a real-time performance. Fire, fuel and smoke all have different complicated spectral properties, it is almost impossible to treat all the elements as a whole, thus different treatments will be designed for each part of them. To accelerate the design the "two step strategy" can be adapted, which basically means solving the most computational heavy radiance estimation parts using grids, and gather the radiance with a simple raytracer.

Figure 2 shows an overview of the pipeline as a process. The components of each module lists some of the important steps required to solve the problem associated with this module. An arrow between modules and components indicates the data flow. An detailed discussion of the fluid module can be found in 3.3.3.

3.3 Grid Based Fluid Simulation

3.3.1 The Incompressible Navier-Stokes Equations

In macroscopic scale, gaseous fuel, fire and smoke act similar to inviscid fluid. The Semi-Lagrangian scheme tracks fluid density at each cell center, and the dynamics of the fluid is governed by equations derived from the Lagrangian perspective. For simplicity incompressible fluids are assumed across the whole thesis, which implies the divergence of the velocity field to always be zero. The change of velocity field $\boldsymbol{u}$ over an infinitesimal time step ∂t can then be described by the incompressible Navier-Stokes equations:

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nabla \boldsymbol{p} + v\nabla^2 \boldsymbol{u} + \boldsymbol{f}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{I}) \tag{3.1}$$

$$\nabla \cdot \boldsymbol{u} = 0. \tag{3.2}$$

In eqa. (3.1) $\boldsymbol{p}$ is pressure, v is viscosity, $\boldsymbol{f}$ is the forces acting on the fluid, which is a function depend on $\boldsymbol{u}$ and $\boldsymbol{q}$, the quantity fields, $\boldsymbol{I}$ is the user interaction

term. The reason that the force is dependent on the velocity field is due to vorticity confinement, as described in 3.3.7.

Eqa. 3.2 ensures the fluid to be divergence-free. Since gas here is inviscid, the viscose diffusion term $v\nabla^2\boldsymbol{u}$ in the first equation can be omitted, which simplifies it to:

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla \boldsymbol{p} + \boldsymbol{f}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{I}). \tag{3.3}$$

By Helmholtz-Hodge Decomposition and the projection method proposed by Stam [1] equation (5.3) become:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{P}(-(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \boldsymbol{f}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{I})) \tag{3.4}$$

where $\boldsymbol{P}$ is the projection operator, explained in section 3.3.5. Equation 3.4 is the fundamental equation for most of the real-time fluid simulations, which can be solved stably by the four-step approch proposed by Stam [1]. The simulation is done by reimplement the "four step approch" in paralell using Compute-Shader. A good guide on paralell implementation of similar fluid solvers can be found in [3].

3.3.2 The Physical Quantities

To simulate and render fire and smoke, four relating physical quantities **smoke density, fuel density, temperature and reaction time** are simulated, together they should be able to capture the basic properties of fire and smoke in a simple combustion process. The quantities are usually represented as scalar fields, and composed into a single vector fields of four components. This vector field can be advected in a similar way using eqa. 3.3. However when incompressiblity was assumed, the coupling between temperature, reaction time and pressure is trivial, thus the pressure term is omitted in the equation, and simplify it to:

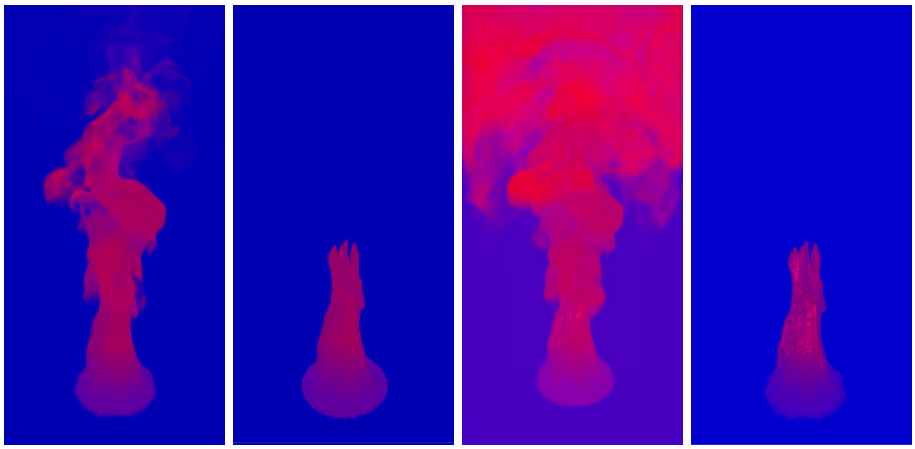
$$\boldsymbol{q} = (\boldsymbol{\rho}_{smoke}, \boldsymbol{\rho}_{fuel}, \boldsymbol{T}, \boldsymbol{\tau})$$

$$\frac{\partial \boldsymbol{q}}{\partial t} = \boldsymbol{C}(-(\boldsymbol{u}\cdot\nabla)\boldsymbol{q} + \boldsymbol{S}(\boldsymbol{I})) \tag{3.5}$$

where $(\boldsymbol{\rho}_{smoke}, \boldsymbol{\rho}_{fuel}, \boldsymbol{T}, \boldsymbol{\tau})$ is the composed vector field. $\boldsymbol{S}(\bullet)$ is the source term, represents the amount of fuel or smoke being injected into the field. A visualization of vector $\boldsymbol{q}$, is shown in Figure 3.2.

3.3.3 An Overview of the Fluid Pipeline

The basic steps in the fluid module consist of: Semi-Lagrangian advection, inject external source, inject force, and pressure projection. Two extra steps, MacCormack correction and vorticity confinement to reduce numerical dissipation. Another extra step used to approximate combustion dynamics is combustion simulation. The overall details of each step and the transition between each step variables, will be listed here, to give the reader an overview or better



(a) $\boldsymbol{\rho}_{smoke}$: smoke density *(b) $\boldsymbol{\rho}_{fuel}$: fuel density* *(c) $\boldsymbol{T}$: temperature* *(d) $\boldsymbol{\tau}$: reaction time*

Figure 3.2: A visualization of the four quantity fields, colored with a blue(min)-red(max) gradient. The ranges of values used in the figure are (from left to right): $[0, 1]$, $[0, 1]$, $[0, T_{max}]$, $[0, 1]$.

insight over what is going on in the complicated solution. Formally, at each time step t, for each position $\boldsymbol{x}$ in the field, the following step variables can be defined:

1. Advection:
$$\boldsymbol{u}_4(\boldsymbol{x}, t - \delta t) \rightarrow \boldsymbol{u}_0(\boldsymbol{x}, t)$$
$$\boldsymbol{u}_4(\boldsymbol{x}, t - \delta t),\ \boldsymbol{q}_3(\boldsymbol{x}, t - \delta t) \rightarrow \boldsymbol{q}_0(\boldsymbol{x}, t)$$

2. MacCormack Correction:
$$\boldsymbol{u}_4(\boldsymbol{x}, t - \delta t),\ \boldsymbol{u}_0(\boldsymbol{x}, t) \rightarrow \boldsymbol{u}_1(\boldsymbol{x}, t)$$
$$\boldsymbol{q}_3(\boldsymbol{x}, t - \delta t),\ \boldsymbol{q}_0(\boldsymbol{x}, t) \rightarrow \boldsymbol{q}_1(\boldsymbol{x}, t)$$

3. Inject External Source:
$$\boldsymbol{q}_1(\boldsymbol{x}, t) \rightarrow \boldsymbol{q}_2(\boldsymbol{x}, t)$$

4. Combustion Simulation:
$$\boldsymbol{q}_2(\boldsymbol{x}, t) \rightarrow \boldsymbol{C}(\boldsymbol{q}_2(\boldsymbol{x}, t)) = \boldsymbol{q}_3(\boldsymbol{x}, t)$$
$$\boldsymbol{q}_2(\boldsymbol{x}, t) \rightarrow \boldsymbol{C}_{div}(\boldsymbol{q}_2(\boldsymbol{x}, t))$$

5. Inject Force,
$$\boldsymbol{u}_1(\boldsymbol{x}, t) \rightarrow \boldsymbol{u}_2(\boldsymbol{x}, t)$$

6. Vorticity Confinement:

$$\boldsymbol{u}_2(\boldsymbol{x},t) \rightarrow \boldsymbol{u}_3(\boldsymbol{x},t)$$

7. Pressure Projection:

$$\boldsymbol{u}_3(\boldsymbol{x},t), \boldsymbol{C}_{div}(\boldsymbol{q}_2(\boldsymbol{x},t)) \rightarrow \boldsymbol{p}$$

$$\boldsymbol{u}_3(\boldsymbol{x},t),\ \boldsymbol{p} \rightarrow \boldsymbol{u}_4(\boldsymbol{x},t)$$

here "$\rightarrow$" indicates the solving direction, the LHS of the arrow listed the step variables required to solve for the variable on the RHS of the arrow. $\{\boldsymbol{C}(\bullet), \boldsymbol{C}_{div}(\bullet)\}$ will be described in section 3.3.6.

3.3.4 Advection and MacCormack Method

The advection term in eqa. 3.4 can be solved using the implicit advection scheme [1] [3]. The basic idea of this method is to trace a position back in time, and set evolve the velocity field accordingly, see Figure 3.3. Formally by computing:

$$\boldsymbol{u}_0(\boldsymbol{x},t) = \boldsymbol{u}_4(\boldsymbol{x} - \boldsymbol{u}_4(\boldsymbol{x}, t-\delta t)\delta t, t-\delta t)$$

$$\boldsymbol{q}_0(\boldsymbol{x},t) = \boldsymbol{q}_3(\boldsymbol{x} - \boldsymbol{u}_4(\boldsymbol{x}, t-\delta t)\delta t, t-\delta t).$$

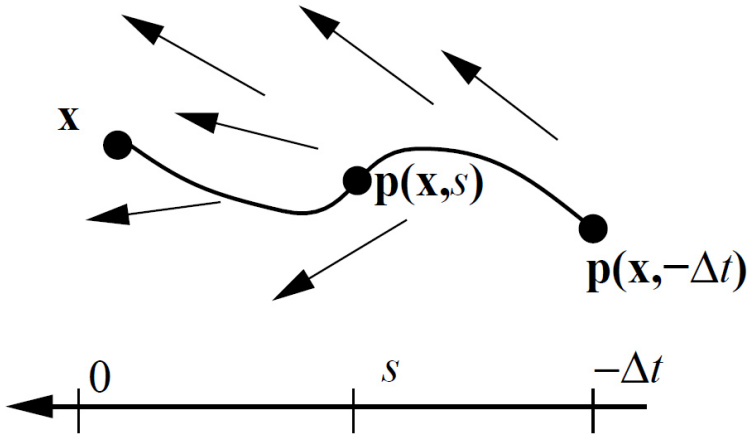


Figure 3.3: Advection scheme by tracing backwards in time. (Image source: [1].)

MacCormack method is a second-order finite difference method that is well suited for solving hyperbolic partial differential equations, which in the case here are the incompressible fluid equations. The modified MacCormack method [10] is an unconditionally stable MacCormack method, which is used in this thesis to reduce numerical dissipation caused by finite difference descretization scheme.

To apply the method with the pipeline in this thesis, same notations as in the article are used here. The following are set separately:

$$\phi^n = \boldsymbol{u}_4(\boldsymbol{x},\ t - \delta t), \phi^{n+1} = \boldsymbol{u}_1(\boldsymbol{x}, t)$$

$$\phi^n = \boldsymbol{q}_3(\boldsymbol{x}, t - \delta t), \phi^{n+1} = \boldsymbol{q}_1(\boldsymbol{x}, t)$$

and perform the correction step through:

$$\hat{\phi}^{n+1} = A(\phi^n)$$

$$\hat{\phi}^n = A^R(\hat{\phi}^{n+1})$$

$$\phi^{n+1} = \hat{\phi}^{n+1} + (\phi^n - \hat{\phi}^n)/2.$$

here $A(\bullet)$ is the Semi-Lagrangian advection scheme, and $\boldsymbol{u}_1$ is the corrected velocity and $\boldsymbol{q}_1$ is the corrected quantities, which is used for rendering later.

Since the physical quantities such as heat or density is assumed to be pressure independent, the computed velocity is the only data needed to advect them, thus a higher resolution grid can be used for the quantities, and interpolation is sufficient to advect it using the lower resolution velocity. A cubic B-Spline interpolation [17] can be implemented relatively efficient on GPU [13], and is used in this thesis to interpolate velocity, see implementation code of function cubicSample3D3() in listings of codes. There are other variants of the cubic interpolation, Figure 3.4 for example shows the curve of a monotonic cubic interpolation used in [2].

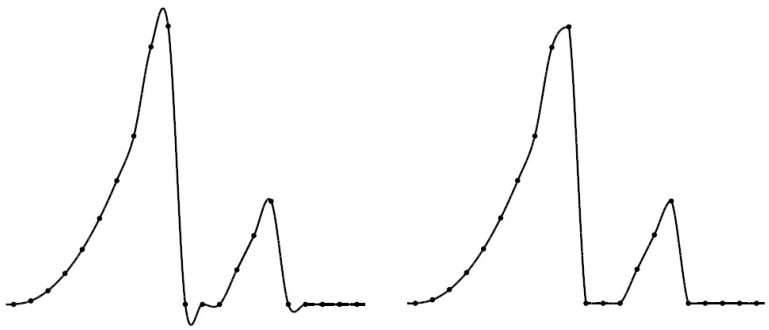


Figure 3.4: Monotonic cubic interpolation used in [2] to obtain higher order of advection accuracy. (Image source: [2].)

