Particle Collision Detection on the GPU Utilizing Voxel Data

Andreas Pointner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2018

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 24, 2018

Andreas Pointner

Contents

Abstract

The focus of this thesis lies on collision handling for GPU-based particle systems. Voxel data is generated on the graphics processor that represents scene geometry. This data is utilized to perform the actual collision detection on the GPU. Particle positions are checked against voxels to detect collisions and use the voxel information to calculate valid collision responses.

Established ways of realizing the individual modules of the whole system are discussed first. The most fitting of those techniques is chosen. Modifications and further implementation-specific aspects of those methods are presented. Furthermore, multiple improvements are introduced to the system to increase efficiency concerning the computational time required and memory consumption. Due to the fact that the computational time available between frames is strictly limited the performance improving aspect is focused on, however. Therefore, the subsequent evaluation is mainly concerned with the performance of the original system as well as the individual contributions of each of those implemented improvements. These results will show how well the optimization measures lower the computational overhead and determine the usefulness of those individual improvement attempts.

Lastly, the outcome of the evaluation process is discussed to draw conclusions about the applicability of such a collision system for GPU-based particle systems in interactive environments. Encountered limitations are described as well as further possible additions and improvement directions that could be investigated and implemented to increase efficiency and thus applicability even more.

Kurzfassung

Der Fokus dieser Arbeit liegt auf der Kollisionserkennung für Partikelsysteme, welche gänzlich auf der Grafikkarte simuliert werden. Eine auf Voxel basierende Repräsentation der Szenengeometrie wird dafür mithilfe des Grafikprozessors erstellt. Diese Voxel werden benötigt, um die Kollisionserkennung auf der Grafikkarte durchführen zu können. Die Positionen der einzelnen Partikel werden auf Überschneidung mit den Voxeldaten überprüft, um Kollisionen zu erkennen und darauf reagieren zu können.

Etablierte Techniken zur Umsetzung der einzelnen Module des ganzen Systems werden zuerst diskutiert. Daraufhin folgt die Auswahl der am besten passenden Methoden. Auf durchgeführte Modifizierungen und implementierungsspezifische Aspekte wird genauer eingegangen. Einen weiteren Schwerpunkt stellt die Vorstellung der durchgeführten Optimierungen, zur Verbesserung der Leistung und Senkung des Speicherbedarfs, dar. Da die verfügbare Zeit zwischen zwei Bildern in interaktiven Umgebungen stark limitiert ist, wird auf die Leistungserhöhung speziell Wert gelegt. Deshalb beschäftigt sich die darauffolgende Evaluierung hauptsächlich mit der Leistungsfähigkeit des gesamten Systems und der Leistungssteigerung, die durch die Integration der einzelnen Optimierungsmaßnahmen erreicht werden kann. Die Resultate werden zeigen, inwieweit diese Optimierungen die benötigte Berechnungszeit verringern und folglich über den Grad der Nützlichkeit dieser Maßnahmen entscheiden.

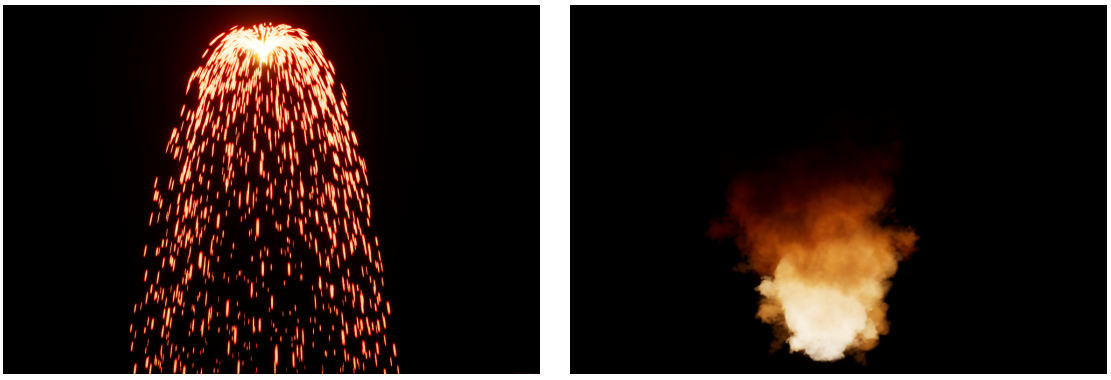
Abschließend werden die Evaluierungsergebnisse diskutiert und Schlussfolgerungen über die Einsatzfähigkeit dieses Kollisionserkennungssystems, für GPU-Partikelsysteme in interaktiven Umgebungen, gezogen. Auch werden Limitierungen des Systems angesprochen und zusätzliche Verbesserungsvorschläge zur weiteren Leistungssteigerung und Erhöhung der Einsetzbarkeit präsentiert.

Chapter 1

Introduction

1.1 Motivation

The abstraction of a complex problem into more efficiently manageable discrete particle entities allows the simulation of a variety of natural phenomena, chemical reactions and artificial visual effects in interactive environments where otherwise the limited amount of time available per frame would not suffice. Due to this flexibility and efficiency particle systems tend to be utilized frequently in video games for instance. Exemplary particle systems used to simulate spark and smoke effects can be observed in Figure 1.1.



(a) (b)

Figure 1.1: This figure shows in (a) the emission of spark particles while (b) displays rising smoke that is also simulated with the help of a particle system. Both particle effects were created within the *Unreal Engine 4*.

Use cases for particle systems continue to emerge not only for pure visual effects. The technique of approximating entities of ill-defined behavior has been adapted to different fields as well. Some techniques for simulating soft body dynamics also rely on abbreviations of particle systems. Mass-spring systems for instance are used for cloth simulation [2; 14]. The cloth pieces to be simulated are approximated using a finite amount of mass particles. Those particles in combinations with additional constraints are then used to calculate the cloth deformation. Another technique for elastic deforma-

tion also relies on particles as control units [13]. The mesh to be deformed is modeled with particles. Each of those particles is responsible for a specific area of the mesh and reacts to incoming forces by position shifting. Additionally, the initial configuration of the particle locations is kept track of. This allows for the deformed set of particles to be gradually brought back into the original configuration using shape matching. Fluid simulation, utilizing *Lagrangian* fluid mechanics, is also achieved with the help of particle representations [3; 12]. In order to display the fluid convincingly, meshes are created from those particle positions using the marching cubes algorithm for instance.

Depending on the use cases of a particle system, influences originating from the environment may need to be taken into account in order to obtain more accurate and visually pleasing results. For instance, raindrops that are simulated with particles need to be able to hit the floor. Cloth pieces should also be able to collide with the surrounding scenery in order to be able to adjust their movement accordingly. This interaction with the environment can only be achieved if some form of collision detection is in place for particle systems.

1.2 Problem Description

This range of possible use cases for particle systems in, addition to the sheer amount of visual effects that can be displayed by variation of particle properties like movement and billboard textures, explains the extensive use of particles in video games for instance. However, since the number of particles that can be updated each frame is limited often GPU-based particle systems are used as they allow for a much higher number of particles to be simulated at once. Consequently when using such a particle system collision handling should be possible as well.

Two fundamental concepts exist for providing collision detection functionality to GPU-based particle systems. The first approach performs a transfer of particle data from video memory to main memory. This transfer to RAM allows the inclusion of particle properties into a physics engine. The physics engine is aware of the surrounding geometry and will then not only resolve occurring collisions between objects of the game world but also include the particles during the collision detection phase. This allows for accurate collision results and even the detection of particles colliding with each other. Particle positions will be altered by the physics engine upon collision and afterward the updated particle data is then transferred back to video memory.

This method, while providing collision detection capabilities to GPU-based particle systems, suffers from performance issues originating from the data transfer that is required to get access to particle data on the CPU side. This negative impact on performance is unavoidable using this technique. The only alternative that allows this approach, utilizing a physics engine to resolve occurring particle collisions, to be utilized for particles is to opt for traditional particle systems that execute the simulation step on the CPU. No expensive additional data transfer operation needs to take place, and the communication with the physics engine happens directly without further intermediate steps. However, this form of particle system limits the amount of simultaneously simulatable particles as data transfer to the graphics processor is still required for the rendering process.

Hence in order to preserve the advantages of GPU-based particles two independent particle systems would be required. Effects that require interaction with the environment could be simulated using a standard particle system with the help of the physics engine. Other effects would be simulated with a GPU-based particle system. A fraction of the particles would gain the ability to collide while the possibility of simulating up to millions of particles is retained as well. This hybrid system, however, would still require memory transfer of particles simulated on the CPU to the GPU for rendering purposes.

Furthermore, the memory transfer bottleneck dictates the maximum particle count that can be simulated on the CPU and therefore have access to the collision detection feature. The amount of particles in need of collision detection within a specific scene is however dependent on the amount and complexity of the utilized effects. This unpredictability lowers the general applicability of the hybrid solution significantly.

The second possible approach for identifying and resolving collisions between particles located in video memory requires information about the scene geometry to be stored there as well. Providing this kind of information can be achieved in different ways. All of them have their own set of advantages and disadvantages regarding accuracy and computational overhead. The chosen representation then evidently influences how the collision detection and response algorithms operate.

Consequently, particle collisions with the environment are identified by utilizing this scene description data. The most significant benefit of this strategy is the avoidance of the explicit transfer of particle data from and to video memory. Additionally, collision handling can be executed per particle and therefore describes a highly parallelized workload that is well suited for the graphics processing unit. Thus the additional computational time required for collision handling calculations remains low.

Without the data transfer limitation and the advantage of being able to provide collision handling capabilities for each particle emitted by a GPU-based particle system, this approach outperforms the previously described technique. While for this approach the number of particles does not have a significant influence on the general performance the provision of scene information itself can impact the computational time severely.

In addition to different description techniques, the precision of the data representing the environment can vary as well. This leads to different degrees of inaccuracies during the collision detection phase. High amounts of abstraction lead to visible artifacts like particles colliding early, without actually touching any geometry, and therefore behave unrealistically. A compromise between accuracy and performance of such a system is often unavoidable as the general level of detail correlates to the computational time required for generating the scene representation data.

1.3 Requirements

The focus of this thesis lies on one variant of this second approach of providing scene information to the graphics processor in order to identify and resolve collisions on the GPU directly. Therefore the introduction of such a particles system, where the graphics processor executes the simulation step, is required.

Furthermore, as previously described, some form of scene representation needs to be present in video memory in order to be able to identify and resolve occurring collisions on the GPU. This information is chosen to be provided in the form of voxel data. The field

of voxel data generation from meshes is well researched. Several use cases for voxels also exist that require calculations to be executed by the graphics processor and therefore also demand the voxel information to be present in video memory. These applications led to the development of voxelization algorithms that exploit the parallel computation power of the graphics hardware and allow for rapid voxel generation. Therefore also for this implementation voxel data is chosen to describe scene geometry.

Another aspect in favor of utilizing voxel data is the fact that voxelization equals a uniform subdivision of 3D space. Each voxel represents one chunk of the subdivided volume and holds information about the geometry located within this particular volume. This uniform scene representation yields the advantage of a predictable memory footprint as the maximum amount of voxels available is known up front as well as the maximum memory requirement of a single voxel. Data access times are consistent as well as the voxel lookup time is consistent throughout the voxel grid. Furthermore, the voxel representation complexity is independent of the actual scene geometry. Only the amount of voxels and the volume covered by those voxels determine the representation accuracy. This property allows for scenes of arbitrary complexity to be represented.

Particle properties, as well as voxel data describing the scene geometry, need to be present in video memory. The implementation of the collision handling module accesses this data in order to be able to identify particles colliding with the surrounding scene geometry. This module is furthermore responsible for resolving detected collisions by altering the properties of affected particles.

1.4 Goals

The thesis focuses on providing collision handling for GPU-based particles utilizing a voxel scene representation for extracting required information about the surrounding environment. Therefore different techniques for creating and managing particles on graphics hardware are covered first. The comparison of method unique advantages and disadvantages leads to the selection of the most promising approach used for implementing a particle system that manages resources entirely on the GPU. Subsequently, methods for generating voxel data from input geometry are investigated. However only graphics hardware accelerated approaches are considered. Preexisting techniques for handling collision detection between particles and scene geometry are furthermore described with their respective benefits and weaknesses.

Aspects concerning the actual realization of the GPU particle collision detection technique are presented afterward. The implementation of the required modules is covered individually to give insights to specific design choices and necessary compromises for the whole system to function as intended. Consequently, optimization measures are introduced to increase the performance of the system by lowering the overall computational time needed for generating the voxel scene representation.

Visual results will provide valuable information about the collision detection precision itself. This precision is solely dependent on the granularity of the voxel scene representation. The main aim of this thesis, however, revolves around the performance evaluation of the particle collision detection technique. Due to the importance of particle systems in interactive environments, the system should be qualified to operate within the limits of the time available between frames. The desired framerate and correspond-

ing frame-time available in video games vary from platform to platform. The framerate requirement for virtual reality games is higher than on mobile platforms for instance.

In order to maximize the applicability of this collision detection system the computational time required should lie well below the available frame-time. The reasoning behind this statement is the application dependent further resources that are processed and render operations that are executed by the graphics processor. The frame-time available thus should not merely be consumed by the particle collision system as those additional features require computational time as well. The scenarios where such a collision handling system can be applied are otherwise limited considerably.

The evaluation will not solely focus on the overall performance but further investigate individual components of the system themselves. This way the computational overhead of each part of the whole system is put in perspective. Additionally, the implemented optimization measures are tested and evaluated to determine their impact on the efficiency of the system.

In order to be able to perform this thorough evaluation, performance data is gathered by testing the collision detection system on different hardware configurations. The resulting information offers additional insights into the general applicability of this system and helps to identify potential hardware-specific bottlenecks on older GPUs.

Ultimately these evaluation results will show to which degree this system is feasible to be introduced in interactive environments and which graphics hardware generation would be sufficient to run this system. The gathered performance data will furthermore give a clear answer on whether or not performing voxelization for solely one purpose is practicable. If the voxelization computational cost for this sole purpose is too high, although it still lies within the frame time limits of conventional video games, then this system could still be useful in combination with other voxel applications.

Chapter 2

State of the Art

2.1 GPU-Based Particle Systems

2.1.1 General Concept

The concept of particle systems is nowadays widely established in the field of computer-generated imagery. In interactive environments, such systems are essential as well. Complex problems like simulating the behavior of clouds or fire are exemplary use cases. These gaseous phenomena cannot be simulated like rigid objects by merely applying different types of transformations as they do not behave in such a predictable manner.

Therefore small particles are defined to model the entity that is to be simulated [15]. Only those particles are considered in the simulation process. Depending on the use case different sets of information and properties are stored for each particle in order to model an approximation of that entity.

All particles that are created for one entity together fully describe its state. The simulation process itself happens by applying specific sets of rules to control their properties and thus behavior. These newly calculated or altered pieces of information are then utilized to display the simulated effect on the screen.

The flexibility and usability of particle systems led to their popularity in interactive environments like video games. Different types of effects can be easily displayed by using particles to simulate the behavior and drawing billboards to the screen to display the results. The billboarding technique describes the rendering of a two-dimensional shape that is always rotated towards the camera.

Advantages of GPU-Based Particle Systems

Two different implementation types of particle systems are well established. On the one hand, CPU-based systems store particle information in memory and perform the simulation process on the central processing unit. For rendering purposes, relevant pieces of information from those particles are transferred to the graphics processor.

The main advantage of CPU-based particle systems is flexibility. The central management of the particle system allows fast and easy access to particle data for other systems. Providing particle positions to the physics engine in combination with collision shapes thus allows for particles to collide with scene geometry.

On the other hand, GPU-based particle systems are frequently implemented as an alternative. The reason being that data transfer between CPU and GPU is minimized as the particle information is stored in video memory. Therefore the simulation step is executed using the graphics processor as well. Without this memory transfer requirement between CPU and GPU, the number of particles possible to simulate rises significantly.

Kolb [11] claims that up to 10,000 particles are possible to be simulated at once on the CPU while introducing a technique to implement a GPU-based particle system featuring over a million particles. Due to the rapid advancement of technology the numbers are most likely not accurate anymore. However the ratio still perfectly communicates the relative performance difference between those two implementation techniques.

Simulating particles does not fall into the category of standard graphics calculations and rendering. The results of those calculations, which are performed during the simulation phase to update particle data, are not rendered to the screen. Over the last two decades, different tricks and implementation techniques for executing such general purpose calculations emerged due to rapid advancement in graphics hardware.

2.1.2 Storage Texture and Multiple Render Targets

One common approach involves tricking the rendering pipeline to perform particle update calculations using a special fragment shader. The particle information is stored within a set of textures and updated by rendering a screen space quad. Particle textures are then bound for the render pass where the shader is then responsible for generating and storing new particle information.

The demonstrated GPU-based particle system by Kolb [11] utilizes the *OpenGL* API. Particle properties are stored in 2D textures as can be seen in Figure 2.1. For dynamic properties that are changed continuously like position or velocity information, two textures are allocated to create a form of double buffering. Double buffering was required as reading and writing from/to the same texture during a shader stage was not supported before *OpenGL* 4. One texture is bound for reading during the simulation process while the results are written to the other texture. For the next invocation of this particle system update shader, the textures are flipped.

Multiple particle properties correspond to multiple textures that need to be updated. The required property textures for reading are bound, and with the support of multiple render targets, the updated property results can be written back in a single render pass. The newly updated particle textures afterward provide the necessary information for drawing the particles in a second render pass. One obvious limitation of this form of particle system is the limited amount of render targets supported. Depending on the complexity of the particles to be simulated the number of textures required may exceed this limit.

2.1.3 Transform Feedback and Stream Output

Another method to implement a GPU-based particle system relies on the transform feedback feature of *OpenGL* or the stream output feature when using the *DirectX* API. Both allow storing results computed in vertex and geometry shader stages into buffer objects instead of further progressing the render pipeline. The transform feedback early exit in the *OpenGL* graphics pipeline is shown in Figure 2.2.

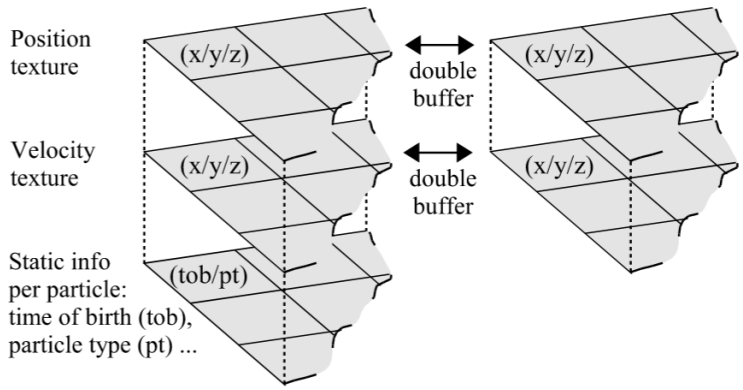


Figure 2.1: Textures are shown for storing per particle properties. Each texel position corresponds to a specific particle. Double buffering is necessary for properties that are subject to change. Source [11].

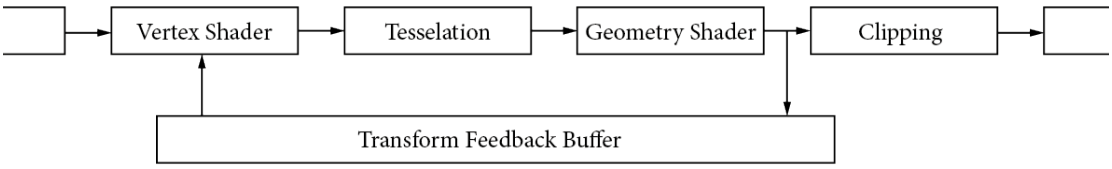


Figure 2.2: The transform feedback premature exit from the *OpenGL* render pipeline is shown. Resulting data is stored in the transform feedback buffer for further use later on.

This feature can, amongst other things, be utilized to implement GPU-based particle systems. In [19] such a particle system is described using the *OpenGL* context. Setting up the particle properties is done first. Similar to specifying per vertex mesh information each particle is treated as a vertex with a 3D position and further vertex attributes like velocity or lifetime for instance. After this specification step is complete, the particle data is stored in a vertex array object.

Two transform feedback buffers with attached vertex array objects are created subsequently. While one of those buffers serves as particle information input, the other one is utilized as storage for the updated particle data. For the following update phase, the buffers are switched so that always the most recent particle states are used as input in the update and render phase.

Each particle is processed as vertex and updated. Depending on the presence of a geometry shader stage the results are further processed or stored back into the buffer object immediately. After this process is finished, the updated particles are drawn to the screen. Hence another render pass is executed that follows through the whole render pipeline. Additional geometry can be constructed during a geometry shader stage, to create billboards for instance.

