Efficient Collision Detection for Dynamic Objects in Distributed Virtual Environments

Diplomarbeit

an der

Rheinisch-Westfälischen Technischen Hochschule Aachen

Fakultät für Mathematik, Informatik und Naturwissenschaften

vorgelegt von

Jan Ohlenburg

geb. am 28.10.1975
Matrikel-Nr.: 210931

Eingereicht am:	20. Mai 2003
Erstgutachter	Prof. Dr. Matthias Jarke
	Lehrstuhl für Informatik V
Zweitgutachter	Prof. Wolfgang Prinz, PhD
	Lehr- und Forschungsgebiet Kooperationssysteme, Informatik V

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Hilfe Dritter erstellt habe und dass ich keine außer den erwähnten Hilfsmitteln verwendet und alle verwendeten Quellen angegeben und alle Zitate gekennzeichnet habe.

Jan Ohlenburg, Köln, den 20. Mai 2003

Abstract

A lot of scientific research has been done on the topic of real–time collision detection and a number of efficient collision detection systems exist. But only special case approaches for collision detection in distributed virtual environments have been developed. The absence of relevant papers, addressing this problem, indicates that this problem is still an open research topic.

Firstly, a collision detection system is designed and implemented. After that, the topic of collision detection in distributed virtual environments is discussed and new approaches are provided to find a trade–off between low bandwidth usage, as a result of dead reckoning techniques, and high update rates to support collision detection.

Throughout this work exhaustive experiments have been performed to support the design process and to show the efficiency of the proposed approaches.

Additionally, the implementation has been integrated into a multi–user virtual environment system to show the usability of the implementation and the new approaches.

Zusammenfassung

Obwohl sehr viel Forschung auf dem Feld der Kollisionserkennung betrieben wurde, die in einer Vielzahl von effizienten Kollisionserkennungssystemen resultierte, beschränken sich bestehende Ansätze zur Kollisionserkennung in verteilten virtuellen Umgebungen auf Spezialfälle. Der Mangel an relevanten Veröffentlichungen, die dieses Thema bearbeiten, zeigt, dass dieses Problem immer noch ein offenes Forschungsgebiet ist.

In dieser Arbeit wird zunächst ein Kollisionserkennungssystem entworfen und implementiert. Im Anschluß daran wird das Problem der Kollisionserkennung in verteilten virtuellen Umgebungen diskutiert und neue Ansätze werden präsentiert. Diese Ansätze stellen einen Kompromiss zwischen niedrigen Bandbreitenbelastungen, als Resultat von Dead Reckoning Techniken, und hohen Aktualisierungsraten zur Unterstützung der Kollisionserkennung dar.

Während der gesamten Arbeit sind ausführliche Experimente durchgeführt worden. Zum einen, um den Entwicklungsprozess zu unterstützen, zum anderen, um die Effizienz des präsentierten Ansatzes zu zeigen.

Darüber hinaus wurde die Implementierung in eine Anwendung für virtuelle Umgebungen mit mehreren Benutzern integriert, um die Benutzbarkeit der Implementierung und der neuen Ansätze zu zeigen.

Preface

This thesis (Diplomarbeit) has been written during my time as a co-student worker for the Fraunhofer Institut Angewandte Informationstechnik (FIT) in the research department Collaborative Virtual Augmented Environments (CVAE) in Sankt Augustin, Germany.

First of all, I would like to thank Prof. Dr. Matthias Jarke for giving me the opportunity to work at Fraunhofer-FIT and for being the first assessor for this thesis. I am also very grateful to Prof. Wolfgang Prinz, PhD, the second assessor of this thesis, for his helpful advice throughout my research and for giving me important feedback for my thesis.

I would like to thank Dr. Wolfgang Broll, head of the research department CVAE and my supervisor, for his many suggestions and support during the research. He had the idea for this interesting research topic and provided a lot of relevant references for my thesis. I am also thankful to all members of the research group for their support and the constructive discussions I have had with them, especially Dr. Ulrich Berntien, Iris Herbst, Irma Lindt, Eckhard Meier, Thomas Schardt and Michael Wittkämper.

Additionally, I want to thank Prof. Michael Zyda, James Klosowski and Dr. Gabriel Zachmann for their quick responses to my requests.

Of course, I am grateful to my parents for their support. Without them I would not have made it so far. Also, I would like to thank my sister, Dr. Anna Ohlenburg, for her advice.

Last but not least, I would like to thank Marion Hossfeld, Martin Gyampoh and Marianne and Alec Brown for proof reading my thesis.

Cologne, 20^{th} of May, 2003 Jan Ohlenburg

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Motivation

Although there has been a lot of scientific research on the topic of collision detection, which has resulted in a number of efficient collision detection systems, only special case approaches for collision detection in distributed virtual environments exist. The absence of relevant papers, addressing this problem, indicates that this problem is still an open research topic.

The hardest challenge for all distributed virtual environments is network latency. It prevents a distributed virtual environment from being highly dynamic and absolutely consistent at the same time [SZ99]. Additionally, the available bandwidth is still a limited resource, and approaches to reduce network traffic aggravate the effect of network latency. The approach described in this thesis takes those methods into account and presents a tradeoff between reduction of network traffic and high update rates to ensure acceptable collision detection results. It will be shown that exact collision detection in distributed virtual environments cannot be provided at interactive frame rates, due to the *Consistency–Throughput Tradeoff*.

A real–time collision detection system is always embedded in a larger application. Therefore all design choices on which the implementation is based on will be made to allow an easy integration into a larger application. Collision detection is a rather time consuming process and always competes with the application for the available processing power, therefore the approach will be embedded into an application to show its usability. An analysis based on simulations will incorporate the design process. The analysis will include an examination of methods used in the state–of–art collision detection systems and methods developed for this work. After the analysis has been done, the most appropriate and efficient methods and algorithms will be integrated into the design.

The following sections provide a short introduction of virtual environments, real–time collision detection and distributed virtual environments. The last section presents an overview of the following chapters.

1.2 Virtual Environments

"Virtual Environments are synthetic sensory experiences that communicate physical and abstract components to a human operator or participant" [Kal93]. A virtual environment consists of virtual objects, such as trees or houses, located in 3D space. In general, these objects are composed of a variable number of polygons. The participant, usually represented by an avatar, is able to navigate through the virtual world, manipulate virtual objects and — if the virtual environment is distributed[1] — interact with other participants. The virtual objects can be manipulated in terms of position and orientation as well as their visual appearance, i.e. an object's location and its geometry can change over time. There is no difference whether a participant, an animation script, a sensor, or other processes such as physical simulations perform this manipulation. Virtual environments are used in many different areas. These include scientific visualization, training and education, architectural visualization, design, computer supported cooperative work, entertainment and many more [EGJ93].

Virtual environments bring great benefits to **training and education applications**. E.g. flight simulators lower the operating costs for the training of pilots and are safe to use. They also allow the simulation of dangerous and rare situations, which are difficult to simulate in a real aircraft.

In **architectural visualization** the customer is able to examine the interior of a house before it is built, which minimizes possible misunderstandings between the architect and the customer about the result. The customer can also play around with furniture and lighting schemes. In an architectural virtual environment the user gets a feeling of the space, which 2D images and drawings are not able to provide. [EGJ93]

1.3 Real–time Collision Detection

Real–time collision detection is one of the most challenging tasks in computer graphics, visualization, simulation of physical systems, robotics, solid modeling, manufacturing, molecular modeling and many more. Additionally, it is of critical

[1] Throughout this work *distributed virtual environments* usually implies a multi–user scenario. Although all aspects apply to other distributed virtual environments also, e.g. if the movement of an object is simulated by a remote process.

importance. In most of these applications, the most important factor is speed. The usefulness of most virtual environments is closely related to efficient collision detection.

Taking a look at the flight simulator again, it is obvious that it would be useless without collision detection. The system is expected to react immediately when the plane collides with something else.

In **solid modeling applications** the designer of a model for a new car needs to verify that no parts of the car collide with other parts that should not collide. Imagine a tire rubbing against the fender every time it is steered to the left.

