ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΣΤΗΝ ΕΠΙΣΤΗΜΗ ΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία
Μεταπτυχιακού Διπλώματος Ειδίκευσης

«*Ray Tracing Acceleration using Displacement Fields on GPU*»

Ανθούσης Ανδρεάδης

Επιβλέπων: Γεώργιος Παπαϊωάννου

ΑΘΗΝΑ, ΙΟΥΝΙΟΣ 2008

Abstract

Ray Tracing is a rendering method for the presentation of three dimensional images on two dimensional displays. One of the greatest challenges of ray tracing is efficient execution, since many times the method is dismissed as being too computationally expensive. In the current work we present the application of a novel method to accelerate the most intensive part of the computation of secondary-rays (reflection, refraction and shadow rays) the ray-primitive intersection tests. The main idea behind this acceleration technique is the use of four-dimensional fields (displacement fields) for each object that describe the distance from an augmented bounding sphere to the actual object surface, in every direction, for all positional samples on the sphere. Ray tracing is accelerated by replacing the often numerous distance-sorted ray-polygon hit queries, by a simple and of constant time displacement field indexing mechanism. Furthermore the GPU implementation of the method allows the exploitation of the several parallel pipelines offered by the modern hardware graphics cards, as ray-tracing can be greatly improved form parallel computation.

CONTENTS

1. Introduction

Finding a way to create photorealistic images has been a goal of computer graphics for many years. Several techniques have been developed in the last decades aiming at the representation of the physical principles that govern the light interaction in the real world. Ray Tracing is one of these techniques producing the desired output by tracing the path of light through pixels in an image plane.

The main idea behind ray tracing is that rays, cast from one or several light sources, interact with the physical objects and some of them reach the eye of the observer. Considering this, if we follow the ray along its path from the light source to the viewer and consider all its interactions with the physical objects, we would be able to give the corresponding light value to each pixel of our screen.

The described approach leads to an unbound problem that cannot be plausible solved if one is to implement it since a light source could casts infinite number of rays. Since one needs to take into consideration only the rays that reach the camera through a viewport pixel, one could approach the problem in the opposite way known as Backwards Ray Tracing or simply (Recursive) Ray Tracing. In Recursive Ray Tracing a ray is cast from the observation point, passes through the projection plane and is traced back to its source. The notion of tracing back the rays to their source instead of following the infinite rays from the source to our scene is what made ray tracing computationally feasible and applicable in many simulation applications.

The main advantage of ray tracing over direct rendering is that reflection and refraction phenomena can be accurately modelled. Moreover, shadowing is part of the ray tracing algorithm and no extra work has to be done in that direction.

Each ray in ray tracing can interact with an intersected object in three ways, according to the properties of that object. In case of opaque objects, the propagation of the ray is stopped. In case of transparent objects, a new refracted ray is created for the intersection point. Finally in the case of reflective objects, a reflection ray is spawned. At each intersection event, in order to calculate the local illumination of the interacting surface a visibility test is performed between the intersection point and each light source. The visibility test is nothing more than a ray that is cast with direction to each light source. This ray interacts as any other ray with the environment. If the visibility ray reaches to the light source, then the origin point of the ray is lit while in the opposite case it is considered in shadow. All the rays are considered of infinitesimal width while refraction and reflection rays have no dissemination. These assumptions of course degrade the realism of the final output but are considered necessary if we want the results in feasible time.

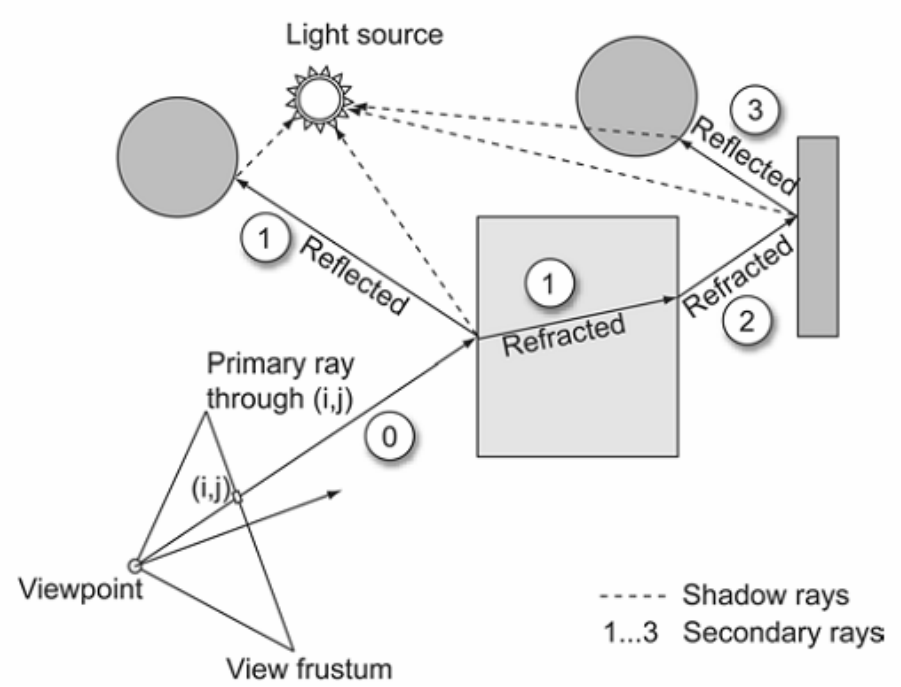


Figure 1: Ray Tracing Rendering Algorithm

In the current work we present the application of a novel method to accelerate the most intensive part of the computation of secondary-rays (reflection, refraction and shadow rays) the ray-primitive intersection tests. The main idea behind this acceleration technique is the use of four-dimensional fields (displacement fields) for each object that describe the distance from an augmented bounding sphere to the actual object surface, in every direction, for all positional samples on the sphere. Ray tracing is accelerated by replacing the often numerous distance-sorted ray-polygon hit queries, by a simple and of constant time displacement field indexing mechanism. Furthermore the GPU implementation of the method allows the exploitation of the several parallel pipelines offered by the modern hardware graphics cards, as ray-tracing can be greatly improved form parallel computation.

2. Related Work

One of the greatest challenges in ray tracing is efficient execution since the method is often dismissed as being too computationally expensive to be useful. Efficiency is therefore a critical issue and has been the focus of much research. This has led to creative approaches involving novel data structures, numerical methods, computational geometry, optics, statistical methods, and distributed computing among many others.

The generality of ray tracing is due to its almost exclusive dependence upon a single operation; calculating the point of intersection between a ray in three dimensional space and a geometrical primitive object. The term primitive object includes elementary shapes such as polygons, spheres, and cylinders, as well as more complex shapes such as parametric surfaces.

For each ray, this ultimately reduces to computing the point of intersection closest to the ray origin which results from any of the individual primitive objects in the environment. This ray-object intersection is the most computationally expensive part of the ray tracing algorithm and it typically overshadows every other operation included in the algorithm. Despite dramatic algorithmic improvements, the demand for ever increasing complexity, keeps the search for even more efficient techniques a lively topic of research.

Three very distinct strategies exist for the acceleration of intersection testing:

- Reducing the average cost of intersecting a ray with the environment.
- Reducing the total number of rays intersected with the environment.
- Replacing individual rays with a more general entity.

These distinct categories appear in Figure 2 as 'faster intersections,' 'fewer rays,' and 'generalized rays,' respectively.

The category of *"faster intersections"* further separates into the subcategories of 'faster' and 'fewer' ray-object intersections. The first sub-category consists of efficient algorithms for intersecting rays with specific primitive objects, while the second one addresses the larger problem of intersecting a ray with an environment using a minimum of ray-object intersection tests. The distinction between these two subcategories is blurred somewhat by algorithms which decompose what is normally thought to be a single primitive object into many simpler pieces for the sake of efficiency. An example of such a method is the intersection testing of B-spline surfaces proposed by *Sweeney et al [1]* where a single surface is subdivided into easily handled fragments and constructs a bounding volume hierarchy. Other primitive object intersection algorithms are extremely special purpose, since mostly they use analytic solutions for the point of intersection with a ray.

The category labeled *"fewer rays"* consists of techniques which allow us to reduce the number of rays which need to be intersected with the environment. This includes both first-generation rays and second-generation rays (rays created by reflection, refraction, and shadowing). The first such technique was adaptive tree-depth control by *Hall et al [2].* These techniques instead of terminating the ray tree at a predefined depth or at non-reflective opaque surfaces, take into consideration the maximum contribution to the pixel color which could result by continuing the recursion. By setting a threshold on this contribution, elimination

of many deep rays which would not alter the final result perceptibly was feasible. In that way Halls terminating criterion led to considerable savings even for environments with many highly reflective surfaces. Other techniques for reducing the number of rays are applicable when anti-aliasing through super-sampling. By detecting situations in which a relatively small number of samples produce statistically reliable results over some region of the image, many first-generation rays (hence entire ray trees) can be eliminated. Though often thought of as part of the anti-aliasing algorithms, these statistical techniques are mostly performance optimizations.

The last category, labeled *"generalized rays"*, consists of a number of techniques which begin by replacing the concept of a ray with that of a more general entity which subsumes rays as a special (degenerate) case. Such entities are cones of both circular *[3]* and polygonal *[4]* cross sections which have been used successfully. The basic idea behind these techniques is tracing many rays simultaneously which implies many interesting advantages, but limitations as well.

All the following methods and work focus on the 'intersection problem' and assume that a negligible amount of time is spent in all remaining tasks, such as shading calculations and common bookkeeping operations. The intersection problem has a trivial but usually impractical solution which is commonly referred to as 'standard' (or 'traditional') ray tracing. This solution entails intersecting each ray with the environment by testing each and every primitive object and retaining the nearest point of intersection (if one exists). As it is expected, this has linear time complexity in the number of objects. Nevertheless, exhaustive ray tracing is by far the most intuitive solution, and it continues to play a role in the processing of sub-problems within more complicated techniques.

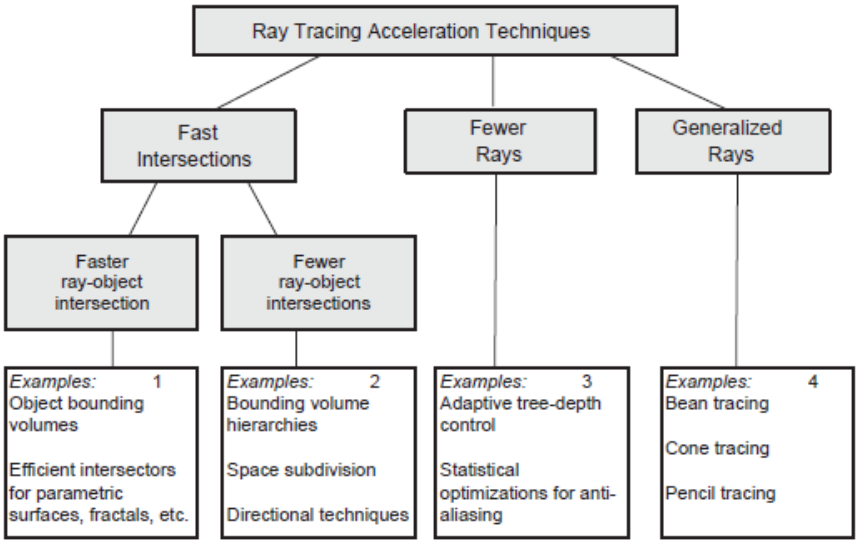


Figure 2: Ray Tracing Acceleration Methods

2.1 Bounding Volumes and Hierarchies

The most fundamental and ubiquitous tool for ray tracing acceleration is the bounding volume. This volume containing the object permits a simpler ray intersection test than the object. Only if a ray intersects the bounding volume

does the object itself need to be checked for intersection. Though this actually increases the computation for rays which would pierce its bounding volume, in a typical environment most rays closely approach only a small fraction of the objects. This results in a significant net gain in efficiency.

Although bounding volumes substitute costly intersection tests with much simpler ones, they do not decrease their number. From a theoretical standpoint this may reduce the computation by a constant factor, but cannot improve upon the linear time complexity of exhaustive ray tracing. *Rubbin and Whitted [5]* in order to alleviate this problem introduced the idea of a hierarchy of bounding volumes. This hierarchy can be used in order to attain a theoretical time complexity which is logarithmic in the number of objects, instead of the linear. By enclosing a number of bounding volumes within a larger bounding volume it was possible to eliminate many objects from further consideration with a single intersection check (hierarchy of bounding volumes). For example, if a ray does not intersect a parent volume, there is no need to test the contained in it volumes for intersection.

To further improve the efficiency of bounding volumes, *Weghorst et al [6]* investigated the trade-offs between two competing factors: tightness of fit and cost of intersection. For example by selecting a sphere, box, or cylinder as bounding volume depending of course on the characteristics of each object (or cluster of objects) to be enclosed, one can increase the efficiency of individual and hierarchically organized bounding volumes. The criterion for this selection begins with the observation that the total computational cost associated with an object and its bounding volume is given by the formula:

$$Cost = n * B + m * I$$

n stands for the number of rays tested against the bounding volume, B for the cost of each test, m for the number of rays which actually hit the bounding volume, and finally I for the cost of intersecting the object within. If we assume that both n and I are fixed, we would like to select a bounding volume which is both inexpensive, making B small, and as tight fitting as possible, minimizing m.

Compromisation is usually needed at the selection of the bounding volume. If one is to make the right trade-off choice, estimation of both cost and fit is needed. *Weghorst et al* proposed the enclosed volume as a measure of fit. This comes from the observation that enclosed volume is related to the *"projected void area"* with respect to any direction; that is, to the difference in the projected areas of the bounding volume and the enclosed object. This difference in area indicates how likely a ray is to hit the bounding volume without hitting the enclosed object. A large void area, resulting from a loose fit, can increase m and cause many unnecessary object intersection checks. Reducing m even at the expense of an increase in B is sometimes justified. Common bounding volumes for ray tracing are *"Bounding Spheres"*, *"Axis-Aligned Bounding Boxes - AABB"* and *"Oriented Bounding Boxes – OBB"*. An AABB is the intersection of half-spaces defined by six planes perpendicular to the WCS axe, while an OBB is a bounding box that is aligned with a coordinate system local to an object. While the AABB requires no transformation of the casted ray for the intersection test, the OBB produces a better fit almost in all cases but carries the extra cost of a ray transformation for every ray-bounding volume intersection test (see Figure 3). These two cases are effectively different types of bounding volume because the offer different cost/fit trade-offs. However when one deals with complex objects,

the ray transformation in the bounding volume intersection test is paid back many times from the great reduction of rays object intersection tests. These types of transformations can also be applied to other type of bounding volumes. For example one could use oriented cylinders or replace bounding spheres by bounding ellipsoids.

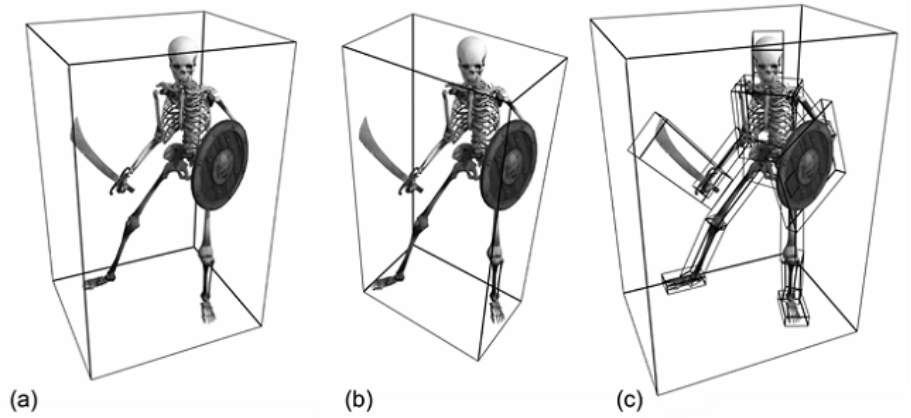


Figure 3: Bounding Volumes for Ray Tracing.
a) AABB, b) OBB, c) Bounding Volume Hierarchies

Further improvements to the direction of better fitting bounding volumes, is the use of multiple bounding volumes for an object. For example one could use the *intersection* or the *union* of two different bounding volumes. In the case of intersecting bounding volumes, a ray must intersect all the volumes before testing the enclosing object. The final cost would be the sum of the individual volume intersection costs but only in the case of a ray which penetrates all of them. In a case of miss, only one of the volumes is being tested. Now in the case of union of two or more bounding volumes, each ray must be tested with all the volumes making the cost of a miss more expensive.