3.3.5 Ensure Incompressibility

As described in [1] eqa. 3.2 can be ensured by the use of the projection operator $\boldsymbol{P}$, which is defined as follows:

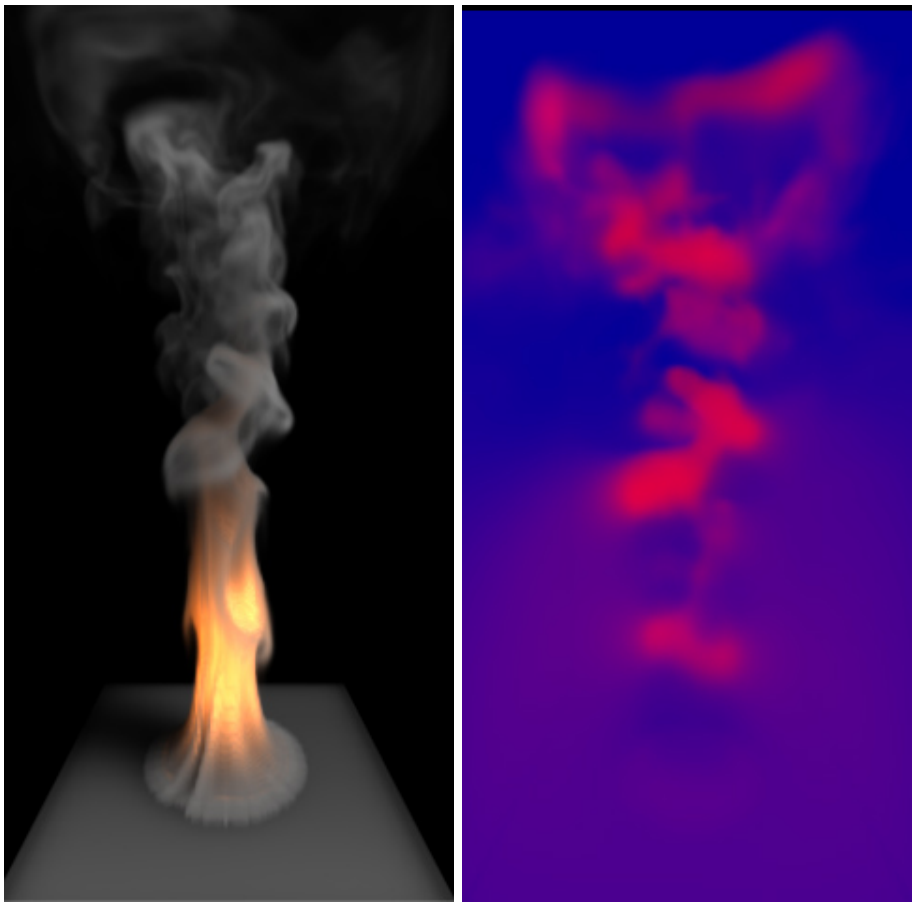
$$\nabla^2 \boldsymbol{p} = \nabla \cdot \boldsymbol{w} \tag{3.6}$$

$$\boldsymbol{P}\boldsymbol{w} = \boldsymbol{w} - \nabla \boldsymbol{p} \tag{3.7}$$

The pressure projection step is carried out by first computing the pressure $\boldsymbol{p}$ from $\boldsymbol{u}_2$ using eqa. 3.6, with a Poisson Solver, assuming the Neumann type boundary condition [1]. Then the pressure is "projected" into the velocity field simply through:

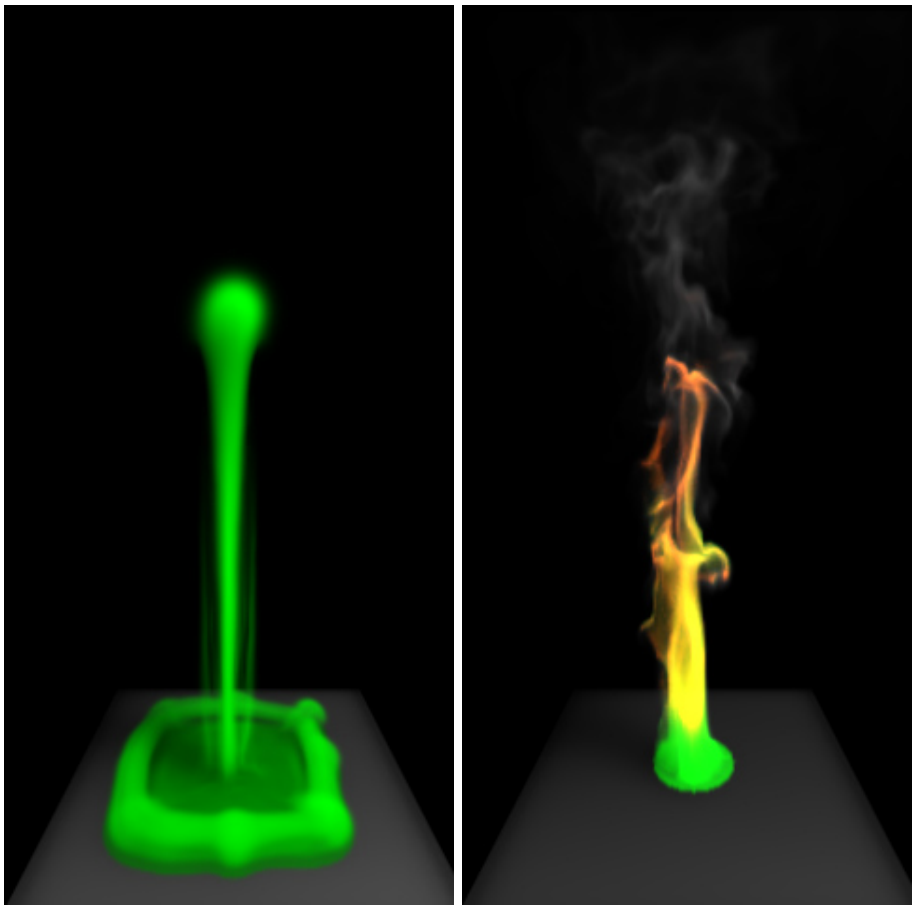
$$\boldsymbol{u}_4 = \boldsymbol{u}_3 - \boldsymbol{p}.$$

A computed pressure field is visualized in Figure 3.5.



(a) rising smoke *(b) pressure field*

Figure 3.5: (a): the rendered image. (b): the underlying pressure field visualized with blue-red gradient of range $[0, -0.5]$.



(a) fuel sinking *(b) fire rising*

Figure 3.6: Fuel(green) combusting, releasing more heat and soot, causes air to expand weakly, heat causes buoyancy force that lift up the fire and smoke.

3.3.6 Combustion Modeling, Source Term and Force Term

The force term $\boldsymbol{f}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{I})$ in eqa. 3.4 consist of three terms:

$$\boldsymbol{f}(\boldsymbol{u}, \boldsymbol{q}, \boldsymbol{I}) = \boldsymbol{f}_{extern}(\boldsymbol{I}) + \boldsymbol{f}_{buoy}(\boldsymbol{q}) + \boldsymbol{f}_{vort}(\boldsymbol{u})$$

where $\boldsymbol{f}_{extern}(\boldsymbol{I})$ is the user applied force, $\boldsymbol{f}_{buoy}$ the buoynacy force. Various formulas for approximating buoyancy force for heated gas can be found in [4][3][9]. The last term $\boldsymbol{f}_{vort}(\boldsymbol{u})$ is used by a vorticity confinement method that will be described in section 3.3.7. Here $\boldsymbol{I}$ represents the user interaction, contains a set of inputs such as mouse or keyboard states.

The source term $\boldsymbol{S}(\boldsymbol{I})$ in Eqa. (3.5), is used for injecting different sources, such as fuel, smoke or heat, it causes change in the quantities field. A Gaussian

function can be used to distribute the fuel evenly in the sphere around the 3D position of the mouse cursor, this will allow the injection of fuel to be smooth:

$$\boldsymbol{S}(\boldsymbol{I}) = C_{fueling}\frac{1}{C_{spread}^3(2\pi)^{\frac{2}{3}}}e^{-\left(\frac{|\boldsymbol{p}-\boldsymbol{c}|}{C_{spread}}\right)^2}$$

here $C_{fueling} > 0$ is the amount of fuel, $\boldsymbol{p}$ is a a position in the simulation volume, $\boldsymbol{c}$ is the mouse position projected onto some 2D plan inside the volume, all of these information can be extracted from $\boldsymbol{I}$.

The source term and force term is simply computed through:

$$\boldsymbol{q}_2(\boldsymbol{x}, t) = \boldsymbol{q}_1(\boldsymbol{x}, t) + \delta t \boldsymbol{S}(\boldsymbol{I}, \mathbf{x}, t)$$

$$\boldsymbol{u}_2(\boldsymbol{x}, t) = \boldsymbol{u}_1(\boldsymbol{x}, t - \delta t) + \delta t \boldsymbol{f}(\mathbf{x}, t).$$

The combustion extension $\{\boldsymbol{C}(\bullet), \boldsymbol{C}_{div}(\bullet)\}$ is introduced to cope with the combustion effects. Here $\boldsymbol{C}(\bullet)$ is a simple functional representing the change of the quantities caused by combustion, formally, it interacts with the pipeline as follow:

$$\boldsymbol{q}_3(\boldsymbol{x}, t) = \boldsymbol{C}(\boldsymbol{q}_2(\boldsymbol{x}, t)).$$

The combustion might also have an effect on the pressure field, where the release of gaseous products can cause the gas to expand, this will break the assumption of incompressibility. However, inspired by the trick used in [12], instead of solving the compressible pressure equations, the expansion can be handled simply by injecting the $\boldsymbol{C}_{div}(\boldsymbol{q})$ into the RHS of eqa. 3.6, to manually expand the pressure caused by air expansion:

$$\nabla^2 \boldsymbol{p} = \nabla \cdot \boldsymbol{u}_3 + \boldsymbol{C}_{div}(\boldsymbol{q}_2(\boldsymbol{x}, t)). \tag{3.8}$$

The extension $\{\boldsymbol{C}(\bullet), \boldsymbol{C}_{div}(\bullet)\}$ can be described with a simple process as below:

> For each cell in the simulation volume, if there is no reaction in this cell, however if there exists fuel and the temperature is high enough, then start the reaction in this cell. Otherwise there is already a reaction in this cell, in that case:
>
> 1. Consume some of the fuel in this cell, according to the Arrhenius formula.
> 2. If there is still fuel left in this cell then heat it up and release soot particles(smoke), otherwise terminate the reaction process.
> 3. Record the reaction(combustion) time.
>
> for simplicity heat will vary linearly with the change of fuel $d_{fuel} = C_{rate}$, although more sophisticated models can be used.

Algorithm 1 Combustion Simulation Pseudo Code

```
if no reaction in this cell then
    if fuel is enough and temperature reaches ignition point then
        start the combustion in this cell.
    end if
else                                   ▷ There is already a reaction in this cell.
    consume fuel in this cell.
    if there is fuel left in this cell then
        increase smoke density linearly to fuel consumption.
        increase temperature as an linear function of fuel consumption and
smoke release.
        add divergence linearly to fuel consumption to approximate release
of gaseous product.
        increase the reaction timer
    else                               ▷ There is no fuel at position x.
        reset reaction timer.
    end if
end if
```

This simple model can be implemented as an algorithm. For each cell, at time t do: An graphical visualization of the model can be seen in Figure 3.6b.

$\boldsymbol{C}_{div}(\bullet)$ approximates the volume expansion caused by release of gaseous products, formally:

$$\boldsymbol{C}_{div}(\boldsymbol{q}) = -C_{expand}\boldsymbol{I}_c$$

$$\boldsymbol{I}_c = \begin{cases} 1, \text{if there is reaction in this cell.} \\ 0, \text{otherwise.} \end{cases}$$

where $C_{expand} > 0$ is the expansion strength, and $\boldsymbol{q}.\boldsymbol{\tau}$ is the reaction time for this cell. However doing so could introduce instability and discontinuity, thus C_{expand} should be limited in order for the fluid to stay stable. It can be shown by experiment that for small values of C_{expand} the solution is relatively stable.

3.3.7 Vorticity Confinement

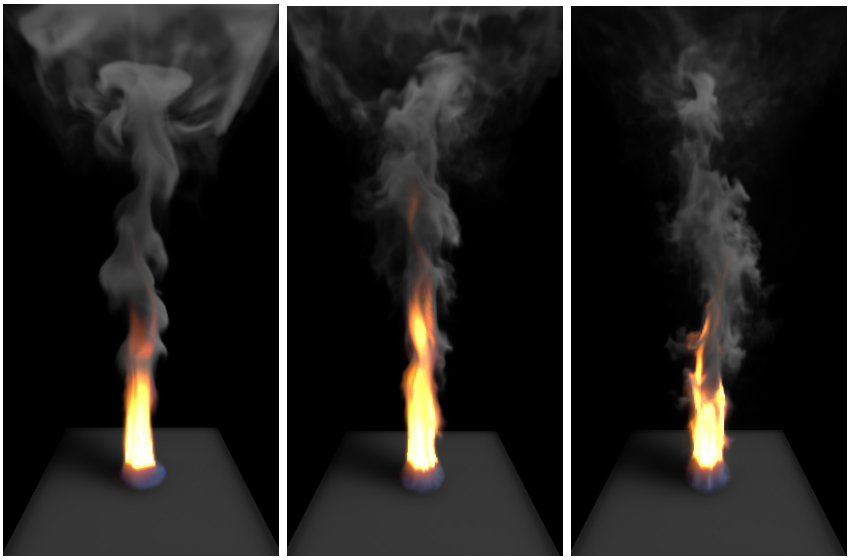
The projection step can cause dissipation in the voriticity of the velocity field. A simple and efficient method that can be used to add some of the lost vorticity back to the velocity field is called Vorticity Confinement, as first described by Fedkiw et. al. [2], which works fine on uniform discretized grid. The method first computes a confinement force based on the curl of the velocity field and add it back as a force:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{u}_2, \ \boldsymbol{\eta} = \nabla|\boldsymbol{\omega}|, \ \boldsymbol{n} = \frac{\boldsymbol{\eta}}{|\boldsymbol{\eta}|} \tag{3.9}$$

$$\boldsymbol{f}_{conf}(\boldsymbol{u}_2) = C_{vort}\ h\ (\boldsymbol{n} \times \boldsymbol{\omega}) \tag{3.10}$$

where $C_{vort} > 0$ is the strength of the confinement, and h is the grid size. Figure 3.7 shows a comparison of different C_{vort} values.

It is worth noting that to avoid false advection artifacts, when using high-resolution methods like MacCormack and/or vorticity confinement, a proper velocity damper should be applied on the advection scheme, however this issue will not be discussed here.