2.1.4 GPGPU Technique

The most recent technique for implementing particles systems on the GPU exploits the GPGPU capabilities of graphics cards. General purpose calculations to be executed on graphics hardware was made possible through the introduction of APIs like CUDA by *NVidia*, *DirectCompute* by *Microsoft* and *OpenCL* by *Khronos Group*. Tricking and bending the render pipeline to perform particle update calculations is not necessary using this technique as GPGPU computation is executed independently.

Along with general purpose calculations more flexible buffer objects were introduced to avoid the input data to be stored in textures or vertex array objects. This allows for more flexibility in defining the particle data. Thomas [20] describes the setup of a GPGPU-based particle system in the *DirectX* context. For updating the particle system, a compute shader is dispatched with access to the particle buffer object. For each particle, a different thread is assigned that performs the required update calculations.

Massive amounts of particles can be simulated using this or the previously described techniques while maintaining reasonably high framerates. The performance of the particle system is however not only determined by the update phase itself but also by the drawing procedure that is executed subsequently. Billboard particles, depending on the size of the quad used, can impose a significant strain on the graphics hardware due to fill-rate issues.

In order to reduce overdraw, a technique is introduced that splits up the screen into tiles. For each tile, a view frustum is constructed that is then used to identify all the particles that overlap or reside inside. This way a list for each tile can be generated that stores the indices of these particles. Subsequently, the particles are sorted by their distance to the camera per tile instead of considering the whole amount of particles. For each pixel location, only particles that are contained in the same tile as the pixel location are considered in the color accumulation process. The particle color contributions are calculated and blended until the alpha value of the resulting color reaches 1. If this happens, the other particle contributions do not influence the final color anymore, and thus the process terminates early.

2.2 Voxelization

2.2.1 General Concept

Volumetric pixels or voxels are the equivalents of pixels in three-dimensional space. Instead of an area that is taken into consideration encompassing a pixel, the surrounding volume is of importance when talking about voxel data. Huang [10] describes the properties of voxels thoroughly. Voxels reside within a three-dimensional discrete Euclidean space. Only integral numbers can be chosen for each Cartesian coordinate axis to describe voxel locations within this 3D space. This way an equidistant grid of positions is formed. In addition to the equidistant placement of the grid positions, the volume asserted with a specific voxel is defined to encompass all locations in the 3D Euclidean space that are nearest to this voxel position.

Therefore the resulting volume is a unit cube with its center coinciding with the grid point the voxel is located at. These two properties ensure that the voxel grid only

features uniformly placed equally sized voxels that are tightly packed together. Hence the grid of voxels uniformly subdivides and thus represents the 3D space.

A mesh located in 3D space thus is approximated by the generated voxel data. Instead of having arbitrary vertex positions, only uniformly positioned voxels are used to describe the mesh. Several techniques for generating voxel data out of meshes exist. Over the last years, methods emerged that create a highly parallelized workload allowing for the voxel generation process to be executed on the graphics processor. These approaches complete the voxelization procedure fast enough to be considered suitable for interactive environments where the available time per frame is limited.

Two different forms of voxel representations can be obtained. Surface only voxelization is one of those two possibilities. This method will yield voxel data that corresponds to the surface of the processed mesh. Taking into account the shape of meshes the second approach generates a solid voxel representation, which means that voxel data can be generated for volumes that are enclosed by mesh surfaces. The following subsections discuss methods for achieving both types of voxelizations.

2.2.2 Solid Voxelization

Solid voxelization as described before not only results in voxel data representing the surface but additionally yields voxel information describing the insides of meshes as well. A rapid technique to perform solid voxelization is presented by Eisemann [7].

This solid voxelization technique uses multiple slicemaps as data storage. The term slicemap was introduced earlier by Eisemann [6] as well and describes a texture that holds multiple slices of the 3D voxel grid. More precisely the x and y coordinates of a voxel are defined by the texel position while the depth component z is represented by the color of that texel which can be seen in Figure 2.3. The color channels are interpreted as one 32-bit binary number where each bit position corresponds to a different voxel grid z coordinate. Stacking all those textures up behind each other forms the total voxel grid storage. The number of textures thus determines the resolution of the voxel grid in the z direction as more or fewer bits can be used for view volume depth division.

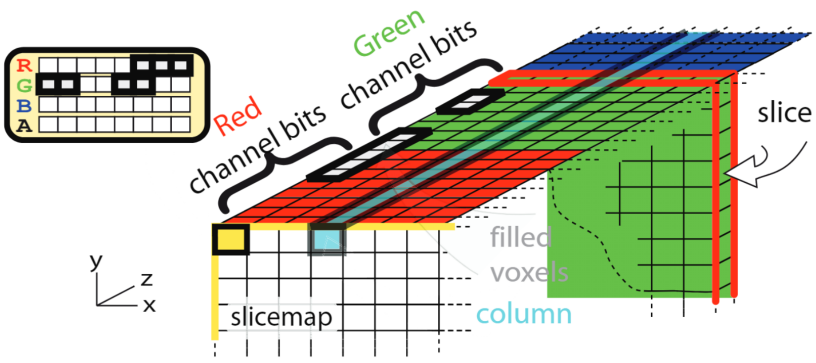


Figure 2.3: The structure of a slicemap is shown. The left image shows the correspondence of the x and y coordinates of the voxel grid to the texel position. Whereas on the right the encoding of the z coordinate is demonstrated. Source [7].

In order to determine if a voxel is located on the inside of a mesh the number of fragments n, that is created in front of the voxel position, is examined. In front is defined by the depth of the individual fragments. If the rasterizer creates an odd number of such fragments with smaller depth, this indicates that the observed voxel location is inside a mesh. An even number corresponds to an exterior position.

Allocating more memory to keep track of this number of voxels in front of every possible voxel location in the 3D grid is not necessary. The only relevant information is whether those numbers end up even or odd which is expressed by the modulo operation $n\%2$. When an even number n is processed, 0 will be the result and 1 otherwise.

Therefore this information can be stored for each voxel position in the slicemaps as only one bit suffices per voxel. Furthermore, this even/odd bit directly signifies if a voxel is present at this location since an odd number specifies a voxel position to lie within a mesh.

The voxelization process itself uses those properties described before by rendering the meshes to be voxelized without discarding back-facing triangles. The general process can be observed in Figure 2.4. The essential part of the procedure occurs in the fragment shader stage. For each generated fragment, a bitmask is created depending on the fragment's depth value.

Figure 2.4: This image demonstrates the voxelization technique using a simplified example. The bitmask creation, as well as the combination process of those masks, is shown. Source: [7].

The depth is mapped to the corresponding depth coordinate z in the voxel grid first. Then a bitmask can be created by setting the bit at position z to 1 as well as all other bit positions that represent voxel depth coordinates smaller than z. This process can be imagined as a ray being cast from the camera, along the z direction, traversing one voxel position after the other until the fragment is hit.

For all those hit voxels their counting bits in the slicemaps need to be updated. This happens by applying the exclusive OR operation to the bitmask that was created to the one already stored in the slicemap textures. The resulting bitmask is then written back into the slicemap. After this procedure has been applied to all fragments, the solid voxelization procedure terminates.

In order to avoid the issue of inverted voxel generation, depth clipping is disabled. This measure ensures that all fragments that would otherwise be discarded due to their depth value are still generated. Otherwise, parts of a mesh might be clipped and thus yield in exterior being interpreted as interior and vice versa.

Furthermore, input meshes are required to be watertight. Watertight meshes do not feature cracks or holes on the surface. Only for those types of geometry, it is possible to separate the exterior from the interior. For meshes that do not meet this requirement, the voxel generation cannot be performed satisfactorily.

This technique offers high performance in generating voxel data from input meshes while keeping the memory consumption low. However, the result is a binary description of voxels only. No additional information can be stored during the voxelization process. Although further information can be derived from this binary voxel representation, applications exist that require additional information and thus cannot rely on this voxelization method.

2.2.3 Surface Voxelization

Crassin [4] presents a surface-only voxelization method that also relies on the hardware rasterizer to perform the voxel generation. The procedure is shown in Figure 2.5. This method operates on a per-triangle basis. Therefore a geometry shader is introduced that processes individual triangles. The first responsibility of this voxelization geometry shader is the calculation of the dominant axis for each input triangle $T = (\vec{v}_a, \vec{v}_b, \vec{v}_c)$. Two vectors $\vec{v_{ab}}, \vec{v_{ac}}$ are created using the triangle input vertices. The normal vector of the triangle $\vec{v}_n$ is created subsequently using the cross product with the previously prepared pair of vectors.

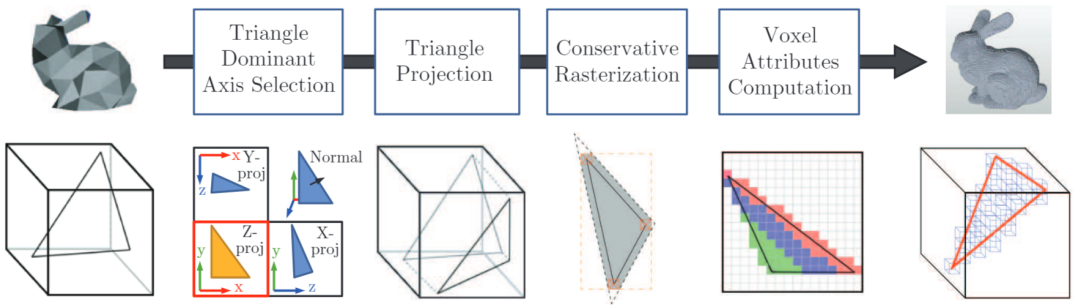


Figure 2.5: The voxelization process is shown on a single triangle. First, the determination of the dominant axis and the projection is executed. Then the triangle is modified to ensure conservative rasterization. Final processing steps on a fragment basis include the calculation of the final voxel positions and attributes. Source [4].

Utilizing the normal vector $\vec{v}_n$ the dominant axis can be determined. The dot product is applied to the three primary axis vectors, and the results are compared. The highest resulting number describes the smallest angle (using the arcus cosine function) between the normal vector and axis vector and therefore is selected to be the dominant axis for this triangle. Projecting onto the dominant axis will yield the projection result with the largest area. Larger triangle projection area results in better voxelization results as more voxel fragments will be generated to cover the original triangle better.

In order to be able to perform the projection, specific view matrices are created beforehand. Each one of those three matrices is set up to perform a view transformation onto one of the different main axes. Using the appropriate view matrix and an orthographic projection matrix thus performs the projection of the individual triangles.

The coordinates of the projected triangle are passed on to the fragment shader subsequently. The hardware rasterizer creates fragments to cover the projected triangle.

Those voxel-fragments, as they are called, are further processed to generate the actual voxel data. One fragment will create at least one but up to three voxels. Partial derivatives in x and y direction determine the actual amount of voxels per fragment as this indicates the tilt amount of the original triangle. More tilt means that each fragment covers more depth which translates to a higher amount of voxels being required to cover that depth range sufficiently.

To ensure that the x and y coordinates of the fragment match the voxel storage texture the viewport has to be configured before starting the voxelization render pass. The chosen voxel grid resolution is set as size for the viewport to enable this mapping capability. Furthermore to fully recreate the mesh surfaces using voxels the mesh information needs to be processed in its entirety. For rendering purposes, the opposite is usually the case where all triangles of a mesh that face away from the camera are discarded to speed up the rendering process. Therefore the backface culling feature has to be disabled during voxel generation. Furthermore, no pixel output is generated nor does depth testing, and writing to the depth buffer need to be executed.

This technique offers high performance as it makes use of the hardware rasterizer. Furthermore, the possibility of storing arbitrary data during the voxelization process per voxel allows for a broader range of voxel-based applications to operate on the voxel data. However, this feature also suffers from high memory consumption.

Using 3D textures as storage means that for every texel position, hence every possible voxel in the voxel grid, the memory has already been allocated. This allocated space will however never be utilized to the full extent. On the contrary, most of the available space will be left untouched because meshes are not voxelized in a solid way. Which means precious video memory on the graphics card is wasted. An alternative technique also presented by Crassin [4] uses an octree data structure to store voxel information. While this reduces the memory overhead significantly the additional computational time required for constructing and updating this octree is significant.

Atomic Average

Two triangles that share a vertex or an edge will generate duplicate voxels. This circumstance is unavoidable since the per-triangle approach does not take into consideration what the surrounding geometry might contribute to the voxel data. If the algorithm proceeds as described before the data is written to the voxel storage without considering data that may already have been stored there before by other triangles. Therefore the last written data at a particular voxel location will be the final one for that voxel.

However not always the same fragment is processed in the same order. The highly parallel architecture of the graphics hardware does not allow for accurate assessment of the triangle processing order. Therefore if the voxelization has to be executed frequently the data stored in the voxels may differ from the results obtained before. In case of color data, this would yield in flickering effects.

Depending on the information to be stored per voxel and the use case, this peculiarity can cause visible artifacts that impair the quality of the desired visual effect to be displayed. This problem of incoherence can, however, be solved by averaging instead of simply overwriting the pre-existing data. This happens with the help of atomic operations to avoid race conditions from occurring.

If for instance the diffuse color should be stored as per voxel information then there may be for instance three color channels and one alpha channel with a total capacity of 32 bit. With eight bit per channel, the maximum number to store is limited to 255. Thus summing up all the incoming voxel fragment colors beforehand and dividing the result afterward is not possible using 8-bit color channels.

To overcome these storage limitations without further allocating memory to accommodate for the accumulation step of the averaging process an iterative method is used. For each voxel location, the current average value A_i is stored in the R, G and B components while the alpha channel holds the number of values i that already contributed to that average.

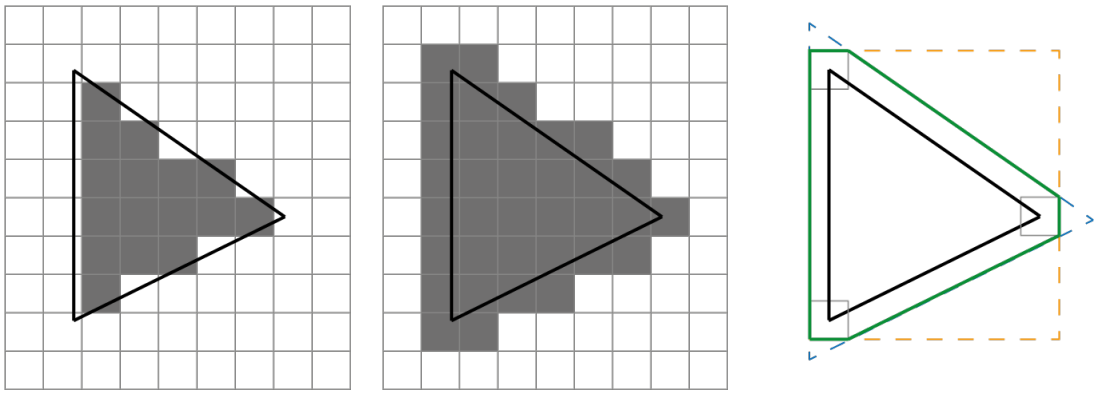
Every voxel fragment updates the previously stored average value at the corresponding voxel location. Before the new voxel data can be added to that current average value A_i, this value first needs to be multiplied by i. After this step a division by $i + 1$ yields the new average value A_{i+1} for that voxel location. Lastly, i is required to be increased by one. This averaging process works until the counter i exceeds the maximum value of 255 since only eight bits are available as counter storage.

Conservative Rasterization

This voxelization method relies on the hardware rasterization to generate fragments that are then used to generate voxel data. The standard rasterization process is not performed conservatively. Which means that the rasterizer will only emit fragments for pixel locations where the overlap with the geometry to be rasterized is big enough also to cover the center point of those locations. Only for those situations, fragments and thus voxels are generated. However, depending on the shape of the input triangle there may exist pixel locations that overlap but fail to cover enough of the pixel area to be significant enough for the rasterizer to generate a fragment. This leads to discontinuities and holes in the resulting voxel hull. Conservative rasterization is used to capture all overlapping pixel locations and thus eliminate this issue. The difference of standard vs conservative rasterization is illustrated in Figure 2.6.

However conservative rasterization cannot merely be enabled in *OpenGL*. Therefore each triangle to be voxelized needs to be modified in such a way that all overlapping fragments will be considered during the rasterization process as described in [9]. The perfect place for this triangle modification is in the geometry shader stage after the main axis projection of the triangle was executed. Each triangle is enlarged by using the edge normals to move the individual edges outwards. Adjacent edge pairs are then intersected to find the vertex positions of the newly enlarged triangle.

Additionally, an axis aligned bounding box is also calculated as the bigger triangle may lead to the creation of too many fragments when the original triangle features shallow angles. Figure 2.6(c) demonstrates how the enlarged triangle in combination with the bounding box form the new bounding polygon to enforce conservative rasterization. While the new triangle vertices are sent to the rasterizer, the bounding box is sent to the fragment shader directly as a variable. In the fragment shader stage, each fragment is checked against the bounding box. Fragments that are not contained are discarded.



(a) (b) (c)

Figure 2.6: Image (a) shows a sample triangle and the corresponding fragments the rasterizer generates while in image (b) the results using conservative rasterization can be seen. Image (c) displays the modified triangle and bounding box utilized to perform conservative rasterization.

2.3 GPU Collision Detection

Particles, highly dependent on the effect to be simulated, often need to interact with the environment. Detecting collisions occurring between particles and scene geometry as well as responding to such collisions is required to enable the particle system to simulate such effects satisfyingly. For CPU-based particle systems supplying collision detection support can be done by using the physics engine for instance.

This solution is however not viable for GPU-based systems as information from the particles would need to be transferred back from GPU to CPU in order to use the physics engine for collision handling. In order to overcome this problem of expensive data transfer between CPU and GPU, a representation of the scene geometry needs to be made available on the graphics hardware. Without this information, it is not possible for the particles to interact with the environment. Techniques emerged for providing this data in different ways that feature different traits regarding performance and accuracy.

2.3.1 Screen Space Collision Detection

Tchou [21] demonstrated a technique for GPU particle collision detection at GDC in 2011 that relies on depth information. When rendering geometry a depth buffer, often called z-buffer, is used to determine whether generated fragments should be drawn or not. This depth buffer covers the screen and holds depth values for each pixel location. This depth test happens in screen space. The depth value of a fragment is compared to the value stored in the depth buffer at the fragments x and y coordinates. If the fragment depth is smaller than the retrieved depth buffer value, the fragment is drawn and the fragment's depth value replaces the previously stored value in the depth buffer at that location. Bigger depth values lead to the fragment being discarded.

This depth buffer is complete when the render pass finishes. Then the particle collision can make use of this depth texture to perform collision detection. At first, each

particle position $\vec{p} = (x, y, z)$, which is stored in world space coordinates, is transformed into screen space multiplying the position with the current view and projection matrix. The perspective divide and viewport transform are applied subsequently to retrieve the screen space coordinates $\vec{p}_s = (x_s, y_s)$ of $\vec{p}$. Using those coordinates as lookup position in the depth buffer retrieves the depth value z. A point $\vec{d}$ is constructed using this z value that is then transformed into view space. $\vec{p}$ is also converted into view space to enable a comparison. If both points are located close enough together, a collision occurred.

Performing a single depth lookup at the calculated position in the depth texture is not sufficient however to calculate a reflection based collision response for colliding particles. Therefore the adjacent depth values of the lookup texel position are also taken into consideration. Using these additional depth values from the neighborhood enables to determine the local gradient which allows the construction of a normal vector. With this information, the collision response can be calculated and applied to a collided particle.

The normal vector can also be retrieved alternatively if deferred shading is used within the render engine. There the G-Buffer holds per pixel properties like position, normals, diffuse and various other information about the visible parts of the scene geometry to be drawn. Instead of creating an approximation of a normal vector at the collision location, the normal vector information from the G-Buffer is accessed. The same lookup location for fetching the depth value to perform collision detection is used to additionally perform the texture lookup for in the G-Buffer normal texture.

This technique offers an efficient way of detecting collisions on the GPU. Constructing the depth texture does not lead to significant drops in performance as this process can be combined with a render pass. For many situations, the depth texture provides enough information about the visible scene information. Collisions happening on the frontmost meshes of the visible geometry can be detected and reacted to satisfyingly.

The disadvantage of this method for GPU collision detection becomes evident when particles move outside the view frustum. For those particles, no depth information is available to perform collision detection, and therefore those particles will pass through geometry. This behavior also occurs when the transformation properties of the camera change as the depth information is updated every frame and the previous depth values are overwritten. Thus particles that already collided before and in another frame are not visible have no means of further detecting collisions. Depending on the effect that is simulated it may suffice to freeze the movement of those particles when this happens.

2.3.2 Signed Distance Field Collision Detection

Another approach for detecting collisions uses signed distance fields [8; 16]. A signed distance field created for a specific geometry will provide information about the shortest distances to the geometry surface from a given point.