In general: "When solid objects collide, they do not penetrate one another (unless they flex or break into pieces). A computer simulation of the physical world will seem more natural and believable if its objects exhibit this property" [Hub95]. Thus, in the realization of 3D interactions and complex object behaviors in virtual environments the issue of detecting collisions between two or more 3D objects is very important.

Especially for interactions in virtual environments collision detection has to be performed in real–time. However, owing to the complexity of virtual scenes (concerning the number of objects as well as the objects themselves) exact collision detection is not trivial. Furthermore, collision detection is a rather time consuming process. It depends on the individual application context whether performance (real–time detection) or quality (detection resolution) is more important. E.g. applications for the construction of CAD models of cars do not depend on real–time performance but require a precise detection of the contact points and contact times to prevent interconnectivity and self–intersection.

As mentioned before, virtual objects are composed of a variable number of polygons. Two virtual objects collide with each other if they have at least one point in 3D space in common, i.e. a polygon of object O_1 intersects with a polygon of object O_2. A naive approach to collision detection is to test all polygons of one object against all polygons of all other objects in the scene. It is obvious, that this brute–force method is not applicable to provide real–time detection for complex scenes, since the complexity of this calculation is $O(n^2)$, where n is the number of polygons, and complex scenes can consist of several million polygons.

1.4 Distributed Virtual Environments

Distributed virtual environments (e.g. [MZP+94, Bro98, FS98, PHM98]) are virtual environments in which multiple users interact with each other in real–time. Each participant can "enter" the environment from a computer, which is connected to the virtual environment by a network, e.g. the internet. Even if the participants are located around the world, they should see the virtual environment as if it exists

locally [SZ99]. Applications for distributed virtual environments include computer supported cooperative work, multiplayer games and training simulations.

In distributed virtual environments each participants host has a replicated scene graph of the environment. Due to network latency temporal inconsistencies between the local replications can hardly be prevented. Therefore, additional requirements have to be fulfilled, in particular it has to be ensured that all participants have a unique view on a particular scene.

Due to the high dependency of collision detection on the actual rendering time, collision detection cannot be performed independently in a distributed environment. Usually collision detection is performed just before a frame is rendered. Thus, additional synchronization mechanisms have to be established.

The main problem is network latency. This can lead to incorrect collision detection results, due to temporal inconsistencies of the local replicated scene graphs. E.g. collisions can be determined which did not occur, since the positions of remote objects are only approximated until the next update messages arrives. A rollback is also not always possible; at the time of the rollback significant changes to the virtual scene could have been made.

E.g. in a multiplayer shooting game a collision between player A and a bullet is temporally missed owing to the network latency. Shortly after the collision should have occurred and player A would have died, he shoots at player B who dies from this bullet. Then the initial update message arrives and the collision between player A and the bullet is detected. The rollback should let player B rise from the dead and let player A die. This would make the game very unrealistic and confusing for the participant.

1.5 Overview

In Chapter 2 a global overview about the most relevant research efforts on the topic of collision detection and distributed collision detection is given.

In Chapter 3 a collision detection system is designed and implemented. All important aspects are described and experiments are carried out to support the design process. Finally, the collision detection system is integrated into a multi–user virtual environment system and experiments show the efficiency of the system.

In Chapter 4 an approach to achieve acceptable collision detection results in distributed virtual environments is presented. An overview of relevant network architectures and frameworks for distributed virtual environments is given.

In Chapter 5 the provided approach is summarized and evaluated. The chapter also gives an outlook for future work and further challenges.

Chapter 2

Previous Work

Due to the importance of collision detection in the applications presented in Chapter 1, a large number of approaches exist on the problem of collision detection. Most of the proposed methods make specific assumptions about the objects of interest and put restrictions on the objects as well as on the application [LMCG96].

To reduce the costly polygon–polygon intersection calculation many approaches use **bounding volume hierarchies**. The purpose of using bounding volumes (BV) is to approximate the object's geometry and therefore reduce the number of polygon–polygon intersection tests, since intersection tests on any two of these volumes are very fast and efficient.

Definition 1. A bounding volume $BV(V)$ for an object O with vertices V, is a volume that encloses all vertices V of O. $BV(V)$ is an outer approximation of object O. For each class of bounding volumes, e.g. boxes or spheres, a minimal bounding volume $BV_{min}(V)$ exists with minimal volume.

Bounding volume hierarchies are used to incrementally obtain more accurate approximations of the objects, until the exact geometry of the object is reached [Klo98]. These BVs usually are

- bounding spheres (e.g. the algorithm by Hubbard),
- axis-aligned bounding boxes (e.g. I-COLLIDE),
- oriented bounding boxes (e.g. RAPID),
- discrete orientation polytopes (e.g. QuickCD).

If bounding volume hierarchies are used for the collision detection process, the procedure is as follows. Starting with the roots of the hierarchies the algorithm recursively descents down the hierarchies if the BVs of two nodes intersect. E.g.

node A is tested for intersection with node B. If they do intersect, all children of A are tested for intersection with all children of B. Otherwise, it is known that the two subhierarchies are not colliding.

The performance of collision detection systems based on bounding volume hierarchies depend on the choice of BVs, the techniques used to build and update the hierarchy and the efficiency of the overlap tests between the BVs. At the core of every collision detection system is the process of efficiently reducing the number of polygon–polygon intersection tests. See Section 2.1 for a detailed discussion about the common BVs.

Another way to reduce the number of polygon–polygon intersection tests is the use of **spatial decomposition techniques**, e.g. *Octrees* [MW88], *BSP-trees* [NAT90], *brep*–indices and (regular) grids (as quoted in [Klo98]). The division of the space which is occupied by the objects has the advantage that the intersection tests only need to be performed for those pairs of objects which are in close proximity, i.e. in the same or nearby cells of the division. These decompositions can also be used in a hierarchical manner (as in Octrees, BSP–trees, etc.) and thus speed up the collision detection process [Klo98].

Since the spatial decomposition techniques can be used to separate geographically distant objects in large virtual environments, it can be easily combined with bounding volume hierarchies. In a broad phase the objects which are in the same geographical decomposition are passed to a narrow phase using bounding volume hierarchies. The focus of this work will be on bounding volume hierarchies, since they are mandatory for handling highly complex objects in close proximity.

2.1 Bounding Volumes

BVs differ in the amount of work, which has to be done to construct them, to rotate and translate them and to test two such volumes for overlap. They also differ on how tight they can fit polygonal models. See Figure 2.1 for an example of how the most common BVs approximate an object.

Klosowski [Klo98] has extended a cost function proposed by Gottschalk, Lin and Manocha [GLM96] for two large input models and hierarchies built to approximate them. Accordingly, the total cost to update the models and check the models for intersection is as follows:

$$T = N_v \times C_v + N_p \times C_p + N_u \times C_u \tag{2.1}$$

T is the total cost for the collision detection, N_v is the number of pairs of BVs tested for overlap, C_v the cost of testing a pair of BVs to overlap.

Similarly, N_p and C_p are the number of pairs of primitives (triangles or polygons) to test and the costs of this test N_u and C_u are the numbers of nodes

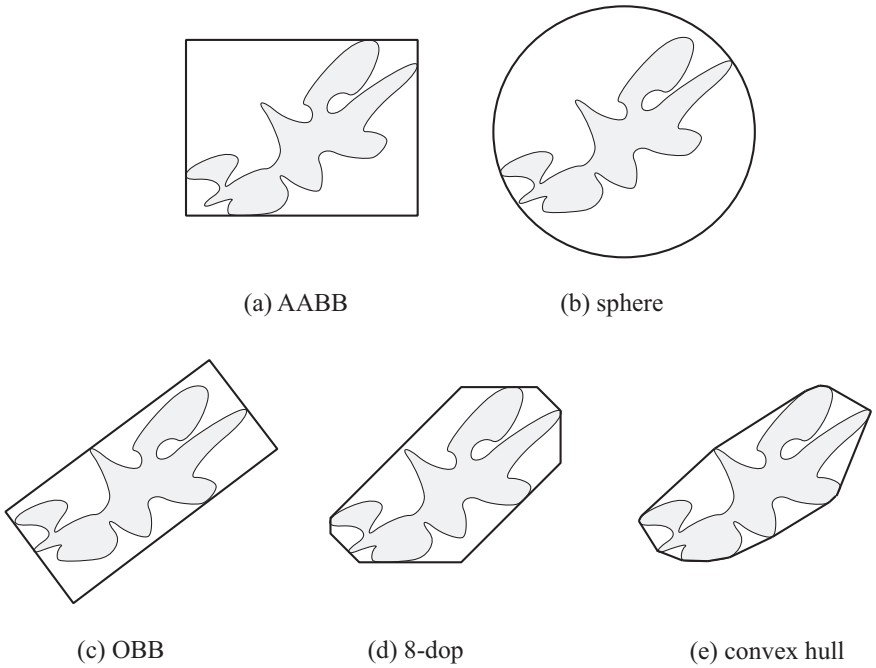


Figure 2.1: Approximations of an object by five bounding volumes (in 2D): an axis-aligned bounding box (AABB), a bounding sphere, an oriented bounding box (OBB), a k-dop (where k = 8) and the convex hull [Klo98].