Finally another approach is the use of bounding slabs. The origin of this approach lies in the attempt to use convexity which as a geometrical property offers great advantages. The convex hull of an object is a uniquely defined convex volume which contains the object and is by definition the smallest such volume. While the facts suggest that convex hulls would be excellent bounding volumes, the computation and the representation of an exact convex hull of an object or a collection of objects is not trivial. Therefore an approximation of a true convex hull which would eliminate these drawbacks and moreover offer efficient intersection tests, would offer many advantages. The best example of this method was introduced by *Kay and Kajiya [7]* and is known as called bounding slabs. The algorithm uses the concept of *plane-sets* which are families of parallel planes. Each of these plane-sets is defined by a single unit vector called *plane-set normal* and each plane within a family is uniquely determined by its signed distance from the origin. Given a plane-set normal and an arbitrary object, there are two unique planes of the family which most closely bracket the object. The infinite region between these planes is called a *slab*, and is conveniently represented by a min-max interval associated with the plane-set normal (see Figure 4). For polyhedral object, these values can be computed by forming the dot product of the plane-set normal with each of the objects' vertices (in world coordinates), and then finding the minimum and maximum of these values. For implicit surfaces such as quadrics the values defining the slab can be computed using Lagrange multipliers. The intersection of several different slabs, can define a bounded

region enclosing the object (see Figure 5). In three dimensional spaces, this requires three slabs which plane-set normals are linearly independent, but this is not a limitation as the greater the number of the slabs, the more accurate the approximation of the convex hull of the object will be. In order to perform ray intersections with bounding slams volumes first compute the interval along the ray, measured from its origin, which lies within each side of the slabs. This amounts to computing two ray-plane intersections for each slab. If the intersection of these intervals is empty, then the ray misses the volume. Otherwise, the ray hits the volume and the maximum of the minimum interval values is the distance to the point of intersection. The shape of this type of bounding volume is unaffected by object translation (min-max plane constants remain unchanged). On the other hand, object rotation affects the quality of approximation as the tightness of the volume changes. This implies that the number and the orientation of planes must be set individually for each object in order to achieve tight-fitting volumes. However there are many computational advantages to using the same collection of plane-set normals for all the objects of the environment, despite their individual orientations. The most significant advantage is that the procedure of ray intersection can be greatly accelerated.

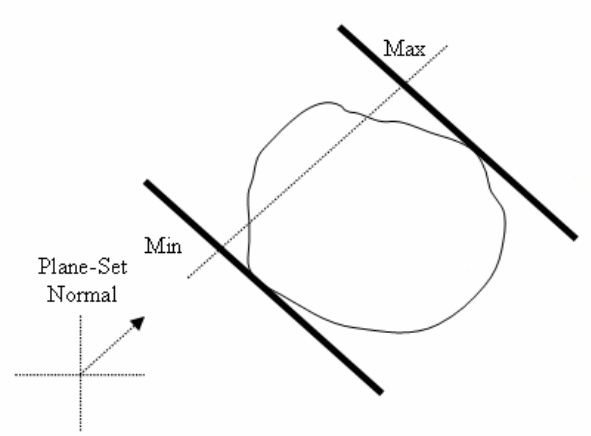


Figure 4: A single slab bracketing the object

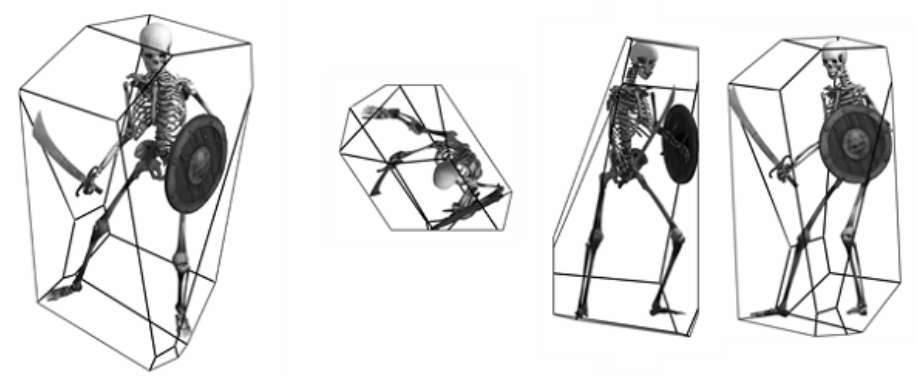


Figure 5: Bounding Slabs

2.2 3D Spatial Subdivision

The further an object is from the path of a ray, the less work we can afford to do in eliminating it from consideration. As we have seen, bounding volume hierarchies provide a means of recursively narrowing the focus of the search to more promising candidates for intersection. This is a natural divide-and-conquer

approach for the examination of a collection of objects, seeking the member producing the closest intersection. Spatial subdivision begins with a different philosophy. Although this approach also relies upon simple volumes to identify objects which are good candidates for intersection, these volumes are constructed by applying a divide-and-conquer technique to the space surrounding the objects, instead of considering the objects themselves. Rather than constructing the volumes in a bottom-up approach by successively enveloping larger collections of objects, this technique proceeds in a top-down approach, partitioning a volume bounding the environment into smaller pieces. The smaller volumes thus formed are then assigned collections of objects which are totally or partially contained within them. Therefore a fundamental difference between bounding volume hierarchies and spatial subdivision techniques is that the former selects volumes based on given sets of objects, whereas the latter selects sets of objects based on given volumes. This leads to a very different approach which places the emphasis on space instead of the objects.

A concept common to all the current techniques of this family is the voxel. A voxel is a *"cuboid"* or axis-aligned rectangular prism, and it is the fundamental element created as a result from the partitioning of space. A pre-processing step is responsible for constructing nonoverliapping voxels which, taken together, constitute a volume containing the environment. Within these constraints there are different methods of defining the voxels, and these differences lead to significant variations. The differences between uniform versus nonuniform spatial subdivision size are particularly important. Once defined, however, the voxels play the same role in all cases. They are the means of restricting attention to only those objects which are close to the path of a ray.

If the point of intersection between a ray and an object lies within a voxel, both the ray and the object clearly must intersect that voxel. Since voxels contain the entire environment, every possible point of intersection must lie within some voxel. Therefore, the only objects which we must be tested for intersection are those which intersect the voxels pierced by the ray. For any given ray, this can potentially eliminate the vast majority of the objects in the environment from consideration. An observation of equal importance is that a ray imposes a strict ordering on the pierced voxels based on the distance to the point at which the ray first enters each voxel. As we mentioned voxels are nonoverlapping and based on that this ordering guarantee that all intersections occurring within one voxel are closer to the ray origin than those in all subsequent voxels. Consequently, if the voxels are processed in the order in which they are encountered along the ray path, we don't need to take into consideration the contents of any further voxels once we have found a point of intersection. This feature drastically reduces the number of objects which need to be tested and it's one of the most attractive features of these techniques. In other words, spatial subdivision techniques offer an efficient means of identifying the objects which are near the path of a ray while at the same time performing a virtual 'bucket sort' on those objects.

2.2.1 Nonuniform Spatial Subdivision

Nonuniform spatial subdivisions are the techniques that discretize space into regions of varying size based on the features of the scene. This variation in size allows detailed subdivision (small voxels) to be performed in densely populated regions of the scene and, simple subdivision (large voxels) to cover

regions which are sparsely populated or entirely void areas. Proposed data structures for storing the non-uniform data are octrees, BSP and kd-trees.

Octrees are hierarchical data structures used for efficiently indexing the data associated with points in the 3-D space. Octrees are constructed by recursively subdividing rectangular volumes into eight subordinate Octans until the resulting voxels meet some criterion of simplicity. The partitioning stops either when a maximum number of subdivisions is reached or when the number of primitives a cell contains is small enough to make further refinement unnecessary. The maximum number of subdivisions performed defines the depth of the tree. Octrees data structure intersection tests are unbalanced, but intersection test distribution at the leaf nodes is more even. *Glassner [8]* introduced an octrees variation for use in ray tracing. In his approach each voxel is assigned a list of objects which intersect it. The candidate list for each voxel is created by testing each object with the six planes of the volume. One intersection of course is sufficient for the object to be added to the voxels list. In case all planes fail the intersection test, another test for complete containment of the object by the voxel is performed. The creation of these candidate lists leads to a top down construction approach of the octree. A box containing the environment is recursively subdivided until each voxel reaches the simplification criterion we mentioned before.

A very similar approach based on binary space partitioning trees (BSP) was introduced by *Kaplan [9]*. BSP trees partition space into two pieces at each level based on a separating place. While in general case of BSP trees arbitrary splitting planes can be used, the most usual case is to use only planes that are perpendicular to one of the coordinate system axes. This special case of BSP trees is known as kd-trees and it's the one mostly used in ray tracing algorithms. Although the construction of kd-trees can be performed in many ways, since there are many ways to choose axis-aligned splitting planes, the final voxel subdivision is almost the same with the octrees.

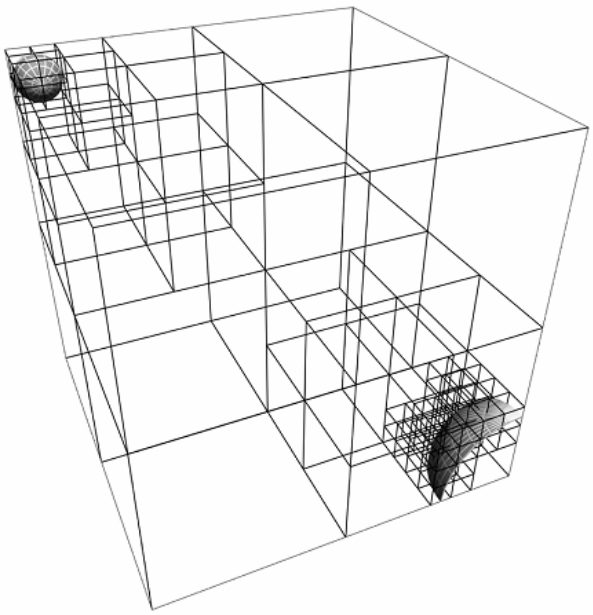


Figure 6: *Non – Uniform Spatial Subdivision of a Scene*

2.2.2 Uniform Spatial Subdivision

A different approach to spatial subdivision is the uniform which was introduced by *Fujimoto et al [10]*. In this approach voxels of same size are organized in a regular three dimensional grids (or lattice). This organization was given the acronym SEADS, for Spatially Enumerated Auxiliary Data Structure. The overall strategy is quite similar to the nonuniform subdivision techniques. Lists of candidate objects are retrieved from voxels which are pierced by a ray and these voxels are processed in the order they are pierced. However, there are two distinguishing features of this approach which are direct consequences of the voxel regularity:

1. the subdivision is totally independent of the structure of the environment, and
2. the voxels pierced by a ray can be accessed very efficiently by incremental calculation

The first is a disadvantage which must be weighed against the obvious benefits of the second. In certain test cases the speed of voxel access proves to be the dominant factor, indicating that the SEADS approach can sometimes offer significant gains in performance over nonuniform subdivision techniques.

The key to efficient voxel access is that finding the voxels along the path of a ray in a regular lattice is the 3-D analogy of representing a line on a regular array of pixels. In that way the selection of voxels for each ray is done with an incremental algorithm similar to the 2D DDA[1], only for voxel space instead of image space. One minor difference is that the 3-D DDA must step through each voxel which is pierced by the given ray whereas line rasterization algorithms identify pixels which are merely close to a line in some sense. This requires a departure from the common property of line rasterization algorithms which forces a step to be taken along the dominant axis unconditionally at all iterations.

The advantages of this approach cannot always compensate for the lack of adaptivity, however. Though the voxels which 'digitize' a ray can be made to approximate the ray with arbitrary precision by increasing the resolution of the grid, two limitations begin to emerge as we do so. First, it becomes more costly to pass rays through empty regions of space, and second, the storage for the corresponding three-dimensional array quickly becomes unmanageable. The storage problem can be alleviated of course by only storing the voxels which have non-empty candidate lists. This could be accomplished through a voxel look-up scheme. This is another space-time trade-off because of the overhead which the hash table look-up adds to the voxel walking process. Because the 3-D DDA is applicable only to uniform subdivision, this restricts its use to walking 'horizontally' among sibling voxels of the octree. Each group of eight siblings can be viewed as a small uniform grid and. as such; the 3-D DDA provides an efficient means of passing a ray through them. After, stepping through at most four voxels, 'vertical' traversal must be performed again in order to locate the next block of eight siblings.

1 *Digital Differential Analyser (DDA) is an algorithm used to determine which points need to be plotted in order to draw a straight line between two given points. It employs the equation for line representation (example: y=mx+c), then scan through an axis. Each scan it would determine the value on the other axis using the equation, this way the proper pixel can be located.*

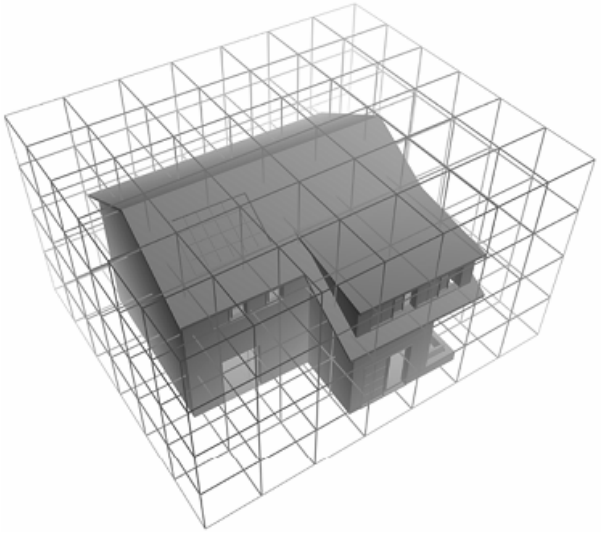


Figure 7: *Uniform Subdivision of a scene*

2.3 Directional Techniques

The most recent category in ray tracing acceleration is that of directional techniques. Though every ray tracing approach must take ray direction into account, the directional techniques are those which exploit this information at a level above that of individual rays. In order to understand how this technique differs from other approaches, we have to consider the use of ray direction within a typical 3-D spatial subdivision scheme. In this case the direction is used in selecting the subset of voxels pierced by the ray. While this eliminates most of the voxels from consideration and defines an efficient order for processing those which remain, still the ordering of voxels must be performed in a ray basis because direction is not taken into consideration during the construction process. In contrast, directional techniques explicitly incorporate directional information into data structures which allow more of the overhead to be moved from the 'inner loop' into a less costly stage. Operations such as back face culling and candidate sorting can be done on behalf of many rays instead of individual rays. A common penalty which accompanies these advantages is the very large storage requirement.

There are currently three different approaches in the directional techniques: the *"Light Buffer" [11]*, the *"Ray Coherence" [12]* algorithm and *"Ray Classification" [13]*. Before analysing the methods, we will present the *"Direction Cube"*, a concept that appears in all approaches.

2.3.1 Direction Cube

The Direction Cube plays a similar role to that of the *"Hemi-Cube"* used in the radiosity method *[14]*. Its main role is the discretization of rays into a finite number of square direction cells. By that we could say that it is analogous to the spatial subdivision where instead of rays, we have discretization of space. The direction cube is an axis-aligned cube centered at the world coordinate origin and the six faces of the cube correspond to the six dominant axis: *+X*, *-X*, *+Y*, *-Y*, *+Z* and *-Z*. Each of these 6 faces corresponds to a solid angle of $2\pi/3$ steradians[2]. In that way, the direction cube allows us to translate 3-D directions to 2-D

rectangular coordinates.

All faces account for all possible directions (4π steradians). For any given ray, we can construct an alternative representation for its direction by imagining it translated to the coordinate origin, determining thus which face of the cube it intersects. This can be done if we first find the dominant axis of the ray and then compute the u-v coordinates of the point of intersection with the corresponding face.

This translation to 2-D rectangular coordinates allows us to easily and efficiently apply subdivision techniques in the context of directions. Both uniform and nonuniform subdivisions can be used. Each direction cell formed from subdivision defines an infinite skewed pyramid with the top of it at the coordinate origin and its edges through the cell corners. These are called directional pyramids and define the volume of space that is accessible to the rays that begin at the coordinate origin and pass through the given direction cell. In each of the following three methods, [2]each one of them begins by associating direction cubes with specific collections of rays (see *Figure 8*).