While for simple geometry a signed distance function might be constructed where a coordinate as input parameter returns the distance value, for more complex geometry like polygonal meshes a regular three-dimensional bounding grid is created where each grid position holds the corresponding distance value. Distance values of an exemplary signed distance field are illustrated in Figure 2.7.

For checking whether or not a particle is colliding with another object the position of that particle is used to calculate a lookup position in the signed distance field of the

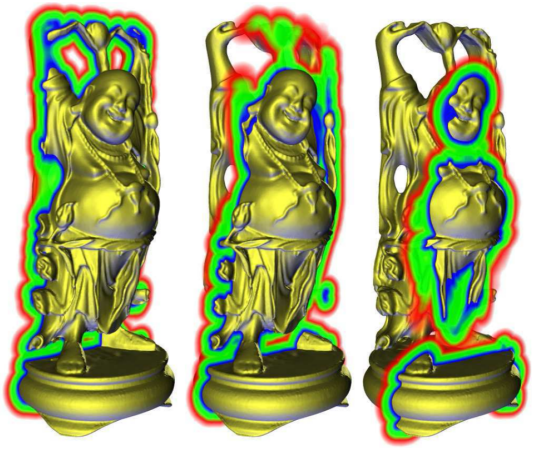


Figure 2.7: Different 2D slices of a 3D signed distance field for the Buddha model are shown. While blue bands indicate closeness to the model surface, red color indicates larger distance values. Source [16].

object. The distance value that lies at that lookup position is interpolated in a trilinear way using the depth values from the eight surrounding grid positions. An interpolated value above zero indicates the particle being located outside the object. If this value equals zero the point lies on the surface of the object and should the value be negative the tested point is situated inside the object.

Whenever a particle collision has been detected, the position of that particle is shifted to the surface of the colliding geometry. In order to be able to perform collision response calculations, a normal vector needs to be acquired. This normal vector is constructed by analyzing the gradient of the distance values used for the trilinear interpolation.

It is required to create such signed distance fields for the desired colliding entities. However, this creation process is computationally expensive and therefore is often done as a pre-processing step. The advantage of this approach is that collision detection results are consistent throughout the whole field as distance values can be interpolated in a trilinear way to achieve sub-cell accuracy.

Continuous Collision Detection

A further technique involving signed distance fields for collision detection was introduced by Xu [17]. There a method for performing continuous collision detection is proposed. Continuous collision detection algorithms pinpoint collisions before they occur. This a priori collision detection form stands in contrast to discrete, a posteriori, algorithms that identify collisions that have already taken place. On the one hand discrete algorithms feature better performance regarding computational time but on the other hand are not guaranteed to detect the complete set of collisions that occurred.

So if precise detection is required continuous methods are utilized. Detecting collision for particles before they appear requires prediction of their movement for the next timestep. The movement is assumed to be linear using the velocity information of the individual particles.

A particle position at the current time $\vec{p}_t$ and the predicted position $\vec{p}_{t+1}$ at the next time-step $t+1$ are mapped into the distance field space to mark the start and end point of the cell traversal that is executed subsequently. The assumed linear movement means that collision can only occur on the straight line that is described by $\vec{p}_t$ and $\vec{p}_{t+1}$. In order to determine which cells in the signed distance field need to be investigated, the algorithm proposed in [1] is used. Similarly to the Bresenham line drawing algorithm, cells along the line are traversed. The difference being that all cells touching the line are traversed no matter the amount of overlap. This way from the start point onwards one cell after the other is traversed and checked for collision. Upon identification of a collision, the predicted contact point, as well as the predicted time of collision, are then used to calculate the appropriate response.

Chapter 3

Implementation

The following sections will provide an overview of the project structure itself and more information about integral parts that are necessary for realizing this particle collision detection technique. The project was implemented using the *OpenGL* graphics library version 4.4. Specific features of this graphics library were used, and therefore some aspects of the following description of the realization process refer to these features in order to explain and justify certain implementation choices.

3.1 System Structure

This project dealing with particle collision detection on graphics hardware using voxel data can be broken down into three main parts that are required for the system to provide the desired functionality:

1. The first part involves the implementation of a particle system that relies on the graphics processor for updating particles. Furthermore, this includes building sample use cases for that system. In order to be able to conduct meaningful collision tests at a later stage, it is vital to provide mechanisms for controlling the behavior of particles as well.
2. Next the voxelization pipeline needs to be set up. The implementation of the shader set responsible for the 3D scan conversion is vital as well as managing the generated voxel data. The resulting voxel-based representation of the scene geometry is then provided to the collision detection algorithm.
3. Finally collision detection needs to be implemented. Both prior described modules need to be in place for this module to function as expected. Particle positions are checked against the voxel data to detect collisions. For collided particles, a response has to be calculated to alter their movement accordingly.

Since some modules depend on the results of other modules the execution order has to be specified beforehand. The order in which the modules perform their tasks and how these parts are integrated with necessary other operations like general scene geometry rendering is shown in Figure 3.1. The first tasks to be executed each frame are summarized under the term world update. Calculations are performed that alter the state of the game world which include for instance input processing and network updates. After this world update phase concludes, the update calculations for the particle system are

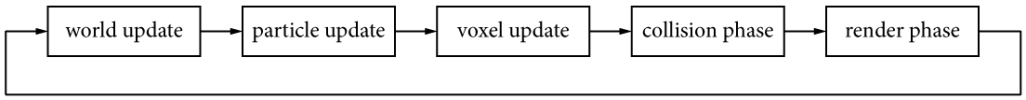


Figure 3.1: A simplified illustration of the tasks performed each frame for the particle collision detection to function correctly including their execution order.

run. The execution of the voxelization process is initiated next. With updated particle properties and voxel data the collision detection phase, including collision response, subsequently identifies and resolves collisions. Rendering the scene geometry and particles conclude the tasks to be performed every frame.

Not all modules are strictly bound to this routine as the outcome will be equal if, for instance, the rendering of the scene geometry happens before or after the voxelization phase. The collision detection however always needs to be executed after the voxel data has been generated and the particles updated. Otherwise outdated particle and voxel information would be utilized originating from the previous frame. This one frame delay would cause the collision detection to produce inaccurate results with an error severity depending on object movement velocity and camera shift.

3.2 GPGPU Particle System

The GPU-based particle system implemented for this project can process large amounts, up to several million, of particles simultaneously reflecting the main advantage of particle systems running entirely on graphics hardware. Features and mechanisms to control particles as well as aspects of the simulation process are described in the following sections. The structure of the particle system has to be compatible with another module, the collision detection and response. In order to fulfill this requirement while retaining as much flexibility as possible, the GPGPU technique is chosen for implementing the particle system. This allows for the update of the particles in general as well as the collision detection and response processes to be carried out with compute shader.

3.2.1 Particle Storage

Per particle properties are saved in video memory using *shader storage buffer objects*, in short SSBO. These buffer objects are capable of storing arbitrary data and therefore allow the particle information storage to be built like an array. Updating the particle system is done by dispatching a compute shader with access to that storage. One thread will then be responsible for updating the properties of a single particle.

This implementation uses four properties to define the state of a particle. Storing position and velocity data is vital to be able to drive the simulation forward. Additionally, each particle holds color information and a lifetime value as well. The lifetime property is used to decide whether or not a particle should continue to exist or has outlived the desired lifespan. These particle properties lead to the configuration of variable types and memory requirements listed in Table 3.1.

Table 3.1: Particle property memory consumption overview.

	type	amount	bytes
position	float	3	12
velocity	float	3	12
color	unsigned int	1	4
lifetime	float	1	4

The resulting memory requirements for one such particle adds up to 32 bytes in total. Given an amount of one million particles the buffer size, holding these per particle information chunks amounts to 30.5 megabytes. Shader storage buffer objects are specified to be able to handle large datasets. While an upper limit towards storage capacity is not defined a minimum of 128 megabytes has to be supported. Thus storing the data of one million particles does not impose any difficulties in terms of buffer capacity. Further per particle information could be introduced to add more flexibility and functionality without having to deal with hardware dependent memory limitations.

In order to gain access to the individual data sets of particles conveniently, the data is stored in an array. The update phase of the particle system consists mainly of the dispatch of a single compute shader. The amount of threads invoked is defined by the amount of currently available particles. One thread is responsible for one particle. The assignment of a thread to a specific particle is done with the help of the global invocation index of that thread. This index then allows retrieving the corresponding particle information from the data array stored in the SSBO.

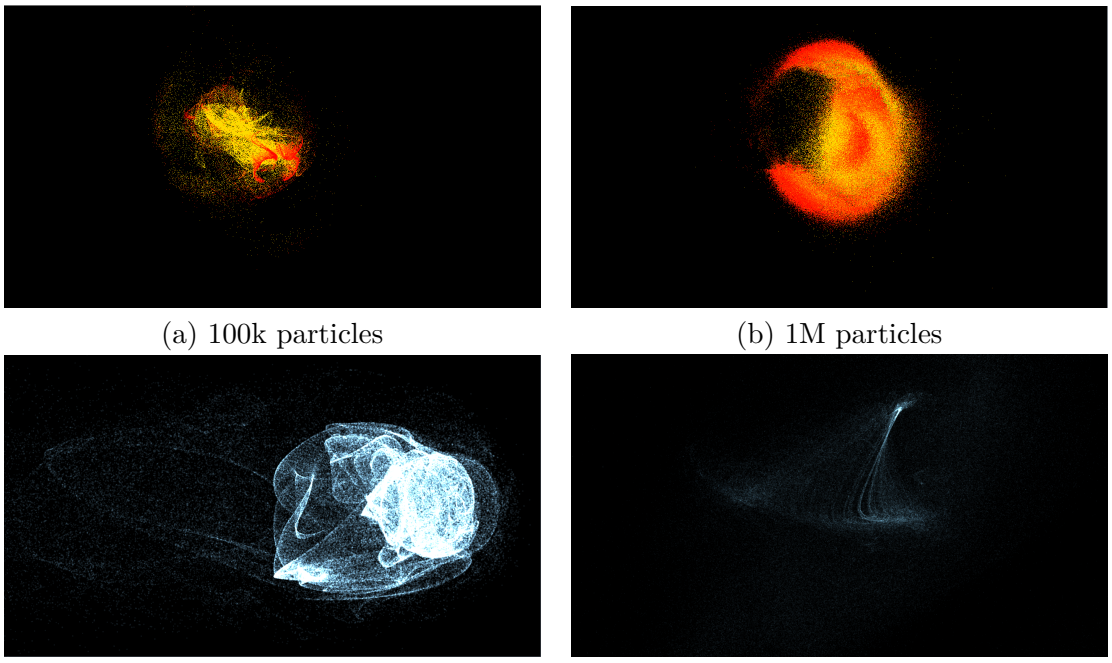
3.2.2 Visual and Behavioral Features

This implementation of a GPU particle system supports point rendering as well as displaying rectangular billboards. This well-established method uses different kinds of two-dimensional textured meshes that are always rotated towards the camera and allows to display various types of effects by merely switching the textures or texture atlases. Instanced drawing is performed in order to be able to render as many billboard quads as possible. This system is capable of displaying several hundreds of thousands of particles simultaneously as can be seen in Figure 3.2.

Furthermore, various parameters were introduced to control particle behavior. On a high level one of two different patterns can be selected and further customized:

- Attraction: One or more attraction points can be placed that exert gravitational forces on the particles. The gravitational pull depends on the squared distance to the attractor points. Attractor points can be configured to move in space in order to create more diverse particle behaviors.
- Gravitation: Particles are spawned in a specific volume and accumulate speed solely through gravitational pull. Gravitation force and direction can be configured freely. Additionally, the lifetime property is utilized. Each particle is only falling for a specific amount of time before being reset again.

For each behavior pattern, a separate compute shader is created in order to avoid unnecessary parameter setting every frame and further branching in the shader code. With



(a) 100k particles (b) 1M particles

(c) 120k particles (d) 150k particles

Figure 3.2: Particle simulation results for different amounts of particles are presented. Images (a) and (b) display point particles while (c) and (d) show particles that were rendered using the billboard technique.

these different behavior patterns the performance of the collision detection algorithm, regarding robustness and reliability, can be evaluated.

Particle Emission

In order to be able to simulate a variable amount of particles, instead of operating on a fixed set of particles, means of spawning and despawning have to exist. This functionality is achieved by the introduction of a dead list where the indices of inactive particles are stored. Inactivity is defined by lifetime only. Values above zero indicate particles still being actively updated to fulfill their purpose while lifetime values of zero and below indicate that those particles have completed their lifecycle and thus are disabled.

The emission process is handled by a separate compute shader that is executed before the actual particle simulation step commences. Whenever a particle is to be spawned, the index of an inactive particle is retrieved from this dead list and used to perform a lookup in the particle data SSBO. For performing the dead list lookup, a dedicated index is utilized. After the look-up in the dead list has been performed this index is incremented to avoid spawning the same particle again. Since one thread is invoked per particle to be spawned concurrency issues can arise upon accessing this dead list index. Hence instead of using an ordinary integer value as an index, an *atomic counter* is used to read and write from/to this index value atomically.

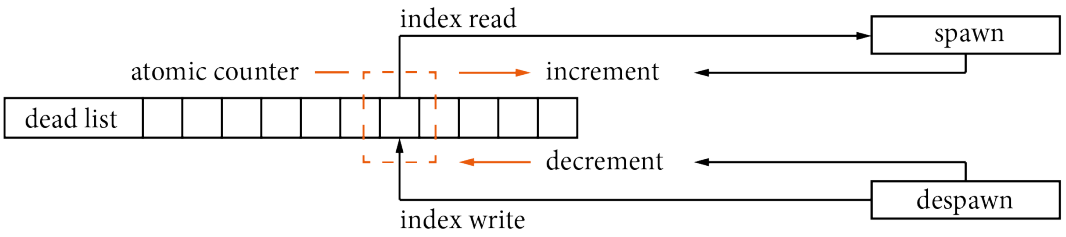


Figure 3.3: This image shows the interactions of the spawn and despawn logic with the dead list. The atomic counter, managing the data access of that list of indices, is shown in orange.

After the retrieval of an inactive particle from the SSBO, the data is modified to fit the chosen style of emission. The first step involves resetting the lifetime to the desired duration in order for the particle to become active again. Changes to position and velocity are performed as well to adjust the particle location and movement accordingly.

In order for this dead list to stay up to date, the code for updating the particles each frame has to be modified. The lifetime of each active particle is reduced with the help of the delta-time. In the event of a particle's lifetime falling below zero, the particle has to be added to the dead list. This despawn process is done by decrementing the atomic counter pointing at the current index in the dead list and adding the particle index at the new counter value into the dead list. This interaction of the spawn and despawn logic with the dead list is illustrated in Figure 3.3.

Simulation Process

The chosen integration scheme for updating position and velocity information of particles is the semi-implicit Euler method. Every frame the simulation calculation steps are executed using the elapsed time Δt between the last frame and the current one. New values for velocity $\vec{v}_{t+\Delta t}$ and position $\vec{x}_{t+\Delta t}$ are calculated as

$$\begin{aligned} \vec{v}_{t+\Delta t} &= \vec{v}_t + \vec{a} \cdot \Delta t, \\ \vec{x}_{t+\Delta t} &= \vec{x}_t + \vec{v}_{t+\Delta t} \cdot \Delta t, \end{aligned} \tag{3.1}$$

where $\vec{v}_t$ and $\vec{x}_t$ denote the position and velocity values calculated in the previous frame and $\vec{a}$ describes the acceleration that is applied to a particle. Acceleration of particles is caused by external forces like gravitational pull for instance. In order to obtain the acceleration vector $\vec{a}$ from such a force $\vec{F}$ the particle mass m needs to be taken into consideration according to Newton's second law of motion $\vec{F} = m\vec{a}$. For this implementation forces are directly used as acceleration. This simplification is valid under the assumption of each particle having an exact mass of $m = 1\text{kg}$.

3.3 Voxel Generation

The chosen method for this implementation is described in Section 2.2.3. Only surface voxel data is generated from input meshes. This approach is more computationally expensive than the solid voxelization technique explained in Section 2.2.2 because an additional shader stage is involved, conservative rasterization calculations have to be performed, and the atomic average procedure is executed. However, the advantage of being able to store arbitrary voxel properties makes this approach the more flexibility one. More applications can benefit from the generated voxel scene representation when the type of data stored per voxel can be chosen individually.

Applying this set of voxelization shaders to all meshes will result in a discrete voxel representation of the scene. 3D textures serve as storage units for generated voxel data. Each texel of such a texture represents a specific voxel position. Per voxel, a color with four components is stored. The R, G and B components of the color of one voxel hold the x, y and z values of the interpolated normal vector of the originating triangle fragment. This vector is needed later in the collision phase whenever a collision response has to be calculated. The alpha value is used for the averaging process and to indicate occupancy.

Voxel data storage units are located at the origin of the world. The chosen size, and thus voxelization resolution, determines to which extent the surrounding scene geometry is affected. Higher resolution will yield better voxel coverage of the scene geometry at equal voxel scales. Such a storage texture can be imagined as a cube that is located within the scene. Meshes or parts of meshes that are contained within or overlap this cube will be voxelized and stored in this storage texture.

3.3.1 Separation of Static and Dynamic Geometry

In game engines entities are often categorized as either static or dynamic. Static objects do not alter their transformation properties or underlying meshes during runtime which allows, for instance, pre-calculation of global illumination information for those objects. On the other hand, dynamic objects are subject to change in terms of transformation and appearance.

This distinction is also considered during the voxelization process. In order to keep the voxel scene representation as accurate as possible, dynamic objects are voxelized each frame. In contrast, due to their immutability, models of static entities are not required to update their voxel data continuously.

Therefore static and dynamic voxel data is generated independently of each other. Static voxels are only generated once while dynamic geometry is processed every frame to keep the voxel scene representation up to date. Voxel data generated from static meshes is furthermore stored in a separate texture. This allows the texture holding dynamic voxels to be cleared and refilled while the static voxels can be left untouched.

For this implementation, static voxels have two available storage unit textures. Utilizing only one storage texture to encompass the whole static geometry of a scene would be inconvenient. Either the voxel size would need to be scaled up extensively leading to a scene representation with an inadequate level of detail or the resolution of the texture would have to be configured much higher leading to a surge in memory consumption and computational time required.

Therefore in order to avoid those issues, two textures are utilized that hold static voxel data of the immediate surrounding scene geometry only. Whenever the camera moves away from both storage units such that the available static voxel data does not cover the surrounding geometry sufficiently anymore, a static voxelization is triggered at the current camera position. The furthest away texture out of the two is shifted to that location, and the newly generated voxels then replace the previously stored ones.

If the camera continued to translate in the same direction at some point where the static data would again not sufficiently cover the camera surroundings anymore, another static voxelization would be performed. This time, however, the other texture would be further away from the camera leading to the location shift and voxel data overwriting of that texture.

This technique serves as a compromise between a pre-voxelization step of the whole static environment requiring extensive amounts of memory and the constant voxelization of static meshes decreasing the overall voxelization performance significantly. Consequently, all textures, dynamic and static ones, holding voxel information have to be considered during the subsequent collision handling phase.

3.3.2 Scene Coverage Customization

While this voxel generation system produces voxel data as required, the coverage area cannot be configured yet. The origin of the world is used as the origin for the voxel storage as well. Depending on the chosen storage texture resolution more or less area will be covered by the voxelization. In order to enable custom placement of voxel storage units, the desired location needs to be considered during this process.

The voxelization itself is not executed at the specified location directly. Instead, a reversed version of that position is used as translation vector. That translation is applied to each mesh in need of voxelization. This shift in position allows the voxelization procedure to continue taking place around the origin and simultaneously enables the processing of scene geometry from different locations.

Another essential feature besides translation is scale support to maximize voxel coverage of the surrounding area. A scale factor is introduced to enable customization of voxel sizes. Similar to the position that is used as a translation vector the scale value is incorporated into the voxelization process. The individual meshes are scaled using this factor instead of scaling the storage directly.

3.3.3 Voxel Visualization

Visualizing the generated voxel data is vital to ensure the correctness of the voxelization algorithm implementation. The first approach relied on displaying single 2D slices from the 3D storage texture. This allows the user to control which slice should be displayed by specifying a depth value. In the long run, this method proved to be cumbersome. Whether or not voxels were created at the right locations was challenging to find out when only one layer of voxels could be observed at a time.

Hence a 3D visualization for the voxel storage was implemented using a raycasting approach. A quad is rendered that covers the whole screen. The vertex shader passes on the quad coordinates, which range from -1.0 to 1.0, unmodified. This way a fragment is created for every pixel on the screen. In the fragment shader for each of those fragments,

a ray is created and sent into the scene. Only those rays that collide with the axis aligned bounding box of the storage texture are further processed, the other ones are discarded.