which have to be updated, and the costs of a node update. The *convex hull* (Figure 2.1(e)) forms the closest approximation of an object. Therefore, the closer the BV's volume is to the volume of the *convex hull*, the fewer overlap tests will signal an intersection even if the objects do not intersect. Thus, the BV's volume for a specific object must be compared to the volume of the *convex hull* of this object. The values for N_u and C_p are constant for all BVs.

Axis–aligned bounding boxes

Axis–aligned bounding boxes are easy to compute and the costs for updates and overlap tests are low. That is why they are often used in hierarchies. However, they have the disadvantage that they approximate some objects very poorly, as one can see in Figure 2.1(a) the AABB leaves empty corners. In addition, the size of an AABB is not fixed; it depends on how the object is rotated. I.e. there are low values for C_v and C_u, but due to the poor approximation the values of N_v and N_p will be higher than those of other BVs, providing better approximations of the objects.

Bounding spheres

Bounding spheres (Figure 2.1(b)) have the advantage of very efficient overlap tests, while the update for a moving object is trivial. However, they normally approximate an object even worse than AABBs; this yields lower costs for C_v and C_u at the expense of increased costs for N_v and N_p. In addition, the costs for the creation of bounding sphere hierarchies are extremely high [Hub95].

Oriented bounding boxes

Oriented bounding boxes (Figure 2.1(c)) approximate objects much better than AABBs and spheres and the update costs are small as well, but the costs of overlap tests are roughly an order of magnitude larger than for AABBs [Klo98]. The problem when creating an OBB is to find a tight–fitting OBB. Although the algorithm from Gottschalk et al uses first and second order statistics to summarize the vertex coordinates, one problem still remains. A large number of vertices lying close to each other in the interior of the model can cause the bounding box to align with them although they should not influence the selection of the OBB. Even if the convex hull of the model is used this problem is not eliminated. See [GLM96] for a description on how to build an OBB hierarchy.

Discrete orientation polytopes

A discrete orientation polytope (dop) is "a convex polytope whose facets are determined by half spaces whose outward normals come from a small fixed set of k orientations" [Klo98]. These k-dops are a natural generalization of axis–aligned bounding boxes. Klosowski was the first – to my knowledge – to use k-dops, also known as bounding slabs, for collision detection. Kay and Kajiya [KK86] have used bounding slabs for speeding up ray tracing.

K-dops (Figure 2.1(d)) outperform AABBs and spheres with a much better approximation of objects and the costs for testing two k-dops for overlapping is an order of magnitude smaller than testing two OBBs [Klo98]. Klosowski also provided a simple approach to update the k-dop for rotating objects to keep the cost C_u low. The k-dop, like the AABB, bounds the vertices in $k/2$ directions. The k directions come from a fixed set of k outward normals, making the computation very simple by carrying out a dot product between the vertices and the k vectors. The vertices of the boundary k-dop are stored and used to realign a rotated k-dop after a transformation has taken place.

Discussion

Each of the BVs has its individual advantages and disadvantages. An application should choose carefully the BV which meets its requirements best. One should also take into account that the effort to implement such BVs differs significantly. Axis–aligned bounding boxes are by far the easiest to implement and to debug. The fact that they are commonly used should not be underestimated. If the poor tightness is not an issue, they should be used. Bounding spheres are also easy to create if they do not have to be minimal. This can be achieved by calculating an axis–aligned bounding box, using the center of the box for the center of the sphere and the longest diagonal as the diameter. In the case an application relies on tight–fitting BVs, it should choose between k-dops and oriented bounding boxes.

Refer to Chapter 3.1 for experiments with the different BVs.

2.2 Collision Detection Systems

The existing algorithms not only differ in the choice of BVs but also in the limitations they put on the input and the output, respectively.

I–COLLIDE

The collision detection system *I–COLLIDE* [CLMP95] uses spatial and temporal coherence in addition to a sweep–and–prune technique and works well for many simultaneously moving objects, but is limited to handling convex objects. Non–convex objects can be handled by partitioning the object into convex objects, e.g. the virtual model of a human hand is not convex, but if it is partitioned into the palm and the five fingers, each of these objects itself is convex[1].

The sweep–and–prune technique reduces the number of object pair intersection tests to the pairs within close proximity by sorting axis–aligned bounding boxes surrounding the objects. Since bounding boxes intersect if, and only if, the intervals of at least one dimension overlap, the bounding boxes are sorted in 3–space. For each dimension a sorted list is held with the intervals of the bounding boxes. Additionally, the sweep–and–prune technique takes advantage of the property of temporal and geometric coherence, which means that the application state does not change significantly between time steps. Thus the sorted lists are only locally unsorted after an update has taken place, and they can be resorted by *Insertion Sort* in $O(n)$.

If two bounding boxes overlap, exact collision detection is performed to test the faces of the polytopes for intersection.

[1] If the fingers are flexible, they have to be further partitioned.

RAPID

The *RAPID* interference detection system is able to perform collision detection among arbitrary polygonal models undergoing rigid motion. It is based on a hierarchy of tight–fitting oriented bounding boxes, called OBBTrees, and detects all contacts at interactive rates between large complex geometries. *RAPID* is a two body system, which means that it requires the application to execute the collision detection between a pair of objects whenever the contact status is needed [GLM96].

Gottschalk et al utilize the *separating axis theorem*, which can speed up the intersection test between two OBBs compared to other approaches using OBBs. The boxes are projected onto some axis and the resulting intervals are tested for overlap. If the intervals do not overlap, the two boxes do not intersect and this axis is called a *separating axis*. There are 15 potential separating axes to test and if no *separating axis* exists, the two OBBs intersect.

V–COLLIDE

The system *V–COLLIDE* [HLC+97], for arbitrary polygonal models undergoing rigid motion in VRML environments [VRM97], combines the sweep–and–prune technique from *I–COLLIDE* with the pair wise collision detection test from *RAPID*. The library provides exact and fast collision detection. To meet the requirements of VRML environments the API of *V–COLLIDE* was designed for the needs of VRML browsers.

QuickCD

Klosowski implemented a system called *QuickCD*, which is based on a hierarchy of k-dops [Klo98]. In his PhD thesis he has examined a wide range of different values for k and compared his system to *RAPID*. *QuickCD* is a two body system and can detect contacts of large complex geometries at interactive rates.

Hubbard's Space–Time Bounds

Hubbard proposed an algorithm using *Space–Time Bounds* and four–dimensional geometry to approximate motion. His algorithm is able to pinpoint the time of contact exactly. Based on his decision to use bounding spheres, his algorithm gains from very efficient overlap tests but suffers from a very long preprocessing step. In his paper [Hub95] he outlined three common problems of collision detection algorithms. He called the inability to adaptively change the time step between two collision detections the fixed time step weakness. The need to check every pair at every time step is the all–pairs weakness. And an algorithm suffers from

the pair–processing weakness if its pair–processing algorithm is not robust and efficient.

Discussion

The *Space–Time Bounds* proposed by Hubbard provides a good approach to eliminate the common problems of CD algorithms, but his choice of using bounding spheres is too inefficient for complex scenes.