(a) $C_{vort} = 0$ *(b)* $C_{vort} = 0.25$ *(c)* $C_{vort} = 0.5$

Figure 3.7: A comparison of various vorticity confinement factor.

3.4 The Poisson Equations

By using forward difference in time and center difference in space to approximate the derivatives the Poisson equations can be descretized. The discretized equation results in a large sparse linear equation system. The system can then be solved iteratively using the Red Black Successive Over-Relaxation(RBSOR) Method, which is a parallel implementation of the SOR method. Here experiment shows that this method is stable when the over-relaxation coefficient $C_{SOR} < 2$. The result is also faster than the Gauss Seidel Method in most cases.

In general, Poisson equations have form:

$$\nabla^2 \boldsymbol{v} = \boldsymbol{b} \tag{3.11}$$

where $\boldsymbol{v}$ and $\boldsymbol{b}$ are some vector field. There is also a time dependent form of the equation, that is used for example in in [5]:

$$\frac{\partial \boldsymbol{v}}{\partial t} = s\nabla^2 \boldsymbol{v} \tag{3.12}$$

where $\boldsymbol{v}$ is some vector field that could for example represents energy density in a light field, and s is the scattering intensity. After discretization the equations become a large sparse linear equation with the unknown v_{xyz}, and b_{xyz} as the RHS in eqa. 3.11.

To parallize the SOR method on GPU, the Red Black modification is used, by dividing the computation grid into a chess board pattern, colored red and black. During an iteration, all the even numbered cells (red cells), are computed first using the formula below, after that, all the black cells are computed in the same iteration. This approach avoids the dependency of neighboring cells during parallel computation.

One cell with index (x, y, z) are computed as follow, at iteration n, let:

$$\boldsymbol{m} = \boldsymbol{v}^{n-1}_{x+1,y,z} + \boldsymbol{v}^{n-1}_{x-1,y,z} + \boldsymbol{v}^{n-1}_{x,y+1,z} + \boldsymbol{v}^{n-1}_{x,y-1,z} + \boldsymbol{v}^{n-1}_{x,y,z+1} + \boldsymbol{v}^{n-1}_{x,y,z-1}$$

then for the time independent eqa. 3.11:

$$\boldsymbol{v}^{n}_{x,y,z} = (1 - C_{SOR})\boldsymbol{v}^{n-1}_{x,y,z} + C_{SOR} \cdot \frac{h\boldsymbol{b}_{x,y,z} - s\boldsymbol{m}}{2s}$$

and for then for time dependent eqa. 3.12:

$$\boldsymbol{v}^{n}_{x,y,z} = (1 - C_{SOR})\boldsymbol{v}^{n-1}_{x,y,z} + C_{SOR} \cdot \frac{h\boldsymbol{v}^{n-1}_{x,y,z} + s\boldsymbol{m}}{h + 2s}$$

3.5 Grid Based Coefficient Estimation

3.5.1 Absorption Coefficient

When assuming that the absorption and emission in the simulation are caused and dominated by soot particles, then the complicated emission process [7] caused by fire can be reduced significantly. The absorption coefficient used in the rendering model is simply computed as below:

$$\boldsymbol{\sigma}(\boldsymbol{x}) = C^3_{size}\boldsymbol{\rho}_{\boldsymbol{smoke}}(\boldsymbol{x})$$

where particle size $C_{size} > 0$ is the size of the smoke particle.

3.5.2 Distant Light

In this thesis only a single external light source, i.e. a distant light will be assumed, which enables efficient simulations of its global illumination effects. According to [5] if for each frame, a constant light direction is assumed and by not considering the effects of multiple scattering, the radiance energy within each cell can be encode with a single value, instead of a spherical function. The radiance energy can then be modeled with a PDE, and computed with the upwind scheme. Multiple scattering effects can be simualted after the energy has been propagated through the volume, by simulating a diffusion process.

Global Light Propagation

Assume constant light direction and single scattering, by follow the two step approach used in [5], global light propagation can be modeled with the time varying light field equation [5]:

$$\frac{\partial \boldsymbol{E}_{light}}{\partial t} = -c\, \boldsymbol{l} \cdot \nabla \boldsymbol{E}_{light} - C_{absorb} \boldsymbol{\sigma} \boldsymbol{E}_{light} \tag{3.13}$$

where c is a constant, $C_{absorb} > 0$ is absorption intensity. This equation can be solved by the upwind scheme described in the same article, with the hybrid Dirichlet-Neumann boundary condition. To achieve faster convergence, the layered propagation technique [5] is utilized. The implementation can be done by shoot an energy ray, from the volume faces of Dirichlet boundary type along the light direction, to the volume face of the Neumann boundary type. Further acceleration may be achieved by considering the view dependent aspects of the problem and only propagate lights from and to the position of visible pixels.

Local Multiple-Scattering

When absorption coefficient of the target medium is very high or very low, multiple scattering within medium can be approximated fairly well with diffusion with the equation:

$$\frac{\partial \boldsymbol{E}_{light}}{\partial t} = s_{light} \nabla^2 \boldsymbol{E}_{light} \tag{3.14}$$

where $s_{light} > 0$ is the scattering coefficient for distant light. Equation (3.13) and (3.14) can be solved independently with the Dirichlet-Neumann boundary condition described in [5], the result of theirs scattering test can be seem in 2.6.

Light Field and Light Emission

After solving eqa. (3.13) and (3.14), a light field consisting of the intensities of light energy is computed as a approximation of the in-scattering term in the RTE for the distant light and the volume. Although this term can be used to solve the full RTE with the emission of fire accounted, the solution will be extremely expansive, unless uniform out-scattering is assumed. This assumption will simplify the full RTE and allow approximations to be obtained with the simple emission-absorption [14] model .

When solving the two equations, satisfying results can be achieved even with low resolution grids. However if low resolution grids are used, high frequency shadows will be smoothed out, which will affect the realism of high density smoke. To compromise this, a simple but efficient approach can be introduced, by using high resolution local gradient based shadow to add back the some of the lost high frequency shadows, formally let:

$$\boldsymbol{\sigma}_{shadow} = C_{shadow} \cdot max(-\nabla \boldsymbol{\sigma} \cdot \boldsymbol{l}, 0)$$

where $C_{shadow} > 0$ is a positive constant controlling the intensity of the shadow, and $\boldsymbol{l}$ is the light source direction. This term will be used to subtract the light field with below.

The computed radiance $\boldsymbol{E}_{light}$ can be seem as estimated emission coefficient for light from the distant light source. This is possible since isotropic scattering was assumed, where the phase function will become a constant, and the out-scattering intensity will only depended on the absorption $\boldsymbol{\sigma}$ in this cell. From the discussion above the out-scattering term can be modeled as isotropic emission, which in this cell can be computed as $\boldsymbol{\sigma}\boldsymbol{E}_{light}$, when considering shadow and light color the emission $\boldsymbol{L}_{light}$ become as follow:

$$\boldsymbol{L}_{light} = \boldsymbol{C}_{light}(1 - \boldsymbol{\sigma}_{shadow})\boldsymbol{C}_{fluid}\boldsymbol{\sigma}\boldsymbol{E}_{light}$$

where $\boldsymbol{C}_{light}$ is light color and $\boldsymbol{C}_{fluid}$ is fluid color, which can be determined by the fraction of fuels in one cell:

$$p = \frac{\boldsymbol{\rho}_{fuel}}{\boldsymbol{\rho}_{fuel} + \boldsymbol{\rho}_{smoke}}$$

$$\boldsymbol{C}_{fluid} = (1 - p)\boldsymbol{C}_{smoke} + p\boldsymbol{C}_{fuel}$$

where $\boldsymbol{C}_{fuel}$ is fuel color and $\boldsymbol{C}_{smoke}$ is smoke color.

3.5.3 Fire

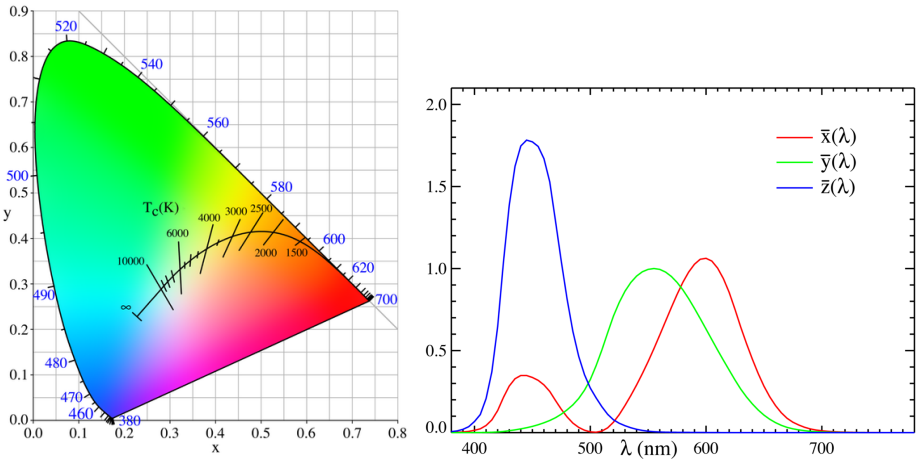
Fire is similar to plasma, the extreme heat causes electrons in the fuel and soot atoms to enter an excited state. When the electrons exit the excited state and jump down to an lower energy level, energy is released in form of electromagnetic radiation of photons. These photons then reach a standard observer, for example human eyes, the brain will then interpreters these light information and the perception of colors occurred. This complicated process can be hard to model, assumptions will be made to simplify the models to obtain a practical solution.

Black-Body Radiation and Fire Color

A simple way to render fire color is to sample the color-temperature gradient, see Figure 3.8a. However the intensity of fire emission is inaccurately modeled using this approach, which resulting low dynamic fire color. A sudden cut in the temperature spectrum introduces unnatural transition between fire and other media. To correctly reproduce the color of fire, the black-body radiations over the visible spectrum are convolved with the color matching function. The color matching function can be seem as a mathematical model of the physiological way of how humans perceive color information. More formally, it maps the spectral intensity values to color intensities as perceived by human.

Blackbody radiation is first computed in unit of spectral radiance ($\frac{J}{s \cdot m^2 \cdot sr \cdot nm}$) [7]:

$$\boldsymbol{B}(\boldsymbol{T}(\boldsymbol{x}), \lambda) = \frac{2hc^2}{\lambda^5} \cdot \frac{1}{e^{\frac{hc}{\lambda k \boldsymbol{T}(\boldsymbol{x})}} - 1} \tag{3.15}$$



(a) Planckian Locus *(b) Color Matching Functions*

Figure 3.8: Left: The curve inside the Planckian Locus indicates the color temperature gradient at different temperatures(Kelvin), which can be used to color fire. Right: CIE 1931 XYZ color matching functions. (Image source: [18], [19], data from [20] according to the author.

where T is temperature, λ is wavelength, c is the speed of light in vacuum, k is Boltzmann constant, h is Plank's constant. Then the radiation is integrated with the color matching function, formally for each cell at time t:

$$\boldsymbol{E}_{fire} = (\boldsymbol{E}_r, \boldsymbol{E}_g, \boldsymbol{E}_b)$$

is the emission at each of the XYZ value.

$$\boldsymbol{E}_{fire} = \int_{480}^{780} \boldsymbol{g}(\lambda) * \boldsymbol{\sigma_a}(\lambda) * \boldsymbol{B}(\boldsymbol{T}(\boldsymbol{x}), \lambda) \mathrm{d}\lambda$$

where the "*" operator is the component-wise product for each color channel, $\boldsymbol{B}()$ is the black-body radiation computed according to eqa. 3.15, $\boldsymbol{g}(\lambda) = (\overline{x}(\lambda), \overline{y}(\lambda), \overline{z}(\lambda))$ is the color matching function, a so called CIE standard observer, that represents an average human's cognitive perception of color, shown in Figure 3.8b. The color matching function can be constructed using tabular values recorded by experiment conducted by International Commission on Illumination(CIE). The tristimulus values are also made available by the Colour & Vision Research Laboratory at the Institute of Ophthalmology. The simulation program can be seem in the ComputeShader program "#pragma kernel ComputeCoefficients" in the Listings.

Fire Emission and Tone Mapping

Integrating radiation values multiplied by color matching function over the visible spectrum, generate color triples in XYZ color space. This triple can be converted into RGB color space by multiplying a XYZ-RGB conversing matrix. A CIE standard conversion matrix is used:

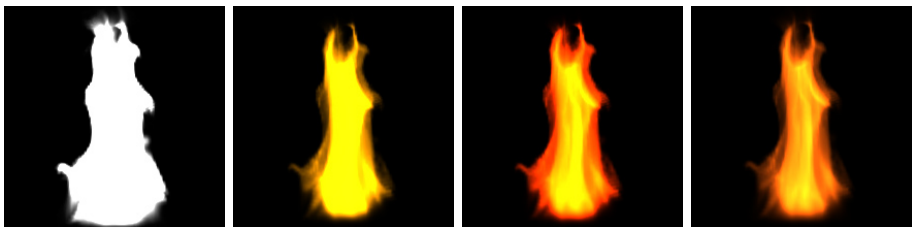
$$\begin{pmatrix} 2.3706743 & -0.9000405 & -0.4706338 \\ -0.5138850 & 1.4253036 & 0.0885814 \\ 0.0052982 & -0.0146949 & 1.0093968 \end{pmatrix}$$

This conversion usually generates high dynamic range colors, which need to be encoded into a correct dynamic range. The easiest way to do this is gamma compression. This is done below together with exposure adjustment:

$$\boldsymbol{L}'_{fire} = \left(2^{C_{exposure}} \boldsymbol{E}_{fire}\right)^{\frac{1}{C_{gamma}}} \tag{3.16}$$

where $C_{exposure}$ is the exposure value and $C_{gamma} > 0$ is the gamma value. The whole color reproduction process is visualized in Figure 3.9.

The color correction can be done as a post-processing process [7], in this thesis tone mapping was done before the rendering of final image, because large radiance values could cause bad condition numbers when solving the diffusion equation.