Entry locations of the remaining rays on the voxel storage bounding box are determined. Obtaining the location of the starting voxel coordinates for beginning the ray traversal through the voxel grid is the next step. Then the algorithm proposed in [1] is utilized to find all voxels along a ray. Whenever an occupied voxel location is traversed, the data is acquired and the algorithm terminates. The data is then set as output color for the fragment where the ray originated from. Exemplary visualization results can be observed in Figure 3.4.

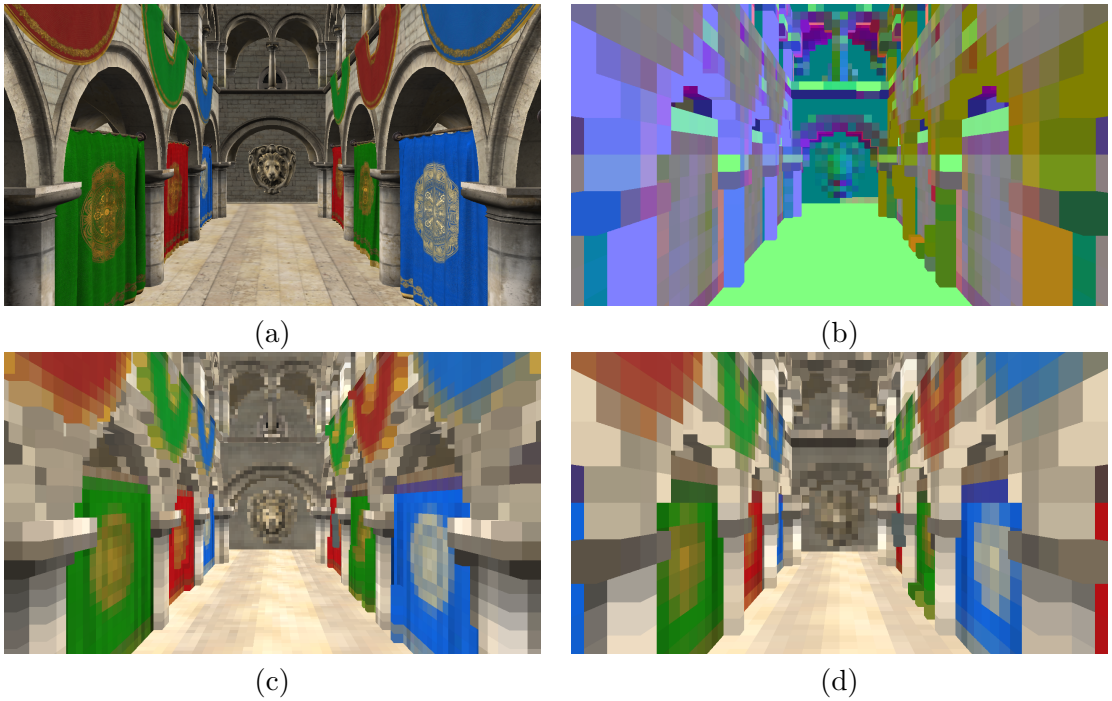


Figure 3.4: Image (a) shows a screen capture of the sponza model. The other pictures contain visualizations of voxel data generated from this model. While in (c) and (d) each voxel holds diffuse color information in (b) per voxel normal information is presented encoded as color.

3.4 Collision Handling

Identifying and reacting to collisions between particles and voxels is done by dispatching another compute shader. This shader accesses the particle data SSBO similarly to the particle update shader. Again one thread is invoked per particle to prevent concurrency issues. Each thread starts by converting the position of the corresponding particle into the voxel space of the current voxel storage where the collision check is going to be performed in.

Each 3D texture is accessed with coordinates ranging from zero to the respective dimensions of each coordinate axis of the texture. Storage textures located at the origin with no scale modification however also cover areas with negative coordinate values. Therefore the conversion involves the addition of half the texture storage resolution in order to map negative coordinate numbers to this positive range that can then be used to access individual texels. For textures that have been relocated and scaled differently to maximize scene coverage, the conversion process involves furthermore the incorporation of that location and scale values to shift the particle position accordingly.

The converted coordinates are then used to perform a lookup in the voxel storage texture. The retrieval will yield a four-component color vector that is stored at this location. The alpha channel describes whether or not this is an occupied voxel. If the alpha value happens to be zero no voxel is present indicating no collision has occurred.

In order to be able to support more than one source of voxel data, multiple voxel storage textures have to be taken into consideration for identifying collisions. The collision detection algorithm prioritizes these voxel storage units by their distances to the camera. Whenever a position has to be checked for collision within a section of two overlapping textures the nearer texture is checked first. Only if no voxel is present at that location, the other texture is examined. The reasoning behind this sequence of actions comes from the distance-dependent level of detail. The view frustum segments near the camera are required to be represented in more detail to allow for more precise collision results.

As mentioned before, the collision detection only continues from one texture to the next one if no collision could be detected before. Otherwise, the following textures are skipped, and the collision response calculations are performed immediately.

3.4.1 Collision Response

Whenever a voxel happens to be retrieved at the lookup location with a non zero value being stored in the alpha channel a collision occurred. This bounce-off collision response that is to be calculated for each collided particle can be observed in Figure 3.5. This is done by utilizing the other three color components of the retrieved color as they describe the normal vector of that voxel. Next, the movement direction of the particle is calculated by normalizing the velocity vector. This direction vector can then be reflected with the help of the voxel normal. In order to let colliding particles bounce off the surface, the reflection vector is set to be the new particle velocity value. Before that, the vector is scaled using the previous particle velocity vector's length. A damping factor is introduced additionally to simulate energy loss upon collision.

Depending on particle velocity and delta-time a previously colliding particle may still be colliding with the same voxel next time the collision detection algorithm is executed. This would lead to further movement changes and thus to incorrect bouncing behavior. Therefore not only the particle velocity vector is altered, but also the position itself is corrected. Using the inverse velocity vector of the particle itself the position is offset to not collide with the voxel anymore.

One edge case occurs whenever the velocity and normal vector face into similar directions, which can be determined by calculating the angle between those vectors. A steep angle indicates this similarity in direction. This situation can be resolved by either

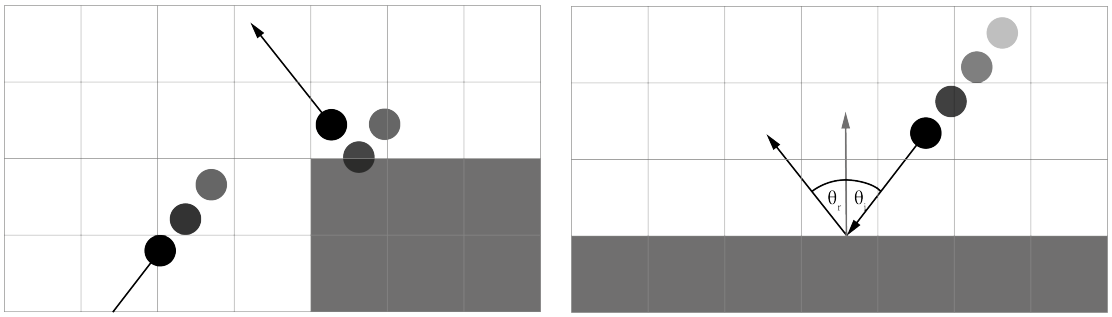


Figure 3.5: In image (a) the collision detection method is shown. Particles (black dots) overlapping voxels (grey squares) indicate collision. In (b) the reflection of collided particles is shown.

continuing with the collision response calculations as described before or by despawning the colliding particle immediately.

The reasoning behind despawning a particle upon identification of such an edge case scenario is one connected to the origin of this issue. Moving particles inside a mesh, for instance, may collide with the shell voxels of that mesh. This behavior would lead to the particle being trapped inside the mesh bouncing around indefinitely. Therefore despawning the particle would be one way to avoid this scenario from happening.

However, the voxel representation of scene geometry is limited in detail which means that for thin-walled geometry often both sides of such walls can only be represented by the same set of voxel data. In such a case collision response would only be calculated correctly for particles moving from one side while other particles coming from the opposite direction would be despawned instead of reflected.

The severity of this erroneous behavior is dependent on the scene geometry, the alignment with the voxel grid and the resolution and scale of the voxel storage unit. Therefore the impact of this issue is hardly predictable. Hence in order to prevent this issue from occurring at all the despawn method is not applied. Instead, every colliding particle is reflected accordingly.

Normal Vector Interpolation and Modification

Noise is furthermore introduced to modify the voxel normal slightly. A particle colliding with a specific voxel will bounce back in the same direction no matter where the collision happened with this specific voxel. If a high amount of particles, moving in the same direction, colliding with a small patch of voxels bouncing patterns occur. This slight modification of the normal vector is executed for each particle and thus helps to mitigate the severity of those artifacts.

Additionally, trilinear interpolation is used to determine the normal vector instead of performing direct lookups in the voxel storage textures. This interpolation is performed with the help of *OpenGL*'s sampler objects. Visual artifacts are counteracted with this approach because particles near each other will now receive different normal vectors in contrast to the simple voxel normal lookup variant. In combination with the slight

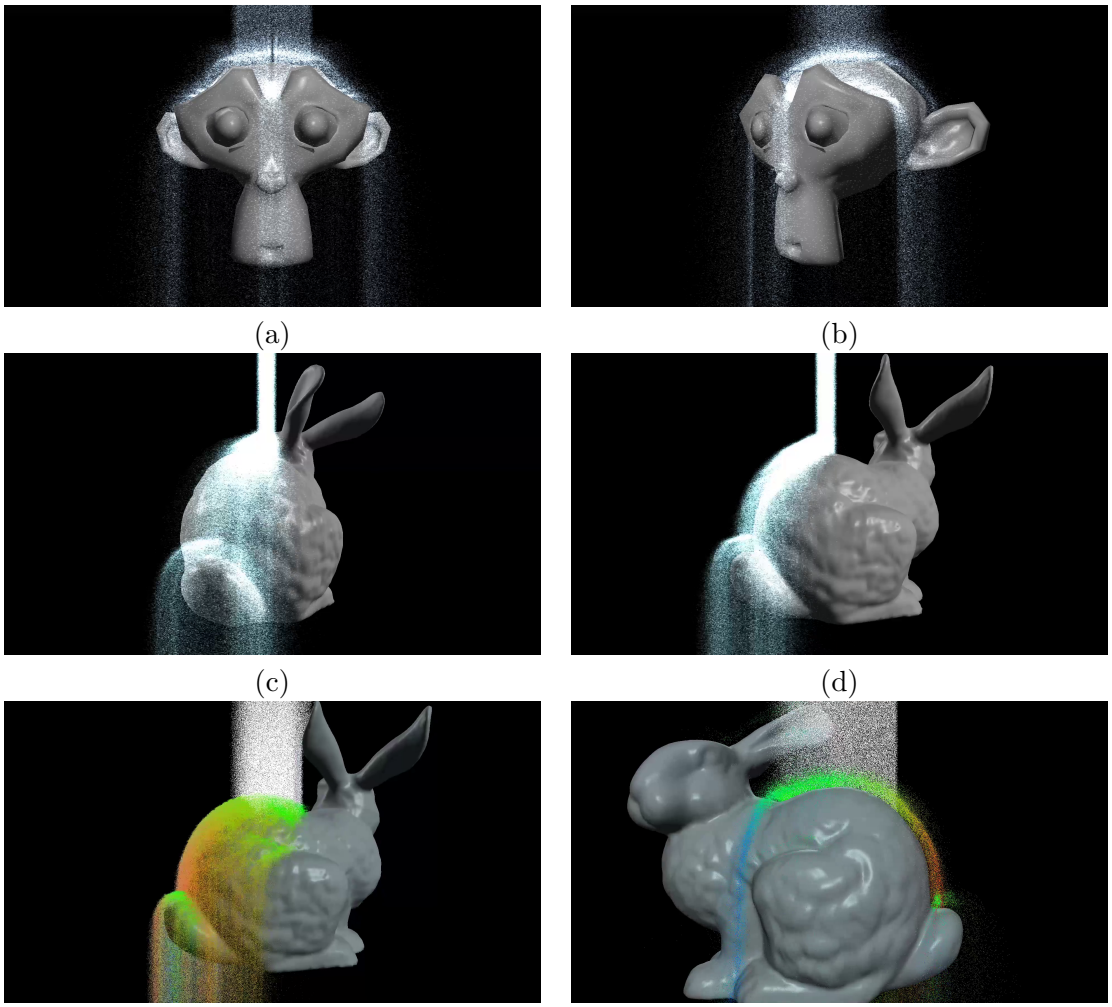


Figure 3.6: Collision detection/response results can be seen for one million simultaneously simulated billboard particles. Images (e) and (f) illustrate the voxel normals the particles were reflected with.

normal modification, artifacts are significantly reduced. Collision detection and response results obtained with this system, using normal vector interpolation and modification, can be observed in Figure 3.6.

In order to obtain these results, the meshes were voxelized at a resolution of 256. Over one million particles collide with different meshes to test the collision detection and response capabilities. As can be seen, the resolution is sufficient for the particles to be able to collide and gather in the indention spots of the *Blender Monkey* model for instance.

Chapter 4

Optimization Measures

4.1 Progressive Voxelization of Static Geometry

The differentiation between static and dynamic geometry allows for more specific optimization measures to be implemented. While most optimization steps are taken towards dynamic geometry, the progressive approach of generating voxel data is dedicated to solving performance issues for static geometry.

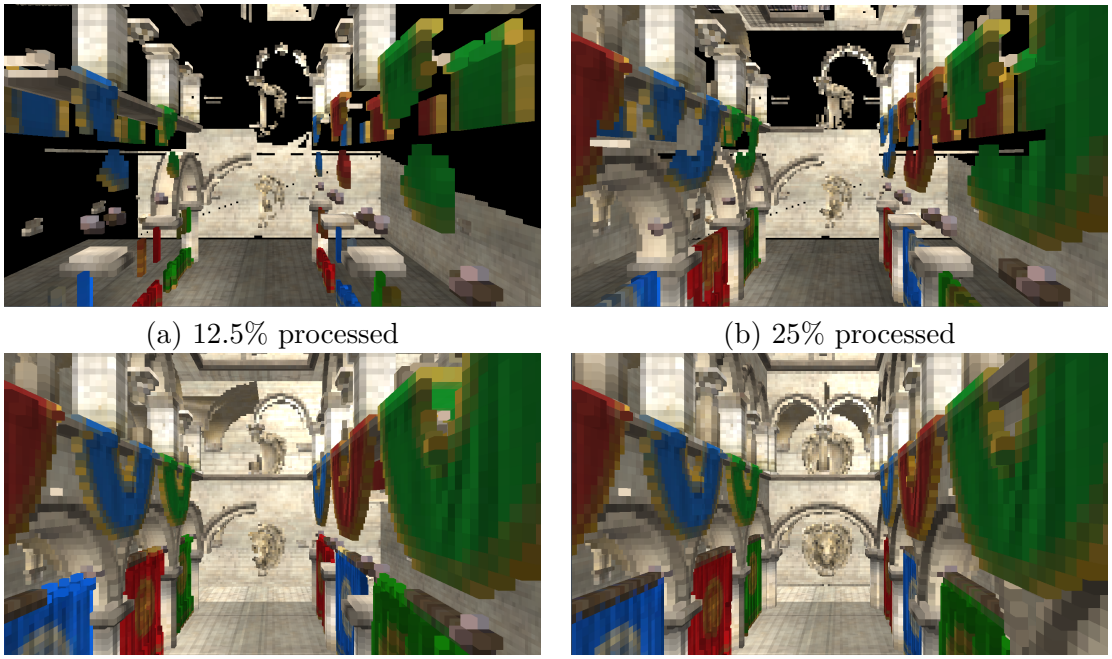
Voxelization of surrounding meshes happens every frame for dynamic geometry. In theory voxel data of static meshes does not require to be updated once it has been generated. One voxelization render-pass would be sufficient. However, depending on the amount of available video memory and the size of the scene a voxel pre-calculation of the whole static scene geometry may not be feasible.

Therefore, as described in Section 3.3.1, static voxelization is carried out on demand when the camera position changes such that the available static voxel data is insufficient to represent the current surrounding environment. Whenever static voxelization is performed the workload for the voxelization module rises abruptly from one frame to another. For one frame, in addition to the continuously executed re-voxelization of dynamic geometry, static meshes are required to be processed as well. Depending on the amount and complexity of nearby static geometry this increased workload leads to a temporal hit in performance as the computational time required is higher. This issue can lead to significant variations in frame-times which is noticeable as stuttering.

In order to mitigate this temporary spike in computational time required in a single frame, the procedure is divided up into several consecutive frames. The number of meshes to be processed per frame can be adjusted. This distribution flattens out the spike and thus provides a more consistent frame-rate. However, not only the number of meshes need to be considered in this workload distribution process. Mesh complexity plays a key role as well. One additional complex mesh, consisting of several tens of thousands of triangles, to be processed along the usual dynamic geometry can single-handedly cause performance problems.

Therefore in order to further distribute the number of triangles to be processed by the voxelization algorithm each frame, the individual meshes are not dealt with as a whole either. *OpenGL* allows the specification of the number of indices to draw as well as the starting point in the index list. This allows to split up the voxel generation for

a single mesh over several consecutive frames. Results of different voxelization stages of one mesh can be observed in Figure 4.1. The parameters for controlling the mesh amount and fraction to be processed per frame need to be chosen wisely however as the distributed voxelization not only helps to reduce the workload for individual frames but also delays the completion of the voxelization process noticeably.



(a) 12.5% processed (b) 25% processed

(c) 50% processed (d) 100% processed

Figure 4.1: This figure contains different intermediate voxelization results obtained by progressive voxelization. While (a) displays voxel data generated from a single voxelization step, the other images show voxel data accumulated over multiple such voxelization steps.

4.2 Normal Vector Compression

As described before the voxel storage used is a 3D texture with equal length sides. The chosen resolution of this texture determines the amount of voxel data that can be stored and therefore corresponds to the size or rather the amount of detail of the scene that can be captured. With increasing texture sizes memory consumption also rises significantly. Encoding a normal vector with 32-bit floating point values requires at least 12 bytes of data per voxel. The resulting memory requirements are listed in Table 4.1 for different voxel resolutions.

Doubling the voxel resolution leads to an eight times increase of possible voxel locations. Thus the memory consumption also rises by that factor. Therefore alternative storage methods have been proposed. Crassin [5] suggests encoding the data into a sparse octree which enables to cut down the memory requirement significantly. However as the construction as well as the traversal of such an octree structure also increases the computational overhead this implementation continues to utilize 3D textures as storage.

Table 4.1: 3D voxel texture memory consumption comparison in megabytes between the original lossless and the compressed normal vector storing approach.

	voxel resolution			
storage technique	64	128	256	512
lossless	3 mb	24 mb	192 mb	1,536 mb
compressed	1 mb	8 mb	64 mb	512 mb

In order to achieve a decrease in memory requirement using such 3D textures, the normal vector voxel components are now encoded into one byte each. This lossy compression of the vector components enables the reduction of required memory to a quarter. The resulting reduction of memory consumption can also be observed in Table 4.1.

However, this compression leads to the introduction of errors due to the limited amount of bits available to encode floating point numbers. Eight bits allow for 256 different values to be represented which translates to a relative error value of $\delta x = 1/256 = 0.0039$. Hence each component of a vector stored this way can deviate up to 0.39% from the original value leading to a significant reduction in precision. On the other hand, there are still approximately 16 million (256^3) possible direction vectors that can be constructed using eight bits per vector component. Therefore the severity of the inaccuracies introduced by this compression is low enough to avoid visible artifacts in the collision response phase from occurring.

4.3 Non-Indexed Drawing

In *OpenGL* the geometry shader stage is executed after vertex processing is completed but before the actual primitive assembly is performed. Such a geometry shader processes primitives instead of single vertices. Hence early primitive assembly is commenced in order to be able to provide the geometry shader input in the form of primitives. Since the output primitives of the geometry shader stage are not required to match the input another primitive assembly process has to be executed after such a geometry processing stage. This additional workload leads to higher frame-times when executing the additional geometry shader stage in comparison to executing only vertex and fragment shader, independent from the actual code that is executed within the individual shaders.

Therefore this attempt to increase the overall performance of the voxelization process focuses on the disposal of the geometry shader stage. Within this stage, in the simplest case, the face-normal calculation is executed as well as the projection of the input triangle onto the dominant main axis. In order to be able to perform the projection in the vertex shader stage, the individual meshes need to be modified. Many meshes, especially with high triangle count, rely on indexed drawing. Each vertex is stored only once and therefore can help to reduce the memory footprint of meshes where triangles commonly share vertices. Additionally, an index list is created that specifies which vertices are connected to form, for instance, triangles.