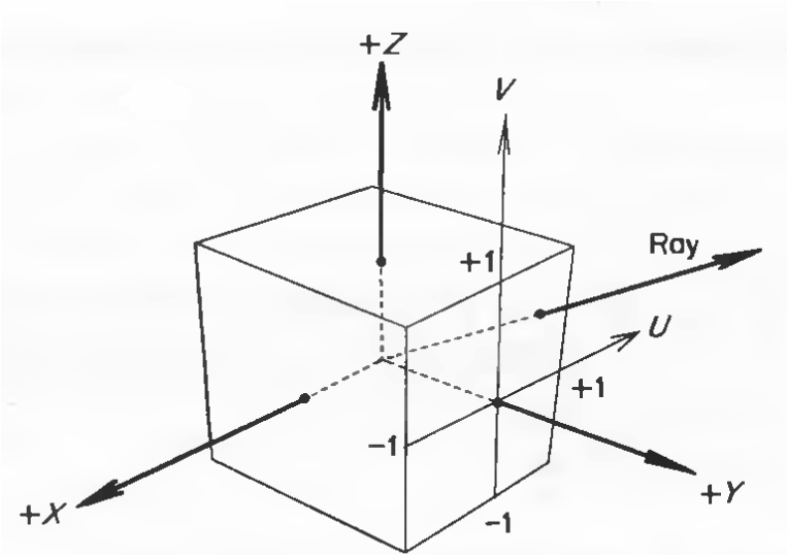


Figure 8: The Direction Cube

2.3.2 Light Buffer

The Light Buffer is a directional technique which accelerates the calculation of shadows with respect to point light sources. The main fact exploited by this algorithm, which was introduced by *Haines and Greenberg [11]*, is that points can be determined to be in shadow without finding the closest occluding object. Since any opaque occluding object will suffice, shadowing operations are inherently easier that normal ray-environment intersections. Furthermore, constraining light sources to be single points allows a particularly effective application of the direction cube to these operations.

The search for an occluding object can be narrowed to a small set of objects by making use of the direction from the light source to the point in question. The light buffer algorithm accomplishes this by associating a uniformly subdivided direction cube with each light source, and a complete list of candidate objects with each of the direction cells. In that way, each candidate list contains every

[2] **Steradian** is the SI unit of solid angle and describes 2-D angular spans in 3-D space.

object which cannot be seen through the corresponding direction cell. These candidate lists are retrieved by finding the direction cell pierced by each light ray which is (conceptually) cast from the light source. The objects in this list are the only ones which can block the ray and thereby cast a shadow.

The light buffers are constructed in a pre-processing step. The candidate lists are created by projecting each object of the environment onto the six faces of each direction cube, adding them to the candidate lists of those direction cells which are partially or totally covered by the projection. Once all the lists are created, they are sorted into ascending order based on depth.

In order to determine if a point on a given surface is in shadow, at first the orientation of the surface must be checked with respect to the light source. If it is facing away, then the object is in shadow. Otherwise the list of potentially occluders is retrieves from the light buffer using the direction of the light ray. The objects are tested for intersection in order of increasing depth. The procedure continues until an occlusion is found or until we reach an object whose depth is beyond the point we are testing. In the first case the result is shadow while in the second the object is illuminated.

2.3.3 Ray Coherence Algorithm

The Ray Coherence Algorithm introduced by *Ohta and Maekawa [12]*, is a mathematical tool for placing a bound on the directions of rays which originate at one object and then hit another, making it possible to broaden the application of direction cubes from single points to bounded objects. In its simplest form the ray coherence theorem applies to objects which are bounded by nonintersecting spheres.

Any ray which originates within a sphere *S1* and terminates within sphere *S2* defines an acute angle θ, with the line that passes through the spheres centers (see *Figure 9*). The following inequality is a bound on this angle in terms of the sphere radius and the distance between their centers:

$$\cos\theta > \sqrt{\left(1 - \frac{r1 + r2}{\|C_1 - C_2\|}\right)}$$

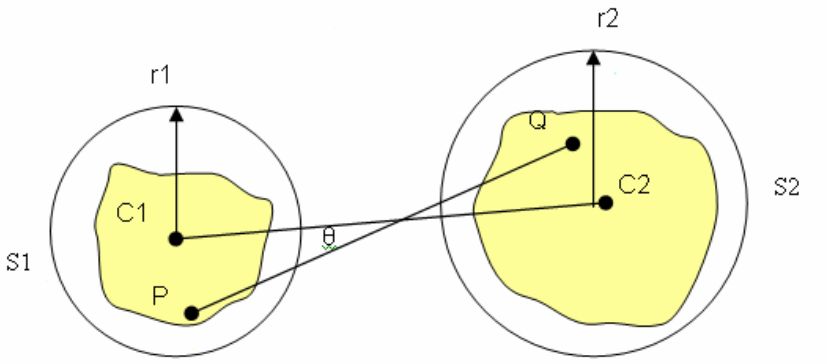


Figure 9: Bound on the angle between lines C1C2 and PQ

Approximations of these direction bounds are stored by means of uniformly subdivided direction cubes associated with each entity in the environment from which rays can originate. This includes the eye point, light sources, and reflective

or refractive objects. Each of these direction cubes is constructed and used in nearly the same manner as a light buffer. A pre-processing operation creates depth-sorted lists of intersection candidates for each direction cell of each direction cube. These candidate lists determine the objects which need to be tested for intersection with any ray based on its direction and the entity whence it originated. The direction cube therefore accelerates the process of finding the 'next' object hit, providing an efficient way of progressing from object to object as the path of a ray is traced. This essentially reduces to a light buffer in the case of shadow tests with respect to point light sources.

The difference in testing a candidate list for intersection between Light Buffer and Ray Coherence algorithm is that in the second the nonshadowing rays require that the closest point of intersection must be found. Objects in the list are tested in order until the list is exhausted or the minimum distance, is greater than the distance to a known point of intersection.

2.3.4 Ray Classification

The Ray Classification algorithm, which was described by *Arvo and Kirk [13]*, does not use explicit direction cubes except in the special case of first-generation rays. The data structure that is used in order to accelerate the intersection process for other rays is closely tied to the concept of the directional cube. The observation based on which the Ray Classification algorithm is build is that rays in 3-D space have five degrees of freedom and correspond to the points of $R^3 \times S^3$, where S^2 is the unit sphere in 3-D space. The algorithm proceeds by partitioning the five dimensional spaces of rays into small neighbourhoods, encoded as 5-D hypercube and associating a complete list of candidate objects with each one of them. A hypercube is the representation of a collection of rays with similar origins and similar directions, while its associated list contains all objects which are hit by any of these rays. In order to intersect a ray with the environment, we locate the hypercube which contains the 5-D equivalent of the ray and test only the objects of its associated list of candidates.

Based on the observation that a 5-D hypercube represents a collection of rays which originate from a 3-D voxel and possess directions given by a single direction cell, the candidate lists are constructed. This collection of rays' sweeps out an unbounded 3-D polyhedral volume called *beam* (see *Figure 10* & *Figure 11*). The candidate list of a hypercube must contain all objects that intersect the beam. As the nodes of the hyper-octree are subdivided, a child's candidate list can be obtained from the parent list by removing those objects which fall outside of its narrower beam.

As with the other directional techniques, the candidate lists are sorted by depth in order to most effectively apply the distance interval optimization. A difference spotted in the ray classification approach is that only the candidate lists associated with the original bounding hypercube need to be sorted as these lists contain all the objects in the environment. Since all subsequent lists are derived from these, the sorted order is passed down to them.

The constructed hyper-octrees and the associated candidate lists can potentially become very large and space-saving measures are required. One of the most critical measures is the restriction of subdivision only to the regions of 5-D space which is actually populated by rays of intersection.

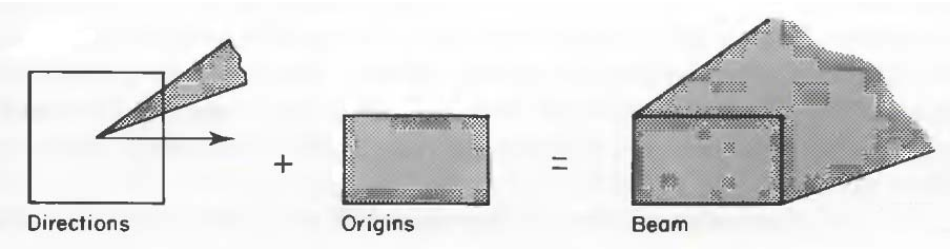


Figure 10: Beams in 2-D space

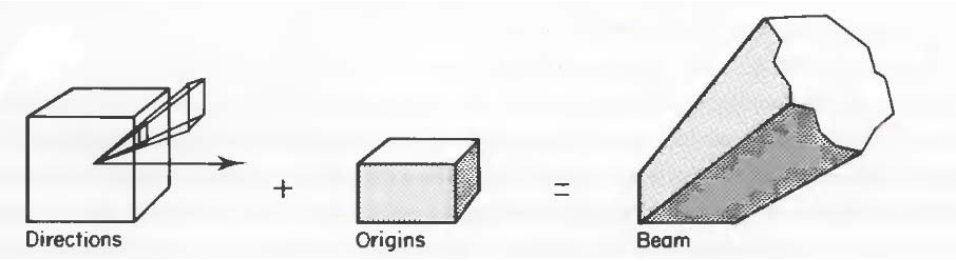


Figure 11: Beams in 3-D space

2.4 Parallelization of Ray Tracing

Ray tracing is a very computationally expensive algorithm but fortunately, beyond the already discussed acceleration methods, the algorithm is quite easy to parallelize. This is easy to comprehend since the contribution of each ray to the final image can be computed independently from the other rays. For this reason, there has been a lot of effort put into exploiting the best parallel decomposition for ray tracing *[15]*.

The simplest parallel schema for ray tracers is to replicate the scene database for every processing node. For such systems the biggest challenge is the load balancing. However if the entire scene is too large to fit on a single node or if the overhead that comes with the replication of the whole scene must be avoided, then an additional challenge comes up since scene geometry and rays have to be distributed between the nodes. In that way the geometry is often split across multiple processors but this come with the mandatory communication between them which can be very expensive. More than that, from an implementation point this approach is much more difficult.

Even though ray tracing algorithm is trivially parallelized, very few interactive ray tracing systems exist. Interactivity requires the ray tracing system spend very little time on communication and synchronization. Simply adding processors does not necessarily increase the performance of a system unless it is properly engineered. Two types of systems have recently yielded interactive systems: shared memory supercomputers and clusters of commodity PCs. In both cases, these systems use a collection of standard microprocessors, each of which is designed for maximum performance on single threaded programs.

An alternative to the super computer is a cluster of commodity PCs. These systems are often more cost effective for the same amount of peak compute performance. However, clusters are limited by having lower communication bandwidth and higher communication latency.

In the recent years CPU ray tracers have gained parallelism due to SSE, distributed processing, and multi-core CPUs. However, visual simulation applications increasingly rely on the CPU for physics, AI, animation, and other

tasks which diminish ray tracing performance.

2.5 Graphics Processors Approach

In recent years the performance of graphics hardware has increased far more rapidly then that of CPUs. While CPUs are normally optimized for sequential code, GPUs are optimized for highly-parallel vertex and fragment shading code. GPUs offer fast floating point operations and a complete orthogonal instruction set. These features make graphics hardware an excellent choice for many highly-parallelizable algorithms, including of course ray-tracing. However while the GPU outperforms the CPU on streaming kernels such as ray intersection, the CPU remains much more efficient at maintaining and traversing the complex data structures needed to trace rays efficiently.

In the modern programmable graphics pipeline (see *Figure 12*), the scene is passed to the hardware as a sequence of triangles defined by vertices, colors and normals. The Vertex Program Stage is generally used to transform the vertices from model space to screen space using matrix multiplication. The Rasterizer takes the transformed triangles and turns them into fragments. At this step the Rasterizer also interpolates the values of each vertex between other vertices in the triangle so that each fragment has a color and normal value. These fragments then pass through the Fragment Program Stage where the color of each fragment is modified using texture look-ups or mathematical functions simulating the light interactions. The resulting fragments are finally compared against the stored value in a depth buffer. The fragments that pass the test are saved and displayed while others are discarded.

The programmable Vertex and Fragment engines found on today's graphics chips (such as NVIDIA's and ATI's cards) execute user-defined programs and allow fine control over shading and texturing calculations. Each stage is programmable through OpenGL ARB extensions (OpenGL Shading Language), vendor specific extensions (NVIDIA's Cg), or the DirectX (Microsoft's HLSL).

The programming model for the programmable fragment engine is shown in Figure 11. Most GPUs have a set of several parallel execution units (at present up to 128) that implement the fragment engine. However, the exact number of parallel units is not exposed to the programmer. Furthermore, the SIMD scheduling puts the limitation that GPU fragment processors must run identical kernels lowering in that way the overall versatility of GPU.

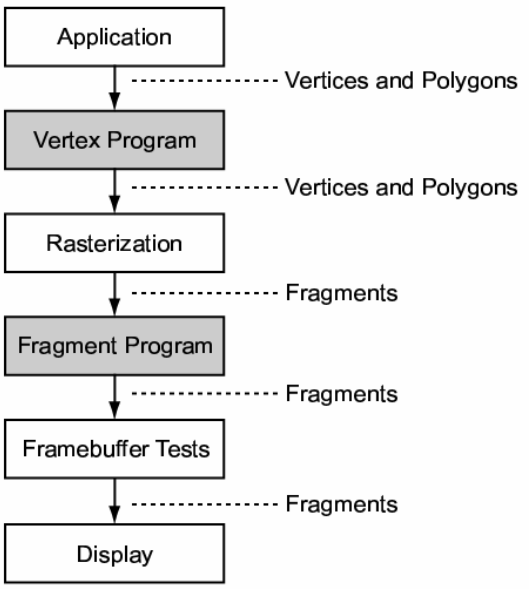


Figure 12: *The Programmable Graphics Pipeline.*
Gray boxes show the programmable stages of vertex and fragment engines. The types of data passed between each stage are represented with dotted lines.

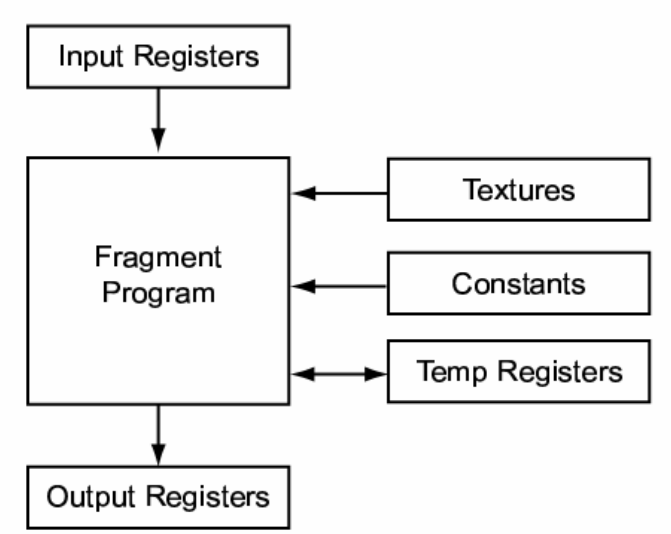


Figure 13: *Fragment Processors read from input registers, constants, texture memory and temporary registers. The output registers store the final color of the fragment.*

3. Method Overview

Initially we will present the general idea of displacement fields used in the Ray Tracing which was introduced in [16]. In the next chapters follows a detailed discussion about our Ray Tracing approach. In section 3.1 we describe the general idea of displacement fields, while in section 3.2 we show their application for ray tracing. We further discuss the sampling techniques used and the storage requirements of the method along with the compression scheme.

3.1 Displacement Fields

Displacement Fields are the collection of distance information stored in textures. Each texture contains distance information regarding an object from a certain sample point of the object's bounding sphere. Each of these textures is known as displacement texture. The method used *[16]* bears some similarity to the parameterization of Huang et al. *[18]* where each ray was described as a vector of the parametric incident location (u, v) on the bounding volume and its corresponding incoming direction (θ, φ).

3.1.1 Displacement Fields Computation

The main idea of encoding displacement fields into maps is as shown in *Algorithm 1*. Consider a rigid object possibly moving through a scene. At a pre-processing step, from a discreet set of sample points on the bounding sphere, given as spherical coordinates (u, v), a hemisphere of rays is cast around the inward normal direction (see *Figure 12*). For each ray (u, v, θ, φ), the closest distance between the bounding volume and the model surface is found and recorded as a compact integer value after being normalized by twice the sphere radius. Thus, for each sample point (u, v) a displacement grayscale map is obtained (*see Figure 13*) that represents the distance traveled along the ray in the direction (θ, φ) before hitting the model surface. As said in the definition above, the displacement field for a certain object is the collection of all the displacement maps generated from all sample points on its bounding sphere.

```
generate bounding sphere sample points
generate samples of hemisphere of rays
for all bounding sphere sample points (u, v) do
      align hemisphere of rays to normal at (u, v)
      for all rays (φ, θ) do
            if ray intersects the object then
                  normalize the distance (divide by 2 * R)
                  record distance in displacement map
            else
                  record distance in displacement map as 2 * R
      end
end
```

Algorithm 1: *Pseudo code of basic algorithm for displacement fields computation at pre-processing time.*

3.1.2 Displacement Fields Indexing

During the real time part of ray tracing, an incident ray to the object intersects its bounding sphere and the distance between the ray origin and the intersection point is stored. The intersection point q is transformed into the object's coordinate system: $q' = M^{-1} \times q$ where M is the transformation matrix with respect to the reference frame of the ray. Depending on the sampling on the surface of the sphere (in our method this was fixed as the Concentric Sampling method, presented later), the inverse function is applied to q' in order to get the closest corresponding point (u, v) on the sphere for which we have a displacement map and therefore the index of the corresponding displacement map. Corresponding direction (θ, φ) of the ray expressed in the local coordinate system of the intersection point has to be found. Depending on the ray sampling method, the appropriate inverse function is applied to the ray, recovering in that way the (θ, φ) values of the ray. We can now index into the displacement field for the given ray (u, v, θ, φ) and extract the distance information which is then added to the intersection distance above and this is the final approximated distance between the ray origin and the object's surface.