(a) black-body radiation *(b) tri-stimulus* *(c) RGB* *(d) tone mapping*

Figure 3.9: A visualization of the fire color reproduction process: (a) integrated black-body radiation only. (b) integrated black-body radiation multiplied by tri-stimulus values to get XYZ values. (c) XYZ values converted to RGB. (d) RGB color after tone mapping.

Local Multiple-Scattering

If the smoke density is very high or very low, one can approximate the local multiple scattering caused by emission of fire use a similar approach as in section 3.5.2. The local multiple-scattering of fire emission can be estimated with a

vector valued version of eqa. (3.14):

$$\frac{\partial \boldsymbol{L}'_{fire}}{\partial t} = s_{fire} \nabla^2 \boldsymbol{L}'_{fire} \tag{3.17}$$

with a small scattering value for high density smoke, and large for thin smoke. The final emission of fire is composed in a similar way as in section 3.5.2:

$$\boldsymbol{L}_{fire} = max(\boldsymbol{E}_{fire},\ \boldsymbol{\sigma} \boldsymbol{L}'_{fire})$$

3.6 Volume Ray Tracing

Above the radiative coefficients of distant light emission and fire emission have been computed, with the grid based approach. Now the radiance can be gathers with raytracing, using a proper reconstruction filter. Here a simple linear filter is used to reconstruct the volume radiance data, along with the simple emission-absorption optical model (section 1.2.1 in [14]). The raytracing step is done by following the same approach with alpha blending, by first introducing the opacity term [14]:

$$\boldsymbol{A}_i = 1 - e^{-\boldsymbol{\sigma}(\boldsymbol{r}_i)\Delta t}$$

where $\boldsymbol{r}_i$ is segment i of ray r. Also let:

$$\boldsymbol{C}_i = \boldsymbol{L}_i \Delta t$$

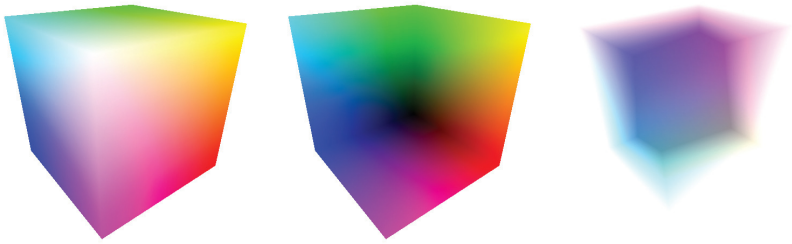
where the composed emission $\boldsymbol{L}_i$ is calculated by:

$$\boldsymbol{L}_i = max(\boldsymbol{L}_{fire}(\boldsymbol{r}_i), \boldsymbol{L}_{light}(\boldsymbol{r}_i))$$

then the front-to-back pre-multiplied alpha blending is evaluated recursively through:

$$C'_i = C'_{i-1} + (1 - A'_{i-1}) C_i$$
$$A'_i = A'_{i-1} + (1 - A'_{i-1}) A_i$$

The computation of ray $\boldsymbol{r}$ can be done as described in Chapter 2 in [14], by render the world coordinates of the volume faces as shown in Figure 3.10.



(a) front face *(b) back face* *(c) ray texture*

Figure 3.10: Volume face rendering for ray generation. (Image source: section 2.1 in [14].)

Chapter 4

Evaluation

4.1 Methodology

If there is enough time and resource, there are standard methods available for evaluation of the results. For example expansive simulations can be implemented to generate the ground truth data to compare the results with. Alternatively, real physical experiments could be carried out to recreate the true physical combustion phenomena as comparison. However in this thesis, evaluation is simply done by demonstrate result images rendered by the implementation of the pipeline, comparisons will be used and reflections to the hypothesis will be made.

4.2 Setup

The simulation is run on a desktop computer with 16GB DRAM, i7-5820K as CPU and GTX670 as GPU.

4.2.1 Scene

A same scene setup is used for most of the renderings. The scene consist of the following: at the bottom of the scene the ground is rendered by the setting of a high absorption value. Around the center of the ground, fuel is being injected in a circular shape, with some velocity pointed upwards. During the start of the injection, the fuel is being heat up for a short moment, the temperature is high enough to cause the fuel to combust into flames, the heat is then turned off and the high temperature flames will ignite the fuel injected below and start a continuous combustion. Various parameters for the simulation and rendering are set differently for each specific figures, the parameters could also have been be changed during run-time.

4.2.2 Base Resolution

All the physical fields, are stored by arrays of 2D textures. The resolution of the discretized grids are: $64 \times 128 \times 64$ for the velocity, $128 \times 256 \times 128$ for quantity, $128 \times 256 \times 128$ for the absorption coefficient, $128 \times 256 \times 128$ for fire emission, $32 \times 64 \times 32$ for local multiple-scattering of fire emission, $32 \times 64 \times 32$ for distant light illumination. The final ray tracing resolution is 550×550 with a maximum sampling steps of 150.

4.3 Results

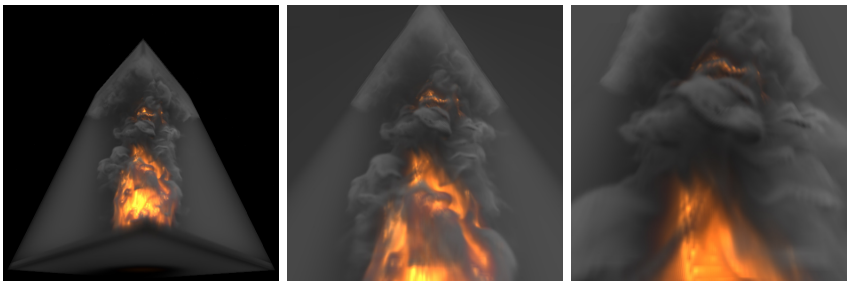
The observations and hypothesis made in section 1.1 and 1.2 will be treated as foundations to evaluate the results. Here rendered images are presented and referenced to the key points in the hypothesis. Beyond the images, real-time performance will be measured in 4.3.2.

4.3.1 Dynamic

Since the pipeline was built on physical models, it is flexible and able to produce a wide range of combustion phenomena. Figure 1.2. shows four different combustion situations for which the implementation is capable of. In Figure 1.2a there is the turbulent fire, where soot production is very low, and light emission is mostly caused by heated fuel. In Figure 1.2b soot production is very high, which becomes high density smoke and is illuminated by a single distant light source. In Figure 1.2c an explosion was simulated by setting a high divergence value see 3.3.6, the smoke was locally lit by the emission of fire. Figure 1.2d shows another type of explosion. Beyond internal parameters, camera viewing angle is also not constrained by pre-computations, thus dynamic views are possible, even rendering inside the volume is allowed, as seem in Figure 4.1 and Figure 4 in the Appendix. Due to no pre-computation, fuel can be injected and ignited into the anywhere in the scene at any time, which allows fire to propagate through the uncombusted fuel, as shown by Figure 3.6 and Figure 5 in the Appendix shows. These image demonstrated the dynamic nature of these combustion phenomena has been recreated.

4.3.2 Real-Time Performance

Another important highlight of the implementation is real-time performance. The performance of the implementation can be measured in milliseconds with the Unity's built-in profiler. The profiler is able to capture GPU dispatches and low-level rendering time. The performance can then be measured by switching on and off certain features in the pipeline, and record the time gap. Table 4.1 shows the recorded time for the listed features. The screen resolutions, iteration and sample counts remain unchanged while base resolution changes. Also note the MacCormack advection step performed poorly under double of the base resolution, this could have been caused by GPU memory overhead. As



(a) outside volume *(b) inside volume* *(c) even closer*

Figure 4.1: Images generated by zooming the camera inside and outside the volume.

it seems in the table, the implementation performs very well under base and half resolutions, where the total simulation and rendering time are respectively below 10ms and 30ms, which is more than safe to be considered as real-time. The double resolution performances demonstrated the performance scales poorly with the increase of resolutions.

Table 4.1: Performance Table

		Time(ms) under Resolution		
Module	Block	Half	Base	Double
Fluid Simulation	MacCormack Advection	< 1	4 ± 1	> 1000
	Pressure Solver (34 iterations)	1 ± 0.2	5 ± 1	50 ± 30
Coefficient Estimation	Fire Color Integral	< 1	2 ± 1	60 ± 10
	Local Multiple Scattering Fire (30 iterations)	< 1	1 ± 0.1	10 ± 5
	Global Distant Light Propagation	1 ± 0.1	5 ± 0.2	60 ± 10
	Distant Light Local Multiple Scattering (10 iterations)	< 1	1 ± 1	4 ± 2
Deterministic Ray Tracing	Rendering (150 samples per pixel)	< 1	2 ± 1	143 ± 8

4.3.3 Realism

The previous section demonstrated the pipeline performs well under the base resolution, in this section the quality of the simulation and rendering will be evaluated. As discussed in the Hypothesis section, the requirements for realism was distributed to different parts in the pipeline, i.e. fluid simulation, fire rendering and smoke rendering, see Figure 1, thus we evaluate each parts differently.

Fluid Simulation

According to the hypothesis, there are three factors affecting realism of fluid simulation, those are fluid motion, combustion details and density details.

Regarding to fluid motion, Figure 3.7, 1.1, 4.2 and 3.5 are able to demonstrate the accuracy of the fluid solver. Figure 3.7 as the result of the use of Vorticity Confinement method, shows some small scale turbulence was captured and added back to the velocity field. Figure 1.1 shows the pipeline is able to produce interesting vorticity details in a large scale explosion. Figure 4.2 shows the difference between first-order Semi-Lagrangian scheme and second-order MacCormack scheme for velocity advection. Figure 3.5 shows the computed pressure field, after a few (34) iteration with the Poisson solver, interesting features can be seem.

Figure 1.2, 3.2, 3.6 and 5 showed the implementation is able to capture most of the basic properties of simple combustion processes.

In 4.2, a increasing trend of accuracy can be seem in the images. The high-resolution details achieved with the second-order MacCormack scheme can also be seemed in 3.2, 4.3 and 1.2.

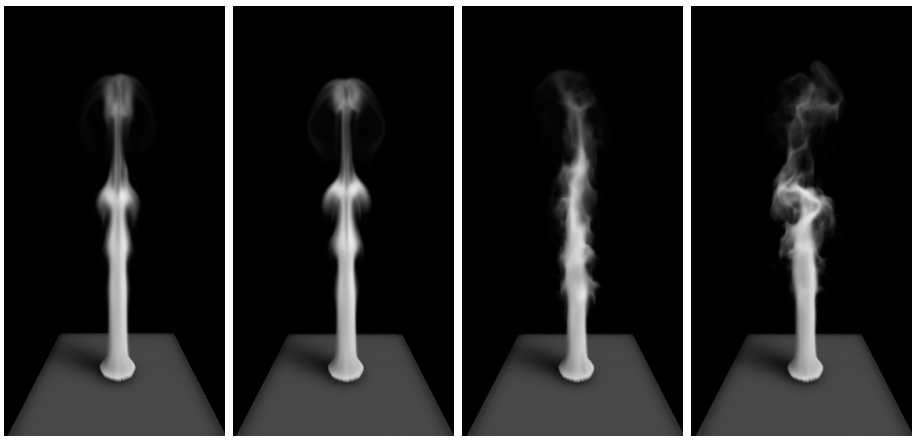
The figures mentioned above demonstrated even with all simplifications and trade-offs made to achieve real-time performance, the implementation was still able to produce fairly accurate fluid simulations.

Fire and Smoke Rendering

By simulating the light emission process and human perception, a more natural and realistic fire color was reproduced, as seem in Figure 3.9 and 4.5. The physically based approach is also capable of rendering accurate fire color as a function of fire temperature, as shown in Figure 4.3.

The local illumination of smoke caused by fire was approximated, as shown in Figure 4.4 the approximation is fairly good when the smoke density is very high.

By the assumption of single distant light source applied on the PDE based approach described in section 3.5.2, the implementation was able to produce high-quality global illumination of the smoke. A local shadow approach was also introduced in that section. Figure 3 in the Appendix shows a comparison of the approaches, an trend of increasing realism can be noticed from (a) to (d).



(a) MCC off *(b) MCC quantity* *(c) MCC velocity* *(d) MCC for both*

Figure 4.2: A comparison of using first order Semi-Lagrangian scheme vs second order MacCormack advection, with the same vorticity value 0.25. Images captured 5 second after simulation start. a) Semi-Lagrangian scheme is used for both quantity and velocity. b) MacCormack scheme is used for quantity only. c) MacCormack is used for velocity only. d) MacCormack is used for both velocity and quantity.