Table 4.2: Frame-time comparison of both voxelization methods, indexed with and non-indexed without geometry shader.

resolution	indexed rendering	non-indexed rendering
64	0.250 ms	0.253 ms
128	0.285 ms	0.295 ms
256	0.512 ms	0.480 ms
512	2.506 ms	2.458 ms

For this optimization, another way of storing the mesh data needs to be utilized. The index list is disposed, and vertices are arranged in a specific manner. The ordering implicitly serves as index list as it determines which vertices will be connected to triangles. This setup requires shared vertices to be stored multiple times. A vertex will not be shared with other triangles and thus enables triangle specific attributes like the face normal to be stored. Face normal vectors are pre-calculated upon the mesh loading process, and each of those normal vectors is assigned to the corresponding three vertices that were used to calculate this vector.

The vertex shader, using this mesh structure, receives the face normal from each vertex as attribute input. With the help of this normal vector, the dominant axis can then be computed which consequently is utilized to project the vertex onto this selected axis. The process happens analogously to the original implementation in the geometry shader as described in Section 3.3. The unmodified fragment shader then takes over to finish the voxelization procedure.

4.3.1 Performance Gain

A performance comparison of the standard voxelization approach to the geometry shader lacking method is shown in Table 4.2. This data was gathered before the incorporation of conservative rasterization to investigate the performance of this optimization measure as early as possible. In order to be able to compare this data in a meaningful way, the information that was gathered from the standard voxelization procedure was also obtained with conservative rasterization being disabled.

Interesting to see on this table is that the non-indexed way of creating voxel data is neither clearly outperforming nor falling behind in comparison to the standard voxelization process. While the frame-times using this technique improve when utilizing a voxelization resolution of 256 and higher, lower resolutions lead to a decrease in performance. At best the frame-times can be lowered by 48 microseconds while an increase of 10 microseconds is possible as well depending on the chosen resolution.

One reason why this optimization does not boost the performance significantly might be the *Post-Transform Cache* hardware feature. This cache stores processed vertex data resulting from vertex shader calculations. With the help of this cached data shared vertices that have already been processed once are not processed again. The cached data is used instead. This feature thus leads to faster completion of the vertex shader stage and therefore to an increase in render performance. In this case, however, no speedup can be achieved due to the fact that vertices are never utilized more than once.

Table 4.3: The memory consumption of meshes using indexed drawing as well as the space required for the non-indexed variant with face-normal attributes.

model	triangle count	indexed	non-indexed
Sponza(simplified)	189,813	13.895 mb	23.895 mb
Stanford Bunny	69,630	7.171 mb	8.765 mb
Blender Monkey	967	0.100 mb	0.122 mb

4.3.2 Memory Requirements

Although the gain in performance is inconsistent, this technique could still be utilized to lower computational times at high voxel resolutions slightly. However, the additional memory required to be able to execute the projection step in the vertex shader stage rules out this method. Not only the index list cannot be used to avoid duplicate vertices but further attributes have to be stored on a per-vertex basis. As previously mentioned for this implementation each vertex data is provided with the face-normal vector as an additional attribute.

Thus the memory consumption rises significantly as can be seen in Table 4.3. The displayed required memory in megabytes solely includes vertex and index data, if available. The memory requirement further rises with the introduction of additional vertex attributes. The magnitude of this increase is however dependent on the number of times triangles share individual vertices as this determines the number of duplicate vertices.

10 megabytes of additional space is required to store the *Sponza* model which corresponds to a rise in memory consumption of over 70 percent. While for the other listed models the relative memory footprint increase is lower the additional requirement is nevertheless significant. Every mesh is affected by this memory overhead issue. Depending on the scene geometry, the severity of this increased space requirement varies.

Another issue arises that requires even more additional memory. The implemented part of the non-geometry-shader voxelization variant does not yet feature conservative rasterization. Conservative rasterization is of importance to avoid the introduction of holes in the voxel representation. In order to be able to perform the calculations necessary, to expand each triangle in such a way that the rasterization process is performed conservatively, all vertices of an individual triangle have to be utilized.

Now for this variant of performing all calculations without a geometry shader, the neighboring vertices would be required to be accessible. This again could be achieved by adding the two other vertex positions of the triangle as vertex attributes. The memory requirement would rise significantly.

To mitigate the issue the face-normal vectors and neighboring vertices could be encoded into textures or buffers instead of storing them as per-vertex attributes. Each vertex would then access their respective face-normal with the help of the built-in vertex shader input variable `gl_VertexID`. While this measure would reduce the actual memory requirement, texture or buffer data retrieval would be required within the vertex shader to access the needed information. These access operations would then again negatively influence the computational time required to complete the vertex processing stage.

These memory consumption issues in combination with the unsatisfactory gain in performance led to an early termination of the development of this optimization attempt. Without a significant boost in performance, the additional memory required per mesh leads to the conclusion that this optimization attempt cannot be seen as a viable alternative to the original procedure.

This failed optimization attempt leads to the conclusion that neither the additional primitive assembly step nor the geometry shader stage itself bottlenecks the voxelization process. Therefore most of the computational time is spent on fragment processing.

4.4 Enhanced Scene Coverage through Cascading

Using one voxel storage texture to cover the surrounding area requires either a large voxel resolution or bigger voxel sizes overall. High-resolution voxelization leads to an increase in computation time as more fragments have to be generated and processed. The amount of available video memory furthermore limits such an increase. Doubling the voxel resolution would demand, in the case of a simple 3D texture as voxel storage unit, eight times more memory on the graphics hardware.

On the other hand, scaling the voxel size to achieve sufficient scene coverage leads to coarser voxelization results which correspond to a less detailed scene representation. Collision detection cannot be performed as accurately. This loss in precision is noticeable especially when particles collide with geometry that is located near the camera.

In order to overcome these coverage and performance issues, a cascading approach is chosen. This method was introduced initially to solve similar problems for shadow mapping [18] where one shadow texture is insufficient for providing shadows within the whole view frustum at a reasonable resolution. Instead, multiple shadow textures are generated that cover different segments of the view frustum. This technique is adapted for the voxel generation procedure that is applied to dynamic geometry. The difference between the standard approach and the cascading method can be observed in Figure 4.2.

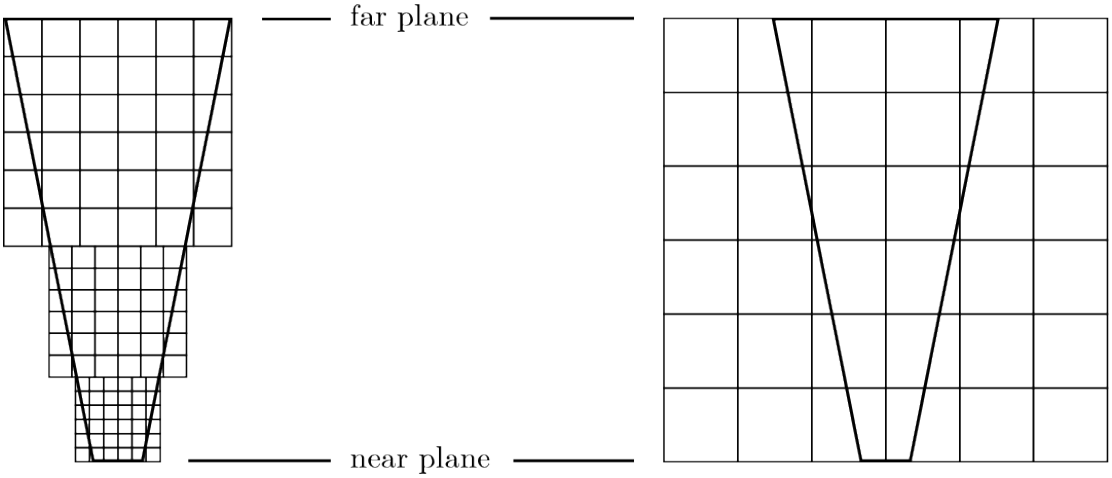


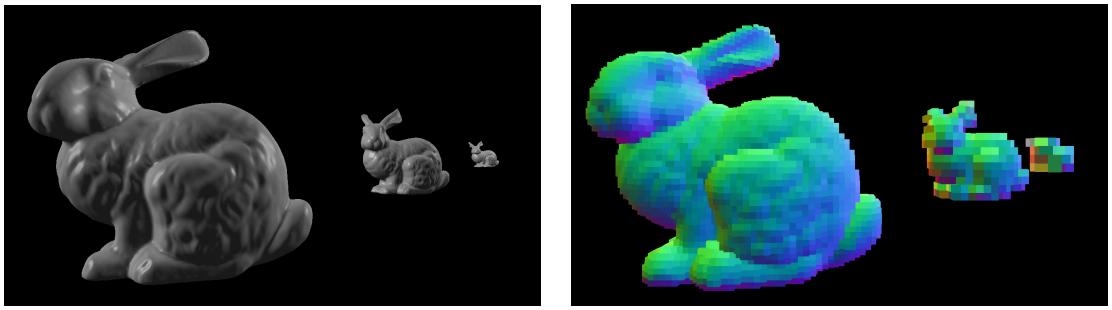
Figure 4.2: Multiple voxel textures with different voxel sizes stacked together can cover the view frustum in more detail in contrast to utilizing one upscaled texture only.

The first texture is placed at the camera position itself in order to cover the immediate surroundings. This allows for particles to further collide with dynamic geometry off-screen and thus minimizes inconsistent behavior. For each following texture the position $\vec{p}_i$ is determined using the texture scale s_i and the scale of the previous one s_{i-1} in the form

$$\vec{p}_i = \vec{p}_{i-1} + \vec{v}_c \cdot \frac{s_{i-1} + s_i}{2}. \tag{4.1}$$

Thus half the size of the first voxel storage and the second one is added together to evaluate the distance of the second storage to the first. The normalized view direction vector $\vec{v}_c$ of the camera is then scaled by that distance factor and added to the first voxel storage position $\vec{p}_{i-1}$ to get the position of the second voxel storage.

Performance is one key aspect using this cascading technique. While good voxel coverage of visible geometry can be achieved as can be seen in Figure 4.3, multiple voxelization render-passes are required. This prerequisite impacts the computational time needed significantly. Therefore the voxelization resolution for the second and following cascade levels should be kept as low as possible. Coarse resolution leads to the creation of fewer fragments that need to be processed by the graphics hardware and thus better performance.



(a) (b)

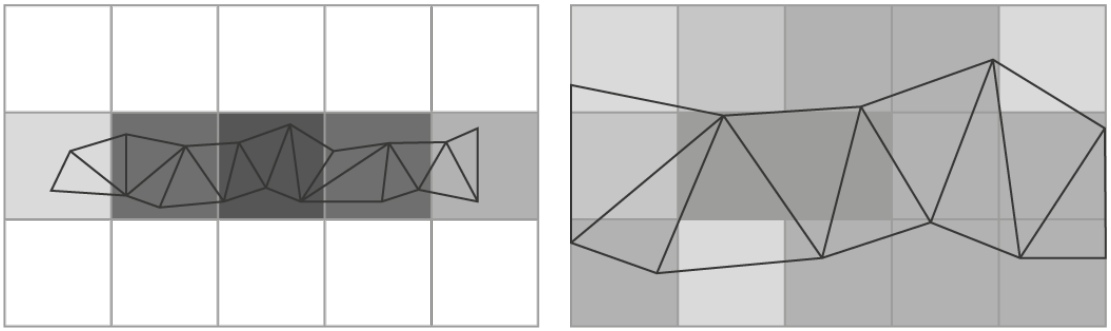
Figure 4.3: Three instances of the *Stanford Bunny* are shown that are positioned at different distances to the camera. While in (a) the reference image displays the models themselves, the cascaded voxelization results are presented in (b).

4.5 Performance Gain through Partial Voxelization

While the voxelization workload distribution of static geometry over several consecutive frames is described in Section 4.1, this improvement reduces the workload imposed by dynamic meshes. The general idea has similarities in the sense that fractions of meshes are processed.

The difference being the lossy nature of this method. While the progressive voxelization measure performs workload distribution, this method relies on reducing the mesh load. While still all necessary dynamic meshes are voxelized each frame the amount of triangles per mesh is cut short. This means that the resulting voxel data might differ from the correct voxelization result. Depending on the reduction amount and the triangle distribution the severity of the error varies.

The reason for this reduction aspect still being an interesting optimization opportunity lies in the impact the relative voxel size and the mesh scale have on the error factor. An exemplary mesh consisting of several triangles is shown with different scales in Figure 4.4. During the conservative voxelization process, the smaller scale mesh will produce more duplicate voxels because multiple triangles intersect with the same voxel location. On the other hand, the larger instance of the mesh, while generating more different voxels, results in fewer duplicate voxels being created in the same voxel locations.



(a) (b)

Figure 4.4: Triangles overlapping voxel cells are shown. Starting from white, the darker the shade, the more triangles intersect. While in (a) the small-scale mesh affects few cells multiple times, (b) shows a large-scale version affecting more voxel cells but less often.

The same principle holds for changing the voxel scale as solely the relative scale ratio between triangle size and voxel size is of importance. Therefore if many triangles are contained by, or at least overlap a voxel then reducing this amount of triangles will still result in the creation of that voxel. The only factor that changes is the data that is stored within that voxel. This observation leads to the conclusion that lower resolution or higher scale voxels increase the mesh reduction potential.

The reduction potential of the *Stanford Bunny* model at a voxel resolution of 256 lies around 30% as can be seen in Figure 4.5. This reduced workload allows for the voxelization process to be completed faster. While minor inaccuracies are introduced due to missing triangles, altogether the resulting voxel representation of the object is still very much resembling the original full voxelization result.

4.5.1 Cascaded Meshload Reduction

This reduction is also used for the cascaded voxelization technique described in Section 4.4. The further away a cascade is placed, the lower the resolution and the higher the voxel scale can be configured. As previously described these aspects allow for a more aggressive mesh reduction to be applied. A comparison of voxelization results obtained width the same amount of mesh reduction at different voxel scales is shown in Figure 4.6.

At a resolution of 256 the voxel representation of the *Stanford Bunny* shows several inaccuracies in the form of missing voxels due to the mesh reduction to 30%. However, at a resolution of 64, enough triangle information about the mesh remains to obtain a hole-free voxel representation of the surface.

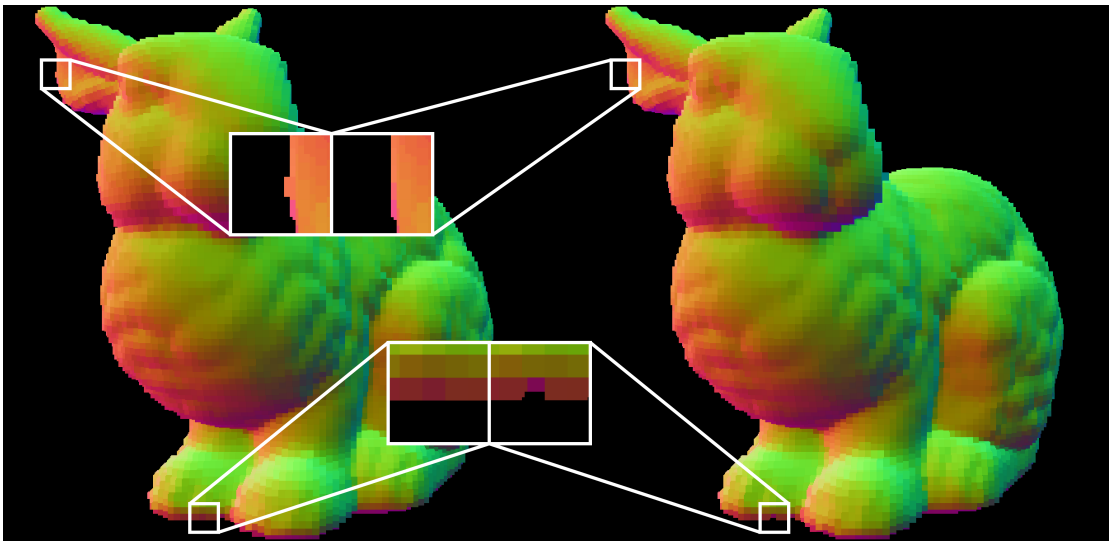
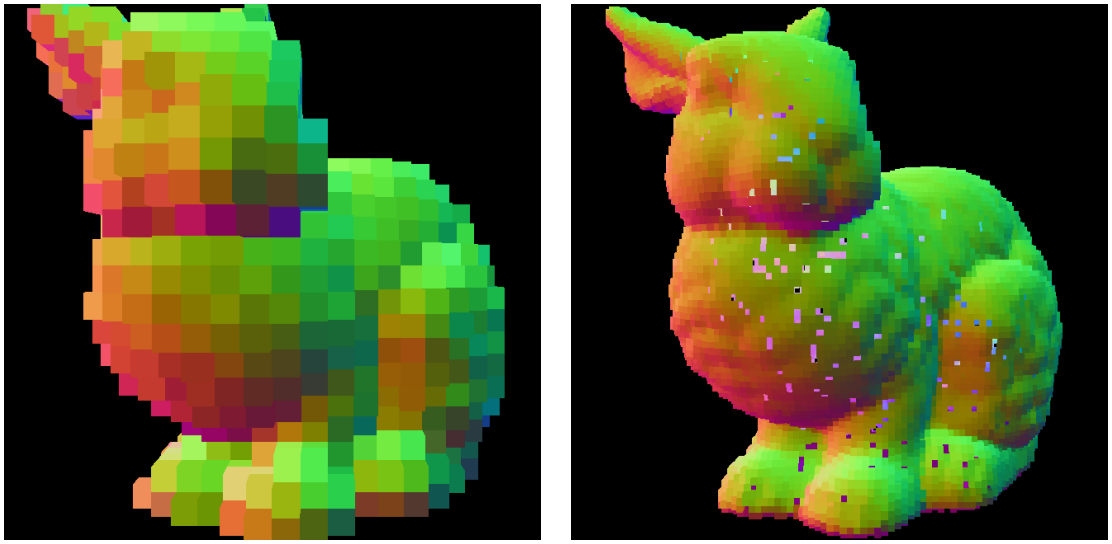


Figure 4.5: This figure shows a comparison of the voxel data obtained by voxelizing a mesh entirely to the results where only 70% of the mesh was processed.



(a) resolution 64 (30% processed) (b) resolution 256 (30% processed)

Figure 4.6: This figure shows a comparison of two voxel representations of the *Stanford Bunny* obtained by mesh reduction to 30% at different voxel resolutions.

This example displays the power of this mesh reduction technique. The workload for further away cascades can be cut down significantly. A pre-processing step is however required. The reduction factors for further away cascades need to be determined on a per mesh basis since each mesh has a different reduction potential.

4.6 Consistent Performance through Cascade Resolution Adaptation

One aspect of the voxel generation procedure that decisively influences the computational time required is the chosen resolution of the voxel storage unit. This resolution corresponds to the viewport size that is configured for the voxelization procedure. A higher voxel resolution is generally desirable as the voxel count that can be generated rises significantly. A higher voxel count enables, depending on the chosen voxel scale, a larger area of a scene to be covered by voxels or allows scene geometry to be represented in a higher level of detail.

The performance, as described before, suffers from an increase in voxel resolution as more fragments are emitted and need to processed individually. The actual amount of fragments to be processed is additionally dependent on the meshes from which voxel data is generated from. Different locations within a scene are constructed from different amounts of meshes with varying complexity. Hence the voxelization performance may vary significantly at different locations within a scene.

Thus finding a voxel resolution that operates satisfactorily performance-wise for the whole scene is a difficult task to accomplish without unnecessarily sacrificing visual accuracy by lowering this resolution overly. Therefore this adaptive voxel resolution optimization measure aims to adapt to the current workload by scaling down the resolution whenever a performance hit occurs and increasing the resolution again when fewer triangles are in need of voxelization.

The implementation of this adaptive voxel resolution adjustment algorithm relies on the cascaded voxel generation technique described in Section 4.4. There the nearest voxel storage unit offers the highest resolution while the individual resolutions of further away cascades decrease. The voxelization process utilizes the same resolutions for generating voxel data as the storage units were created with. The performance of the process thus is dependent on those cascade resolution values. This aspect is utilized to dynamically adapt the voxel resolution whenever the performance impact of the voxel generation process exceeds a specified computational time limit due to high workload.

Instead of generating only the required storage textures with the appropriate resolutions for the individual cascades, one additional texture is introduced that features the lowest resolution of all storage units and can be imagined as another cascade with the least amount of detail. For low workloads this storage is not used as the resolution adaption is not required. Therefore only the original cascade storage units are updated with newly generated voxel data.

In order to achieve the change in resolution for all voxel cascades, without the need for recreating lower resolution storage textures, the additional introduced texture plays an important role. An index is defined that points towards the nearest cascade featuring the best resolution. Incrementing this index means that the highest resolution storage now is set to be the next cascade featuring lower resolution. The introduction of additional storage is therefore required in order to be able to feature the same amount of voxel cascades for this lower resolution voxelization. On the other side decrementing this index restores the original voxel cascade resolutions as now the original texture constellation is again used. One storage unit in these scenarios will always be unused and therefore is not required to be updated by the voxelization process. This aspect ensures the performance gain upon index increase.