I–COLLIDE puts a too strong restriction on the input models to be considered as a solution for general collision detection.

The two collision detection systems *RAPID* and *QuickCD* are both efficient and robust. Both address the same problem: moving a single object amongst a large number of unstructured, complex models. Klosowski [Klo98] has compared *QuickCD* with *RAPID* and has measured that *QuickCD* performs better on average than *RAPID*. In addition, they are both designed in a way that collision detection is executed before a frame is rendered [Klo98, GLM96], i.e. they have the fixed time step weakness.

2.3 Distributed Collision Detection

NPSNET

NPSNET [ZOMP93], a distributed virtual environment with the focus on human–computer interaction and software technology for implementing large–scale virtual environments, is one of the longest existing academic distributed virtual environments. It specializes in military maneuver simulation and has a limited and simple collision detection and response. The collision detection process is divided into three levels. In the first level the object's height above the terrain is compared to the maximum height of all objects within close proximity, i.e. if the object is higher than the tallest object no collision can occur, otherwise the process continues with the second level. Here the distances between the objects are computed. If the distance is smaller than the sum of the cylinder radii, then a collision has occurred. In this case the process moves onto the third level – the resolution phase – where the collision response is computed. This very simple approach, where collision detection is only performed between BVs and collision response is limited to the situation, that objects involved in the collision either stop or die [Pra93], is obviously only applicable to some applications. The NPSNET Research Group has not implemented collision detection especially for distributed virtual environments, that is to say, no special strategy has been applied to handling problems encountered in distributed collision detection, but they do use a special technique, called *Dead Reckoning* [Pra93], to reduce network traffic and to predict movement

of moving objects, owned by a participant. This leads to situations where a possible position for remote objects is calculated, and collisions can be determined between objects and dead reckoned objects which have not occurred. Since the purpose of this collision detection approach is to make the simulation more realistic without performing exact collision detection, this case is ignored, assuming the collision has been determined with the exact object positions. See Section 4.2.2 for a more detailed outline of dead reckoning.

Position History–Based Protocol

Singhal and Cheriton propose a *Position History–Based Protocol* for distributed objects to extrapolate remote object positions [SC94]. The history of remote object positions is used to predict future object positions. Additionally, it uses curve–fitting to model the behavior of the remote objects. The protocol allows hosts to generate a visually accurate animation of remote objects regardless of network latency. Although they do not address the problem of collision detection itself, the same problems are investigated. Distributed collision detection needs an accurate extrapolation of remote objects. See Section 4.2.3 for an outline of their approach.

Distributed Snooker Game

Sandoz and Sharkey [SS96] are discussing the problem of network latency and provide an example where network latency has no side effects. In this example — a distributed snooker game — only the event that the cue ball has been hit by a player is sent. Every participant performs collision detection locally and calculates the positions of the snooker balls independently.

Chapter 3

Collision Detection in Virtual Environments

In this chapter the design and implementation of the collision detection process is outlined. Additionally, the integration into a multi–user virtual environment system is explained. The analysis for the design process includes an examination of methods used in the state–of–art collision detection systems and methods developed for this work. After the analysis has been done, the most appropriate and efficient methods and algorithms will be integrated into the design. An analysis based on simulations will incorporate the selection of appropriate methods throughout the design and implementation process.

In Section 3.1 the different bounding volumes (BV) are tested for their capabilities and efficiency to support the task of collision detection. Since the collision detection will be integrated into a multi–user virtual environment system, Section 3.1 also takes a look at other aspects of real–time rendering which require efficient BVs as well. The objective of this approach is a collision detection system that can be integrated into a large application. In this application the collision detection will play an important part in providing a real–time animation of virtual environments and meeting user requirements of an interactive system. A flight simulator is useless without an efficient collision detection system, but the user is not actually interested in collision detection itself. His requirement is to train how to fly a plane in critical situations. Therefore the selection of a particular BV takes into account that other aspects of real–time rendering need BVs as well. The BVs calculated for the collision detection can be shared among the different tasks. In Section 3.2 all algorithms are outlined which are part of collision detection. In Section 3.3 the integration of this collision detection into a multi–user virtual environment system is explained.

3.1 Choice of Bounding Volume

Since Klosowski [Klo98] has already shown that k-dops are a very good compromise between the poor tightness of bounding spheres and AABBs and the relatively high costs of overlap test with OBBs, this examination will not be repeated.

In Chapter 2 it was mentioned that the size of axis–aligned bounding boxes and k-dops is not fixed. Usually, the corners of these BVs are used to realign them[1]. Also, it depends on how an object is oriented initially. A needle–like object is very well approximated by an axis–aligned bounding box if it is oriented along one of the coordinate axes, but it is very poorly approximated if it is oriented with an 45° angle. Therefore the BV's volumes are compared for a set of rotating objects, to see how well the objects are approximated by the BVs.

With these results it will be examined how other aspects of real–time rendering profit from efficient BVs, namely *View Culling* and *Occlusion Culling* as well as *Picking*, and which BV is the best choice for them.

Finally, the decision will be made which BV to choose for the implementation of this approach.

3.1.1 Tightness of Bounding Volumes

Since it is not obvious which BV fits different objects at different rotations best, this examination compares the BVs' volume for a set of objects, see Figure 3.1 and Figure 3.2. The objects are chosen to be as diverse as possible. Each object should represent a category with specific aspects. See Appendix B for images of the objects (Figures from B.1 to B.8).

The experiments calculate the axis–aligned bounding box and the 18-dop[2] for the object. Then the object is fully rotated around the x-axis, the y-axis, the z-axis followed by the rotation around x-y-axis, x-z-axis, y-z-axis and finally around the x-y-z-axis (look at the top of each diagram). From each 360° rotation the volume is calculated at steps of 0.36°, i.e. 7000 volumes are calculated with the first 1000 displaying the rotation around the x-axis, the second 1000 displaying the rotation around the y-axis and so on. Since the convex hull, the oriented bounding box and the bounding sphere have the same size for all rotations, their volume is only calculated once.

The convex hull and the bounding sphere are computed by the 3ds max^{TM} plug-in Flexporter[3]. To compute the oriented bounding box the algorithm from [Ebe01] was used. The calculation of axis–aligned bounding boxes and 18-dops has

[1]The planes of an axis–aligned bounding box are parallel to one of the coordinate axes. After a rotation they are not necessarily *axis–aligned*.

[2]Klosowski has proposed to use $k = 18$ for collision detection.