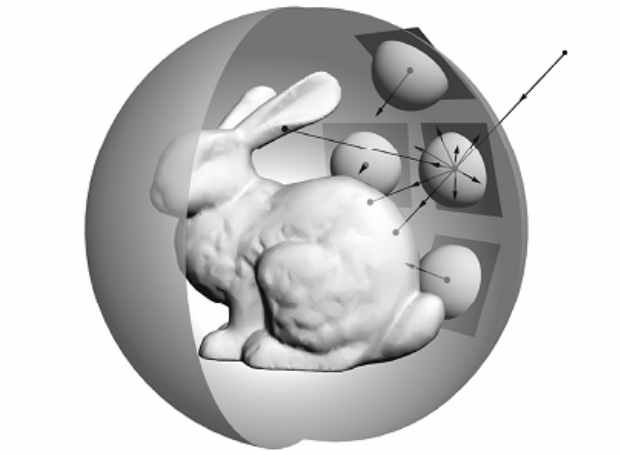


Figure 14: *A hemisphere of rays emanating from the bounding sphere towards the object is pre-computed for a large number of sample points on the sphere.*

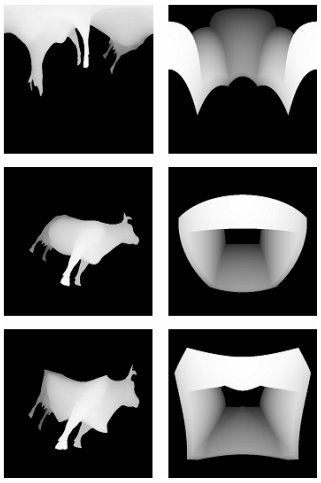


Figure 15: *512x512 displacement maps of a model of a cow and a cube with a hole in it.*
(Top row) Using uniform Sampling of rays
(Middle row) Rejection Sampling
(Bottom row) Concentric Map Sampling
Different (θ, φ) to (s, t) mappings, produce different displacement maps.

3.1.3 Selection of Samples around the Object

In order to generate displacement maps for an object, entry points on the surface of its bounding volume object have to be chosen. The method to select for this procedure must have a quick inverse function that can convert an intersection point into the nearest sampled point. Furthermore the distribution of the sampled points has to be as even as possible.

A common bounding shape that is used to sample the contained geometry is an axis-aligned bounding box (AABB). During the real-time simulation we would perform fast ray-box intersections. Special care though is needed as the AABBs are not transformation invariant and their oriented bounding boxes

(OBB) counterparts require more operations. As most sampling methods deal with sampling over a sphere, if the same methods were used to sample over a cube there would be a high concentration of samples near the vertices of the cube. That's the reason the bounding volume used in this approach is a bounding sphere.

The method used for the selection of samples around the object is *Slater's [17]* method, which generates uniformly, distributed points on a hemisphere using the triangle subdivision method (see *Figure 16*). The same method can be used of course to cover the full sphere as well. At the same time *Slater* suggests a constant time inverse function. In that way when a ray intersects the bounding sphere of the object, we can immediately associate this intersection point with one of the pre-generated displacement maps in order to retrieve the angle and distance information.

Another more straightforward approach would be to use the spherical coordinates which have a fairly easy to compute inverse function. However the samples with this method are concentrated more towards the poles of the sphere, reducing in that way their efficiency as a sampling mechanism.

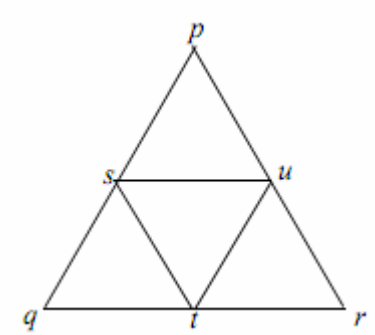


Figure 16: *Triangle Subdivision.*
Each triangle is subdivided to four new triangles

3.1.4 Samples on a Hemisphere of Directions

Several methods exist focusing on the uniform distribution of samples over a hemisphere. The method needed in our case has to uniquely discretize its samples so that they can be stored in the displacement maps. Furthermore an inverse function must exist so that the displacement map entries can be converted back into to the samples space.

One possible choice would be the spherical coordinates where a direction in the hemisphere is given by two angles (φ, θ). The generated rays are concentrated towards the cap of the hemisphere (see *Figure 15a*) producing a good cosine term (close to 1.0) but they are not equally spaced.

Another possible method is the rejection sampling method (see *Figure 15b*). Uniformly distributed points are selected inside a unit disk by selecting points inside the $[-1,1]^2$ square and rejecting the points that fall outside the unit disk. Using *Malley's* method *[18]* the samples are projected on the disk up to the hemisphere above it, producing a cosine distribution of rays. The drawback of this method is that about 21.5% of the samples are rejected and so the corresponding space in the displacement map remains unused.

Shirley et al. [Shi97a] suggest a concentric map (see *Figure 15c*) sampling method that maps samples in the square $[-1,1]^2$ to the unit disk $\{(x,y)|\, x2+y2\le 1\}$

by mapping concentric squares to concentric circles. The map preserves fractional area it is bi-continuous and has low distortion. Combined with Malley's method where samples on the unit hemisphere have density proportional to the cosine term, it gives the best results.

3.1.5 Storage and Error Considerations

A 256x256 map stores the distance to the object for 65536 ray directions emanating from one sample. If that map was to store the values as floats it would require 262144 bytes of storage space while storing them as unsigned chars it would require 65536 bytes. In addition, if lossless compression is used (e.g. run length encoding) then on average less storage would be required. In application areas where integral calculations are performed over the samples or accuracy is not imperative, lossy compression could be used to further reduce the storage requirements. In our case all maps store distance information as unsigned characters (8bits).

3.2. Ray Tracer on GPU using Displacement Fields

First of all we have to describe the storage of our data and the limitations posed by the vast use of texture space. Since we don't use geometry for our ray-tracer, but only displacement fields, the use of equally sized textures to the displacement fields holding the Normal information is inevitable. Displacement and Normal textures are loaded and passed to the GPU as 3D Textures (see *Figure 17* & *Figure 18*). The extensive use of texture space along with the limitations of CPU-GPU communications prevents us from using an acceleration data structure such as a BVH-tree. The reason behind this is that we can't load all the displacement fields and corresponding normal textures for all objects of a given scene together on the GPU. Furthermore current GPUs have limitations on the number of textures passed to them also making it even more difficult to pass the data of an entire scene all together.

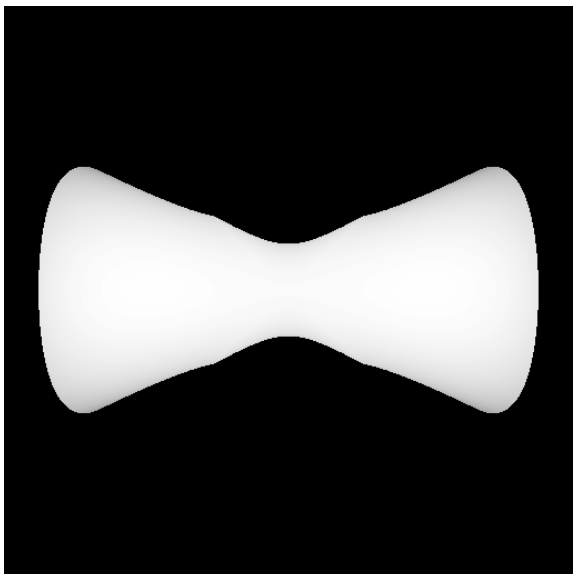


Figure 17: *Displacement Map*

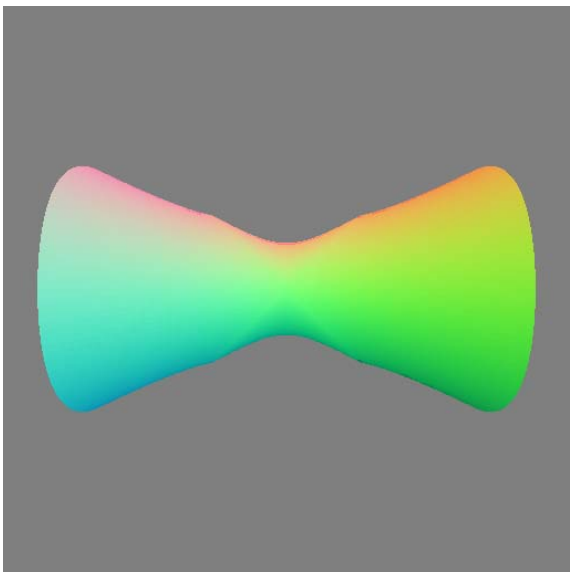


Figure 18: *The Corresponding Normal Map*

3.2.1 General Algorithm

The general algorithm used is described in *Algorithm 2* & *Algorithm 3*. Each operation is performed per object. Intermediate and final results are stored in textures through frame buffer objects. Each part of the algorithm will be described in detail in the following chapters.

```
calculate ray directions
for each object in the scene do
      Intersection Testing
end
for each object in the scene do
      Shadow Test
end
for each object in the scene do
      Light Equation
end
if  max_depth > 1 then
      Trace Ray
```

Algorithm 2: *Pseudo code of the First General Step of Ray Tracing*

```
current_depth ++
for each object in the scene do
      if  current object has reflection then
            Reflection Testing
end
for each object in the scene do
      if  current object has reflection then
            Shadow Test
end
for each object in the scene do
      if  current object has reflection then
            Light Equation
end
if   current_depth < max_depth then
      Trace Ray
```

Algorithm 3: *Pseudo code of the Trace Ray Recursive Algorithm of Ray Tracing*

3.2.2 Intersection Testing Using Displacement Fields

The common input data for our algorithm with the straight-forward approach of the ray tracing algorithm are the ray origin and the ray directions. Ray origin is passed as a vector of three float number for the first general step and as a texture in the recursion step. Ray directions are passed in a 2D texture in all steps. But the similarity of data representation with the common ray tracing approach stops here. Instead of triangle data we pass to the Intersection program the displacement fields in a 3D Texture. The procedure of finding the intersection point with the object is as described in the *Displacement Fields Indexing* chapter of the displacement fields. The only part we need to describe is how we get the final intersection data since each object is intersected alone. Let's consider the first step of intersections. That is the intersection test with the first object of our scene:

I. Perform ray – sphere intersections calculating the intersection point and normal.
II. Locate the sample point on the sphere
III. Find the ray index that matches to our ray based on the direction
IV. Read the corresponding distance value from the 3D texture
V. Calculate the approximated final point of intersection with the object
VI. Store the intersection point and the distance from the origin in a 2D texture

In all other steps, that is for all other objects, we perform the same steps except the final one. At that step we have to check which point is closer to the origin. If the already stored one is the closest, we discard our results. Otherwise we overwrite the pixels value in the 2D texture as the new point of intersection is the one closer to us.

The data we store in this part of the algorithm are the following:

I. A 2D texture for each object that contains the calculated intersection points as RGB values and the distance from the bounding volume to the object as Alpha value.
II. A 2D texture containing the closest intersection points from all objects as RGB values and the total distance between the origin and the intersection point as Alpha value.
III. A 2D texture containing the texture coordinates of the 3D displacement texture. That is the index of the pre-calculated ray corresponding to the ray we just checked. Will need this data in order to read the normal value from the corresponding 3D texture of normals in later stages.

```
for all rays do
      read ray origin and direction from corresponding textures
      ray-sphere intersection → p, n (point & normal of intersection)
      if ray hits sphere do
            locate the sample point corresponding to p
            find the index of the closest stored ray in sample based on n
            use index to read distance from displacement field → d
            calculate final distance from origin → D
            calculate final intersection point → P
            if D smaller than previous calculated value of D do
                  store in 2D texture P,d → modelData
                  store in 2D texture P,D → sceneData
                  store in 2D texture index of stored ray → indexData
end
```

Algorithm 4: *Pseudo code of Intersection Algorithm using Displacement Fields*

3.2.3 Shadow Testing Using Displacement Fields

In the common ray tracing approach whenever a ray hits an object of the scene, a corresponding shadow ray is casted in order to determine whether the current fragment should be normally lit or if it under shadow. In our approach the same procedure occurs. A new ray is cast with direction from the light source to the object and the light source as origin. The opposite approach could also be used but it would require the shadow test to be subdivided in two processes: Test for shadows from other objects and self shadow test. As shown in *Figure 20*, the ray with the point tested for shadow as origin and direction from the point to the

light source, hits the bounding sphere of the object. The method used to find the closest ray direction from the stored directions to the intersection direction, will match falsely with some inwards direction. This would result to a false positive shadow check. This is the reason a different approach would be needed for that case. In *Figure 19* you can see the used approach mentioned before. Of course in order to perform the described approach, we have to pass the previous stored textures containing all the closest intersection points. The shadow information for each pixel is stored in a separate texture which will be used later in the light equation. For multiple light sources the procedure is performed for each light.

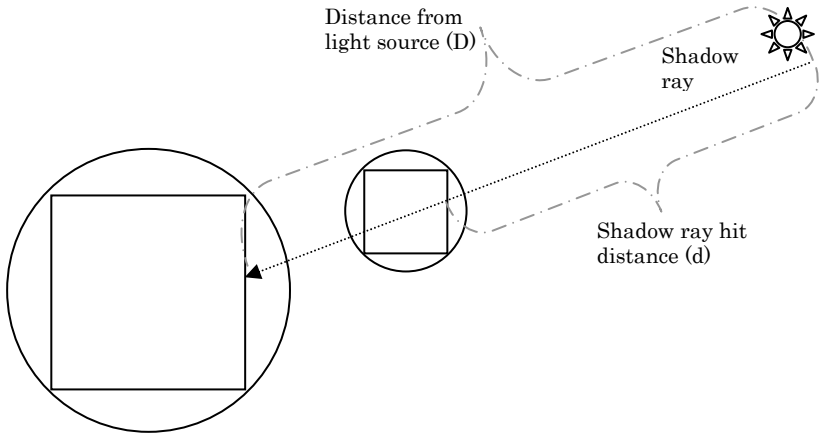


Figure 19: *Shadow ray approach used in our Ray Tracer*

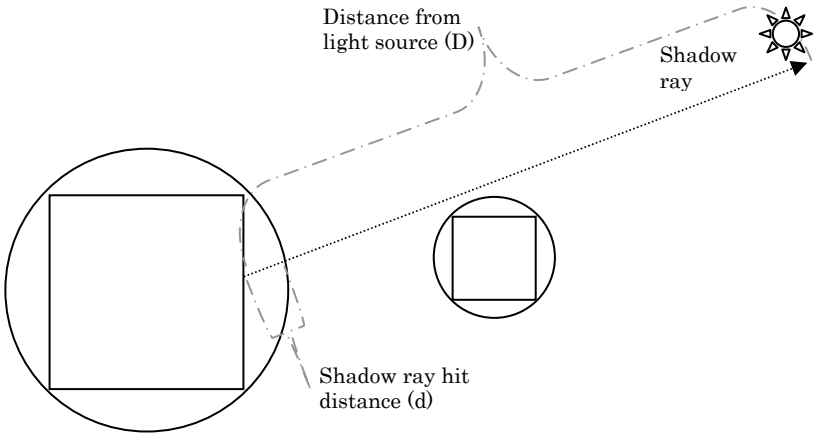


Figure 20: *Inverse shadow ray approach with false positive*

```
for all rays do
      read P, d values from modelData
      if d > 0.0 (pixel has stored value for object) do
            read P, D values from sceneData
            calculate ray direction from light to point P
            ray-sphere intersection → p', n'
            find the index of the closest stored ray in sample based on n'
            use index to read distance from displacement field → d'
            calculate final distance from origin → D'
            if D' < D && ray hits underline shape do
                  store shadow information in shadowTexture
end
```