Additional adjustments have to be introduced in order for this optimization measure to function as intended. The positional information stored for the original cascades are not applicable to the new configuration of lower resolution cascades. Furthermore, the scale information is required to be updated as well to match the extent of the previously active cascades. Therefore a change of the index further triggers the adaption of the cascade scales to match the previously active cascade configuration. With updated scale values the cascade positions can be determined subsequently.

4.7 Management of Implemented Improvements

While normal vector compression, cascaded voxelization and progressive voxelization are always enabled, the other previously described optimization measures are managed centrally and only utilized on demand. The key controlling factors are two thresholds depicting the lower and upper voxelization time limits. Whenever the frame-time exceeds the upper limit continuously for a specific duration, those optimization measures are activated to reduce the computational time required by the voxelization process.

The first optimization that is activated is the mesh reduction measure as described in Section 4.5. Depending on the reduction potential of the scene geometry this optimization can already improve the voxelization performance noticeably without introducing severe inaccuracies to the resulting voxel representation.

Another optimization that is enabled when the previously described approach of reducing the workload fails to increase performance sufficiently is the reduction of cascade textures. For this implementation, three textures were chosen as the standard number for cascaded voxelization. In order to cut down the computational time needed for voxel generation, the furthest away texture can be skipped during the voxel update cycle.

Lastly, the dynamic resolution adaption, as explained in Section 4.6, is enabled when the voxelization time still exceeds the specified upper limit. This lowers the voxel resolution of all cascades to reduce further the computational time required to complete the voxelization process.

The lower voxelization time limit plays an important role as well. Whenever the workload lessens such that the voxelization duration does not even exceed this lower limit anymore, the optimization measures are disabled one after the other in reverse order. Therefore the first action would be to increase the resolution of the voxel cascades again. If the frame-time still stays below that specified lower time limit, the other two optimization measures are disabled as well.

The combination of these performance-enhancing measures with the previously described mechanisms allows to lower the hardware requirements for the system and offers more control over the voxelization performance at the expense of losing accuracy in terms of voxel data detail and lower voxel coverage of the scene.

Chapter 5

Evaluation

Within this section gathered performance data of the implemented and optimized system is evaluated. In order to receive accurate isolated data about computational times from individual modules and their components, a benchmark application was implemented. This benchmark features a variety of test scenarios which allows not solely the evaluation of those individual parts but furthermore enables to draw conclusions on the overall performance of the system. The benchmark was executed on multiple different hardware configurations featuring different graphics hardware as the computationally demanding calculations are executed utilizing the graphics processor. This hardware diversity allows for more accurate assessment of the GPU particle collision system concerning hardware requirements and applicability in general.

5.1 GPU Particle System

The introduced benchmark tests concerning the GPU-based particle system itself focus on the one hand on the number of particles that can be simulated at once and on the other hand help to determine the performance of the graphics hardware in fill-rate intensive situations. For these particle system test scenarios alpha blending was used.

An overview of the average performance in terms of computational time needed to complete the individual particle test scenarios is provided in Figure 5.1. The results extend from low end to top range hardware and show that all the listed graphics cards were able to simulate and render up to one million point particles. On average the computational times did not exceed four milliseconds which would allow the application to maintain a framerate of at least 250 frames per second during runtime.

For the last two tests, each particle is drawn utilizing a textured billboard sprite. The performance impact of this technique is dependent on the number of particles in need of rendering as well as the size of each of those billboard sprites used to display such a particle. Even a limited particle count of 65,000 with small billboard sprites already leads to a significant decrease in performance. While the relative increase in frame-time is similar throughout the tested graphics hardware, older generation GPUs suffer from a more significant absolute rise in computational time.

Running the test with a higher number of particles with larger billboard sprites sizes results in even higher frame-times and thus reduces the framerate even more.

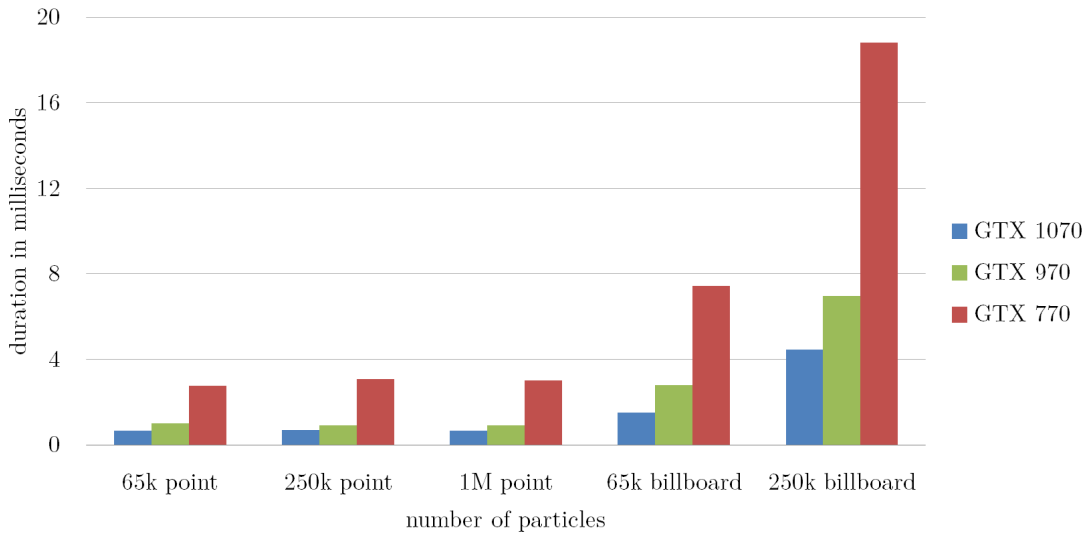


Figure 5.1: Computational times for simulating and rendering different amounts of point and billboard particles are displayed.

This baseline test for the GPU-based particle system leads to the conclusion that the rendering process bottlenecks the particle system as a whole while the simulation process efficiency would allow even more particles to be added to the system throughout the tested graphics hardware.

This realization is of importance since more per particle calculations have to be executed during the collision detection process. If the simulation step of the particle system would already impose a significant strain on the older graphics hardware, then the additional calculations necessary for identifying and resolving collisions could already lead to a drop in framerate below acceptable thresholds and therefore limit the applicability to more powerful hardware.

5.2 Voxelization

Generating the voxel-based scene representation undoubtedly imposes the most significant strain on the graphics hardware. Although the implemented technique creates a highly parallelizable workload, that therefore can be executed efficiently, the performance is highly dependent on the chosen voxelization resolution and the triangle workload that has to be processed per frame.

5.2.1 Resolution Specific Performance Analysis

The performance impact of the voxelization process at different resolutions applied to the 69,630 triangles of the *Stanford Bunny* can be observed in Figure 5.2. All listed graphics cards were able to process the workload under one millisecond for a chosen resolution of 64, enabling voxel generation at a rate of over 1,000 frames per second. A resolution of 512, however, leads to a massive increase in computational time.

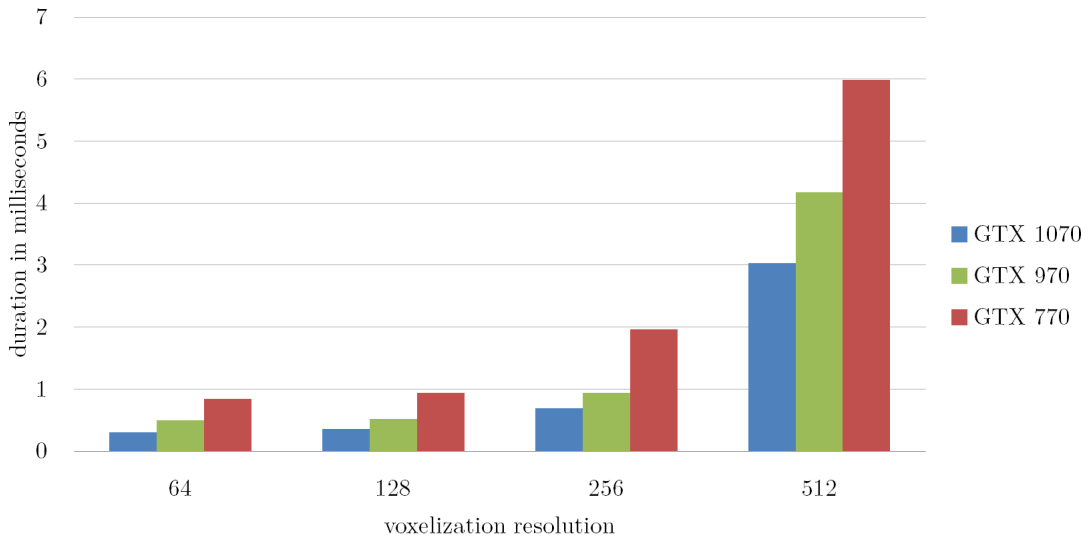


Figure 5.2: The computational time required for completing the voxel data generation at different resolutions using the *Stanford Bunny* model is shown.

The voxel generation process can be seen as a low-resolution render-pass of the scene geometry utilizing a specific set of shaders to perform the actual voxel data calculations. Rendering at lower resolutions results in performance gain as the number of fragments emitted by the hardware rasterizer is reduced. This leads to a faster rendering process as fewer fragments are required to be processed. Similarly, a higher voxel resolution results in a higher workload and therefore slower completion time.

The performance scaling is however not comparable to geometry rendering at different resolutions. Backface culling cannot be introduced to this voxel generation process as meshes are required to be processed as a whole in order to achieve a complete voxel representation. The lack of this optimization measure makes the voxelization process a more demanding one. Additionally, the procedure of generating voxel fragments differs from a typical render pass. Triangles that are rotated towards the camera emit more fragments during the rasterization process than other triangles facing in other directions as the projection of those triangles end up having larger areas.

The voxelization process, however, relies on this maximum area projection of each triangle to generate as many voxels as possible to cover the whole triangle fully. In other words independent of the orientation of the triangles to be voxelized the workload cannot be lowered by utilizing rendering optimization measures. Therefore the voxel generation performance decreases more significantly by increasing the voxel resolution than rendering geometry at higher resolutions would.

Interestingly doubling the voxel resolution leads to a predictable increase in computational overhead except for the jump from 256^3 to the resolution of 512^3. There the performance drops severely on all tested hardware configurations. While for older hardware like the GTX 770 the computational time increases threefold newer generation hardware suffers from an even more severe hit in performance.

Table 5.1: Hardware resource usage of the GTX 1070 during the voxel generation process for the *Stanford Bunny* at different voxel resolutions.

voxel resolution	shader units	texture units	memory bandwidth
64	95.07%	25.97%	34.97%
128	94.76%	55.41%	23.29%
256	96.70%	84.38%	19.20%
512	98.45%	95.30%	6.99%

The steady increase of computational time required originates from the limited amount of shader cores available on graphics hardware as well as the clock frequency. Those two factors determine the amount of shading work that can be performed each second. As the voxelization itself is executed by a set of shaders these factors heavily influence the overall voxelization performance. The rapid increase in computational time required when switching from 256 to a resolution of 512, however, is not explainable by the number of shading cores and clock frequency. Because otherwise a similar performance drop would have been registered for each resolutions doubling step.

Hence in order to investigate the actual reasons for this sudden significant drop in performance *Nvidia Nsight* is utilized. This tool allows to analyze and debug the code that is run on the GPU side of an application. More detailed data about the execution time of individual API calls as well as information about resource usage can be retrieved. A breakdown of individual frames completing the voxelization process at different resolutions is displayed in Table 5.1. For all resolutions, the shader units are utilized to their full extent. An increase in resolution, however, leads to the higher percentage of texture unit usage as more voxels are generated which corresponds to more texture read and write operations being carried out.

Increasing the voxel resolution to 512 or even higher not only again leads to full occupation of the available shader units but additionally fully utilizes the available texture units. Texture access operations cannot be performed as fast as voxel data is being generated and therefore significantly increase the computational time required. This observation explains the sudden jump in computational time required upon voxel resolution expansion to 512. For older generation hardware the performance drop already becomes apparent at a resolution of 256. Due to a lower number of both shader cores and texture processing units as well as lower clock frequency, this loss of performance occurs at an earlier stage.

Important to state here is that the atomic average technique, as described in Section 2.2.3, is utilized to honor all contributions of voxel fragments for constructing the final normal vector of each voxel. For each voxel fragment not solely one storage texture write has to be performed, but additional read operations are executed as well. Hence part of the strain put on the texture units originates from this averaging.

Thus in order for graphics hardware to perform better not only an increase in available shader units or clock frequency would be required, but a higher amount of texture processing units would be necessary as well. Otherwise either the processing of the voxel fragments themselves would be the bottleneck or the texture access operations would cause slowdowns.

5.2.2 Mesh Workload Performance Analysis

Data about the actual limit of triangles that can be processed each frame is gathered by adding models gradually. This test scenario utilizes the *Stanford Bunny* model consisting of 69,630 triangles again. While the first voxel generation performance test solely happens with one instance of that model for subsequent runs the amount of instances is increased. An overview of the voxelization performance at different workloads for different graphics hardware is given in Figure 5.3. For this scenario, the voxelization resolution is fixed to 128.

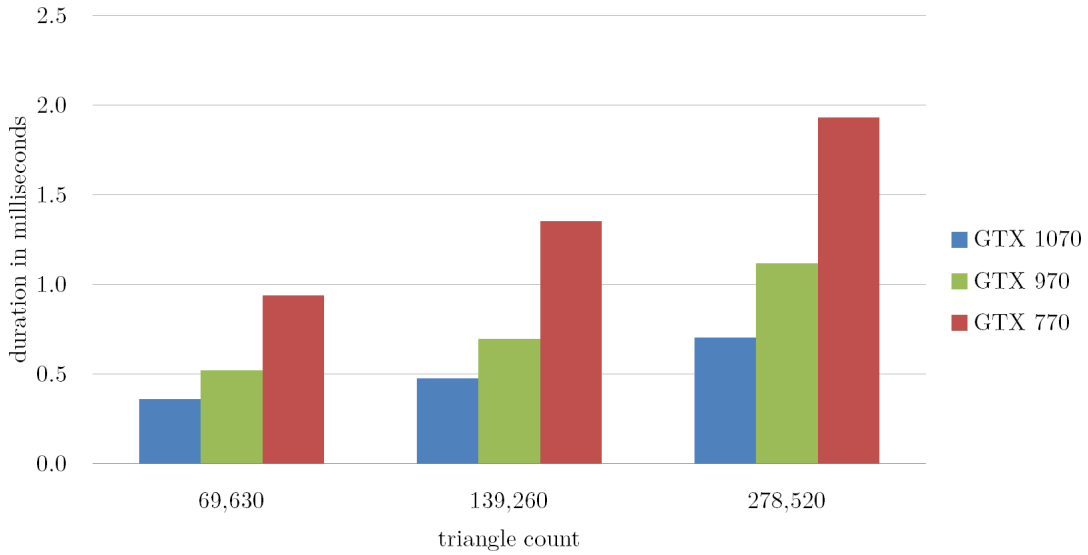


Figure 5.3: This figure shows the computational times required for performing the voxel generation procedure with different triangle workloads, in this case, varying amounts of instances of the *Stanford Bunny*.

While the computational time required for the voxel generation to complete rises when the amount of triangles increases, the impact on the overall voxelization performance remains lower than a voxelization resolution change as discussed previously. Quadrupling the workload leads to an approximate doubling in computational time across multiple tested graphics cards. Older hardware suffers more noticeable from the decline in performance as the initial absolute voxelization duration already starts higher.

With four instances of the bunny mesh being voxelized almost 280,000 triangles are processed per frame. While the performance varies significantly over different hardware generations, even older hardware from the *Kepler* hardware architecture perform well within the desired framerates of for instance 60 frames per second.

5.3 Overall Performance and Relative Workload Breakdown

In order to give an overview of the overall performance of this collision handling technique for particle systems, the benchmark additionally features a more authentic test scenario where again the *Stanford Bunny* model is used. Cascades are not utilized for

Table 5.2: Individual module frame-times in microseconds retrieved by performing frame profiling with *Nvidia Nsight* are listed here. The time difference between utilizing one and five voxel storage units is shown as well.

voxel storage units	total	render	voxel gen.	p. update	p. collide	p. render
1	4,005	146	1,470	573	211	1,605
5	4,269	150	1,483	575	459	1,602

this test scenario as this test model can be fully captured in one voxelization pass. To compensate for that aspect a higher voxel resolution of 256 is chosen. The simulation is performed with 250,000 particles at once. Those particles are visualized by rendering small billboard sprites.

Resulting average frame-times are shown for different graphics cards in Figure 5.4. While the strain put on older hardware is higher even there a frame-rate of over 60 frames per second can be maintained. Newer generation hardware based on *Maxwell* and *Pascal* architecture tested in this scenario produced frame-times under 5.5 milliseconds allowing the application to run at over 180 frames per second.

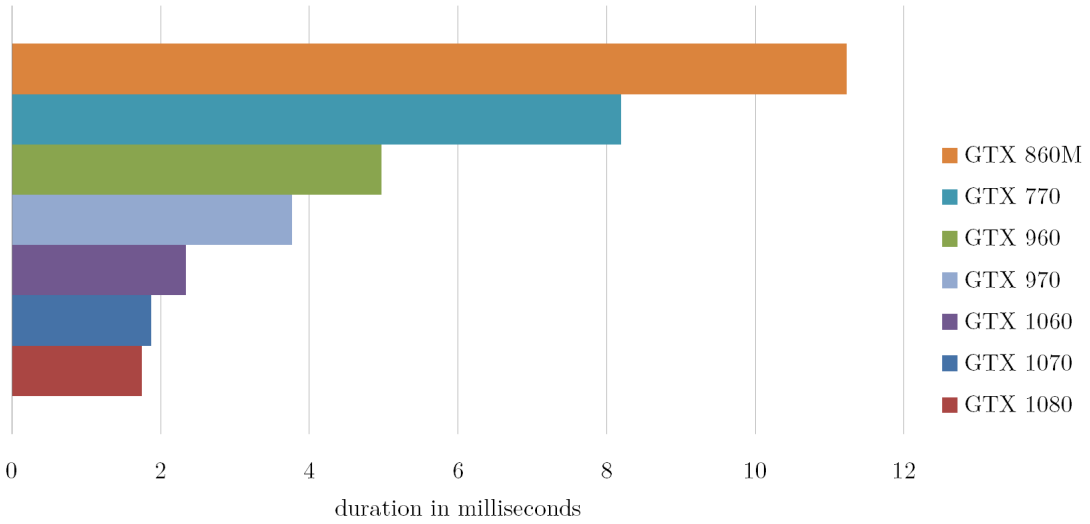


Figure 5.4: Frame-times achieved for the tested scenario involving rendering and voxelization of the *Stanford Bunny* as well as particle simulation, collision handling and rendering.

While those average frame-times convey a good picture of the performance of the whole system in action on different hardware levels another interesting aspect is the analysis of the individual modules involved. In order to gain access to a more in-depth look into the construction of a frame *Nvidia Nsight* is used once more. Exemplary frame-times and how much time was consumed by which module of the whole test application is shown in Table 5.2.

The utilized test scenario consisted of 250,000 particles being simulated and tested for collision, per frame voxelization and rendering of the *Stanford Bunny* model as well as performing billboard rendering of those simulated particles. The resolution of the voxelization was set to 256. Two tests were conducted in order to gain a more detailed view of the performance of the collision detection module. For the first analyzed frame, only one voxel storage texture was provided to perform collision detection with while for the second frame a total of five textures were utilized.

The individual frame-times show that both the voxel generation procedure and the billboard rendering process consumed the majority of the total elapsed time. With a combined duration of 3.08 milliseconds for the first frame and 3.09 milliseconds for the second frame, the voxel generation and billboard drawing operations consume over 70% of the total elapsed time for both analyzed frames. Therefore geometry rendering, particle system update and collision detection only account for less than 30 percent of the elapsed time of each frame.

5.3.1 Collision Detection Efficiency

Useful information about the collision detection performance can be seen in Table 5.2 as well. With the help of one voxel storage, the collision algorithm required 0.21 milliseconds to complete. Including a total of five different voxel storage textures during collision identification phase raised the computational time to 0.45 milliseconds. The additional texture access operations impose a high workload on the texture processing units as a high amount of particles are checked for collisions simultaneously.

Five such voxel storage textures are utilized when cascaded voxelization, as described in Section 4.4, is performed. Three voxel cascade storage units and two additional textures holding static voxel data are required. Thus the additional elapsed time during the collision handling phase encountered when using five lookup textures is caused by the voxel cascades performance optimization. Hence the gained advantage performance wise, during the actual voxel generation phase, is mitigated slightly by this added time requirement for the collision handling algorithm to terminate. Looking at the total frame-times from both analyzed frames, the fraction of time required in comparison to the voxelization or particle render phase is low, however. While the single texture lookup took up 5.3% of the frame-time the five texture variant required 10.8%.