[3]Flexporter Homepage: http://www.codercorner.com/Flexporter.htm

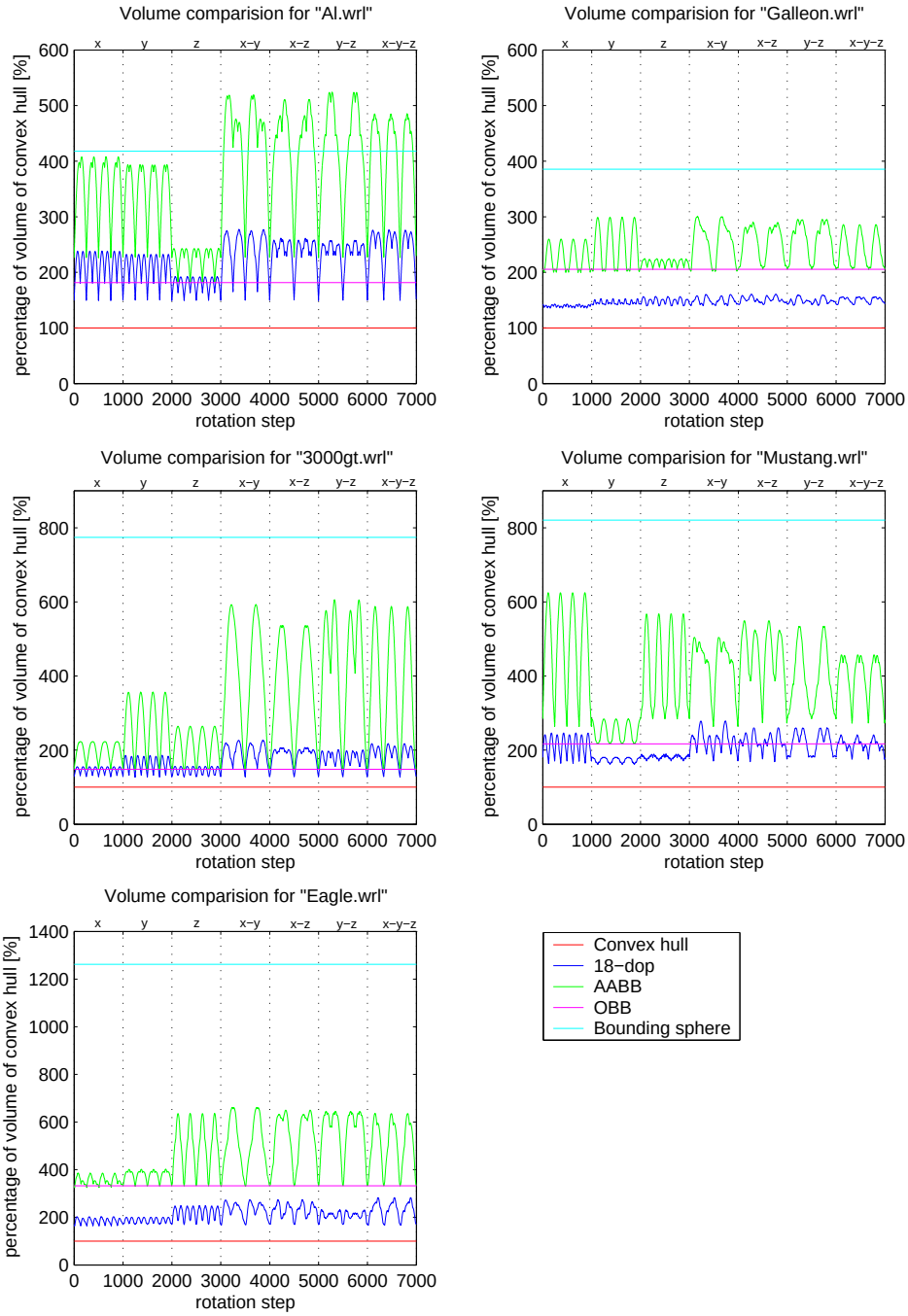


Figure 3.1: Comparison of bounding volumes

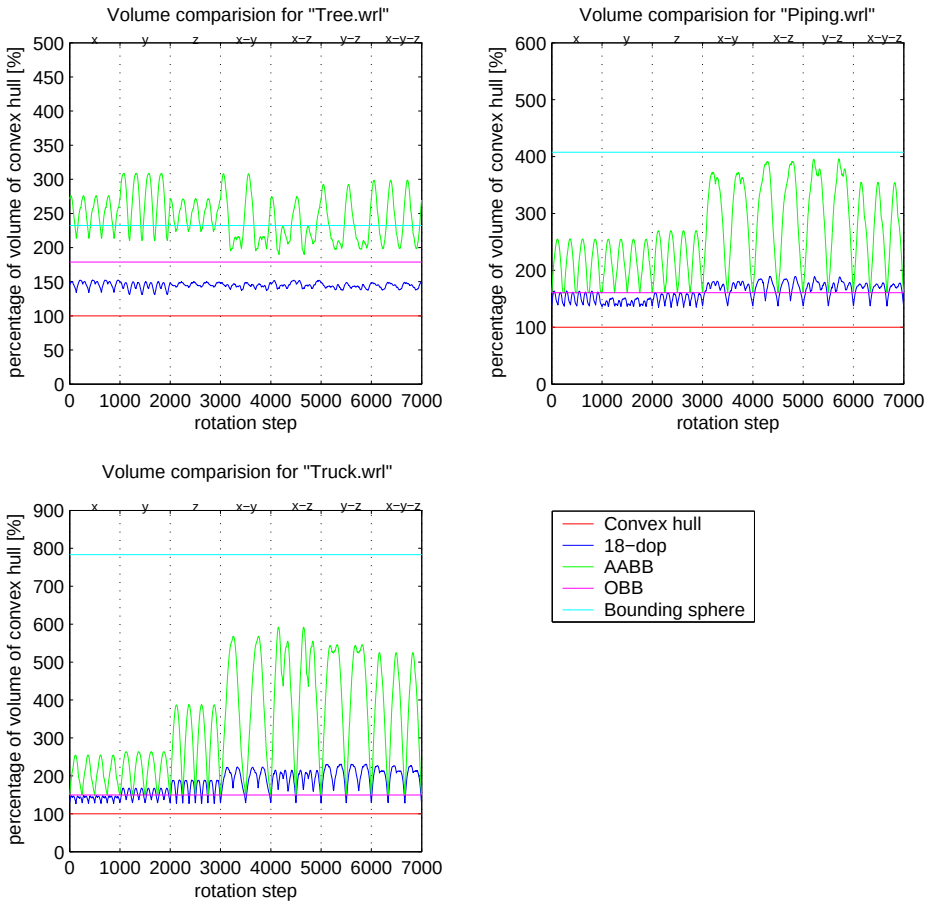


Figure 3.2: Comparison of bounding volumes *(cont.)*

been implemented for this work. An explanation of k-dop creation can be found in Section 3.2.3.

The diagrams show the volume of the BVs as a percentage of the volume of the convex hull, since the convex hull is the tightest convex approximation which encloses the object, i.e. the volume of the convex hull is 100% and the volumes of the other BVs are above than 100%. Table 3.1 lists the average percentage for each object and each BV in relation to the volume of the convex hull.

As one can see in Figure 3.1 and Figure 3.2 the axis–aligned bounding boxes and bounding spheres are a very bad choice for BVs because they do not fit the objects very tightly. The volume of the bounding spheres range from 232.31% up to over 1200% of the volume of the convex hull. Better, but still poor, are the volumes of the axis–aligned bounding boxes which range from 243.91% up to 479.68%. Except for *"Tree.wrl"* axis–aligned bounding boxes fit the objects much

Object	**Convex hull** [%]	**18-dop** [%]	**AABB** [%]	**OBB** [%]	**Bounding Sphere** [%]
Al.wrl	100.0	221.99	373.88	**181.50**	417.99
Galleon.wrl	100.0	**147.62**	243.91	205.40	385.60
3000gt.wrl	100.0	170.99	323.89	**148.26**	774.00
Mustang.wrl	100.0	**206.99**	404.12	216.83	820.80
Eagle.wrl	100.0	**213.28**	479.68	331.96	1261.97
Tree.wrl	100.0	**144.66**	244.86	178.83	232.31
Piping.wrl	100.0	160.68	261.74	**160.57**	407.59
Truck.wrl	100.0	179.23	333.71	**149.36**	783.10
Wins	–	**4**	**0**	**4**	**0**

Table 3.1: Average percentage for each object and BV in relation to the volume of the convex hull.

tighter than bounding spheres.

Of course the volume of the oriented bounding box is always smaller than or equal to the volume of the axis–aligned bounding box. In Table 3.1 the score between oriented bounding boxes and 18-dops is tied, but in taking a closer look at the wins of the oriented bounding box, the average is only a little smaller than for 18-dops. In the other examples, however, 18-dops outperform the oriented bounding boxes. This observation is supported by the means of the results, the mean for the 18-dop is 180.67% and 196.58% for oriented bounding boxes. And the cost for testing two k-dops for intersection is an order of magnitude smaller than testing two oriented bounding boxes, see Section 2.1.

3.1.2 Aspects of Real–time Rendering using Bounding Volumes

Collision detection is not the only task in the process of real–time rendering, which benefits from efficient BVs. View frustum and occlusion culling as well as picking use BVs to reject objects not contributing to the result.