Algorithm 5: *Pseudo code of Shadow Testing*

3.2.4 Light Equation Using Displacement Fields

The light equation used in our Ray Tracer is Phong's [19] classic illumination model. The visibility intensity is estimated as the sum of four components: emission, ambient reflection, diffuse reflection, and specular reflection:

$$I = I_e + I_g + I_d + I_s$$

The ambient component I_g compensates for the fact that the Phong model takes no account of the interaction of light between objects; a surface that is not directly illuminated by a light source would appear completely un-illuminated if it were not for this component. A constant value of ambient light I_α is assumed for the scene and each object reflects this ambient light according to its ambient reflectance coefficient k_α:

$$I_g = I_a k_a (0 \le k_a \le 1)$$

The light that hits an object directly from a light source is split into two reflected components: diffusely reflected light, which is uniformly scattered in all directions and specularly reflected light, which has its maximum value in the "mirror" of the lighting direction. The diffuse and specular reflection coefficients k_d and k_s depend mainly on the object's surface properties. In general, the rougher the surface the more light is diffusely reflected; while the shinier the surface the lighter is specularly reflected. As all incident light must be accounted for: $0 \le k_d, k_s \le 1$ and $k_d + k_s \le 1$. The sum of k_d and k_s may be slightly smaller than 1 to account for light that is transmitted or absorbed by the object.
The diffuse component assumes a Lambertian surface and distributes incident light evenly in all directions. It therefore does not depend on the viewing direction. Its value is proportional to the irradiance which is replaced by intensity Ii according to the photometry law; the distance d of the light source is ignored.

$$I_d = I_i k_d \cos\theta = I_i k_d (n \cdot l), (0 \le \theta \le \pi/2, 0 \le k_d \le 1)$$

Where I_i the intensity of a point light source, θ the angle between the direction of light incidence (l) and the normal vector to the surface (n) (see *Figure 21*), and k_d is the object's diffuse reflection coefficient. Apart from the object's roughness, k_d also depends on the wavelength of the incident light. The vectors l and n should be unit vectors. The value of I_d is constant over a planar surface since both n and l vectors are constant (light source at infinity). In practice negative values of $\cos\theta$ are not accepted:

$$I_d = I_i k_d \max(0, n, l)$$

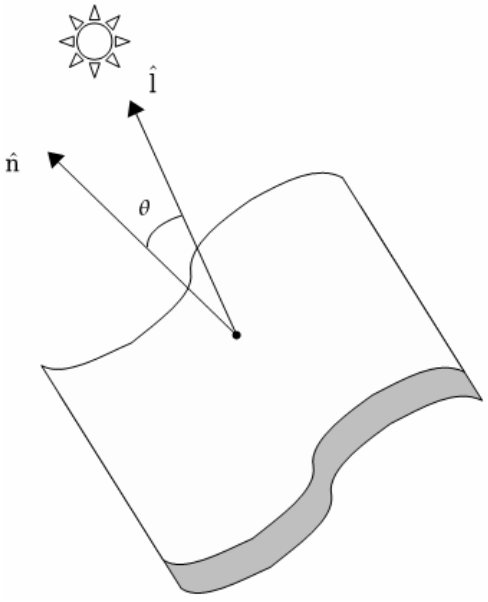


Figure 21: The I and n vectors

The diffuse component alone gives objects a totally matte appearance. The specular component on the other hand follows the rule of the mirror. Perfect mirrors will only specularly reflect in the direction of reflection r (see *Figure 22*). Most surfaces will have a diminishing function of specular reflection that attains its maximum value when the viewing direction v coincides with r:

$$I_s = I_i k_s \cos^n \alpha = I_i k_s (r,v)^n$$

Where r and v are unit vectors and n is an empirical value that corresponds to the surface shininess.

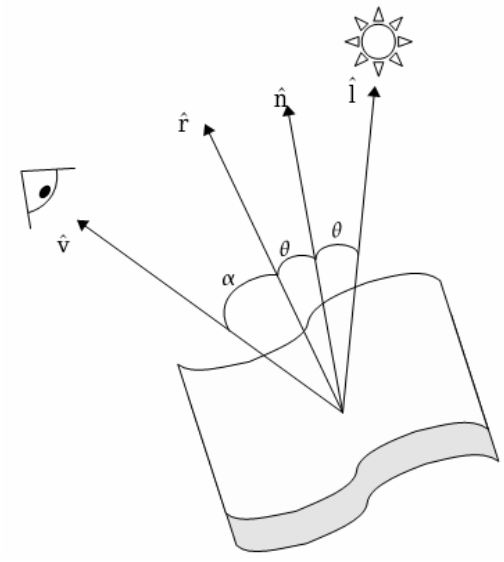


Figure 22: The v, r, n and I vectors

Specular reflection is responsible for the highlights that are visible in shiny objects. The $\cos^n a$ term intuitively approximates the spatial distribution of the specularly reflected light. The effect of the material exponent n and the coefficient k_s can be seen in Figure 23. Small values of n correspond to coarse materials where the size of the highlight is relatively large and scattered.

Conversely, large values of n correspond to shiny objects with a small and crisp highlight. The specular reflection takes the color of the light source. For example, if a blue object is illuminated by a white light source, the color of the diffuse reflection will be blue but that of the specular reflection will be white. Finally, the value of the specular factor $\cos^n a$ should not take on negative values, so we can replace it by $\max(0, \cos^n a)$

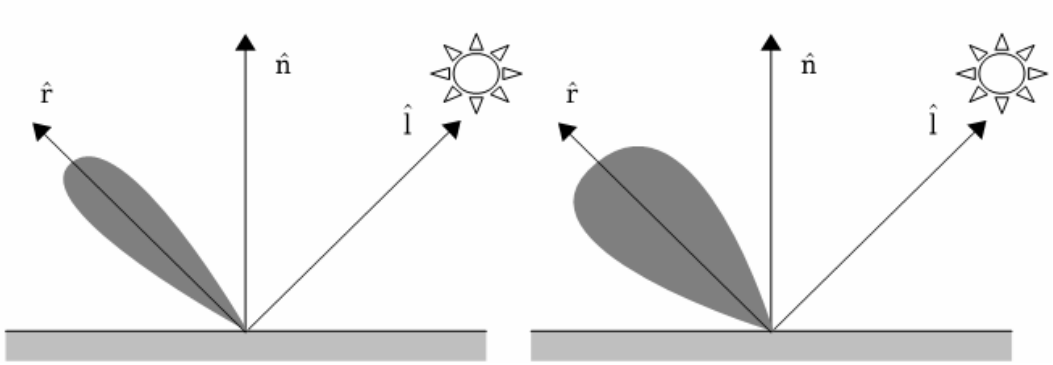


Figure 23: Phong highlight (shaded area) for large n (left image) and small n (right image)

In that way Phong model computes the illumination value as:

$$I = I_e + I_a k_a + I_i \left(k_d (n \cdot l) + k_s (r, v)^n \right)$$

The only critical point to mention about our implementation is that this is the point where the stored index of the displacement maps is used with the 3D normal texture that corresponds to the displacement field of the object. The final color is stored in a texture value and is used in the recursion process of ray tracing. As we said, in ray tracing many rays may contribute to the final color of a pixel. In that way each ray has a contribution factor (ex. Reflection ratio).

```
for all rays do
      read P, d values from modelData
      if d > 0.0 (pixel has stored value) do
            read shadowTexture
            if pixel under shadow do
                  use shadow color
            else
                  read index value from indexData
                  use index to read normal from 3D normal  texture
                  perform Phong's shading model
                  read previous value of color from colorData
                  write new value based on contribution to colorData
end
```

Algorithm 6: *Pseudo code of Light Equation*

3.2.5 Reflection Using Displacement Fields

As we mentioned in 3.2.4 section, in order to predict the direction of maximum specular highlight, we have derived the reflection vector $\vec{r}$ in terms of the normal vector $\vec{n}$ of the surface at the point of incidence and the direction of the incoming light l. The reflected and incident directions lie on a plane perpendicular to the surface and according to the law of reflection, the angle of incidence θ_i equals the

angle of reflection θ_r that is, the incident and reflected light propagation directions are symmetrical with respect to the normal vector. Thus the calculations for an arbitrary ray of light from a direction $\vec{r}_i$, incident on a perfectly reflecting interface between two bodies, the reflected ray in the perfect mirror reflection direction $\vec{r}_r$, is given by (see *Figure 24*):

$$\vec{r}_r = \vec{r}_i - 2\vec{n}(\vec{n} \cdot \vec{r}_i)$$

We have to notice at this point that the incident direction is the opposite of the light direction vector l mentioned in the previous section as we need to emphasize on the direction of propagation for clarity.

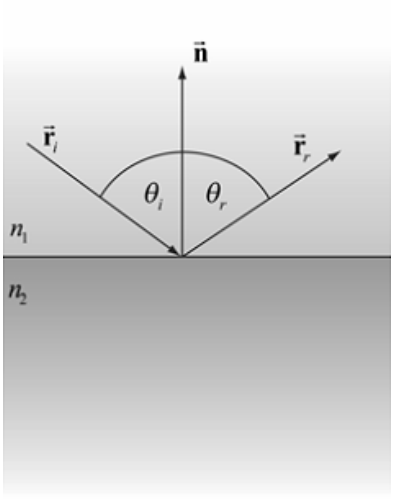


Figure 24: *Reflection*

In our implementation, we create two unique textures common for all objects. The first one contains the new origins which are the previous intersection points, while the second contains the new directions which are the product of the reflection of the previous directions of the rays and the normal of the surface. These two textures are then passed to the intersection algorithm which we described in previous chapter.

```
for all rays do
      read P, d values from modelsData
      if d > 0.0 (pixel has stored value) do
            read index value from indexData
            use index to read normal of surface from 3D normal texture
            read ray direction from direction texture
            create reflection direction → r
            store P in new origin texture
            store r in new direction texture
end
```

Algorithm 7: *Pseudo code of Reflection*

4. Test Cases

In our test cases we used mostly 4226 positional samples on the bounding sphere and 3 different resolutions for the concentric map sampling of the ray directions. More resolutions could also be used for the positional samples of the bounding sphere, but the 4226 resolution gave the best results.

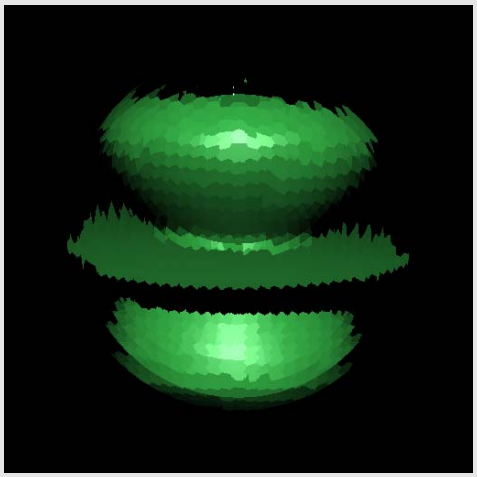
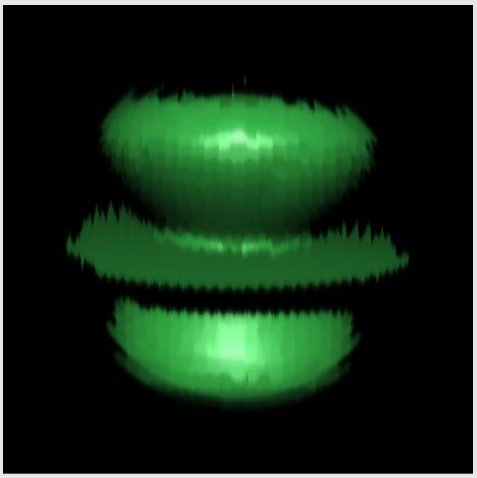
In the following figures we see the rendered images from our method in comparison to standard ray tracing. The comparison is both in time and in visual approximation of the reference image while all images were rendered in a 512x512 viewport. It is critical to mention that the CPU and GPU methods of the standard ray tracing are not the brute force (Intersection test of each ray with each triangle). On the contrary, they are both accelerated by BVH structure which, as proven in *[20]*, consistently outperforms the Kd-tree and the Uniform Grid schemes in GPU ray tracing, sometimes by as much as a factor of nine. At the same time the BVH scheme remains a competitive scheme for acceleration on the CPU.

The quality measure of our rendered images is the *Root Mean Square (rms)* error. Each rendered image is presented two times. One directly from our rendering and the other with Gaussian blur as a post filter to our rendering procedure.

All tests have been performed on a AMD Athnlon 64 X2 Dual with 2GB of RAM and an NVIDIA GeForce 8600GT graphics board, with 512MB Video Ram. The operating system was a 32-bit Linux.

4.1. Super Shape 1

The super shape 1 model is assembled by 16128 triangles and the scene is rendered using one light.

	Original Rendered Image	Post Filtering with Gaussian Blur
4226x32x32		
time / rms	36ms / 14.46	38ms / 12.66