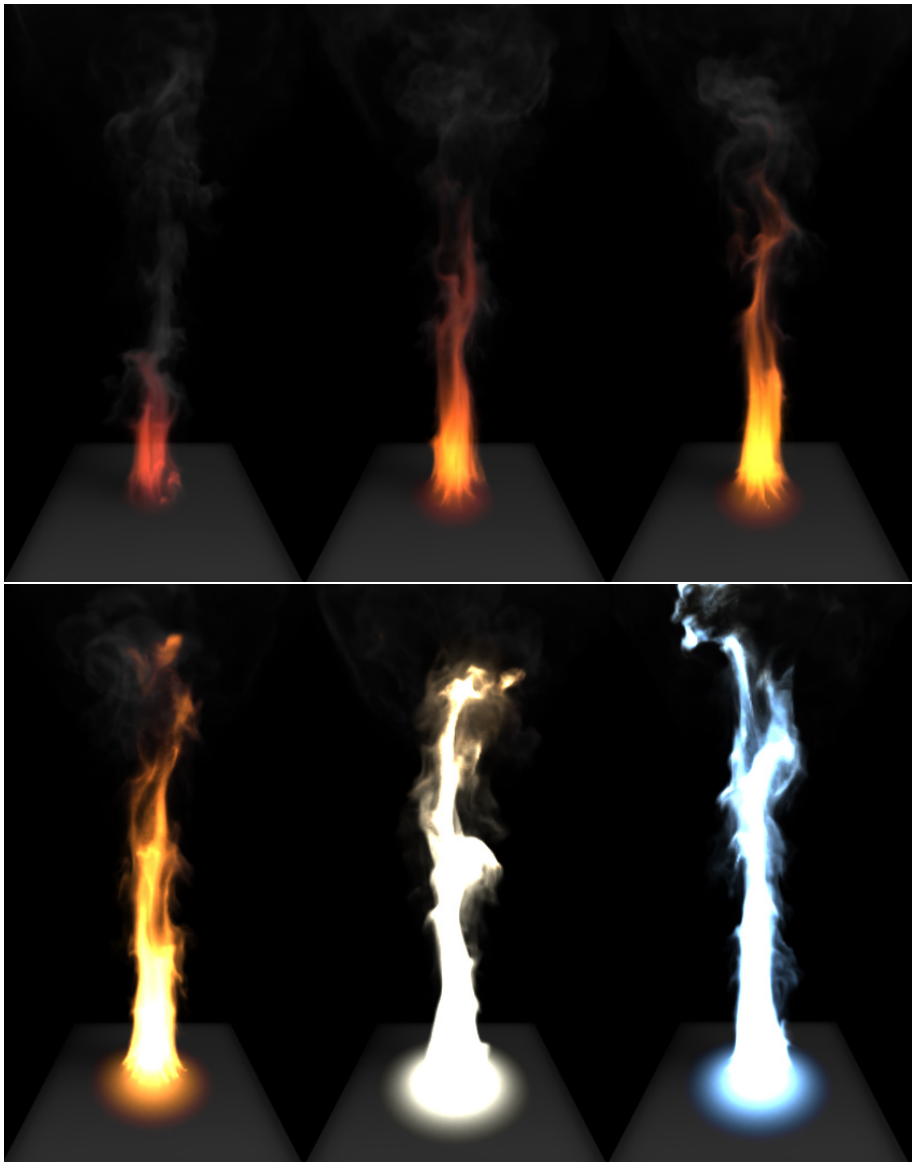
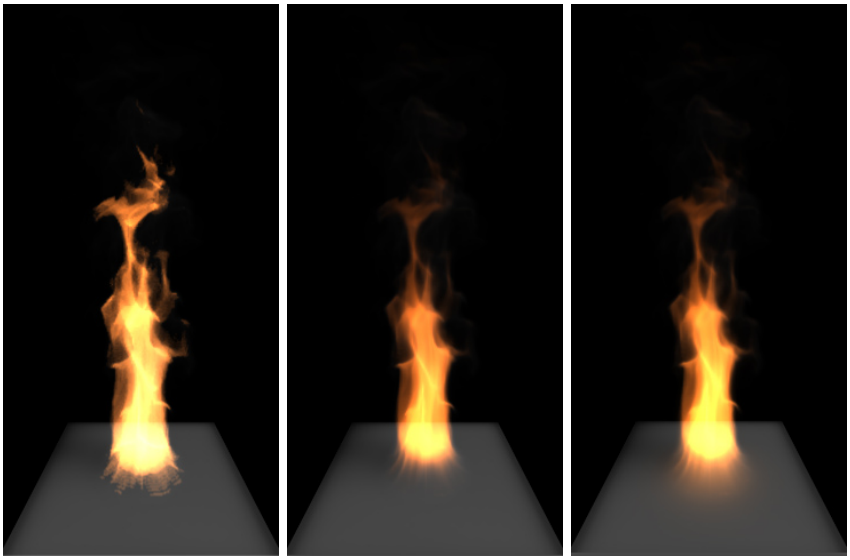


Figure 4.3: Fire simulation with various maximum combustion temperature(in Celsius), top: 700, 1000, 1300, bottom: 2600, 5000, 10000.

(a) without local multiple scattering *(b) with local multiple scattering*

Figure 4.4: Local multiple scattering comparison.



(a) gradient based color *(b) physically based color* *(c) multiple scattering*

Figure 4.5: Comparison of gradient and physically based fire color reproduction. (a) the color is relatively flat, brightness distributed more evenly everywhere, some dis-continuity occur at the root. (b) the color and brightness distributed more naturally across fire media.

Chapter 5

Discussion

5.1 Challenges and Improvements

During the progression of designing and implementing the pipeline, the complexity and scale of the problems leveled up, and new challenges revealed constantly. The processes described in the pipeline covered a wide area of scientific disciplines. To achieve real-time performance required even more careful thinking when selecting solution methods and underlying models. One may ask if there is an simpler way to recreate the beauty of nature, which hopefully was answered by section 4.

One of the major problems while using grid based simulation, is the coupling between these grids of different resolutions. When sampling at a high resolution, a grid with a very low resolution(lower than half of the high resolution grid) the so called Nyquist-frequency will be breached, and strong aliasing will appear, especially if the the gradient of the volume density is high. Usually one deal with this problem with smoothing filters, but When the resolution of the low grid is very low, not even a very good interpolation scheme can save the day. In [7], a non-linear sampling trajectory was mentioned, to simulate the bending of light rays caused by Fresnel effect, this distortion could have the potentials to reduce the problem aliasing.

The Poisson solvers was frequently used, which could easily become the bottle-neck in the performance. To replace it with a multigrid Poisson solver, will accelerate the entire implementation greatly.

One may also observe that the models used for illumination and rendering shares many similarities, this is because the two problems actually shared a common goal, that is to solve the RTE. The equation was divided and solved with multiple steps, in the aim for better performance. However a unified solution is to be investigated and its solution can be compared the ones here.

5.2 Value

After a long road travelled through the physically based models, one may ask whether it was worth the efforts, especially, when traditional sprites, particles based animations are able to quickly produce high-resolution graphics, which have the advantage over the expensive physically based animations. However the physically based approaches have a set of extreme flexible parameters, and a high quality real-time rendering that is ready for the complex environments in modern video games. Figures presented in the Results section shows high quality rendering of different combustion phenomena, most of them are impossible to achieve with the traditional approaches. Imaging a futuristic scene of some next-generation video game, where the players will be able to fly through real-time explosions with near cinematic realism, these possibilities is what makes the research of this topic worth its efforts.

Chapter 6

Conclusion

As evaluated and demonstrated, the implementation of the pipeline in this thesis is able to produce fairly realistic, dynamic fire and smoke simulation and rendering in real-time, which agreed with the hypothesis to a certain standard. This in turn showed that the hypothesis are fairly accurate to describe the dynamic and visual aspects of various combustion phenomena. The implementations were done in Unity, with the help of GPGPU and the power of modern GPU. Even thought The physical models were heavily simplified, and the rendering quality was lowered to achieve real-time performance, the realism was still maintained. This was due to careful selection of implementation and solution methods. The results showed possibilities for real-world applications, where renderings of high quality real-time combustion phenomena are required.

Bibliography

[1] J. Stam, "Stable fluids", in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, (New York, NY, USA), pp. 121–128, ACM Press/Addison-Wesley Publishing Co., 1999.

[2] R. Fedkiw, J. Stam, and H. W. Jensen, "Visual simulation of smoke", in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, (New York, NY, USA), pp. 15–22, ACM, 2001.

[3] M. Harris, "Fast fluid dynamics simulation on the gpu", in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, 2005.

[4] D. Q. Nguyen, R. Fedkiw, and H. W. Jensen, "Physically based modeling and animation of fire", in *ACM Transactions on Graphics (TOG)*, vol. 21, pp. 721–728, ACM, 2002.

[5] Y. Zhang and K.-L. Ma, "Fast global illumination for interactive volume visualization", in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pp. 55–62, ACM, 2013.

[6] J. T. Kajiya and B. P. Von Herzen, "Ray tracing volume densities", *SIGGRAPH Comput. Graph.*, vol. 18, pp. 165–174, Jan. 1984.

[7] V. Pegoraro and S. G. Parker, "Physically-based realistic fire rendering". in *NPH*, pp. 51–59, 2006.

[8] J. Steinhoff, N. Lynn, and L. Wang, "Computation of high reynolds number flows using vorticity confinement: I. formulation", *UTSI Preprint, University of Tennessee Space Institute, Tullahoma, TN*, 2005.

[9] K. Crane, I. Llamas, and S. Tariq, "Real-time simulation and rendering of 3d fluids", *GPU gems*, vol. 3, no. 1, 2007.

[10] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac, "An unconditionally stable maccormack method", *J. Sci. Comput.*, vol. 35, pp. 350–371, June 2008.

[11] P. N. Panchal, *Scientific Computation on Graphics Processing Unit using CUDA*. PhD thesis, INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, 2011.

[12] B. E. Feldman, J. F. O'brien, and O. Arikan, "Animating suspended particle explosions", in *ACM Transactions on Graphics (TOG)*, vol. 22, pp. 708–715, ACM, 2003.

[13] C. Sigg and M. Hadwiger, "Fast third-order texture filtering", *GPU gems*, vol. 2, pp. 313–329, 2005.

[14] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski, "Advanced illumination techniques for gpu-based volume raycasting", in *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, (New York, NY, USA), pp. 2:1–2:166, ACM, 2009.

[15] A. McAdams, E. Sifakis, and J. Teran, "A parallel multigrid poisson solver for fluids simulation on large grids", in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, (Aire-la-Ville, Switzerland, Switzerland), pp. 65–74, Eurographics Association, 2010.

[16] C. Kulla and M. Fajardo, "Importance sampling techniques for path tracing in participating media", *Comput. Graph. Forum*, vol. 31, pp. 1519–1528, June 2012.

[17] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens, "Efficient gpu-based texture interpolation using uniform b-splines", *Journal of Graphics Tools*, vol. 13, no. 4, pp. 61–69, 2008.

[18] en:User:PAR via Wikimedia Commons, "Planckian Locus". https://upload.wikimedia.org/wikipedia/commons/b/ba/PlanckianLocus.png, 2005. [accessed: 2018-01-16].

[19] en:User:PAR via Wikimedia Commons., "Color Matching Function". https://upload.wikimedia.org/wikipedia/commons/8/87/CIE1931_XYZCMF.png, 2005. [accessed: 2018-01-16].

[20] G. Wyszecki and W. Stiles, *Color Science Concepts and Methods, Quantitative Date and Formulae (2nd ed.)*. Wiley-Interscience, 2000.

Appendices

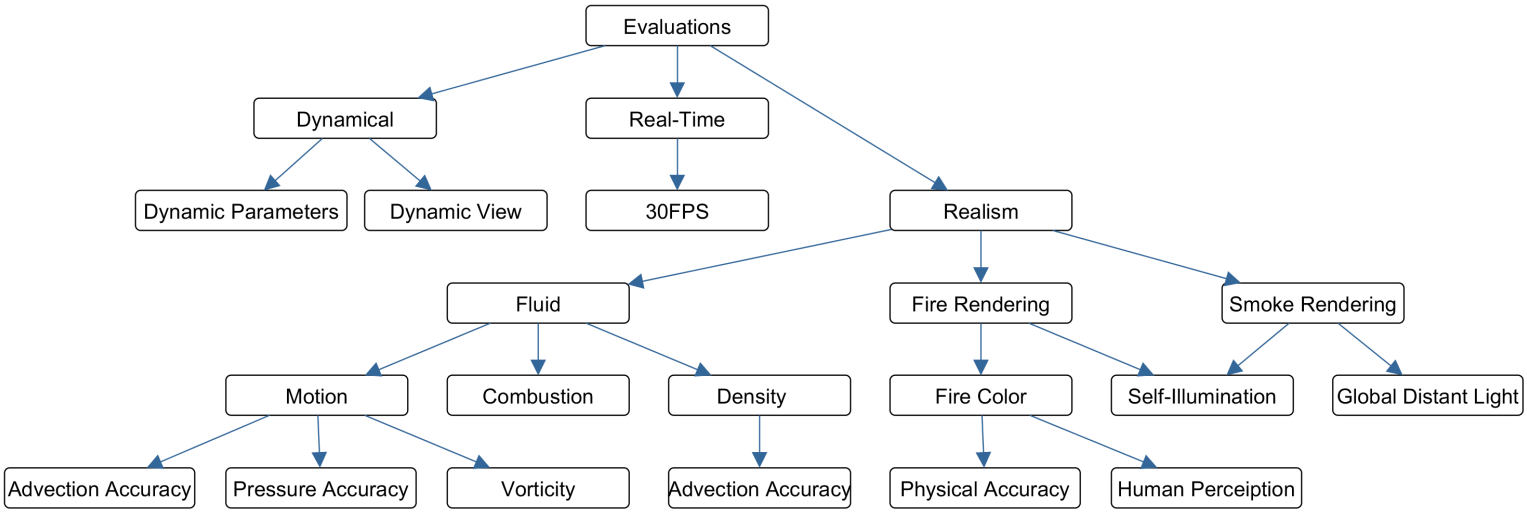


Figure 1: A requirement graph summarizes the important evaluation aspects.