View Frustum Culling

The perspective view frustum defines the part of the scene the user is looking at and has the shape of a truncated pyramid. The view frustum specifies how a virtual object is projected onto the screen. One can think of the view frustum as the view through a camera (see Figure 3.3). Its origin is the current viewpoint and the orientation is specified by the viewing direction. The width and height are

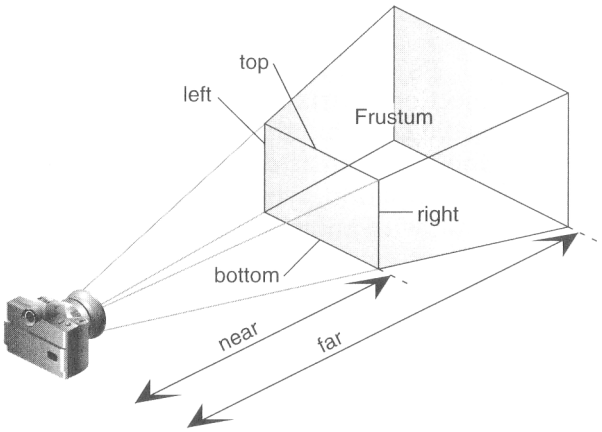


Figure 3.3: View frustum [WNDS00]

defined by the field of view. Perspective views are used to create realism, because it is similar to how our eyes work. Perspective means that the closer an object is to the viewpoint the larger it appears on the screen. It also specifies which parts of an object are drawn. If an object falls out of the view frustum, it does not contribute to the rendered image, i.e. it is culled away [WNDS00].

To discard an object, the BV of the object is tested against the view frustum. If the BV is completely outside the view frustum, it does not have to be drawn. Depending on the complexity of the object a lot of time can be saved. In a virtual world, e.g. a city consisting of a lot of buildings, vehicles and people walking around, most of the objects do not intersect the view frustum of a user. If the chosen BV does not fit an object tightly enough, it is more than likely that the intersection test will succeed, but the object will be outside the view frustum. The complexity of the calculation does also depend on the BV. See Figure 3.4 for examples of culled and unculled objects.

The algorithm for deciding whether the BV of an object intersects with the view frustum or not, is the same for convex hulls, k-dops, axis–aligned bounding boxes and oriented bounding boxes.

Definition 2. A bounding volume is outside the view frustum if all vertices are outside of at least one half–space defined by a plane of the view frustum.

The algorithm is straightforward. For all planes of the view frustum, test if all vertices are on the negative side of the plane. If this test is successful for at least one plane, the object is culled. For axis–aligned bounding boxes and oriented bounding boxes, this test needs eight dot products for each of the six planes of the view frustum. The algorithm stops earlier if one plane is found where all vertices

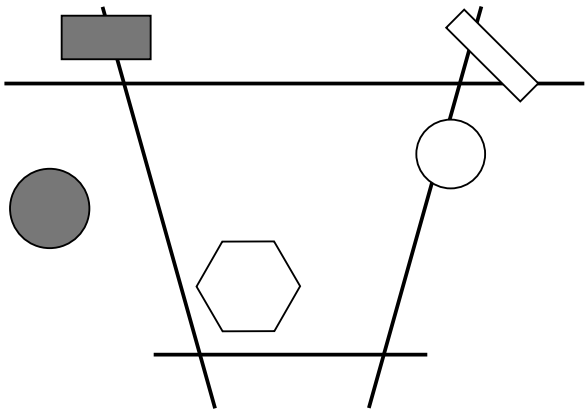


Figure 3.4: Example of culled and unculled objects. The grey objects are culled and the white objects are drawn.

are on the negative side. In the case of a convex hull or a k-dop the number of vertices increases [Ebe01].

The algorithm for bounding spheres is even simpler. If the center of the sphere is on the negative side of a plane and the distance between the center and the plane of the view frustum is greater than the radius, the bounding sphere is outside the view frustum.

The test for oriented bounding boxes can be sped up by projecting the box and the plane onto the line $\overrightarrow{C} + s\overrightarrow{N}$. Where $\overrightarrow{N}$ is the normal vector of the plane and $\overrightarrow{C}$ the center of the oriented bounding box [Ebe01]. This test needs only 15 multiplications and 11 additions, whereas the other algorithm needs 36 multiplications and 40 additions.

The oriented bounding box, see Figure 3.4, in the upper right hand corner is outside the view frustum, but is not culled. This is because the box intersects two planes and none of the planes has the property of having all vertices on the negative side, which means all six plane–box culling tests fail.

At first glance it seems that bounding spheres have won the test, because the test only needs one dot product and one comparison per plane of the view frustum, but the opposite is true. As mentioned before, the volume of the bounding sphere can be more than a thousand percent of the volume of the convex hull; therefore, even if the bounding sphere intersects with the view frustum, their is a high probability of the object not intersecting the view frustum. The oriented bounding box test is faster than the k-dop test, but because k-dops fit the objects better than oriented bounding boxes the extra calculation steps are worth the effort, especially if the objects are very complex. If it is known that the object is not complex and that little time is required to draw the object, the complexity

of the test can be reduced by converting the k-dop into an axis-aligned bounding box and test only eight vertices against each of the six planes.

Occlusion culling

View frustum culling is easy to implement and performs well, but the intersection of the object with the view frustum does not tell whether an object is visible. Objects can be occluded by other objects, e.g. if the engine bonnet of a car is closed, there is no need to render the engine because it is completely occluded by the bonnet.

Some graphic cards implement occlusion culling queries, like Hewlett-Packard and NVIDIA. Hewlett–Packard's VISUALIZE fx is able to return a boolean value whether or not a set of polygons will change the Z–buffer. It is easy to use, the BV of a complex object is sent to the hardware and the query returns if the Z–buffer would have changed. If not, the object does not need to be drawn otherwise the object is sent into the rendering pipeline. The latency of this query is normally very long, so this technique should only be used for objects consisting of a large number of polygons. There is no need to use this technique if drawing the object is faster than the query itself [AMH02]. In [BKS01] the authors have tested a variety of "real world" MCAD datasets. They achieved a 50 % speed up using k-dops instead of axis–aligned bounding boxes since k-dops fit objects much tighter and therefore reduce false positive visibility queries. In our example the edges of the axis–aligned bounding box of the engine would be outside the hood and therefore the visibility query would return a positive answer.

NVIDIA uses a different extension to allow occlusion queries. Instead of returning a boolean value they return the number of not occluded pixels, n. Therefore, if $n = 0$ all pixels are occluded and the object is culled. This query is also very useful to determine the *level of detail* (LOD) of an object. The LOD specifies which detail level should be used for displaying an object. Normally, the LOD is linked to an object's distance to the viewpoint, if it is far away it does not need to be drawn in full detail [AMH02]. With the return value n the LOD can be decided by how much an object contributes to the displayed frame.

Picking by ray–tracing

Another aspect of real–time rendering is picking. Picking is used to calculate which objects are at a specific coordinate on the screen, normally below the mouse pointer but also which object is looked at. Usually, the picking process provided by graphic libraries, such as OpenGL, is used and works as follows.

The scene is rendered into a very small projection plane — from 1x1 to 5x5 pixels — around the pick origin. Before an object is rendered its identification

number is pushed onto the select buffer and popped after rendering it. This information is used when rendering the objects by writing the identification number into the depth buffer. After that it is possible to query which objects are actually drawn into the frame buffer and read the distance to the pick origin out of the depth buffer. The object with the smallest distance to the pick origin is the picked object. View frustum culling can speed up this rendering process too.

Since a hierarchy for all objects in the scene has been built (see Chapter 3.2.4), picking can be implemented by ray–tracing. A picking ray is cast from the pick origin into the pick direction. If it intersects the BV of an object, its hierarchy is traversed until the picking ray intersects with the BV of a leaf of the hierarchy. Then the ray is tested for intersection with the polygons of the leaf, for each passed test, the polygon is pushed into the result buffer. Afterwards the minimal distance from the pick origin to the polygons in the result buffer is determined. The owner of this polygon is the picked object. The pseudo code (Listing 3.1) for this algorithm is straightforward.

```
for all objects {
    if (ray intersects bv)
        if (!leaf)
            intersect(left, ray);
            intersect(right, ray);
        else
            if (intersect(polygons, ray))
                pushed object into buffer
}
pickedObject = NULL;
pickedDistance = MAX;
for all polygons in buffer {
    if (distance to polygon < pickedDistance)
    {
        pickedObject = object of polygon;
        pickedDistance = distance to polygon;
    }
}
```

Listing 3.1: Picking by ray tracing

It is clear that this algorithm benefits from tight–fitting BVs, since the objective is to minimize ray-polygon intersection tests, in exactly the same way as with collision detection, i.e. oriented bounding boxes and k-dops are much more efficient than bounding spheres and axis–aligned bounding boxes.