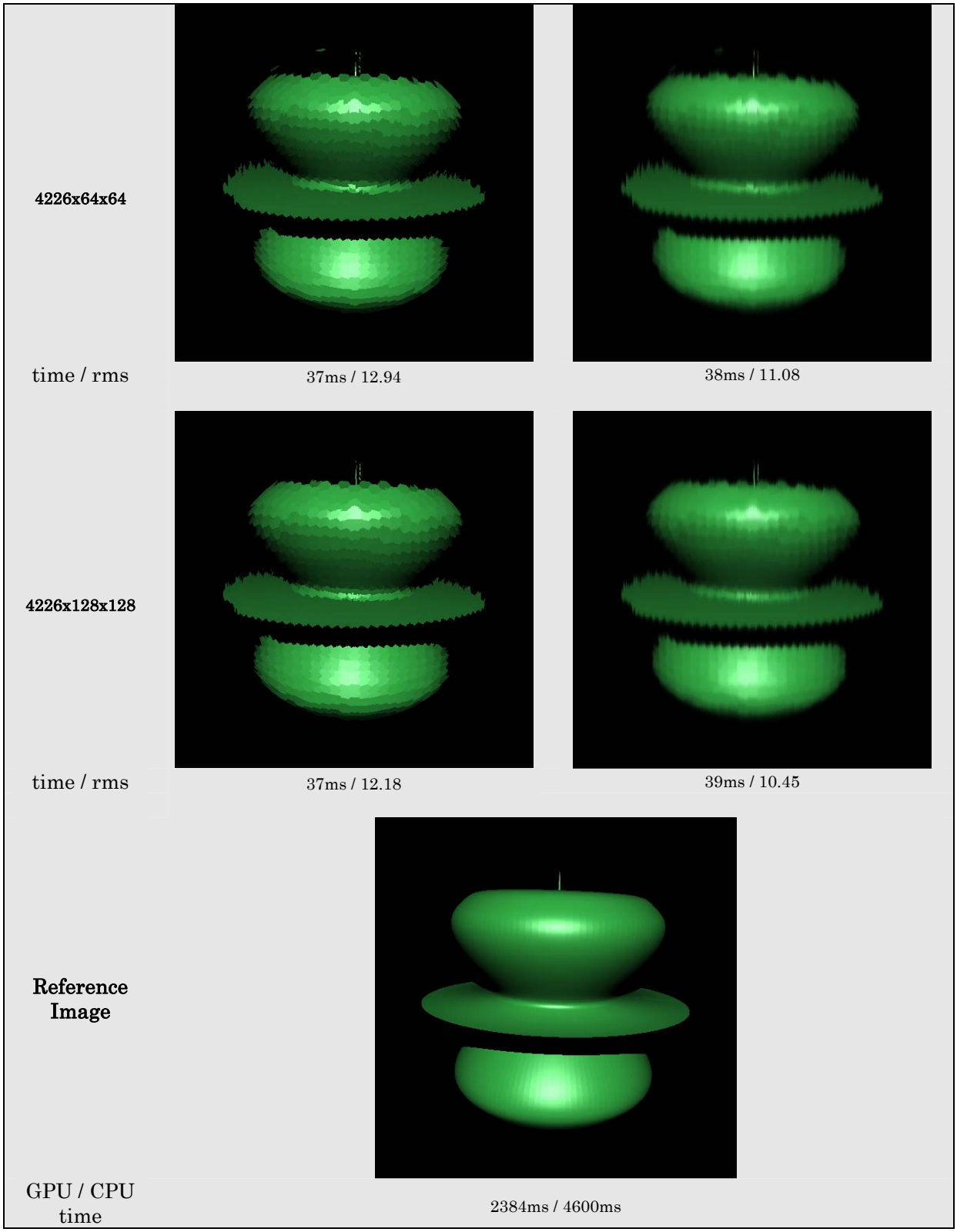
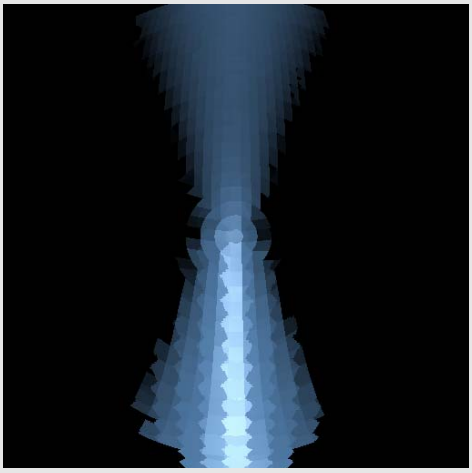
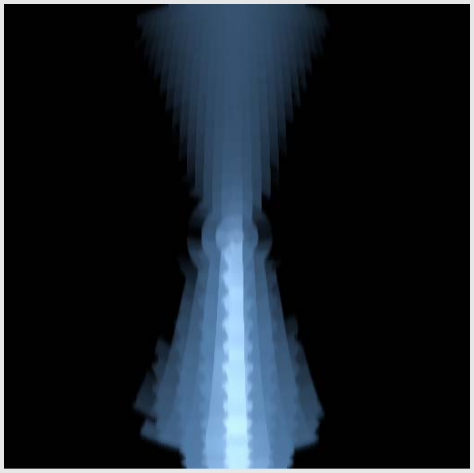
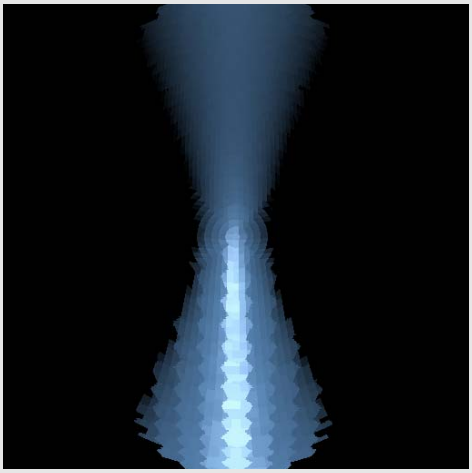
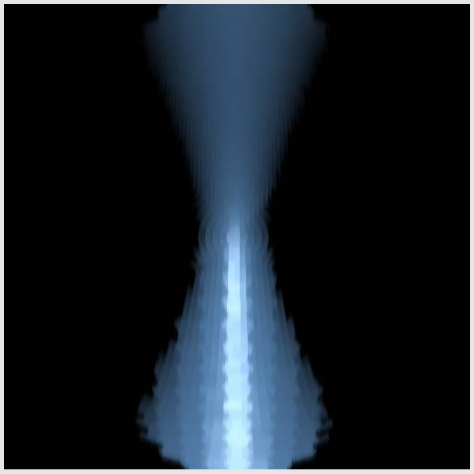
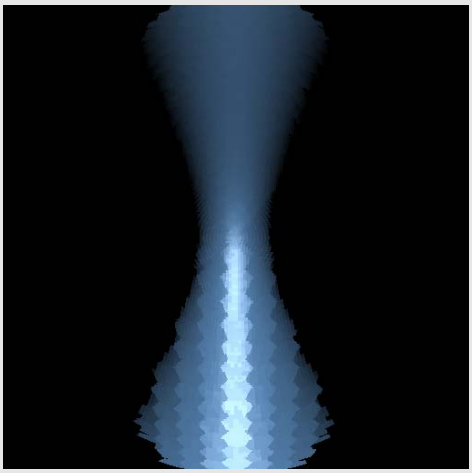
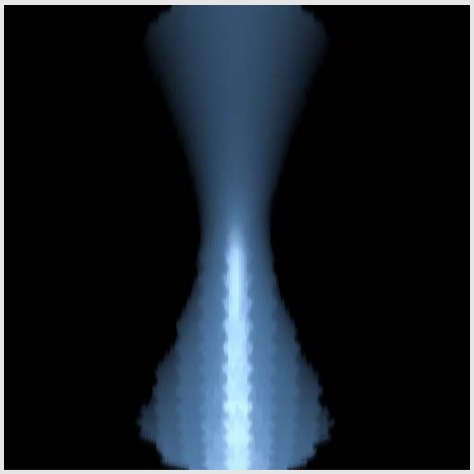


Figure 25: *Super Shape 1. Ray Tracing Results with Displacement Fields & Reference Image using standard Ray-Tracing*

4.2. Hyperboloid

The super shape 1 model is assembled by 16384 triangles. The scene is rendered using one light.

	Original Rendered Image	Post Filtering with Gaussian Blur
4226x32x32		
time / rms	37 ms / 12.38	38 ms / 11.30
4226x64x64		
time / rms	37 ms / 8.58	40 ms / 7.18
4226x128x128		
time / rms	40 ms / 6.68	41 ms / 5.28

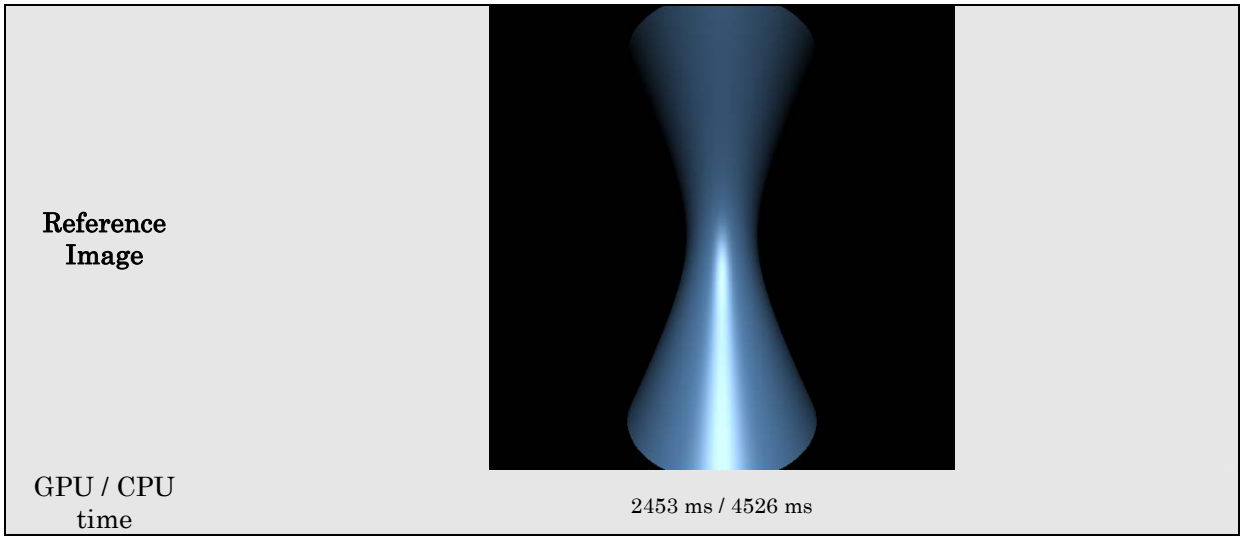
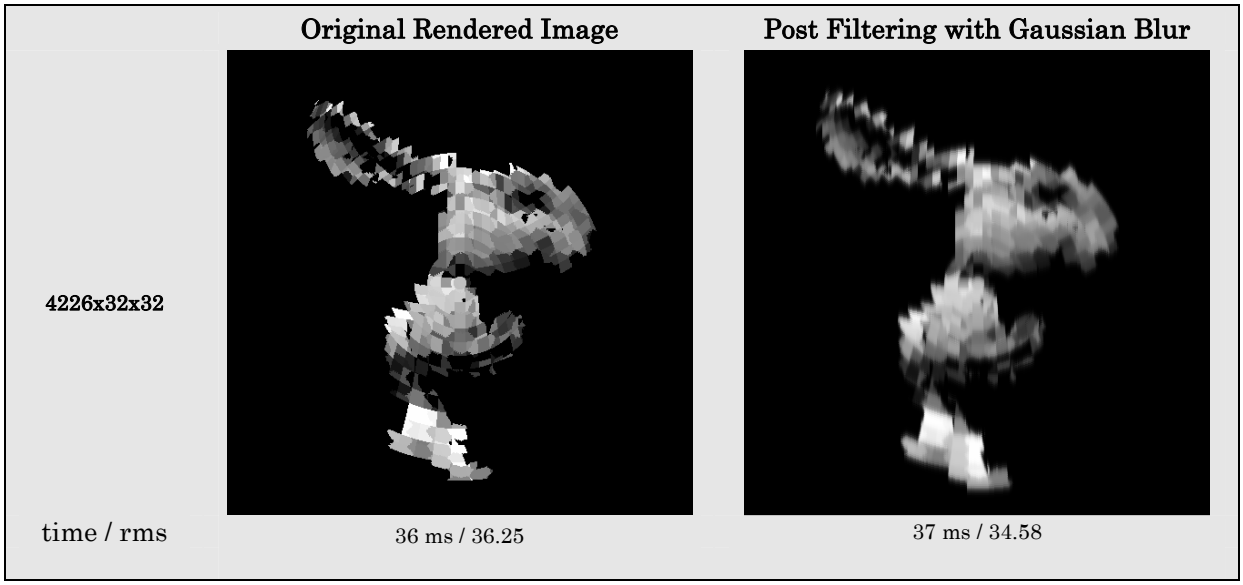


Figure 26: *Hyperboloid. Ray Tracing Results with Displacement Fields & Reference Image using standard Ray-Tracing*

4.3. Bunny

The bunny model is assembled by 38889 triangles. The scene is rendered two times, the first using one light and the second using two lights. This is done in order to compare the differences in the time and the quality of rendering between our method and the classical approach.



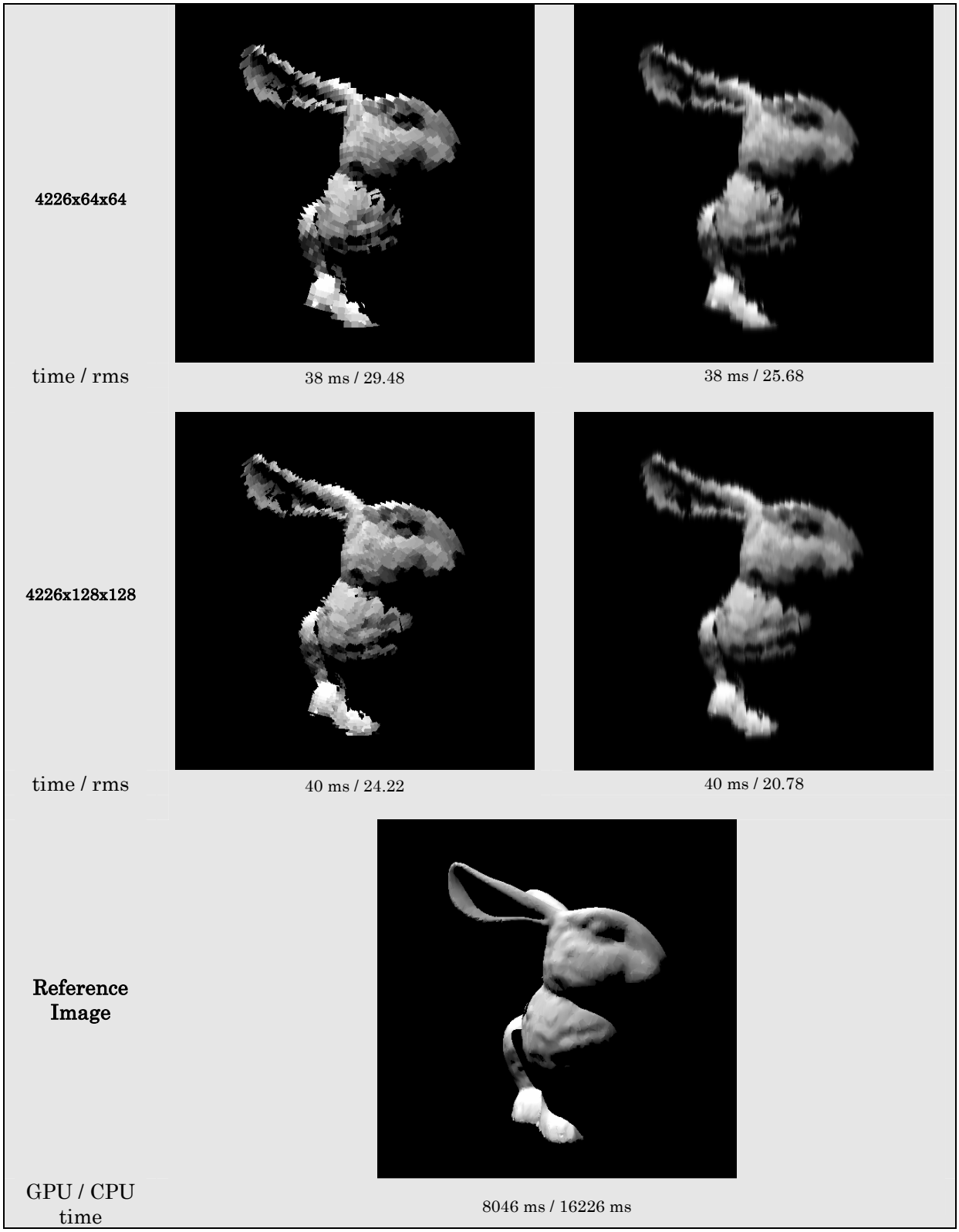
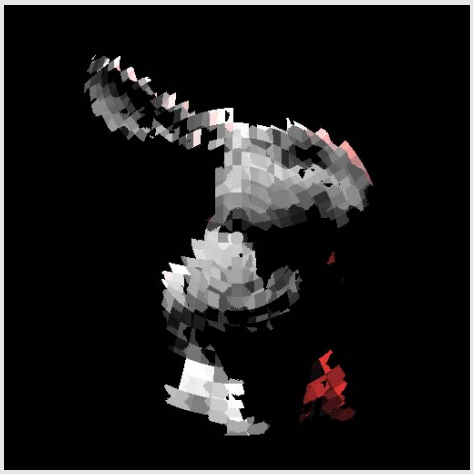
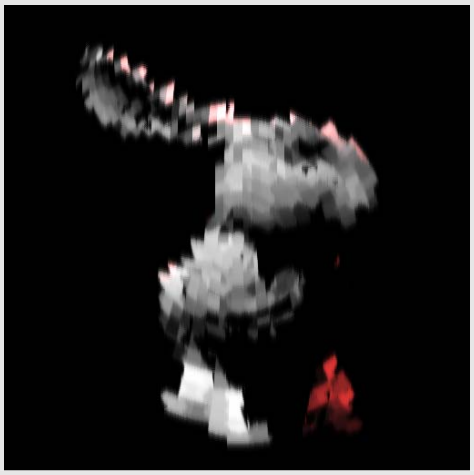
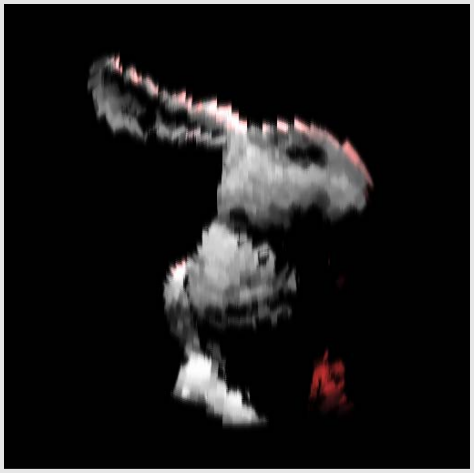
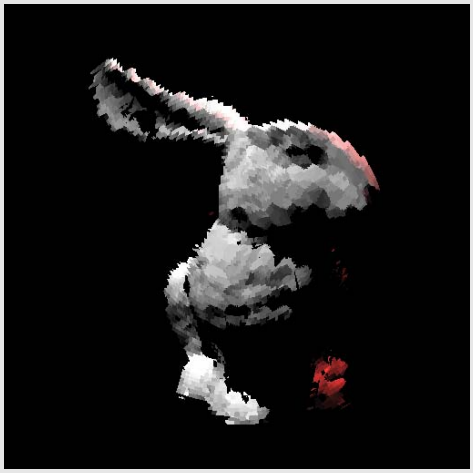
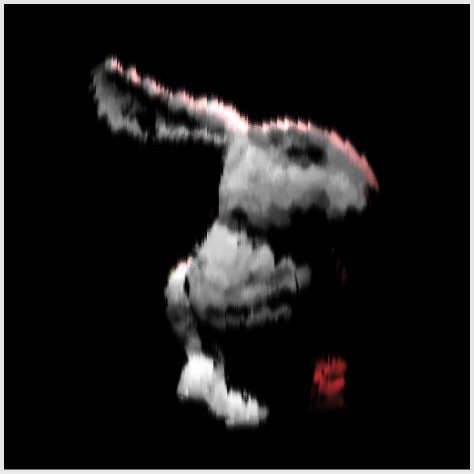