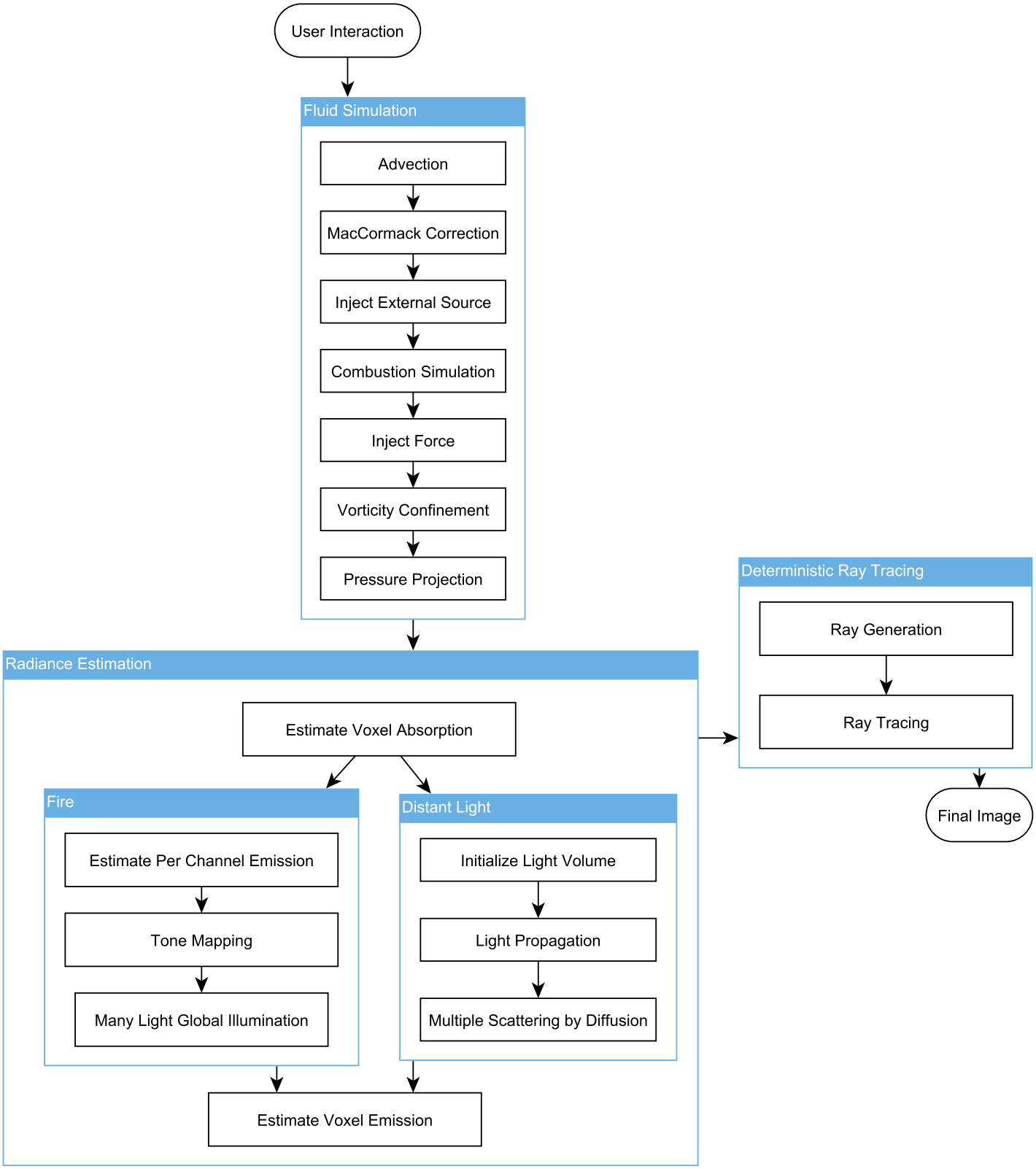
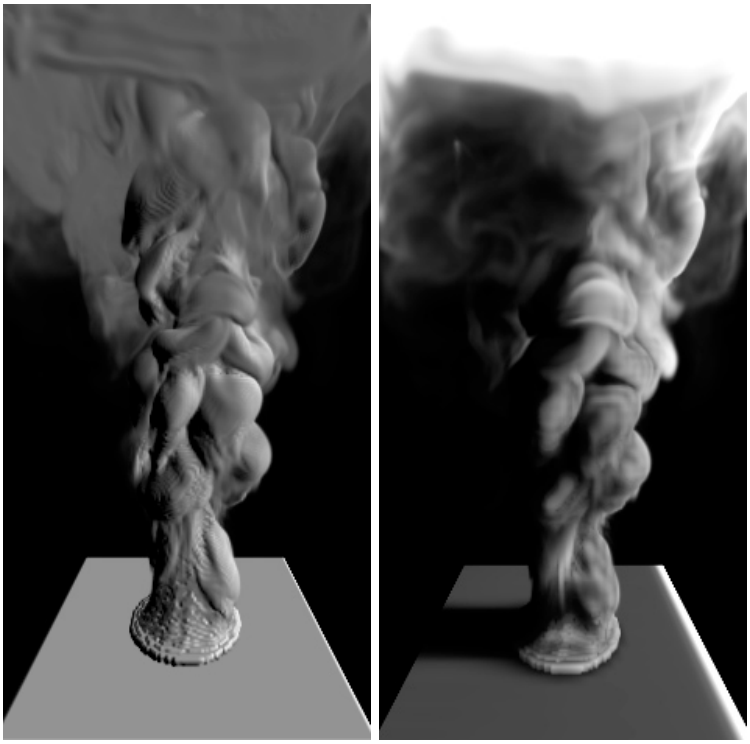
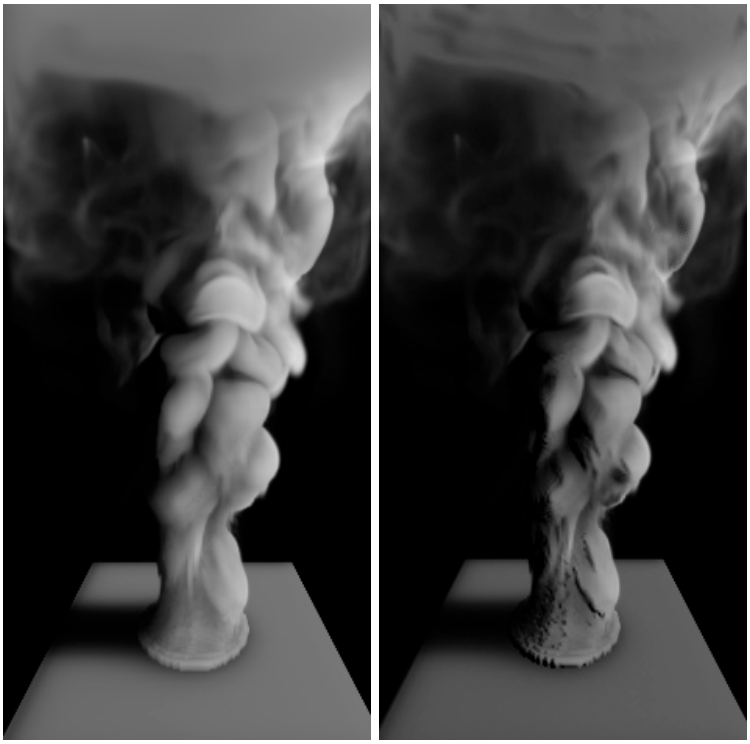


Figure 2: A process as a possible realization of the pipeline.



(a) Phong shading *(b) GDL*



(c) GDL + MS *(d) GDL + MS + direct shadow*

Figure 3: Lighting result comparison: (a) gradient based lighting only. (b) global distant light only. (c) global distant light with multiple scattering. (d) global distant light with multiple scattering, added local direct shadow.

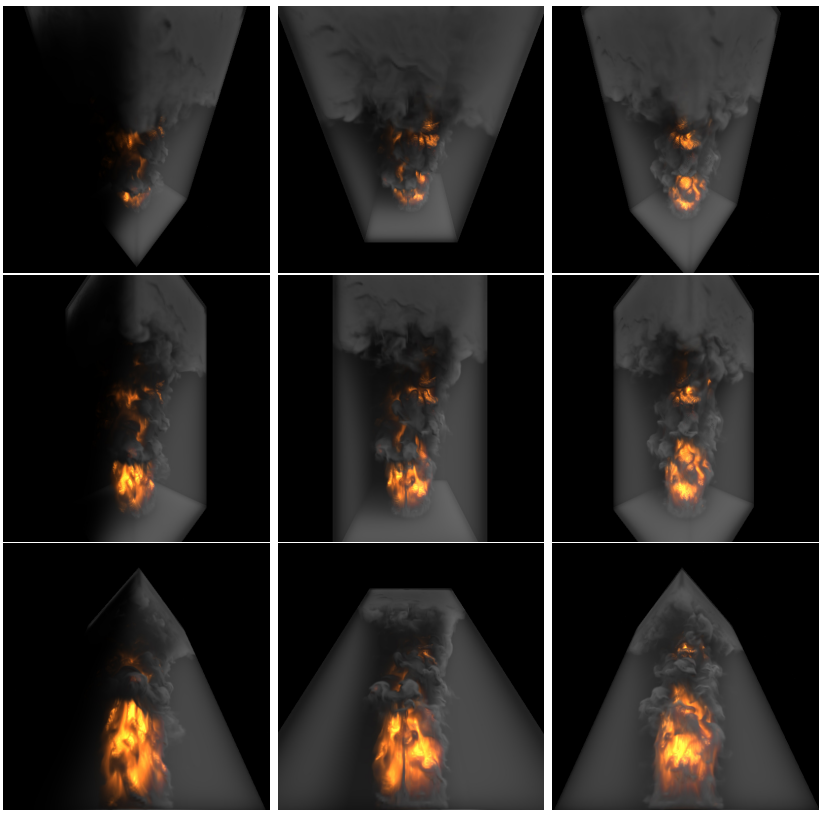


Figure 4: The fluid simulation is paused to demonstrate that camera view is fully interactive during any frame: the image position corresponding to the viewing angle, i.e. top-left image corresponds to top-left view of the volume etc.

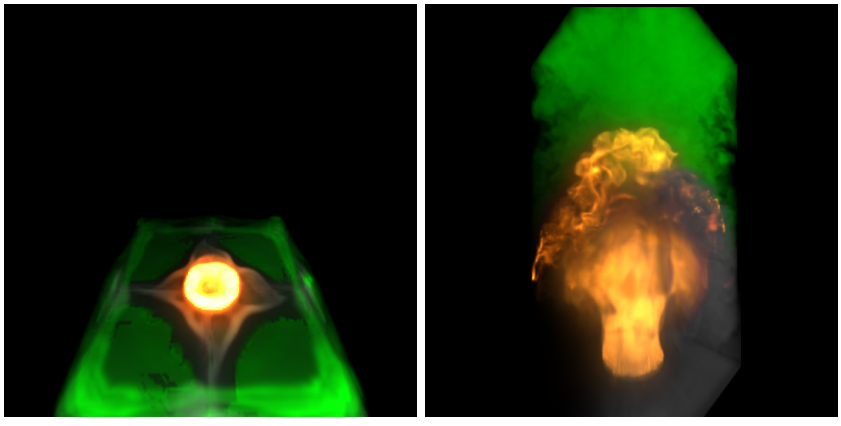


Figure 5: Ignition occurred half way through pre-injected fuel(green).