5.4 Progressive Voxel Generation

In order to be able to investigate the individual optimization measures introduced to the collision detection system, the benchmark furthermore features test scenarios that focus on those techniques and their impact on the performance. This progressive method spreads out the workload of voxelizing the surrounding static geometry over multiple consecutive frames. Hence one test scenario was created where this delayed voxelization process was executed while reference performance data was gathered as well with this optimization disabled.

Frame-times gathered from running both scenarios can be observed in Figure 5.5. At 0.1 seconds the line graph shows the power of the workload distribution technique. For both displayed graphics cards a significant temporal increase in computational time is

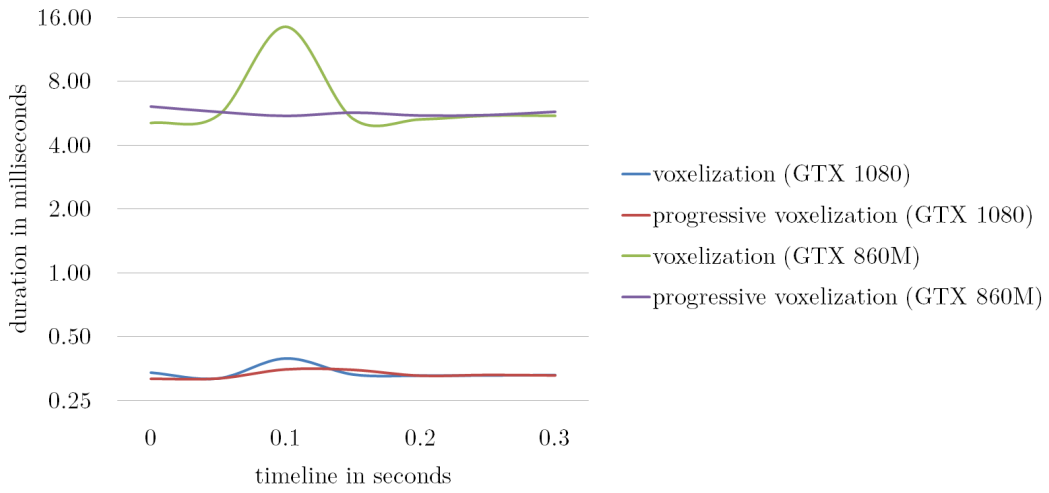


Figure 5.5: This line graph shows the impact of the static voxelization on the frame-times as well as the frame-times achieved by applying the progressive voxelization technique.

visible as the workload of the static geometry is added atop of the continuous dynamic geometry workload. The workload distribution achieved by this progressive voxelization allows for a much more consistent framerate. Hence stuttering originating from sudden changes in frame-times can be effectively avoided with this technique.

Especially noticeable is the frame-time smoothing achieved on low-grade graphics hardware like the notebook graphics card GTX 860M. The already high amount of time required for completing the dynamic voxelization in combination with the added static voxelization workload leads to a temporary frame-time increase of up to 10 milliseconds. With progressive voxelization, these frame-rate jumps, and thus stuttering, can be prevented effectively.

For these tests, the workload was split up into five model groups. Each frame 10% of such a model group is processed which results in the completion of the voxel generation process after 10 frames per model group. For an exemplary frame-rate of 60 frames per second, this means that six model groups could be completed within one second. Depending on the number of meshes this progressive voxelization has to process, this extensive delay, however, might lead to completion times of up to multiple seconds.

This workload distribution optimization measure proves to be useful for avoiding sudden and short-lived performance hits when static geometry imposes additional voxelization workload. However, this distribution should not be performed extensively since the collision detection results obtained from static voxel data will be inaccurate until the whole static scene geometry has been fully processed.

5.5 Voxel Cascades

With the introduction of multiple cascades for generating voxel data of dynamic geometry, the voxel coverage of scenes can be maximized. Instead of having to use one high-resolution voxelization pass to cover as much of the visible dynamic geometry as

possible now multiple lower resolution voxel generation passes, of different parts of the view frustum, are performed. As the impact of the chosen voxel resolution profoundly impacts the computational time required, this optimization measure reduces voxel resolution to improve performance as well.

The test data was retrieved from continuously performing dynamic geometry voxelization on different triangle workloads. The ordinary one pass voxelization was performed with a resolution of 256. The three cascaded voxel passes were performed at a resolution of 128, 64 and 32. The resulting performance gain can be seen in Figure 5.6.

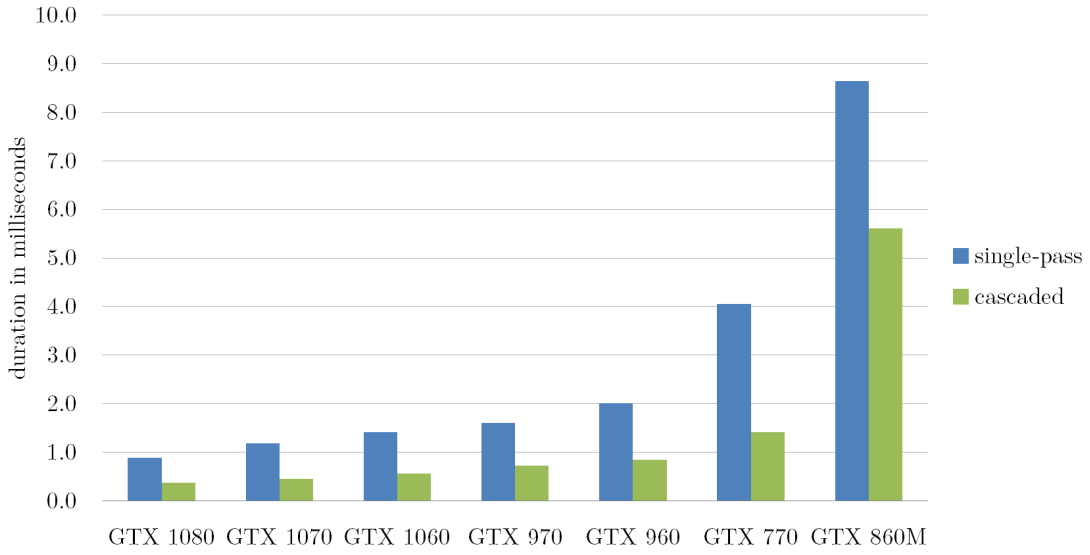


Figure 5.6: The difference in computational time between the single-pass high-resolution voxelization method and the lower resolution multiple cascade approach is shown.

Throughout all tested graphics hardware, a significant improvement of frame-times could be achieved with the help of this optimization measure. The computational required either was bisected or reduced even more. This observation holds true for all graphics cards except for one. The GTX 860M showed lower performance improvements in frame-times. The reason, as already discussed in Section 5.2.1, for this outlier is related to the significantly more limited shading and texturing capabilities of this mobile-grade graphics processing unit. Performing voxel generation at a resolution of 128 therefore already proves to be an issue as the texture unit usage is already maxed out. Hence the reduction from 256 to a resolution of 128 does not have the same performance gain impact that is observed with newer hardware.

However, this technique also possesses the disadvantage of adding additional computational time to the collision detection phase. Because more voxel storage textures are utilized more texture lookups have to be performed for identifying particle collisions. The additional time required is discussed in Section 5.3.1.

This additional time spent during the collision handling phase, however, is acceptable due to the fact that this increase is independent on the scene geometry itself and therefore much more predictable. Furthermore, the performance gain achieved by the cascaded approach during the voxelization phase outweighs this added cost on each

tested graphics card. Decreasing the computational time needed for the highly scene dependent voxelization is worth the much more constant and thus predictable additional expense during collision detection.

5.6 Partial Voxelization

Reducing the number of triangles to be processed each frame leads to lower time consumption, which is desirable. Section 4.5 describes how such a mesh reduction can positively influence the overall performance at the cost of accuracy of the resulting voxel scene representation. Due to this loss of precision, this technique is solely utilized in combination with the cascaded voxelization technique. The nearest cascade covering the area around the active camera performs the voxel generation procedure ordinarily. Further away cascades however only utilize subsets of each model to generate voxel data. For this test case, the second nearest cascade is allowed to ignore 15% of each mesh during voxelization while the third cascade skips 40%.

The gain in performance becomes apparent when looking at Figure 5.7. Varying amounts of models are encountered over the course of the test track to ensure different workloads being imposed. Interesting to see is that the relative influence of this mesh workload reduction is hardware dependent. For the GTX 1080 graphics card an average decrease in computational time of 9.7% is achieved while the GTX 860M only benefits from a 3.3% performance boost.

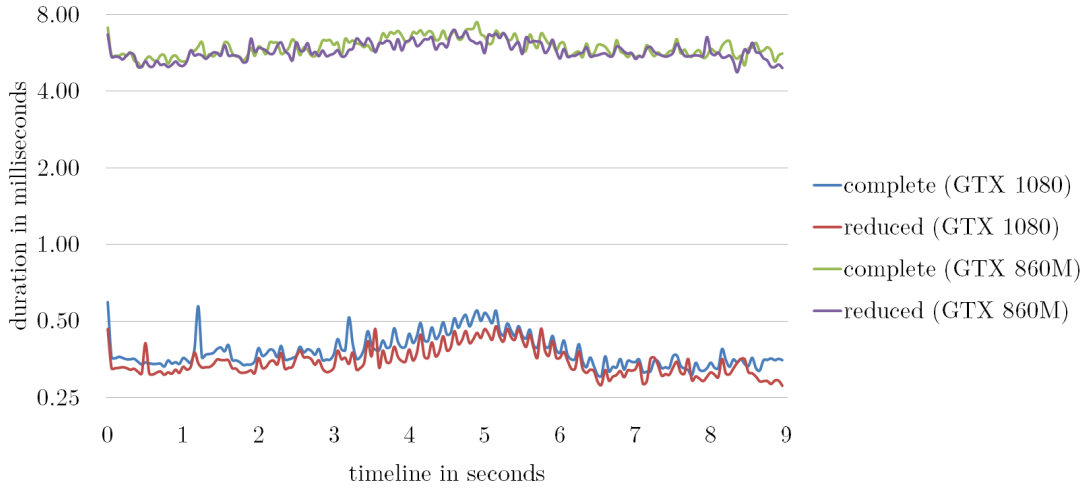


Figure 5.7: Frame-times achieved over the course of the test scene are displayed with and without mesh reduction enabled.

The reason for this lack in performance is explained in Section 5.2.1. The GTX 860M features both the fewest shader cores and texture units combined with the lowest core frequency. This leads to the hardware maxing out the fastest during voxel generation, even at a resolution of 128 where the other tested graphics cards are not utilized to their full extent yet.

Therefore the voxel generation of the nearest cascade with a resolution of 128 is costing that graphics card the most time to complete as the workload cannot be calculated in a fully parallel way. Whereas the voxelization process, of the other two cascades with respective resolutions of 64 and 32, is not limited by the shader or texture units. Hence the voxel data for those cascades can be calculated more efficiently.

The workload reduction for those two voxel cascades thus does not impact the overall performance as much as the majority of the voxel generation time will be spent on the first cascade overstraining the hardware. Reducing the number of triangles of the first cascade would, for this graphics card, improve the performance more significantly. However mesh reduction performed with small-scale voxels, as happens to be the case for the nearest cascade, introduces non-neglectable inaccuracies upon the creation of the voxel scene representation. Therefore the reduction is limited to further away cascades where larger voxel sizes mitigate this accuracy issue.

Furthermore important to state is the fact that this relative performance gain result measured is highly dependent on the utilized test meshes, voxelization resolutions, and the chosen reduction factor. As described in Section 4.5 that mesh reduction factor is hardly definable as a global constant. The amount of triangles that can be skipped is mesh dependent. Therefore the reduction amount should be identified and set on a per mesh basis to avoid omitting too few or too many triangles. Low detail meshes that consist of few triangles should not be further reduced at all during the voxel generation.

While this technique of reducing the triangle workload for the voxelization process proved to be working, the relative gain in performance was inconsistent throughout tested hardware. Another issue with this optimization measure is the additional preprocessing work required to identify the reduction potential of individual meshes for individual voxel cascades. Hence this reduced voxelization technique only pays off if a good part of the scene geometry offers good reduction potential.

5.7 Cascade Resolution Adaptation

Another quite common scenario in video games is an unbalanced distribution of mesh complexity. Different types of meshes with varying complexity populate different parts of the environment. For the voxel generation, this implies that the workload also varies massively depending on the current position of the camera within the scene. This dynamic cascade adaption optimization counteracts increasing triangle counts by scaling down the voxel resolution.

To test the performance of the voxelization procedure under different mesh workloads, the test scene used consists of spatially varying mesh density. One test run is executed to gather baseline performance information while the second run features the dynamic resolution adaption optimization measure. Only the voxel generation procedure is performed, no additional particle system related computations or rendering operations are executed to gain pure voxelization performance data.

A comparison between both techniques is shown in Figure 5.8. The upper threshold for triggering the resolution bisection was set to 0.44 milliseconds, 2,250 frames per second. Dropping below that framerate for more than half a second led to a decrease in resolution to improve performance. On the other end, the lower threshold for triggering the resolution doubling was set to be 0.4 milliseconds or 2,500 frames per second.

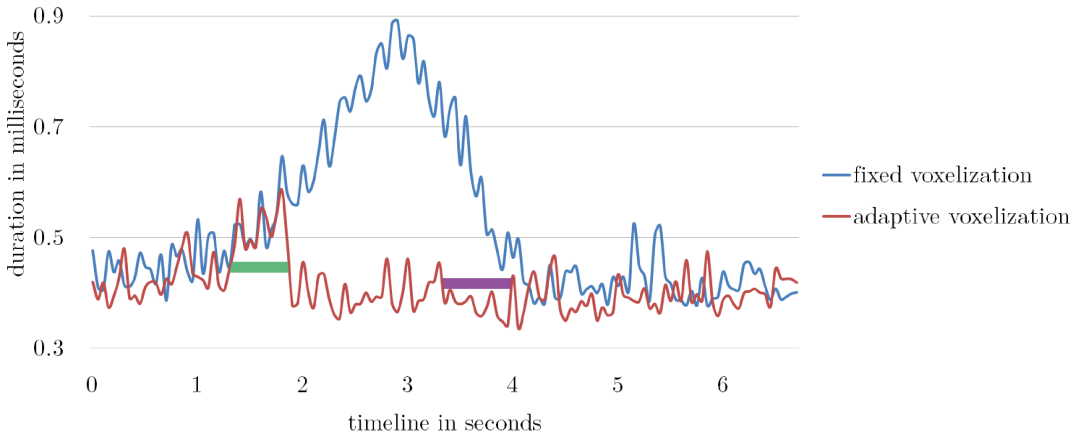


Figure 5.8: This figure shows the frame-times of two voxelization test runs of the same scene consisting of spatially varying amounts of triangles showing the performance advantage of dynamic resolution adaption.

The green area at approximately one and a half seconds shows where the upper threshold was exceeded long enough to trigger a resolution reduction. At second three and a half, the purple dashed line indicates that the time required for the voxel generation drops below the lower threshold for at least half a second as well. This, therefore, triggered the resolution increase back to the original configuration.

The gain in performance is significant. While the frame-times more than double under the increased workload over the course of the test run for the constant resolution variant, the resolution reduction allows keeping the framerate almost constant. These results show how powerful this dynamic resolution adaption is to keep the voxel generation duration from skyrocketing uncontrollably when more detailed sections of a scene are traversed. While the resolution reduction leads to a coarser voxel scene representation, the aspect of being able to control the duration of the highly scene dependent voxel generation process makes this optimization measure a highly desirable one.

Chapter 6

Conclusion

As mentioned in Chapter 1, two main aspects of the presented way of providing particle collision detection capabilities on the graphics processor determine the overall applicability of this system. Efficiency regarding the computational time required to perform update calculations within a single frame is the first key variable. Especially in interactive environments where the available time per frame is heavily limited. Therefore the performance evaluation results concerning basic implementation and optimization measures are discussed. Although the visual output of the system is not the main aspect of this work, nevertheless those results also determine the relevancy of such a collision detection system. Therefore visual results are discussed as well.

6.1 System Performance Discussion

The frame-times gathered from the initial test scenario without optimization measures applied, which are evaluated in Section 5.3, show that the required time differs significantly depending on the utilized graphics card. The computational time required for the *Pascal* generation of graphics cards revolves around two milliseconds. This duration is tolerable considering the fact the billboard sprite drawing process, that is executed for each particle, as well as the geometry rendering process also significantly contribute to this frame-time.

For older generation hardware the frame-times increase drastically up to over 10 milliseconds as can be seen. These computational times are unacceptable although a frame-rate of 60 can still be maintained. However, in video games, the available frame-time on the GPU has to be shared with other graphics related operations like rendering complex scene geometry and applying post-processing effects. Hence wasting 10 of 16.6 available milliseconds on particle effects is not possible for most interactive applications.

The investigation performed in Section 5.1 to evaluate the undisturbed raw performance of the implemented particle system leads furthermore to the conclusion that older graphics hardware not only struggles to update the voxel scene representation in a timely manner but furthermore has problems drawing that many billboard sprites simultaneously. Therefore reducing the number of particles to be drawn can already significantly boost the performance and thus expand the applicability, of the whole particle system with collision handling, to older hardware.

The frame analysis conducted in Section 5.3 shows furthermore that the geometry render, particle system update and collision detection phases consume less than 30% of the frame-time. The most time-consuming process beside the already mentioned billboard drawing is the voxel generation which is responsible for over 30% of the frame-time consuming over 1.4 milliseconds on a GTX 1070 graphics card. This high computational overhead is one reason why multiple optimization measures were introduced to improve the performance.

6.1.1 Progressive Voxel Generation

The distribution of the voxelization process of static geometry over several consecutive frames helps to reduce the temporary spike in the additional computational time required that otherwise would be caused by this increased workload within a single frame. The impact this frame-time smoothing has is however dependent on the configured distribution amount. Splitting up the workload over more consecutive frames leads to a better smoothing result as the additional workload imposed per frame is smaller. The opposite effects are observable when fewer frames are used for the workload distribution.

The resulting framerate smoothing impacts are highest for low-end graphics hardware as can be observed in Figure 5.5. The already high frame-times that are owed to the voxel generation of dynamic geometry skyrocket under the additional workload the static geometry imposes without this optimization enabled.

Hence all in all this optimization is necessary whenever static geometry is also required to be voxelized during runtime. Otherwise, stuttering is unavoidable due to this temporary increase in voxelization workload upon triggering the static voxel generation.

6.1.2 Voxel Cascades

The performance gain of the cascaded voxelization approach is given by the fact that for ordinary voxel generation of dynamic geometry a resolution of 256 is utilized for test scenarios. The highest resolution cascade, however, is set to 128 with each further away cascade featuring half of the resolution of the previous one. This configuration lowers the computational time required as can be seen in Figure 5.6. All voxel generation steps required to generate data for each cascade happen at a much lower resolution and therefore perform better although multiple voxelization steps are required.

The gain in performance is significant. Additionally, a better voxel coverage of the view frustum can be achieved while furthermore reducing the memory footprint. Nevertheless, a compromise in performance has to be made with this optimization as well. Although a speedup during the voxelization process is achieved the collision handling requires more computational time to complete. The reason being the additional texture lookups that have to be performed in order to consider all voxel storage textures. The amount of additional time required, as can be seen in Table 5.2, lies around 250 microseconds. This additional cost is however outweighed by the larger time savings achieved with this cascaded voxelization approach.

This optimization measure is thus considered a valuable one that should be introduced permanently. Not only the better scene coverage that is achieved by the introduction of cascades but furthermore the additional gain in performance makes this optimization a perfect addition to the collision detection system.

6.1.3 Partial Voxelization

Reducing the number of triangles is another way of speeding up the voxel generation process. This mesh reduction inevitably leads to the introduction of inaccuracies in the resulting voxel representation, as explained in Section 4.5.

The amount of reduction that can be performed before severe inaccuracies like missing voxels are introduced is entirely dependent on the individual models. The *Stanford Bunny* model consists of solely one mesh that features similarly sized triangles. This consistency offers a higher reduction potential. Other models like the *Sponza* model consist of up to multiple hundreds of meshes with different sized triangles that aggravate finding a reduction factor that works for all meshes involved. Therefore the resulting reduction is much less effective for such a model. In order to achieve better reduction results, each mesh of that model would need to be processed separately, and thus the resulting reduction factors would then be assigned per mesh instead of per model.

Furthermore, this reduction factor changes whenever the voxel sizes of the cascades or the scale of the model changes. Thus in addition to the pre-processing step required to determine the mesh reduction potential of individual meshes, further logic is required that is responsible for updating the reduction factor values accordingly.

This optimization cannot be recommended fully. The additional pre-processing and update logic required to figure out the reduction potential of each mesh may not be worth the inconsistent performance gain results obtainable as shown in Figure 5.7.