Figure 27: *Bunny rendered with one light. Ray Tracing Results with Displacement Fields & Reference Image using standard Ray-Tracing*

	Original Rendered Image	Post Filtering with Gaussian Blur
4226x32x32		
time / rms	40 ms / 38.38	41 ms / 36.35
4226x64x64		
time / rms	40 ms / 31.31	42 ms / 27.18
4226x128x128		
time / rms	44 ms / 25.82	46 ms / 21.88

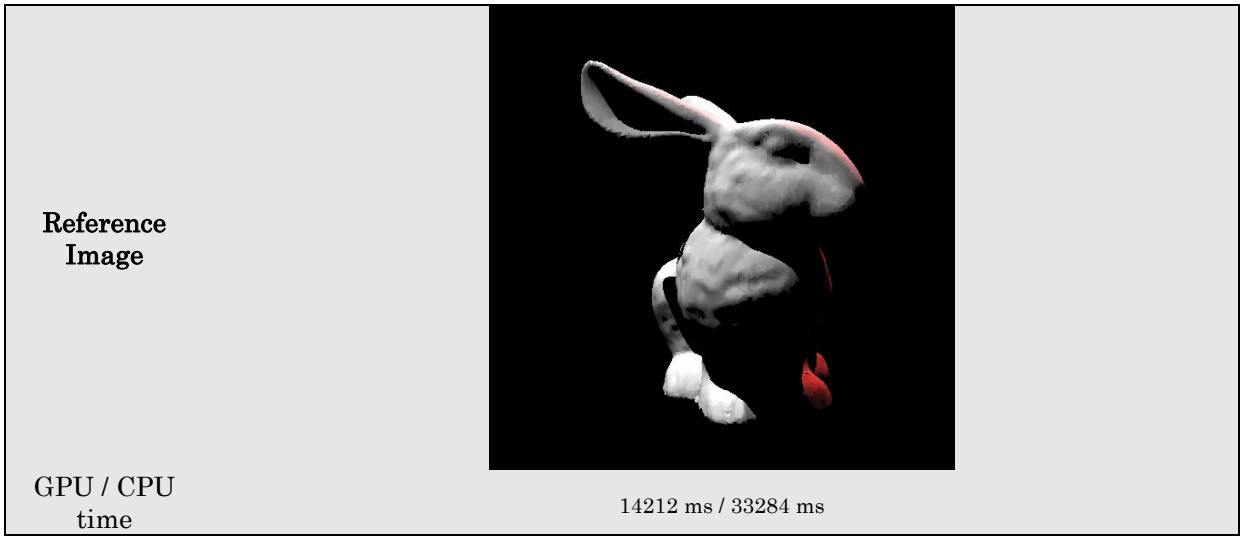
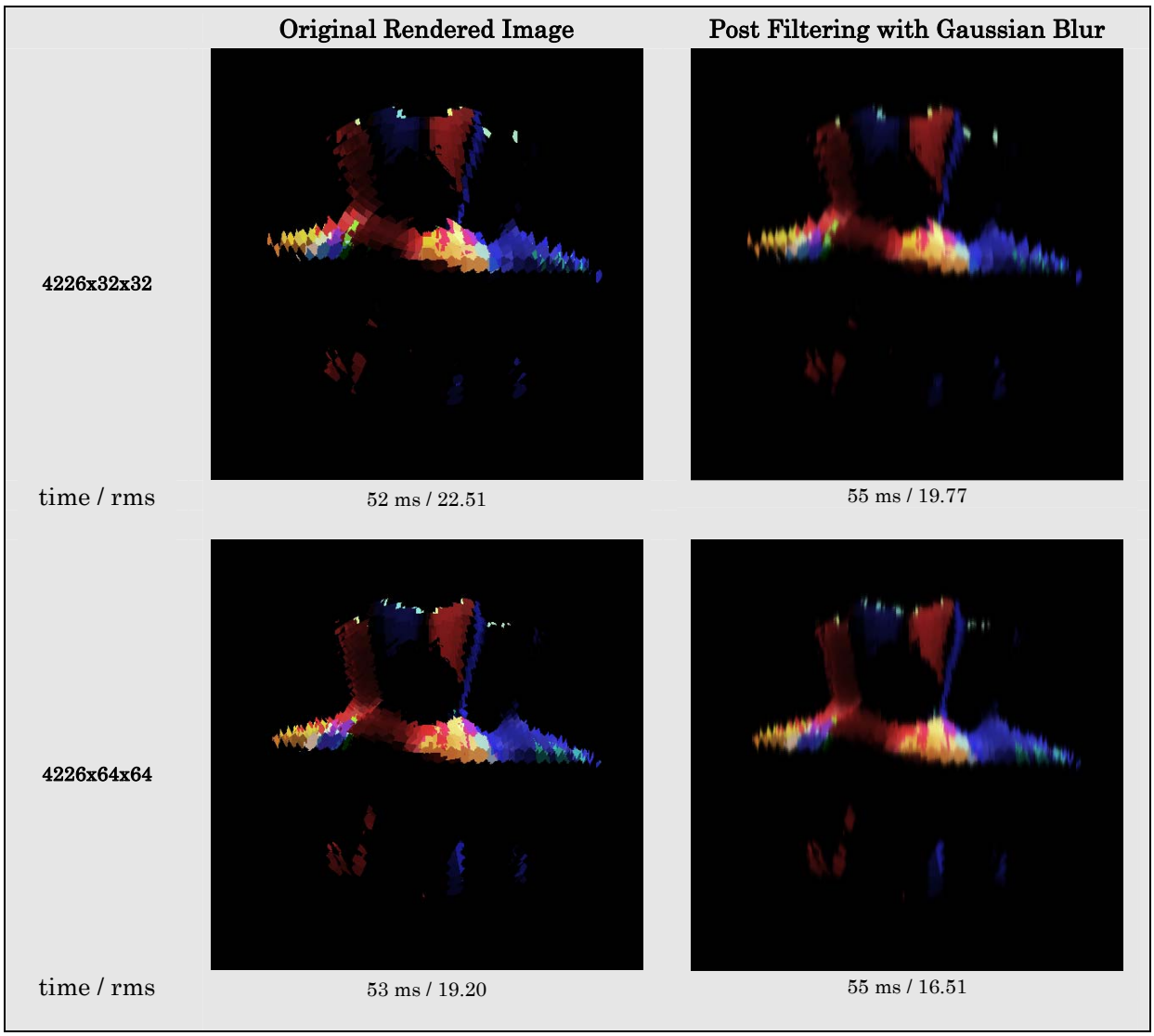


Figure 28: *Bunny rendered with two lights. Ray Tracing Results with Displacement Fields & Reference Image using standard Ray-Tracing*

4.4. Super Shape 2

Another super shape rendered with 3 lights. The object is assembled by 16128 triangles



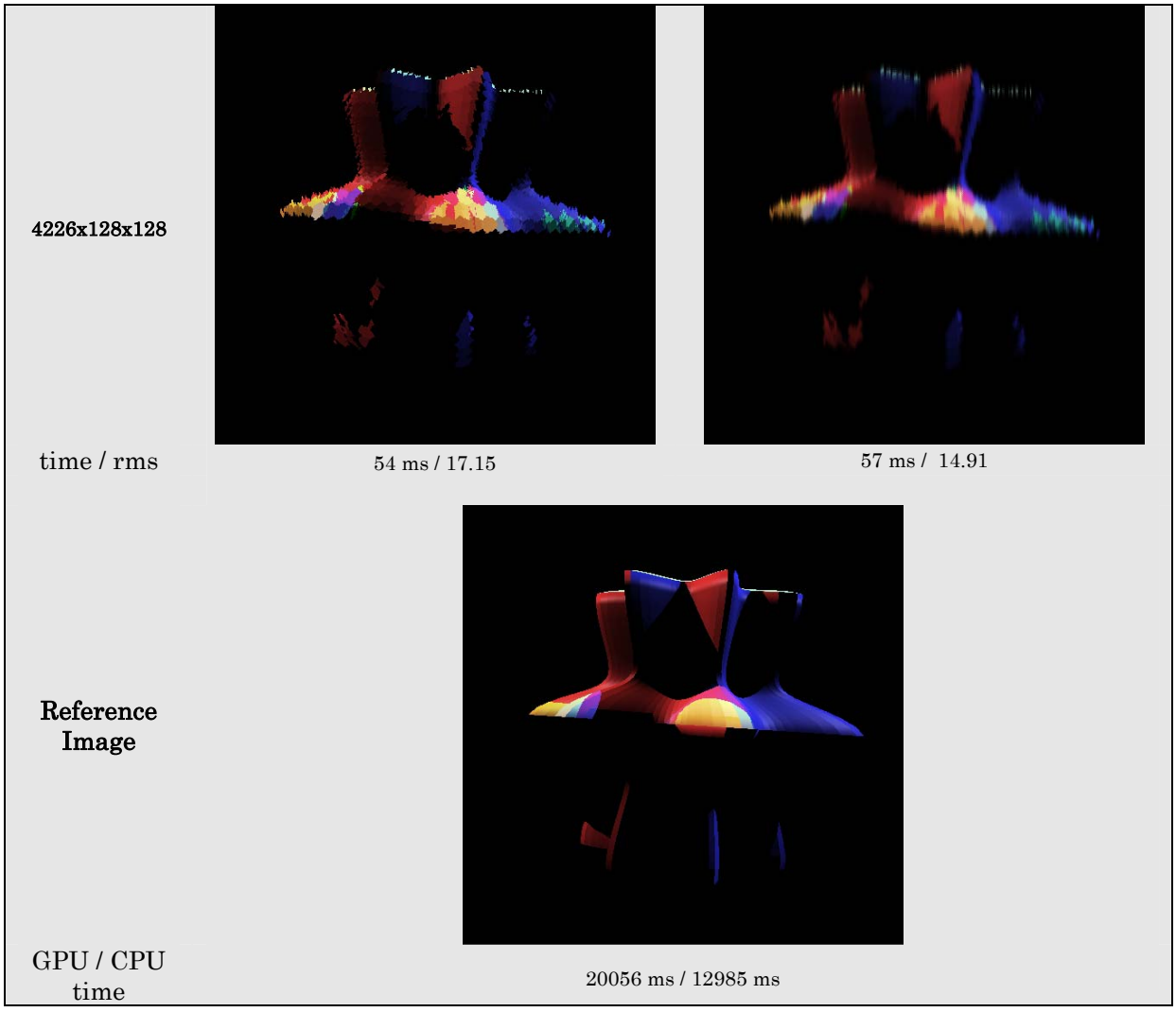


Figure 29: *Bunny rendered with three lights. Ray Tracing Results with Displacement Fields & Reference Image using standard Ray-Tracing*

4.5. Reflection Scene

The first reflection scene is assembled of two objects of total 32512 triangles. The hyperboloid object of the scene is the one with the reflective surface. One thing to notice in this example is that our method performs much better on secondary rays from a quality aspect. The resulting reflected sub region gives a very good approximation of the reference reflected area.

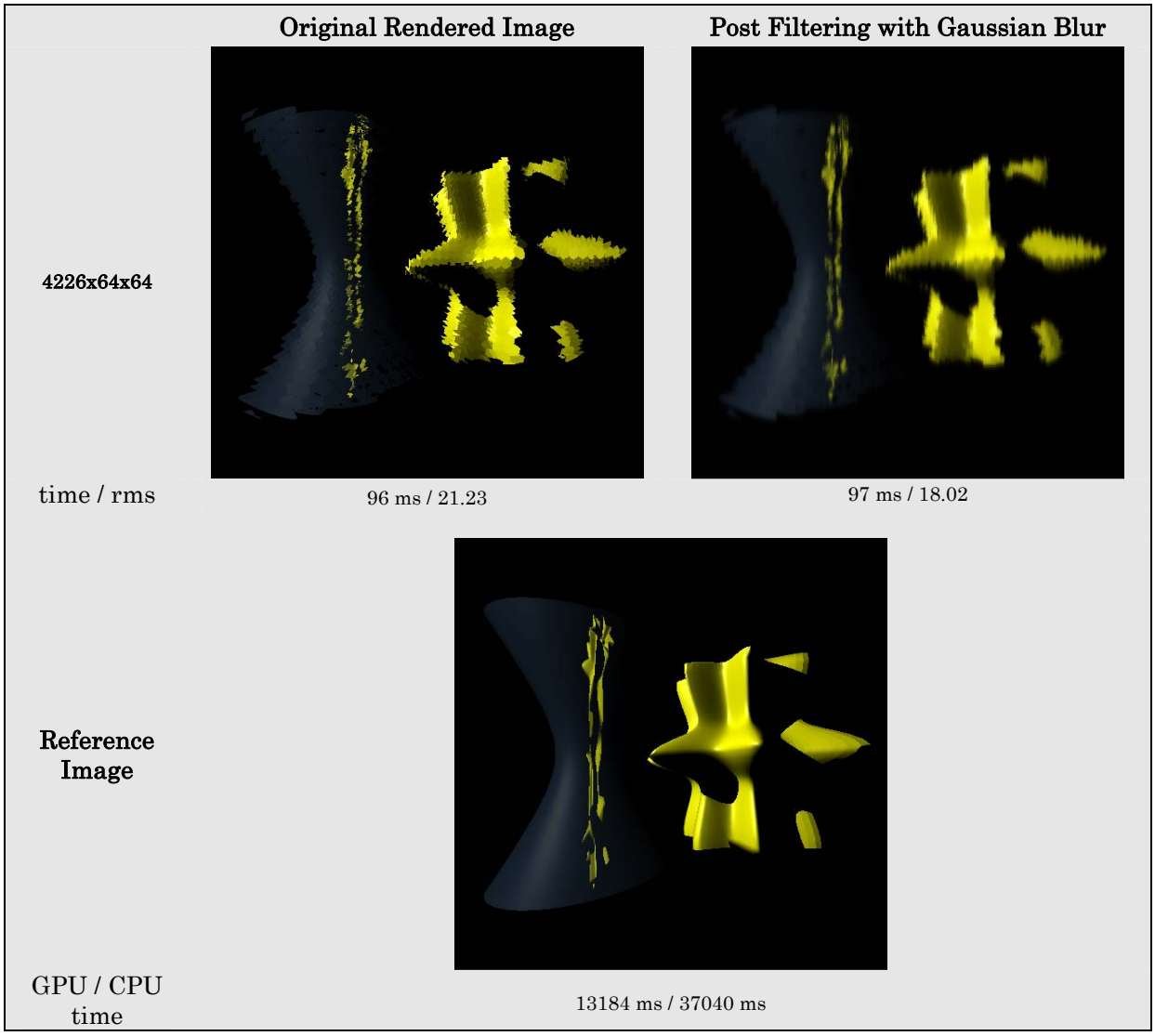


Figure 30: *Reflection scene1.*

The reflection scene 2 is assembled of two objects of total 19384 triangles. The previous example gave us the idea of comparing a reflective scene with polished surface against the same scene with a nearly mirrored surface. The polished reference image is rendered with 4 rays for each reflected pixels which leads to slower rendering times. As we can see from both the images and the rms factor, the reflected sub region of our method is closer to the result of the polished reference image. On the other hand the impact of using a polished surface on the rendering time of reference methods is almost a multiplier of 16.