The benefit of using ray–tracing for picking is that complex objects do not need to be pushed into the frame buffer. Drawing an object is quite time consuming, even if only a small portion of the object is visible the complete model has to

be processed. By utilizing the above approach, only those parts of an object are considered which are in close proximity of the ray.

3.1.3 Conclusions

The comparison of tightness of the BVs and the different aspects of real–time rendering yield k-dops are the best choice as BVs for collision detection and other aspects of real–time rendering. Even if the complexity to test for intersection between view frustum and k-dop and between a ray and a k-dop is higher than for oriented bounding boxes, the complexity can always be reduced by converting the k-dop to an axis-aligned bounding box in constant time. An application should choose if an object is complex enough to accept longer processing times in order to achieve that these objects are culled.

Furthermore, these experiments have demonstrated that an application can process other aspects faster if the hierarchies built for collision detection are shared within the application, i.e. that they are not hidden within a collision detection module. This section has also shown that the design of the collision detection process should not be exclusively focused on the problem of efficient collision detection. The immense required memory and much longer preprocessing times can only be justified if the application also can benefit from the calculations.

Klosowski [Klo98] did exhaustive experiments on which value for k performs best on average. He proposes to use $k = 18$. Although the following implementation implicitly uses 18-dops, all outlined algorithms apply to other values for k as well.

3.2 Implementation of the Collision Detection Process

In this section all aspects for the implementation of the collision detection are described. Aspects discussing the implementation details for collision detection in distributed virtual environments are discussed in Chapter 4.

3.2.1 Requirements

First of all, the general requirements of the collision detection are outlined.

- No restrictions should be applied to the input. I.e. the geometry of objects usually comes from CAD systems and 3D modeling tools, such as 3ds max, and it can contain cracks and gaps, it can be self–intersecting and polygons

can be duplicated and degenerated. Hence this input data is called *polygon soup*.

- Polygon data should not be limited to triangles. Even if triangles can be tested for intersection faster than polygons, triangulation of polygons is time consuming and efficient algorithms for testing polygons exist and can be faster when operating on polygons instead of a larger number of triangles.
- The collision detection should be able to handle deformable objects, i.e. for objects that change their geometry over time.
- The collision detection should be able to pinpoint the exact time of contact, i.e. that no penetration can occur.
- The collision detection should be fast enough even for very complex objects to ensure constant high frame rates.
- The collision detection should be exact.

Of course not all of these requirements can be met at the same time by today's computers and known algorithms. In particular deformable objects cannot use preprocessing time to build hierarchies to speed up the collision detection process. This requirement collides with the requirement for constant high frame rates for complex objects. Since the process of building a hierarchy of tight–fitting BVs takes far more than a second, a constant frame rate cannot be provided, because after a geometry has changed, its hierarchy is out of date and has to be rebuilt. Algorithms for deformable objects exist, but they cannot handle large objects (number of polygons > 100.000) in real–time [Zac00]. E.g. character animation normally transforms each polygon of a character each frame for a smooth animation.

Furthermore, it is very complicated to pinpoint the exact time of contact; Hubbard [Hub95] has shown how to use 4D geometries to achieve this. The trade–off between dynamic and static collision detection is to choose between time exact collision results and high frame rates for the collision detection. For most applications it is sufficient to have the time of collision within a certain interval, i.e. if static collision detection is called more than 50 Hz, the biggest error will be smaller than 20 ms.

The proposed implementation meets the following requirements:

- No restriction on the input data.
- Fast collision detection queries.
- Exact collision detection results.

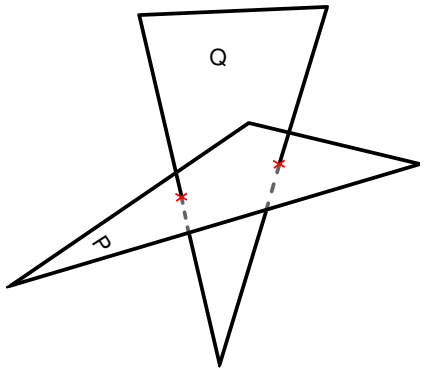


Figure 3.5: Example of polygon-polygon intersection, the crosses mark the intersections of the edges of polygon Q with polygon P.

I.e. it provides constant frame rates but cannot handle deformable objects and the collision detection is static (timeless). Flexible objects are not considered, since this would limit the complexity of virtual scenes, without contributing to the result of distributed collision detection. In order to pinpoint the exact time of contact, the movement of an object is usually mapped by a closed function. Since this is not possible for highly random movement, e.g. a squirrel running around, this aspect will not be considered in this work.

3.2.2 Polygon–Polygon Intersection

This intersection test is performed by two algorithms, the first algorithm calculates the point of intersection for each edge of polygon P with the plane of polygon Q, and these are n edges if P has n vertices. Then the next algorithm tests if one of the points is inside polygon Q.

Edge–plane intersection

To check whether an edge with vertices u and v of polygon P intersects with the plane of polygon Q, we first check if u and v lie on different sides of the plane, then the intersection of the ray from u in direction $v - u$ with the plane is calculated. The intersection point is then passed to the algorithm to test if it is inside polygon Q (see *Point in polygon strategies*).

It is not sufficient just to check polygon P against polygon Q, polygon Q has to be checked against polygon P as well. See Figure 3.5 for an example, here no edge of polygon P intersects polygon Q.

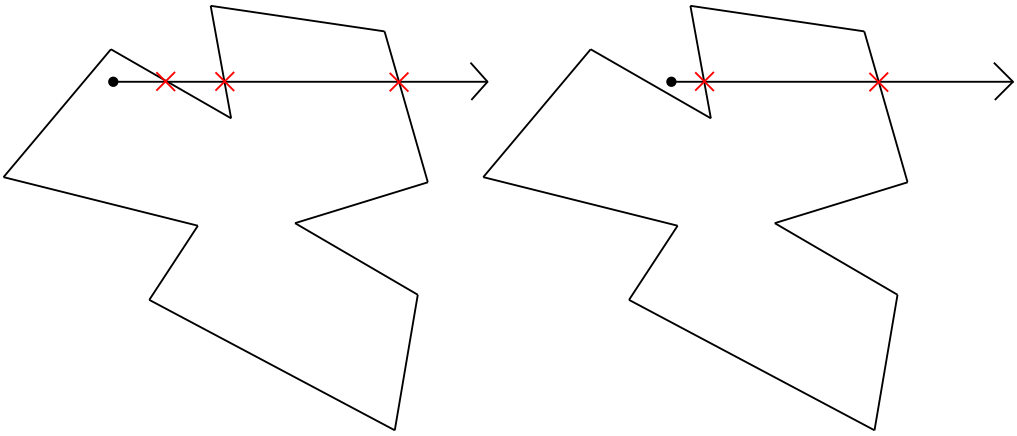


Figure 3.6: Example of crossings test. On the left the point is inside the polygon because the number of crossings is odd. On the right the point is outside.

Point in polygon strategies

Eric Haines [Hai94] has compared different *point in polygon* strategies. The worst of all strategies is the *Angle Summation Test*; Haines calls it "the worst algorithm in world". This test is implemented straightforward: Sum the signed angles from each vertex to its neighboring vertex. If the sum is near zero, the point is within the polygon, otherwise the sum is near 360 degrees. The algorithm performs so badly since it uses the arccosine to calculate the result.

A better strategy — and the one I have chosen — is the *Crossings* strategy, see Figure 3.6 for an example. This strategy is very simple and very efficient. A ray is shot from the queried position into an arbitrary direction — normally along the x-axis — then the number of crossings is computed. If the result is odd, the point is inside the polygon, otherwise the point is outside. To check for a crossing, project the point and the polygon onto a plane which is most parallel to the polygon. Consider the given point to be at the origin and check the y-components of the vertices of the polygon against this point. If the y-components of a polygon edge differ in signs, the edge is a candidate for a crossing. In this case the ray *crosses* the edge if both x-components are positive or if the intersection of the ray and the edge is positive.