6.1.4 Cascade Resolution Adaptation

Another big concern is the change in performance of the voxel generation process during runtime. Due to the fact that the number of triangles processed influences the amount of time required to complete the voxelization, the performance is dependent on the scene geometry. As neither mesh complexity nor distribution is usually consistent throughout a scene, the time that is spent on generating voxel data changes continuously and is therefore hardly predictable.

The dynamic resolution adaption effectively allows for a more consistent frame-time during runtime by reacting to increased workloads by lowering the effective voxel resolution. As discussed in Section 5.7 this reduction is applied when a significant rise in frame-time is detected and thus prevents a further increase in computational time.

Although this resolution lowering process corresponds to a decrease in collision detection accuracy, the additional control gained in regulating the voxelization duration is worth the compromise. The significant time savings that were achieved for the test scenario can be seen in Figure 5.8. The frame-times can be kept almost constant while without this optimization the required time more than doubles.

6.1.5 Conclusion

The evaluation results regarding performance show that the introduction of such a system for detecting collisions of particles in video games is feasible. However since the performance is tied to the composition of the scene and the utilized graphics hardware, a definitive assessment of the applicability of this system is hardly possible.

The discussed optimization measures help to increase the efficiency of the voxelization process. The introduction of voxel cascades allows reducing the effective voxel resolution through the introduction of multiple lower resolution cascades that allow furthermore better coverage of the visible scene geometry. The required time for the voxel generation to complete is cut in half. As described in Section 5.2.2 quadrupling the number of triangles leads approximately to a twofold increase in computational time.

Therefore the cascaded voxelization technique theoretically allows for the quadruple amount of triangles to be processed in the same amount of time as would be possible without this optimization. A baseline workload of 280,000 triangles led to a maximum voxel generation duration of two milliseconds when using a resolution of 128. This enables for complex scenes like the *Sponza* model to be voxelized each frame, for instance.

This number should, however, not be seen in an absolute way but more as a guideline as the voxel generation itself is further dependent on the triangle sizes as well as the available shading and texturing power of the utilized graphics hardware. The actual possible voxelization performance may differ significantly. The GTX 1070, for instance, requires less than half of the time of the GTX 770 to complete the process and therefore could handle a bigger workload.

Since the chosen graphics hardware greatly influences the performance of the voxel generation process, the applicability of this system is limited to newer generation graphics cards based on *Kepler* and *Pascal* architecture as their shading and texturing capabilities exceed those of their predecessor cards greatly. Especially for *Maxwell* generation graphics hardware like the GTX 770 and the GTX 860M, where not only the voxel generation but additionally the billboard rendering test scenarios showed performance issues, this system should not be introduced to provide collision detection. If utilized the voxelization resolution should be kept at 128 or lower at all times and the number of billboard particles should be limited as well.

Keeping in mind that this workload is reserved for dynamic geometry only as static models are not required to be voxelized on a per frame basis. Furthermore solely scene geometry near the camera is considered for the voxel generation. Hence according to these facts and the gathered performance data, the voxelization technique is feasible to be introduced to various game scenarios. Depending on the complexity of the geometry, however, a scene should consist of a majority of static meshes to keep the voxelization workload as low as possible. For games where high-resolution dynamic geometry dominates the scene, this voxelization technique is not applicable as the overall workload would be too high. The voxel generation process would possibly still be completed within the available amount of time per frame. However the little remaining time each frame would limit other GPU related operations significantly.

The adaptive voxel resolution measure effectively reduces the negative impact on frame-times upon temporary increases in workload caused by dynamic geometry. When static geometry voxelization is triggered the additional triangles to be processed are distributed over multiple consecutive frames using the progressive voxelization technique. Thus otherwise occurring stuttering due to temporary spikes in frame-time can be avoided. These two techniques in combination with the advantages of the cascaded voxel generation approach offer good control over the voxel update phase and provide means of handling static geometry as well.

6.2 Visual Performance Discussion

Providing voxel-based scene representations as means of detecting particle collisions not only has to happen in a timely manner but also provide visually satisfying collision handling results. The outcome can be observed in Figure 3.6. The chosen voxelization resolution in combination with the configured voxel size determines the quality of the voxel scene representation. This aspect becomes further apparent when looking at the different collision detection and response results.

Even for meshes with low levels of detail inaccuracies can be observed when the voxel scene representation consists of large-scale voxels. A triangle that leads to the creation of such a larger-scale voxel thus creates a larger volume in which particles collide. The resulting voxel representation of the surface of the geometry is thicker and more inaccurate than with smaller-scale voxels. This thicker surface leads to early collision detection and response of particles because particles collide with occupied voxel locations much earlier than they would with lower-scale voxels or the actual mesh surface.

The smaller the voxel size, the more this issue is mitigated as the mesh triangles can be fitted much more tightly with smaller voxels. While this early detection can still occur the inaccuracy is reduced as the maximum distance of the furthest away location within a voxel to the actual surface of the geometry the voxel represents is smaller.

However, this observation does not rule out voxel generation with a low resolution of 64 for instance. Resolution alone does not control the actual voxel size. This size is dependent on the resolution as well as the scale of the volume that is to be covered by those 64^3 voxels. Hence low resolution in combination with a small volume to be covered can still be utilized to obtain accurate voxel representations.

Additionally, for voxel data that is located further away from the camera low resolutions and higher voxel scales are allowed to be utilized as well. Due to the fact that geometry located in the distance will occupy fewer pixels on the screen than in the front less detail about that object is observable. With less visible detail less detailed voxel data is sufficient to perform collision detection without introducing noticeable inaccuracies.

With the help of the normal vector deviation as well as the trilinear interpolation of those voxel normal vectors, the resulting collision response results hide the finite detail of the block-shaped geometry representation the voxel data provides. As long as the voxel resolution and scale is configured in a reasonable manner the collision handling is performed satisfactorily. Reasonable in this case is defined by the voxel scale to triangle size ratio. Too large voxels encompass vast amounts of triangles and therefore are not fit to represent the geometry in a detailed way.

6.3 Limitations

The limited voxel resolution will always introduce inaccuracies as the collision detection results can never be perfect. While a finer resolution and smaller voxel sizes yield more accurate results, the area that can be covered is significantly smaller as well. Therefore a compromise between voxel scene coverage and representation quality is unavoidable.

Also, triangle sizes of the scene geometry need to be taken into consideration upon voxel resolution and size specification. Small and detailed meshes, for instance, will lead to the generation of duplicate voxels at the same position. If the method described in

Section 2.2.3 is used to store an average of the normal vectors instead of overwriting them whenever a new duplicate voxel is generated, the number of duplicate voxels should not exceed 255. This limitation is imposed by the 8-bit capacity of the alpha value of the voxels. The alpha values are utilized to keep track of the number of values that already contributed in order to be able to weigh new contributions accordingly. If more than 255 values need to be averaged, the result cannot be calculated accurately anymore. Due to the fact that the exact generation order of voxel fragments is nonpredictable, not always the same 255 voxel fragments will be averaged. This can lead to differences in the outcoming final voxel normal vector each time the voxelization process is executed.

Further limitations include the voxel management of static geometry. Depending on the movement speed of the camera through the scene the re-voxelization of static geometry on demand might not be feasible to perform. For instance in racing games where the camera is attached to a car driving at high velocities the change in scenery happens quickly and continuously. In such a scenario the voxelization of static geometry on demand would be triggered at a high frequency as well. In the worst case, this static voxel generation would need to happen at such a high rate that constant re-voxelization, like for dynamic geometry, would be required for static geometry as well.

For these scenarios, this form of static voxelization on demand is not feasible. Instead, voxel generation for static geometry could be performed once as preparational step upon loading the scene. This would allow eliminating the additional static workload being imposed on the voxelization module during runtime. In order to be able to hold static voxel data of large scenes in video memory, an efficient storage method is required. One possibility would be to utilize a sparse voxel octree as proposed by Crassin [4]. However, depending on the actual scene size and the voxel resolution used, multiple instances of such an octree may be required to store voxels of different sectors of the scene. Otherwise, the depth level of the tree would affect the required traversal time negatively.

6.4 Future Work

While this project already explores optimization methods to increase performance, numerous other measures could be taken to further improve the performance in terms of efficiency and visual accuracy.

6.4.1 Triangle Count Limited Progressive Voxelization

The mesh load distribution as described in Section 4.1 allows for a more coherent framerate to be maintained for when static voxelization is required to be executed as well. However, this optimization measure does not take into account the complexity of the static models in need of processing. Models consisting of multiple meshes or otherwise possess high amounts of triangles are treated the same as low-resolution models.

The implemented variant of the workload distribution operates on model groups. A model group consists of multiple static models for which the voxel generation process is executed simultaneously. As only one model group is allowed to be active during a single frame the groups are processed consecutively over multiple frames. To further split up the number of triangles within such a model group over several consecutive

frames, only a fixed percentage of each mesh within the current model active group is processed within a single frame.

While this technique already heavily mitigates temporary framerate surges, this variant does not lead to an equal distribution of triangles. This issue exists due to the fact that only the amount of models is considered for this process instead of further incorporating the triangle counts of the individual models as well. If one model group consists of highly detailed meshes whereas another model group only features less detailed geometry the workload imposed by the first model group is higher.

To solve this issue, the triangle amount is required to be accessible either on a per model or per mesh basis. This information is then used to construct more balanced model groups. Thus mesh groups are not limited in the number of meshes anymore as the workload now is controlled by triangle count.

6.4.2 LOD

An integral part of every game engine that should be able to handle large-scale environments is a level of detail system, LOD in short. For each model several lower quality models, regarding triangle count and texture resolutions, are generated either manually or automatically. If the distance of a model to the camera is small, the full resolution version of the mesh is drawn. The lower resolution versions are drawn instead when the distance to the camera increases.

This lower resolution mesh variants can further be used to boost the performance of the voxelization process. Instead of the highly detailed models, the LOD variants could be used for voxel generation. With fewer triangles to process, the computational time for completing the voxelization procedure would be lowered drastically.

Especially in combination with cascaded voxelization utilizing such a LOD system could improve the performance even more. Each cascade would be assigned a specific level of detail model variant. Only those model variants of the same level of detail are then used for generating the voxel data for individual cascades.

6.4.3 Cascade Priority

As described in Section 4.4 multiple voxelization passes for dynamic geometry ensure good coverage of the view frustum at reasonable resolutions. However, the additional strain that is put on the graphics hardware impacts the performance of the voxelization system. To lower the computational time needed prioritization of the single cascade levels can be introduced. The first voxel storage is assigned the highest priority as the immediate surroundings are captured there. The other cascades are of lower priority the further away the voxel storage is located from the camera. If the workload for the voxelization system causes the specified computational time limit to be exceeded, this priority optimization can be activated.

The reduction of the workload is achieved by filling those the voxel storage texture with high priority first. Dynamically, with the help of the previous frame-time, the system decides how many of the voxel storage textures can be updated and which have to be skipped in order to avoid framerate drops when dealing with heavy workloads.

In other words, the amount of cascades is determined at runtime based on the number of models and their complexity to maintain reasonable frame-times. This performance

optimization, however, imposes the disadvantage of losing valuable voxel data of further away dynamic geometry if the system is forced to skip the update cycle for low priority voxel storage textures. For collision detection two ways of dealing with this issue are feasible. On the one hand, the voxel data of those storage units can be discarded which leaves no possibility of further detecting collisions with dynamic geometry at that locations. On the other hand, the voxel data can be preserved to continue collision detection support with those not up to data voxel storage units. However, using this variant means that the possibility of particles colliding with outdated voxel information exists.

6.4.4 Mesh Priority

Further control over the voxelization performance at high workloads could be achieved by prioritizing the models involved. Similar to the method described in Section 6.4.3 models with higher priority values are considered in the voxelization process whereas lower priority models are skipped depending on the workload that is imposed on the voxel generation system. This optimization measure would allow mitigating the surge in computational time required upon a sudden workload increase of dynamic geometry.

Priority values would need to be calculated each frame for each dynamic mesh that is located close enough to be voxelized. The priority factor is dependent on factors like the distance of the mesh towards the camera for instance. Proximity yields higher priority values in order to avoid missing voxels at close range. Lacking voxel data would lead to particles passing through geometry which becomes much more apparent at close range than at a distance.

Another critical factor that is needed for calculating a model based priority value is volume. Models that have a low volume in comparison to that of the voxel storage units will not contribute much information in the form of voxel data while still occupying the shader units of the graphics card as each triangle of the model still needs to be processed. In order to avoid these unnecessary calculations as much as possible, high volume meshes should be assigned higher priority values compared to low volume meshes.

6.4.5 Early Sub-Mesh Culling

One essential method for reducing the number of triangles to be voxelized each frame is to check each model for overlap with the individual voxel storage units. Axis-aligned bounding boxes are used to determine overlapping regions. This check is sufficient to determine whether or not the *Stanford Bunny* should be included in the voxelization process for instance. For the *Sponza* model this overlap check could be executed more thoroughly.

A comparison of the *Stanford Bunny* and the *Sponza* scene shows the differences in their underlying structure. While the bunny model only consists of a single mesh the *Sponza* scene model is built from over 300 meshes. Over 200,000 triangles are utilized to create this model. Whenever the bounding box check detects an overlap with this model and a voxel storage unit the whole model is voxelized no matter how small or large the overlapping area may be.

Small overlapping regions in reality often only affect a small amount of the meshes from the whole model. Thus instead of always processing the whole mesh, further overlap tests could be performed on a per mesh basis. This measure would allow for unaffected

parts of the model to be culled. Mesh culling requires however for each mesh to be equipped with a bounding box. These bounding boxes furthermore would need to be updated whenever the model is transformed in order to ensure the correctness of the overlap tests.

This optimization could be enabled individually for models where the mesh structure is similar to the *Sponza* model where multiple meshes are used to build the complete model. One aspect of this optimization to investigate, besides measuring the performance difference of the voxelization process, is the additional computational time required for bounding box update calculations and overlap tests on the CPU. If the strain put on the CPU would be too high, then the performance gain on the GPU-side would not be of much value.

6.4.6 Continuous Collision Detection

The current implementation features discrete, a posteriori, collision detection only. This approach suffers from detection misses that are caused by tunneling. Tunneling occurs when a particle travels at high velocity towards a thin wall mesh. Due to high velocity, the particle passes, within one timestep, through the wall. This means that no collision was detected in neither timestep. This issue cannot be avoided when such a posteriori technique is used for identifying colliding entities.

While the discrete collision detection approach works well for the implemented particle system, there are specific corner cases that can cause the system to fail. High-velocity particles in combination with small voxel sizes, caused by high-resolution voxelization, can lead to this tunneling effect. Thus if such a system configuration is desired, an a priori collision detection technique needs to be implemented in order to receive reliable and consistent results.

For this system continuous collision detection could be implemented as described in Section 2.3.2. However, since the calculations involved are more demanding in comparison to the discrete way of identifying collisions, a performance comparison would be required to determine whether or not the more reliable collision detection is worth the additional expense in computational time.

Appendix A

CD-ROM/DVD Contents

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Files

Path: /

_thesis_pointner.pdf .	Master Thesis

A.2 Project-Files

Path: /project_files

_project_source.zip . .	Project Source Code and Executable
_benchmark.zip	Benchmark Used for Performance Data Gathering
_evaluation.xlsx	Performance Evaluation Excel Sheet

A.3 Miscellaneous

Path: /

_screenshots.zip	Images Showing Particle Collision Detection Results
_online_sources.zip . .	Cited Online Sources

References

Literature

[1] John Amanatides, Andrew Woo, et al. “A Fast Voxel Traversal Algorithm for Ray Tracing”. In: *Eurographics ’87*. (Amsterdam, Netherlands). Eurographics Association, 1987, pp. 3–10. URL: https://diglib.eg.org/handle/10.2312/egtp19871000 (cit. on pp. 18, 26).

[2] David Baraff and Andrew Witkin. “Large Steps in Cloth Simulation”. In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. (Orlando, Florida, USA). SIGGRAPH ’98. New York, NY, USA: ACM, 1998, pp. 43–54 (cit. on p. 1).

[3] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. “Particle-based Viscoelastic Fluid Simulation”. In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. (Los Angeles, California, USA). SCA ’05. New York, NY, USA: ACM, 2005, pp. 219–228 (cit. on p. 2).

[4] Cyril Crassin and Simon Green. “Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer”. In: *OpenGL Insights*. Boca Raton, Florida, USA: CRC Press, 2012. Chap. 22, pp. 303–318 (cit. on pp. 12, 13, 58).

[5] Cyril Crassin et al. “Interactive Indirect Illumination Using Voxel Cone Tracing”. *Computer Graphics Forum* 30.7 (2011), pp. 1921–1930 (cit. on p. 31).

[6] Elmar Eisemann and Xavier Décoret. “Fast Scene Voxelization and Applications”. In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. (Redwood City, California, USA). I3D ’06. New York, NY, USA: ACM, 2006, pp. 71–78 (cit. on p. 10).

[7] Elmar Eisemann and Xavier Décoret. “Single-Pass GPU Solid Voxelization for Real-Time Applications”. In: *Proceedings of Graphics Interface 2008*. (Windsor, Ontario, Canada). GI ’08. Toronto, Ontario, Canada: Canadian Information Processing Society, 2008, pp. 73–80 (cit. on pp. 10, 11).

[8] Arnulph Fuhrmann, Gerrit Sobotka, and Clemens Groß. “Distance Fields for Rapid Collision Detection in Physically Based Modeling”. In: *Proceedings of GraphiCon 2003*. (Moscow, Russia). GraphiCon Scientific Society. 2003, pp. 58–65. URL: http://graphicon.ru/html/2003/Proceedings/Technical/paper495.pdf (cit. on p. 16).

[9] Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. "Conservative Rasterization". In: *GPU Gems 2*. Amsterdam, Netherlands: Addison-Wesley, 2005. Chap. 42, pp. 677–690 (cit. on p. 14).

[10] Jian Huang et al. "An Accurate Method for Voxelizing Polygon Meshes". In: *Proceedings of the 1998 IEEE Symposium on Volume Visualization*. (Research Triangle Park, North Carolina, USA). VVS '98. New York, NY, USA: ACM, 1998, pp. 119–126 (cit. on p. 9).

[11] Andreas Kolb, Lutz Latta, and Christof Rezk-Salama. "Hardware-based Simulation and Collision Detection for Large Particle Systems". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. (Grenoble, France). HWWS '04. New York, NY, USA: ACM, 2004, pp. 123–131 (cit. on pp. 7, 8).

[12] Matthias Müller, David Charypar, and Markus Gross. "Particle-Based Fluid Simulation for Interactive Applications". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. (San Diego, California, USA). SCA '03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 154–159 (cit. on p. 2).

[13] Matthias Müller et al. "Meshless Deformations Based on Shape Matching". In: *ACM SIGGRAPH 2005 Papers*. (Los Angeles, California, USA). SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 471–478 (cit. on p. 2).

[14] Xavier Provot. "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior". In: *Proceedings of Graphics Interface '95*. (Québec, Québec, Canada). GI 1995. Toronto, Ontario, Canada: Canadian Human-Computer Communications Society, 1995, pp. 147–154 (cit. on p. 1).

[15] William T Reeves. "Particle Systems-a Technique for Modeling a Class of Fuzzy Objects". *ACM Transactions on Graphics (TOG)* 2.2 (1983), pp. 91–108 (cit. on p. 6).

[16] Matthias Teschner et al. "Collision Detection for Deformable Objects". *Computer Graphics Forum* 24.1 (2005), pp. 61–81 (cit. on pp. 16, 17).

[17] Hongyi Xu and Jernej Barbic. "Continuous Collision Detection Between Points and Signed Distance Fields". In: *11th Workshop on Virtual Reality Interactions and Physical Simulations*. (Bremen, Germany). Eurographics Association. 2014, pp. 1–7. URL: https://doi.org/10.2312/vriphys.20141218 (cit. on p. 17).

Online sources

[18] Rouslan Dimitrov. *Cascaded Shadow Maps*. 2007. URL: http://www.cse.chalmers.se/edu/year/2011/course/TDA361/Advanced%5C%20Computer%5C%20Graphics/cascaded_shadow_maps.pdf (visited on 03/17/2018) (cit. on p. 35).

[19] Etay Meiri. *Particle System using Transform Feedback*. 2011. URL: http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html (visited on 10/06/2017) (cit. on p. 8).

[20] Gareth Thomas. *Compute-Based GPU Particle Systems*. 2014. URL: https://www.gdcvault.com/play/1020002/Advanced-Visual-Effects-with-DirectX (visited on 10/06/2017) (cit. on p. 9).

[21] Chris Thou. *HALO: REACH Effects Tech*. 2011. URL: http://www.gdcvault.com/play/1014347/HALO-REACH-Effects (visited on 10/14/2017) (cit. on p. 15).