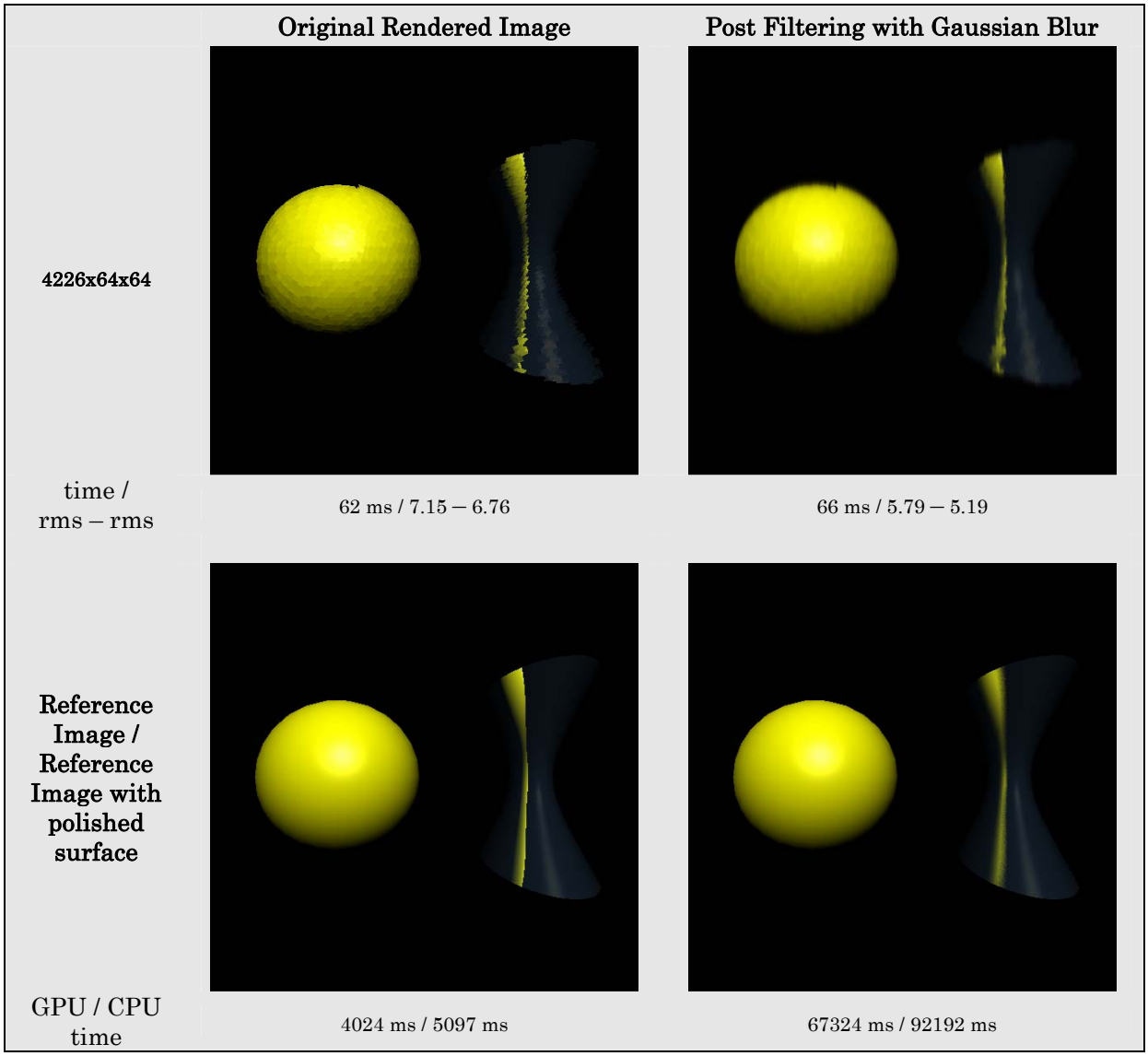


Figure 31: *Reflection Scene 2*

4.6. Shadow scene

A scene with inter-object shadow rendered using three lights. The total triangle count of the scene is 56317. In this example we notice a good approximation of the reference image despite the use of multiple lights. On the other hand the use of multiple lights increases greatly the difference between the rendering times of the reference methods and our method. Another noticeable detail is that from an aspect of quality the inter-object shadowing sub regions of our rendering are very close to the reference ones.

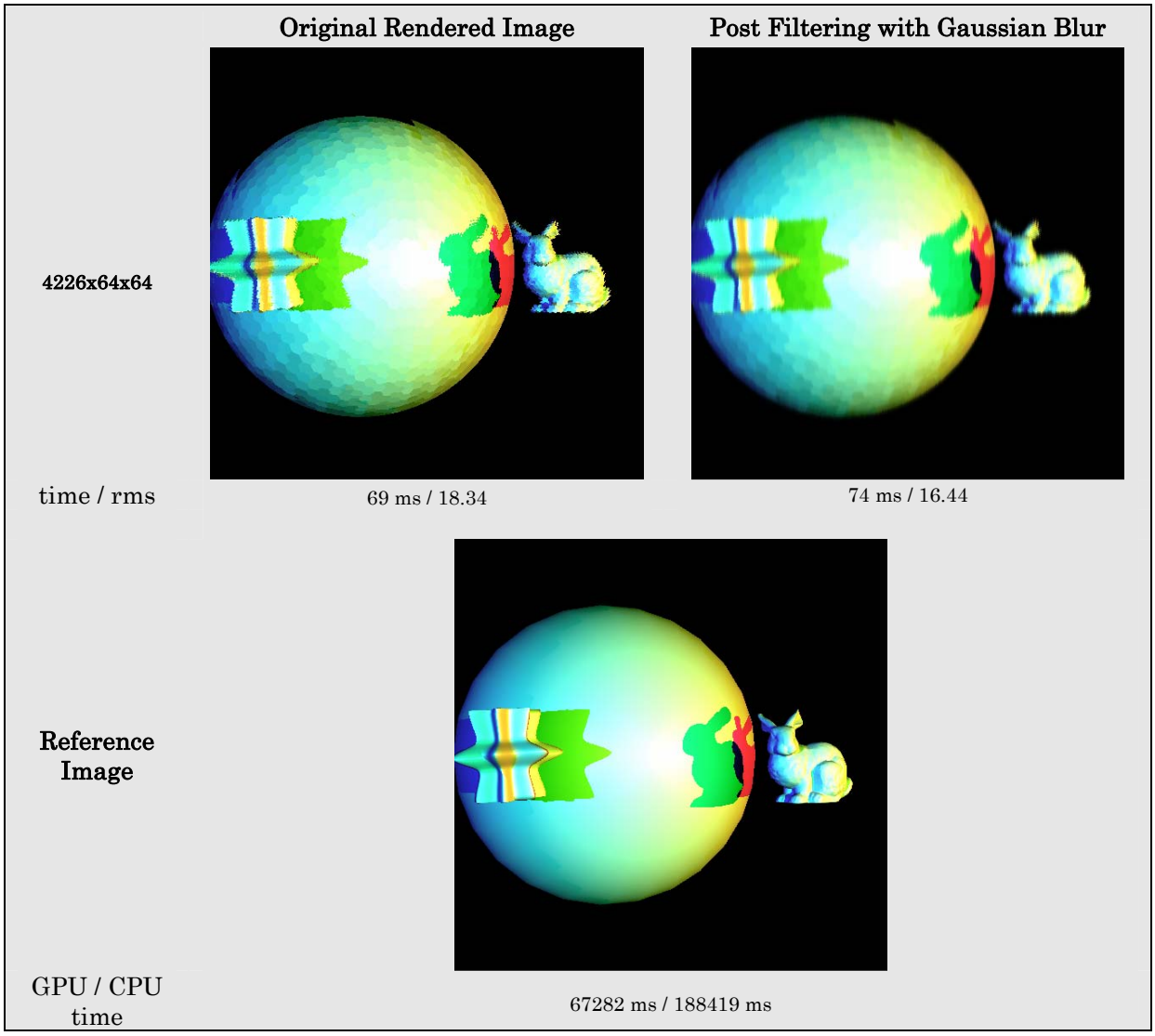


Figure 32: *Scene rendered with three lights, resulting to inter-object shadowing.*

4.7. Summary of results

As we expected the finer the resolution of the concentric maps sampling of ray directions, the better the rms error measure. The visual comparison between the different rendered images differs based on the complexity of the models edges. This also is shown by rms factor. The hyperboloid object (see *Figure 26*) is the one achieving the smallest error while the bunny object is the one with the highest rms error factor (see *Figure 27*).

In terms of memory consumption the 4226x32x32 maps correspond to 4.16MB of depth textures and 12.38MB of normal textures. The 4226x64x64 maps correspond to 16.05MB of depth textures and 49.5MB of normal textures while the 4226x128x128 maps correspond to 66MB of depth textures and 198MB of normal textures. However as we can see from all results since the method uses a constant time query access the desired data the rendering times are almost equal despite of the increase of the resolution of the stored ray direction.

From Figures *25*, *26* and *27* we observe that the cost of using the displacement maps does not depend on the underlying geometry of the objects, while the other two methods have almost linear correlation to the total triangle count. This is an expected result as all triangles intersections are replaced by the

constant time lookup to the displacement fields.

Another great observation from *Figure 27* and *Figure 28* is that adding a second light source causes doubling of rendering time to the standard ray tracing approach. This is expected as well since all intersections are doubled as well. On the other hand doubling the light sources in our method only results to a slight but disproportionally small increase of rendering time. With this result in mind we can conclude that in our method the intersection tests are not the most computationally expensive part. Instead the inter communication and the changing of contexts between CPU and GPU is the most time consuming procedure. The same conclusion is supported by the results of the scene rendered with three lights (see *Figure 29*) as well from the results of shadow scene and reflection scenes (see *Figure 30*, *Figure 31* & *Figure 32*) where render times between our implementation and standard ray tracing have great differences.

All results taken with our method present render times of real time in contrast with the standard ray tracing. These results prove that the use of displacement fields can vastly accelerate the ray tracing algorithm making it possible even for real time rendering.

On the other hand from an aspect of rendering quality our method degrades the final image because of the approximations in the method. This can be alleviated by the use of more sampling points on the bounding sphere or by increasing the pre-calculated ray directions of the sphere samples.

From an aspect of memory consumption our method is forbidding for scenes with a great number of distinct objects, in contrast with the basic ray tracing approach. However in scenes with instantiated objects (ex. Five spheres) regardless of their transformations, displacement textures are loaded only once for memory saving.

4.8. Secondary Rays Approach

While in ray tracing visually correct results are the final goal, there are cases where sharpness is not always the desired result. Such examples are shadows and reflective or permeable materials. Low distortion is only necessary in the case of flat mirror reflection. Reflection blurring due to imperfect polished surfaces or reflection and refraction on curved surfaces favours our solution as secondary rays are significantly more in number than primary ones.

A major improvement to the basic ray tracing algorithm in terms of visual quality is distributed or stochastic ray tracing. In this approach instead of sampling the contributing energy from a single direction as in the basic algorithm, multiple rays randomly distributed over a solid angle centred at the principal ray direction are cast. This is essentially a Monte Carlo approximation of the integral of all energy contributing to the path that was traced from the eye point to the scene. The method dramatically enhances the visual quality of the result in the expense of course of the rendering time required to intersect the extra rays with the scene. The same technique can be used to the shadow rays in-order to achieve soft shadows with the expense again of increasing the rendering time.

Instead of casting multiple rays with randomly perturbed directions, we could use our implementation with displacement fields only for secondary rays or for shadow rays, achieving in that way almost the desired effect at nearly no cost. In *Figure 33* we can see that the addition of reflection in a scene nearly doubles

the rendering time. Even greater is the impact of using polished surface as it can be seen from the results of reflection scene 2 (see *Figure 31*). Instead if we used our method for the reflection rays, the rendering time would had only increased by a constant of around 100ms while achieving at the same time non sharp reflections at no cost.

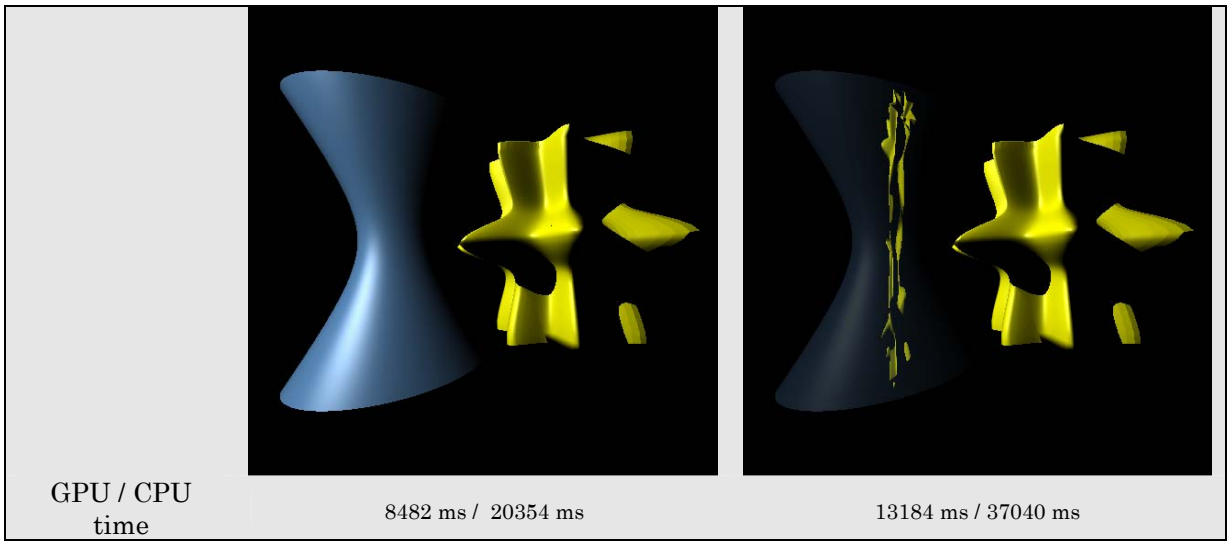


Figure 33: *Same scene rendered with no reflection on the left and with reflection on the right. Rendering Time is nearly doubled*

5. Conclusion & Future Work

We have presented an alternative approach to Ray tracing with the use of displacement fields, a novel discretization of the visibility around an object. The number and resolution of the displacement maps used in the displacement field can be adjusted depending on the required accuracy and the available memory

We have shown that this alternative approach can be used in cases where a degrading of the visual quality for real time rendering times is acceptable. The method is robust and it especially favours scenes with few objects but with high triangle count as it is independent of the model complexity.

Furthermore our method can be applied to specific ray tracing calculations where exact ray hits are not critical improving in that way rendering times. Such examples are soft shadows and secondary rays for non planar reflective/refractive surfaces.

For future work, we could construct a hybrid ray tracer using our implementation for soft shadowing and secondary rays for refraction and reflection and compare it to the results of a Monte-Carlo ray tracer from aspects of time and image quality.

6. References

[1] Sweeney, M.A.J. and Bartels, R.H., Ray tracing free-form B-spline surfaces. ZEEE Comput. Graph. AMl. 6(2), 41 -49, February 1986.

[2] Hall, R.A. and Greenberg, D.P., A testbed for realistic image synthesis. ZEEE Comput. Graph. Appl. 3(10), 10-20, November 1983

[3] Amanatides, J., Ray tracing with cones. Comput. Graph. 18(3), 129-135, July 1984

[4] Hall, R.A. and Greenberg, D.P., A testbed for realistic image synthesis. ZEEE Comput. Graph. Appl. 3(10), 10-20, November 1983

[5] Rubin, S. and Whitted, T., A three-dimensional representation for fast rendering
of complex scenes. Comput. Gmph. 14(3), 110- 116, July 1980

[6] Weghorst, H., Hooper, G. and Greenberg, D., Imprnved computational methods for
ray tracing. ACM Tmns. Graph. 3(1), 52-69, January 1984

[7] Kay, T.L., and Kajiya, J., Ray tracing complex scenes. Comput. Graph. 20(4), 269-278, August 1986.

[8] Glassner, A.S., Space subdivision for fast ray tracing. ZEEE Comput. Graph. Appl.
4(10), 15-22, October 1984.

[9] Kaplan, M.R., Space tracing a constant time ray tracer. State of the Art in Image Synthesis (Siggraph '85 Course Notes), Vol. 11, July 1985

[10] Fujimoto, A., Tanaka, T. and Iwata, K., ARTS: Accelerated Ray-Tracing System.
ZEEE Comput. Graph. Appl. 6(4), 16-26, April 1986.

[11] Haines, E.A. and Greenberg, D.P., The light buffer: a shadow testing accelerator.
ZEEE Comput. Graph. AMl. 6(9), 6-16, September 1986.

[12] Ohta, M. and Maekawa, M., Ray coherence theorem and constant time ray tracing algorithm. Computcr Graphics 1987 (Proc. of CG International '87) (ed. T.L. Kunni), pp. 303-314.

[13] Arvo, J. and Kirk, D., Fast ray tracing by ray classification. Comput. Gmph. 21(4), 55-64, July 1987.

[14] Cohen, M.E, and Greenberg, D.P., The hemi-cube: a radiosity solution for complex environments. Comput. Graph. 19(3), 31-41, July 1985.

[15] Alan Chalmers, Timothy Davis, and Erik Reinhard, “Practical Parallel

Rendering". A K Peters, 2002. ISBN 156881-179-9

[16] A. Gaitatzes, Y. Chrysanthou, G. Papaioannou: Presampled Visibility for Ambient Occlusion, Proc. WSCG 2008, Journal of WSCG, 16(1-3), pp. 17-24, 2008

[17] Slater, M.: Constant time queries on uniformly distributed points on a hemisphere. In Journal of Graphic Tools 7, 1 (2002), pp. 33–44.

[18] Huang, P., Wang, W., Yang, G., Wu, E.: Traversal fields for ray tracing dynamic scenes. In ACM Symposium on Virtual Reality Software and Technology (VRST '06), Limassol Cyprus, November 1-3, 2006.

[19] Bui-Tuong Phong. Illumination for Computer Generated Images. Communications of the ACM, 18(6):311-317, June 1975

[20] Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, Denmark, 2005.