Listings

Here includes listings of ComputeShader codes, to give an intuitive view of how one possible implementation may look like.

```
//*************************************************************************************
//********************************** Constants ****************************************
// The Colour & Vision Research laboratory http://www.cvrl.org/
static const float tristimulus[40][3] =
        {3.77E-03, 4.15E-04, 1.85E-02,
        2.21E-02, 2.45E-03, 1.10E-01,
        8.95E-02, 9.08E-03, 4.51E-01,
        2.04E-01, 2.03E-02, 1.05E+00,
        2.92E-01, 3.32E-02, 1.55E+00,
        3.48E-01, 5.03E-02, 1.92E+00,
        3.22E-01, 6.47E-02, 1.85E+00,
        2.49E-01, 8.51E-02, 1.52E+00,
        1.81E-01, 1.30E-01, 1.25E+00,
        8.18E-02, 1.79E-01, 7.55E-01,
        2.08E-02, 2.38E-01, 4.10E-01,
        2.46E-03, 3.48E-01, 2.38E-01,
        1.56E-02, 5.20E-01, 1.18E-01,
        7.96E-02, 7.18E-01, 5.65E-02,
        1.82E-01, 8.58E-01, 2.44E-02,
        3.10E-01, 9.54E-01, 9.85E-03,
        4.49E-01, 9.89E-01, 3.79E-03,
        6.13E-01, 9.97E-01, 1.43E-03,
        7.97E-01, 9.73E-01, 5.45E-04,
        9.64E-01, 8.96E-01, 2.12E-04,
        1.11E+00, 8.12E-01, 8.49E-05,
        1.15E+00, 6.92E-01, 3.55E-05,
        1.08E+00, 5.58E-01, 1.55E-05,
        9.14E-01, 4.23E-01, 0.00E+00,
        6.92E-01, 2.98E-01, 0.00E+00,
        4.73E-01, 1.94E-01, 0.00E+00,
        3.00E-01, 1.19E-01, 0.00E+00,
        1.71E-01, 6.67E-02, 0.00E+00,
        9.22E-02, 3.56E-02, 0.00E+00,
        4.71E-02, 1.81E-02, 0.00E+00,
        2.26E-02, 8.66E-03, 0.00E+00,
        1.10E-02, 4.20E-03, 0.00E+00,
        5.21E-03, 2.00E-03, 0.00E+00,
        2.46E-03, 9.45E-04, 0.00E+00,
        1.19E-03, 4.56E-04, 0.00E+00,
        5.76E-04, 2.22E-04, 0.00E+00,
        2.86E-04, 1.10E-04, 0.00E+00,
        1.44E-04, 5.58E-05, 0.00E+00,
        7.35E-05, 2.86E-05, 0.00E+00,
        3.81E-05, 1.49E-05, 0.00E+00 };

// http://www.brucelindbloom.com
static const float3x3 xyz2rgb_cie =
        { 2.3706743, -0.9000405, -0.4706338,
        -0.5138850, 1.4253036, 0.0885814,
        0.0052982, -0.0146949, 1.0093968 };

static const float PlancksConstant = 6.6260693*1e-34;
static const float SpeedOfLight = 2.99792458*1e8;
static const float BoltzmannsConstant = 1.380658*1e-23;
//********************************** Constants ****************************************
//*************************************************************************************


// ComputeShader Function.
(TODO:CITE)
float3 cubicSample3D3( uniform Texture3D< float3 > v_, float3 gridRes, float3 samp )
{
        float3 x_ = samp*gridRes - 0.5;
        float3 a_ = abs(frac(x_));
```

```
        // Catmull-Rom-Spline
        float3 w0 = (-pow(a_,3) + 2*pow(a_,2) - a_)/2;
        float3 w1 = (3*pow(a_,3) - 5*pow(a_,2) + 2)/2;
        float3 w2 = (-3*pow(a_,3) + 4*pow(a_,2) + a_ )/2;
        float3 w3 = (pow(a_,3) - pow(a_,2))/2;

        //// B-Spline
        //float3 w0 = (-pow(a_,3) + 3*pow(a_,2) - 3*a_ + 1)/6;
        //float3 w1 = (3*pow(a_,3) - 6*pow(a_,2) + 4)/6;
        //float3 w2 = (-3*pow(a_,3) + 3*pow(a_,2) + 3*a_ + 1)/6;
        //float3 w3 = pow(a_,3)/6;

        float3 g0 = w0 + w1;
        float3 g1 = w2 + w3;
        float3 h0 = 1 - w1/max(g0, MACHINEEPSILON);
        float3 h1 = 1 + w3/max(g1, MACHINEEPSILON);

        float3 p_ = floor(x_);
        float3 p0 = (p_ - h0 + 0.5)/gridRes;
        float3 p1 = (p_ + h1 + 0.5)/gridRes;
        float3 v000 = v_.SampleLevel( _LinearClamp, p0, 0 );
        float3 v001 = v_.SampleLevel( _LinearClamp, float3(p0.x, p0.y, p1.z), 0 );
        float3 v010 = v_.SampleLevel( _LinearClamp, float3(p0.x, p1.y, p0.z), 0 );
        float3 v011 = v_.SampleLevel( _LinearClamp, float3(p0.x, p1.y, p1.z), 0 );
        float3 v100 = v_.SampleLevel( _LinearClamp, float3(p1.x, p0.y, p0.z), 0 );
        float3 v101 = v_.SampleLevel( _LinearClamp, float3(p1.x, p0.y, p1.z), 0 );
        float3 v110 = v_.SampleLevel( _LinearClamp, float3(p1.x, p1.y, p0.z), 0 );
        float3 v111 = v_.SampleLevel( _LinearClamp, p1, 0 );

        // Flat z-direction.
        float3 v00 = lerp(v000, v001, g1.z);
        float3 v01 = lerp(v010, v011, g1.z);
        float3 v10 = lerp(v100, v101, g1.z);
        float3 v11 = lerp(v110, v111, g1.z);

        // Flat y-direction.
        float3 v0 = lerp( v00, v01, g1.y );
        float3 v1 = lerp( v10, v11, g1.y );

        // Flat z-direction.
        return lerp( v0, v1, g1.x );
}

// ComputeShader function.
// "Ray-boundbox calculation Roettger 2003"
bool IntersectBox( float3 pos, float3 dir, float3 boxmin, float3 boxmax, out float3 pNear, out float3 pFar)
{
        // compute intersection of ray with all six bboxplanes
        float3 invR = 1.0 / dir;
        float3 tbot = invR * ( boxmin.xyz - pos );
        float3 ttop = invR * ( boxmax.xyz - pos );

        // re-order intersections to find smallest and largest on each axis
        float3 tmin = min (ttop, tbot);
        float3 tmax = max (ttop, tbot);

        // find the largest tmin and the smallest tmax
        float2 t0 = max (tmin.xx, tmin.yz);
        float tnear = max (t0.x, t0.y);
        t0 = min (tmax.xx, tmax.yz);
        float tfar = min (t0.x, t0.y);

        // The p survived now is pNear.
        pNear = pos + dir * max( 0, tnear );
        // The p survived now is pFar.
        pFar = pos + dir * tfar;

        return ( tfar > tnear ) && ( tfar >= 0 );
};


// Semi-Lagrangian scheme for Advect_VectorField.
#pragma kernel Advect_VectorField
[numthreads(8,8,8)]
//////////////////////////////////////////////////////////////
void Advect_VectorField( uint3 id3 : SV_DispatchThreadID )
//////////////////////////////////////////////////////////////
{
        // Get location to extract velocity
        float3 p_ = kernel2TextureCoords( id3, _c_Grid );

        // Sample velocity in world space.
        float3 vel = 0;
        if( any(_c_Grid > _c3_VelocityGrid) ) { // Interpolate velocity due to higher resolution quantities.
                //vel = _r_t3_velocity.SampleLevel( _LinearClamp, p_, 0 ); // Flow velocity.
                vel = cubicSample3D3( _r_t3_velocity, _c_Grid, p_);
        } else {
                vel = _r_t3_velocity.SampleLevel( _PointClamp, p_, 0 );
        }
        // The Sampler is in texture coords. samp_back is the new coords to sample the old quantity.
        float3 p_back = p_ -  (vel*_deltaTime - _volumeMovement)/_volumeDim;
```

```
        p_back = max( min(p_back, 1 - 1/_c_Grid), 1/_c_Grid );

        // Advect the vector field use the new coordinate.
        float4 field_new = _r_t3_field_vector.SampleLevel( _LinearClamp, p_back, 0 );

        bool boundary = any(id3 == 0) || any(int3(id3) == _c_Grid - 1); // Set values at boundary to zero.
        _w_t3_field_vector[ id3 ] = lerp( field_new, 0, boundary);
}


#pragma kernel MacCormackCorrection
[numthreads(8,8,8)]
//#####################################################################
void MacCormackCorrection( uint3 id3 : SV_DispatchThreadID )
//#####################################################################
{
        float3 p_ = kernel2TextureCoords( id3, _c_Grid );

        float3 vel = 0;
        if( any(_c_Grid > _c3_VelocityGrid) ) { // Interpolate velocity as in Semi-Lagrange Advection.
                vel = cubicSample3D3( _r_t3_velocity, _c_Grid, p_);
        } else {
                vel = _r_t3_velocity.SampleLevel( _PointClamp, p_, 0 );
        }

        float4 phi_n = _r_t3_phi_n[ id3 ]; // Field to be advected.
        float4 phi_hat_np = _r_t3_phi_hat_np[ id3 ]; // Field to be corrected.

        float3 res_inv = 1/_c_Grid;

        // Compute phi_hat_n by reverse advect vel_hat_np with Semi-Lagrange scheme reversely.
        // Reverse advect the field use the old velocity.
        float3 p_vel_hat_n = p_ + ( vel*_deltaTime - _volumeMovement )/_volumeDim;
        p_vel_hat_n = max( min(p_vel_hat_n, 1 - res_inv), res_inv );
        float4 phi_hat_n = _r_t3_phi_hat_np.SampleLevel( _LinearClamp, p_vel_hat_n, 0 );

        // Sampling pos where interpolation take place, the backtracking pos of not advected velocity with semi-lagrangean.
        float3 p_trace = p_ - ( vel*_deltaTime - _volumeMovement )/_volumeDim;

        //>>>>>> Flux limiter.
        // Find the grid lattice point which is f_k.
        float3 id3_trace = floor( p_trace*_c_Grid - 0.5 );
        p_trace = (id3_trace + 0.5)*res_inv;

        float4 phi_min;
        float4 phi_max;
        //Find the closest corner of the advected point.
        // The coords of the center point that is closest to where the interpolation occurs.
        id3_trace = floor( p_trace*_c_Grid + 0.5 );
        //Scale down the index to texture coordinates.
        p_trace = id3_trace*res_inv;
        //Find extrema at where the interpolation occur.
        float3 offset = 0;
        float4 phi_n_t = 0;
        phi_min = 1e20; phi_max = -1e20;
        [unroll] for( int i = 0; i <= 1; i++ ) {
                [unroll] for( int j = 0; j <= 1; j++ ) {
                        [unroll] for( int k = 0; k <= 1; k++ ) {
                                offset = 0.5*( 2*float3( i,j,k ) - 1 )*res_inv;
                                //p_trace = max( min(p_trace + offset, 1 - res_inv), res_inv );
                                phi_n_t = _r_t3_phi_n.SampleLevel( _PointClamp, p_trace + offset, 0 );
                                phi_min = min( phi_min, phi_n_t );
                                phi_max = max( phi_max, phi_n_t );
                        }
                }
        }

        // Clamp extrema produced by the correction step.
        // Turn off correction near boundary and density edges.
        bool boundary = any(id3 == 0) || any(int3(id3) == _c_Grid - 1);
        float gradNorm = _r_t3_gradientNorm.SampleLevel( _LinearClamp, p_, 0 );
        float4 phi_mcc = phi_hat_np + 0.5*( phi_n - phi_hat_n );
        float4 phi_np = lerp(
                                phi_hat_np, // Overshoot -> use Semi-Lagrange scheme.
                                phi_mcc, // Correct range -> use MCC scheme.
                                // Correct range -> 1, otherwise -> 0, disables at fluid edges
                                exp(-gradNorm*3)*((( phi_min <= phi_mcc ) && ( phi_mcc <= phi_max )))
                        );

        _w_t3_phi_np[ id3 ] = lerp( phi_np, 0, boundary);
}

#pragma kernel SetBoundary_DistantLight
[numthreads(8,8,8)]
//#####################################################################
void SetBoundary_DistantLight( uint3 id3 : SV_DispatchThreadID )
//#####################################################################
{
        VolumeLightSource lightSource = _b_lightSource[0];

        // Check if the cell is at the boundary
```

```
        bool isBoundary = cellIsBoundary( id3 );
        if( !isBoundary  ) {
                return; // Not boundary, return.
        }

        // If the cell is a boundary, use Equation (6)(7)
        float3 n_ = unitOutwardNormal( id3 );
        // Tiny tolerance to handle tiny num error.
        _rw_t3_energy[ id3 ] = (dot(n_, lightSource.dir) < -1e-3)*lightSource.intensity;
}

#pragma kernel LightPropagation
[numthreads(8,8,8)]
//#####################################################################
void LightPropagation( uint3 id3 : SV_DispatchThreadID )
//#####################################################################
{
        float3 samp = kernelToTextureCoords( id3, _c3_LightingGrid );

        VolumeLightSource lightSource = _b_lightSource[0];
        float3 lightDir = safeNormalize( lightSource.dir );
        float3 sampStart, sampDest;
        IntersectBox(
                                        samp - 2*lightDir, // Move the origin outside the box, to get pNear.
                                        lightDir,
                                        0,
                                        1,
                                        sampStart, sampDest );

        if( length(samp - sampStart) > _c_PropagationLayerThickness ) return; // Thickness for res: "64=0.2"

        float3 h_ = _volumeDim/_c3_LightingGrid; // Length of one voxel in (m)
        float3 ds = 0.1*_c_LightRayLength*h_*normalize( lightDir );
        float dLen = length(ds);
        //int maxSteps = ceil( length( (sampDest - samp)/h_texture ) );

        int3 id3_samp, id3_samp_old;
        float rho_old, opacity, rho_new;
        float3 d_rho; // Eqa. 18,19,20
        bool converged = false;
        int convergeCounter = 0;
        //for( int i = 0; i < maxSteps; i++ ) {
        (TODO: update code to ensure the boundary condition in here and in Poisson solver, before every iterations)
        while( all(samp > 0) && all(samp < 1) ) {
                id3_samp = textureToKernelCoords( samp, _c3_LightingGrid );

                if( any(id3_samp != id3_samp_old) ) {
                        rho_old = _rw_t3_energy[ id3_samp ];
                        //WARNING: not normalized final opacity.
                        opacity = _rw_t3_opacity_lowRes[ id3_samp ];

            (TODO: pre-compute d_rho, so its outside while)
                        d_rho = 0; // Eqa. 18,19,20
                        if( lightDir.x < 0 ) {
                                d_rho.x = _rw_t3_energy[ id3_samp + int3( 1,0,0 ) ] - rho_old;
                        } else {
                                d_rho.x = rho_old - _rw_t3_energy[ id3_samp - int3( 1,0,0 ) ];
                        }
                        if( lightDir.y < 0 ) {
                                d_rho.y = _rw_t3_energy[ id3_samp + int3( 0,1,0 ) ] - rho_old;
                        } else {
                                d_rho.y = rho_old - _rw_t3_energy[ id3_samp - int3( 0,1,0 ) ];
                        }
                        if( lightDir.z < 0 ) {
                                d_rho.z = _rw_t3_energy[ id3_samp + int3( 0,0,1 ) ] - rho_old;
                        } else {
                                d_rho.z = rho_old - _rw_t3_energy[ id3_samp - int3( 0,0,1 ) ];
                        }

                        rho_new = ( 1.0 - opacity )*( rho_old - 0.5*dot( lightDir, d_rho ) ); // Eqa. 16.

                        converged = (rho_new - rho_old) <= 0;
                        convergeCounter += converged ;
                        if( convergeCounter == _c_MaxConvergedVoxelPerRay ) {
                                return;
                        }
                        _rw_t3_energy[ id3_samp ] =
                                lerp(
                                        rho_new,
                                        rho_old,
                                        converged );
                }
                id3_samp_old = id3_samp;
                samp += ds;
        }
}


#pragma kernel ComputeCoefficients
[numthreads(8,8,8)]
//#####################################################################
```

```
void ComputeCoefficients( uint3 id3 : SV_DispatchThreadID )
//######################################################################
{
        float3 samp = kernelToTextureCoords( id3, _c3_HighResGrid );

        float4 quant = _r_t3_quantity.SampleLevel( _PointClamp, samp, 0 );

        if( id3.y <= 1 ) { // Render ground
                quant.r = 10;
        }

        float absorption_lambda = 0, B_ = 0, absorption = 0;
        float3 tris = 0;
        float3 fireEmission = 0;
        float lambda = 390;

        for( int i = 0; i < 40; i++ ) {
                absorption_lambda = computeAbsorption( quant, lambda );
                absorption = max( absorption, absorption_lambda );

                B_ = computeBlackbodyRadiation( quant.b, lambda );

                tris = float3( tristimulus[i] );
                fireEmission += tris*B_; // Color matching function integral.

                lambda += 10;
        }

        //absorption = computeAbsorption( quant, 700 );
        _w_t3_opacity[ id3 ] = 1 - exp( -absorption );

        fireEmission = max( 0, mul(xyz2rgb_cie, fireEmission) ); // Convert from XYZ to RGB and cut off negative values.
        //fireEmission = fireColor( quant.b ); // Comparison with color gradient.
        toneMapping( fireEmission, _c_Gamma_Fire, _c_FireExposure );
        _w_t3_fireEmission[ id3 ] = fireEmission;
}



// ComputeShader function.
float computeBlackbodyRadiation( float T_, float lambda )
{
        float B_ = 2*1e36*PlancksConstant*pow(SpeedOfLight,2)/
                                (pow(lambda,5)*(exp(
                                                (1e9*PlancksConstant*SpeedOfLight)/
                                                (lambda*BoltzmannsConstant*(T_ + 273.15))) - 1));

        return B_;
}

// ComputeShader function.
float computeAbsorption( float4 quant, float lambda )
{
        float absorption = pow( _c_VisualSootParticleSize,3 )*(quant.r + quant.g*_fuelColor.a*0.1);
        //absorption += quant.g*exp(-_debugFloat*1e-18/BoltzmannsConstant/(quant.b + 273.15))*pow(lambda,4)*
        //                                (exp(
        //                                        (1e9*PlancksConstant*SpeedOfLight)/
        //                                        (lambda*BoltzmannsConstant*(quant.b + 273.15))) - 1)/SpeedOfLight;
        // Fuel absorption. \alpha 1.5 better than 1.39 proposed by "[Hot54]".
        //absorption +=  0.0001*pow( lambda, 2 )
        //        /
        //quant.b*quant.a + (0.2*pow( lambda, 2 )/quant.b*quant.a + (quant.a == 0))*fuelDensity;

        return absorption;
}

// ComputeShader function.
void toneMapping( inout float3 L_, float gamma, float exposure )
{
        float pixelLumin = dot( L_, float3( 0.299, 0.587, 0.114 ) );

        L_ *= pow( 2, 30*exposure ); // Adjust exposure.
        L_ = pow( abs(L_), 1/max(gamma, MACHINEEPSILON) ); // Adjust gamma.
}

#pragma kernel CompositeCoefficients
[numthreads(8,8,8)]
//######################################################################
void CompositeCoefficients( uint3 id3 : SV_DispatchThreadID )
//######################################################################
{
        float3 samp = kernelToTextureCoords( id3, _c3_HighResGrid );
        float alpha = _r_t3_opacity.SampleLevel( _PointClamp, samp, 0 );
        float4 quant = _r_t3_quantity.SampleLevel( _PointClamp, samp, 0 );
        float3 fluidColor =
                lerp(
                        _smokeColor.rgb,
                        _fuelColor.rgb,
                        _fuelColor.a*0.01*quant.g
                                /
                        max( MACHINEEPSILON, (_fuelColor.a*0.01*quant.g + _smokeColor.a*quant.r) ) );
```

```
        // ############## Lighting emission ####################
        float3 lighting = _r_t3_lighting.SampleLevel( _LinearClamp, samp, 0 );
        float highFreqShadow = _r_t3_directShadow.SampleLevel( _LinearClamp, samp, 0 );
        float3 lightFieldEmission = lighting*(1 - highFreqShadow)*fluidColor*alpha;

        // ############## Fire emission ####################
        float3 fireScattering = _r_t3_fireScattering.SampleLevel( _LinearClamp, samp, 0 );
        toneMapping( fireScattering, _c_Gamma_Fire_Scattering, _c_FireExposure_Scattering );
        float3 fireEmission = _r_t3_fireEmission.SampleLevel( _PointClamp, samp, 0 );
        float3 fireEmissionScattering = alpha*max( fireEmission, fireScattering ); // Render Mode 1: Smoke Intensive
        //float3 fireEmissionScattering = max( fireEmission , fireScattering ) ; // Render Mode 2: Glow Intensive

        float3 emission = max(fireEmissionScattering, lightFieldEmission);
        _w_t3_radiativeCoeff[ id3 ] = float4( emission, alpha );
}

// ComputeShader code.
float4 alphaBlend( float4 res, float4 coeff, float stepLen )
{
        //WARNING: not normalized final opacity.
        res.rgb += (1 - res.a)*coeff.rgb;
        res.a += (1 - res.a)*coeff.a;
        return res;
}

#pragma kernel VolumetricRayCasting
[numthreads(8,8,1)]
//#####################################################################
void VolumetricRayCasting( uint2 id2 : SV_DispatchThreadID )
//#####################################################################
{
        float4 ray = _r_t2_ray[ id2 ];

        if( ray.a == 0 ) {
                _w_t2_finalImage[ id2 ] = 0;
                return;
        }
        float absorption = 0, expExtinction = 0;
        float3 emission = 0;
        float shadow = 0, fireLayer = 0;
        float4 finalPixel = 0; // front-to-Back.
        float3 samp_start = world2TextureCoords( _r_t2_rayStartPos[ id2 ].xyz );
        float3 samp = samp_start;
        float3 samp_step = ray.xyz*ray.a/_c_MaxSteps/_volumeDim;
        float samp_stepLen = ray.a/_c_MaxSteps; // Mul by 10 to make it more close to 1 as max?
        float4 coeff = 0;
        // Front-to-Back alpha blending.
        for( int i = 0; i < _c_MaxSteps; i++ ) {
                //coeff = _r_t3_radiativeCoeff.SampleLevel( _PointClamp, samp, 0 );
                coeff = _r_t3_radiativeCoeff.SampleLevel( _LinearClamp, samp, 0 );
                //coeff = cubicSample3D4( _r_t3_radiativeCoeff, _c3_HighResGrid, samp);
                //coeff *= (coeff.a >= _debugFloat && coeff.a <= _debugFloat + 0.001);

                finalPixel = alphaBlend( finalPixel, coeff, samp_stepLen );

                if( finalPixel.a >= 1 ) {
                        break;
                }
                samp += samp_step;
        }

        _w_t2_finalImage[ id2 ] = finalPixel;

}


//#####################################################################################################################
//############################## Start of Poisson Solver  #############################################################
//#####################################################################################################################
// Use this method if the target is a vector field.
public void sorSolveVector(
        // The first texture is the right hand side (constant term) of the linear system
        ref RenderTexture rhs,
        // The fist texture in the array is the unknows in the linear system,
        // which is also the restult of solving. Rest 2 textures are first and second temperary textures.
        ref RenderTexture[] t3_target,
        int iterNum, // Iteration number for the solving.
        float diffuseFactor,
        float sorFactor,
        bool timeDependent,
        int[] gridRes,
        Vector3 volumeDim
 ) {
        _PoissonSolver.SetFloats( "_c_gridRes", new float[] { gridRes[0], gridRes[1], gridRes[2] } );
        _PoissonSolver.SetVector ( "_volumeDim", new Vector4( volumeDim.x, volumeDim.y, volumeDim.z, 0 )
);
        _PoissonSolver.SetFloat( "_c_diffuseFactor", diffuseFactor );
        _PoissonSolver.SetFloat( "_c_sorFactor", sorFactor );
        if( timeDependent ) {
                _PoissonSolver.SetFloat( "_c_timeDependent", 1 );
```

```
        } else {
                _PoissonSolver.SetFloat( "_c_timeDependent", 0 );
        }

        _PoissonSolver.SetTexture(  _h_SORSolveVector, "_r_t3_rhs", rhs ); //IN: Right Hand Side term of the linear system.
        for ( int i = 1; i <= iterNum; i++ )
        {
                _PoissonSolver.SetFloat( "_updateEven", 1 ); // Updating all even(x+y+z ) voxels.
                _PoissonSolver.SetTexture(  _h_SORSolveVector, "_r_t3_temp", t3_target[1] ); //IN
                _PoissonSolver.SetTexture(  _h_SORSolveVector, "_w_t3_res", t3_target[0] ); //OUT
                _PoissonSolver.SetTexture(  _h_SORSolveVector, "_w_t3_temp", t3_target[2] ); //OUT
                _PoissonSolver.Dispatch(  _h_SORSolveVector, gridRes[0]/8, gridRes[1]/8, gridRes[2]/8 );

                _PoissonSolver.SetFloat( "_updateEven", 0 ); // Updating all odd(x+y+z ) voxels.
                _PoissonSolver.SetTexture(  _h_SORSolveVector, "_r_t3_temp", t3_target[2] ); //IN
                _PoissonSolver.SetTexture(  _h_SORSolveVector, "_w_t3_res", t3_target[0] ); //OUT
                _PoissonSolver.SetTexture(  _h_SORSolveVector, "_w_t3_temp", t3_target[1] ); //OUT
                _PoissonSolver.Dispatch(  _h_SORSolveVector, gridRes[0]/8, gridRes[1]/8, gridRes[2]/8 );
        }
}

// Solve Poisson Equation with Red-Black SOR method.
// xp/xm: updated neighbours, xc: the old center cell.

// Solving using RBSOR on a vector field.
#pragma kernel SORSolveVector
[numthreads(8,4,8)]
//#####################################################################
void SORSolveVector( uint3 id3 : SV_DispatchThreadID )
//#####################################################################
{
        //if( any(id3 == 0) || any(id3 == _c_gridRes - 1) ) return;

        // Shift the id3.y up one cell depend on id3.x and id3.z, to create an red-black patterned volume.
        uint rb = abs( !_updateEven - fmod( id3.x + id3.z,2 ) );
        id3.y = 2*id3.y + rb;

        uint3 idle3 = id3;
        idle3.y += rb;
        // Pass on the values at the cells that is not updating during this pass.
        _w_t3_res[ idle3 ] = _r_t3_temp[ idle3 ];

        float3 samp = kernel2TextureCoords( id3, _c_gridRes );

        //// Take care boundaries(TODO).
        //bool isBoundary = ( id3.x == 0 ) || ( float(id3.x) == (_c_gridRes.x - 1));
        //isBoundary = isBoundary || ( id3.y == 0 ) || ( float(id3.y) == (_c_gridRes.y - 1));
        //isBoundary = isBoundary || ( id3.z == 0 ) || ( float(id3.z) == (_c_gridRes.z - 1));
        //if( isBoundary ) {
        // Store a copy of the updated value to be used at next update as a read texture.
        //      _w_t3_temp[ id3 ] = _r_t3_rhs.SampleLevel( _PointClamp, samp, 0 );
        //      _w_t3_res[ id3 ] = _r_t3_rhs.SampleLevel( _PointClamp, samp, 0 );
        //      return;
        //}

        float3 h = _volumeDim/_c_gridRes;
        h = pow( h,2 );

        float4 xc = _r_t3_temp[ id3 ];
        float4 xp = _r_t3_temp[ id3 + uint3( 1,0,0 ) ];
        float4 yp = _r_t3_temp[ id3 + uint3( 0,1,0 ) ];
        float4 zp = _r_t3_temp[ id3 + uint3( 0,0,1 ) ];
        float4 xm = _r_t3_temp[ id3 - uint3( 1,0,0 ) ];
        float4 ym = _r_t3_temp[ id3 - uint3( 0,1,0 ) ];
        float4 zm = _r_t3_temp[ id3 - uint3( 0,0,1 ) ];

        float4 val = (xp + xm)/h.x + (yp + ym)/h.y + (zp + zm)/h.z;

        //float3 samp = kernel2TextureCoords( id3, _c_gridRes );
        float4 rhs = _r_t3_rhs.SampleLevel( _PointClamp, samp, 0 );

        float s_ = _c_diffuseFactor;
        float sign = 2*_c_timeDependent - 1;
        float4 res = ( 1 - _c_sorFactor )*xc + _c_sorFactor*( rhs + sign*s_*val )
                        /( 1 - !_c_timeDependent + sign*2*s_*dot( 1/h,1 ) );

        _w_t3_res[ id3 ] = res; // Store a updated value for the updating cell.
        // Store a copy of the updated value to be used at next update as a read texture.
        _w_t3_temp[ id3 ] = res;
}
//#####################################################################################################################
//################################### End of Poisson Solver ###########################################################
//#####################################################################################################################

//#####################################################################################################################
//################################### Start of Face Shaders ###########################################################
//#####################################################################################################################

// Unity CG shader code for volume face generation.
Shader "Hidden/RayTextureShader"
{
```

```
        SubShader
        {
                Pass // Pass 0: Find the ray start pos which is either box or the near plane.
                {
                        Fog { Mode off }
                        ZWrite Off
                        Cull Back

                        CGPROGRAM
                        #include "UnityCG.cginc"
                        #pragma target 5.0
                        #pragma vertex vert
                        #pragma fragment frag

                        uniform const float3 _volumeDim;
                        uniform const float3 _volumeCenter;

                        // Working Coords: World
                        struct v2f
                        {
                                float4  pos : SV_POSITION;
                                float4 worldPos : TEXCOORD0;
                        };

                        v2f vert( appdata_base v )
                        {
                                v2f o;
                                o.worldPos = mul( unity_ObjectToWorld, v.vertex );

                                o.pos = UnityObjectToClipPos( v.vertex );

                                return o;
                        }

                        float4 frag( v2f i ) : SV_TARGET
                        {
                                float3 Pnear, Pfar;
                                bool hit =
                                        IntersectBox(
                                                i.worldPos,
                                                i.worldPos - _WorldSpaceCameraPos,
                                                _volumeCenter - _volumeDim/2,
                                                _volumeCenter + _volumeDim/2,
                                                Pnear, Pfar
                                        );

                                return float4( Pnear, 1 )*hit;
                        }

                        ENDCG
                }


                Pass... // Duplicated pass since it is hard to randomwrite to a Texture3D in Unity Pixel Shader,
                                // we have to use a second pass to calc RayDirLen.
}

// OpenGL code for draw the quad display of renderings.
void glDrawQuadNearPlane()
{
        // Prepare the frustum dimension for the near plane panel.
        float distance = _camera.nearClipPlane + 0.00001F; // Tiny offset to avoid self near clipping.
        float fov = _camera.fieldOfView;
        float aspect = _camera.aspect;
        float frustumHeight = 2.0F * distance * Mathf.Tan(fov * 0.5F * Mathf.Deg2Rad);
        float frustumWidth = frustumHeight * aspect;
        Vector3 topLeftCorner = new Vector3();
        topLeftCorner.x = -frustumWidth/2.0F;
        topLeftCorner.y = frustumHeight/2.0F;
        topLeftCorner.z =  distance;

        GL.PushMatrix();
        GL.MultMatrix( _camera.transform.localToWorldMatrix );
                GL.Begin( GL.QUADS );
                GL.Vertex3( topLeftCorner.x, topLeftCorner.y, topLeftCorner.z );
                GL.Vertex3( -topLeftCorner.x, topLeftCorner.y, topLeftCorner.z ); // top right
                GL.Vertex3( -topLeftCorner.x, -topLeftCorner.y, topLeftCorner.z ); // bottom right
                GL.Vertex3( topLeftCorner.x, -topLeftCorner.y, topLeftCorner.z ); // bottom left
                GL.End();
        GL.PopMatrix();
}
//#########################################################################################################
//################################### End of Face Shaders #################################################
//#########################################################################################################
```