3.2.3 Building the k-dop

In order to determine the k half–spaces of the k-dop, an iteration loops over all vertices of an object to find the minimum and the maximum for each $k/2$ direction.

```
dop[0..8]  = DBL_MAX;
dop[9..17] = -DBL_MAX;
for all vertices v {
  dop[0] = MIN(dop[0], v_x);     dop[9]  = MAX(dop[9],  v_x);
  dop[1] = MIN(dop[1], v_y);     dop[10] = MAX(dop[10], v_y);
  dop[2] = MIN(dop[2], v_z);     dop[11] = MAX(dop[11], v_z);
  dop[3] = MIN(dop[3], v_x+v_y); dop[12] = MAX(dop[12], v_x+v_y);
  dop[4] = MIN(dop[4], v_x+v_z); dop[13] = MAX(dop[13], v_x+v_z);
  dop[5] = MIN(dop[5], v_y+v_z); dop[14] = MAX(dop[14], v_y+v_z);
  dop[6] = MIN(dop[6], v_x-v_y); dop[15] = MAX(dop[15], v_x-v_y);
  dop[7] = MIN(dop[7], v_x-v_z); dop[16] = MAX(dop[16], v_x-v_z);
  dop[8] = MIN(dop[8], v_y-v_z); dop[17] = MAX(dop[17], v_y-v_z);
}
```

Listing 3.2: Building a 18-dop for a set of vertices

Clearly, this algorithm has $O(kn)$ time complexity and $O(k)$ space complexity if the object P has n vertices. The pseudo code for a 18-dop (Listing 3.2), assuming directions are $(1,0,0)$, $(0,1,0)$, $(0,0,1)$, $(1,1,0)$, $(1,0,1)$, $(0,1,1)$, $(1,-1,0)$, $(1,0,-1)$ and $(0,1,-1)$, is straightforward.

If the convex hull of the object is known, the time complexity can be reduced to $O(kh)$, where $h = \mid convhull(P) \mid$. Obviously $h \leq n$ is always true.

To tumble the k-dop the boundaries have to be calculated during the preprocessing step.

Klosowski [Klo98] uses two different methods to update a k-dop for a tumbled object. The first method computes the convex hull of an object and uses the transformed vertices of the convex hull to compute the new k-dop by a "hill climbing" algorithm to take advantage of the step-to-step coherence. This new k-dop always has minimal volume. Alternatively, this could also be done with the algorithm presented above as well.

The second (*approximation*) method calculates the vertices, $V(P_v)$, where P_v are the vertices of P's k-dop and uses the transformed vertices $V(P_v)$ to compute the new k-dop. Of course, only the boundaries of the initial k-dop are transformed to calculate the k-dop for the tumbled object, otherwise the volume of the calculated k-dops would increase with every transformation. The boundaries of the k-dop are computed by intersecting k half–spaces.

Klosowski uses the first method only for the root node of the hierarchy and the second method for all other nodes of the hierarchy. Since otherwise the convex hull would have to be calculated for each node of the hierarchy and would have to be stored at each node of the hierarchy, the approximation method saves preprocessing time and memory (see Section 3.2.4).

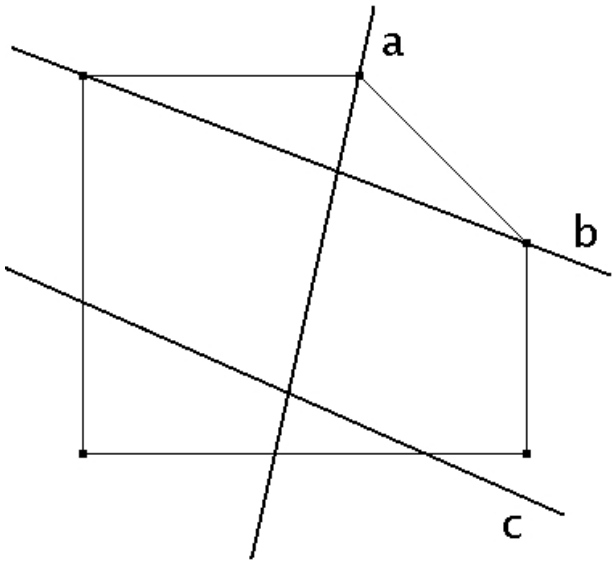


Figure 3.7: Intersection of a polygon by a plane. Planes intersect the polygon whether through one vertex (a), two vertices (b) or no vertex (c).

For this work an alternative approach has been developed. It computes the intersections plane by plane. Starting off with a rectangle for all k half–spaces, then intersecting the edges of each rectangle with the $k-2$ non–parallel half–spaces.

The initial rectangle for each half–space can be determined by using the four perpendicular half–spaces. The corners of the rectangle for the half–space with normal $(1, 0, 0)$ is determined by the half–spaces with normals $\pm(0, 1, 0)$ and $\pm(0, 0, 1)$. Each of the eighteen rectangles forms an outer approximation of the resulting polygon for its half–space. The polygons are then determined by consecutively intersecting the polygon with each non–parallel half–space.

The intersection of a polygon by a plane representing a half–space has five possible scenarios:

- If all vertices of the polygon are on the negative side of the plane delete the polygon since it is not part of the boundary.
- If all vertices are on the positive side of the plane the polygon stays as it is.
- The plane intersects the polygon through vertex a and the edge from vertex u to v (Figure 3.7 (a)). Calculate intersection of the plane through the edge, vertex b, and insert the vertex between u and v. Delete all vertices between a and b which are on the negative side of the plane.

- The plane intersects the polygon through vertices a and b (Figure 3.7 (b)). Delete all vertices between a and b which are on the negative side of the plane.

- The plane intersects the polygon through the edges e_0, from vertex u_0 to v_0, and e_1, from vertex u_1 to v_1 (Figure 3.7 (c)). The two intersections a_0 and a_1 are inserted between u_0 and v_0 and between u_1 and v_1, respectively. Delete all vertices between a_0 and a_1 which are on the negative side of the plane.

Even though this algorithm is slightly slower, the resulting polygons of each k "sides" of a k-dop can be further used to build the hierarchy (see Section 3.2.4).

3.2.4 Building the Bounding Volume Hierarchy

Since the goal of most collision detection systems — including this one — is to reduce the number of polygon–polygon intersection tests, the creation of a bounding volume hierarchy (BVH) is mandatory. See Figure B.9 for images of the hierarchy for Mustang.wrl at different levels.

Definition 3. A bounding volume hierarchy of a geometric object O, BVH(O), is a binary tree. Each node v of BVH(O) corresponds to a subset, $O_v \subseteq O$, with the root corresponding to the full set O. Each internal node of the hierarchy has two children, with $|O_v| \geq 2$. Also each node has a bounding volume, $b(O_v)$, which is an outer approximation of O_v.

A BVH can be created bottom–up or top–down. A bottom–up approach starts with each single polygons of O as a leaf for the tree. Then pairs of nodes are recursively joined together until a single node, the root, remains. In a top–down approach starting with the root node, splitting rules are recursively applied to divide the nodes until no divisions can be performed. The algorithm presented below uses a top–down approach, while comparing different splitting methods.

The creation of a BVH is not trivial and as always a compromise has to be found between fast preprocessing times, memory required and collision detection times (see Section 3.3.2).

Split axis

Klosowski [Klo98] has compared four different methods to find a split axis. The methods to find a split axis, orthogonal to the x, y or z-axis, from fastest to slowest, are:

- **Longest Side**
  Split axis is the axis orthogonal to the longest side of the k-dop.

- **Splatter**
  Split axis is the axis with the largest variance of the projected centroids onto the three coordinate axes.

- **Min Sum**
  Split axis is the axis which minimizes the sum of the volumes of the two children.

- **Min Max**
  Split axis is the axis which minimizes the largest of the volumes of the two children.

Klosowski has decided to use the *Splatter* method to choose the split axis, since it is an order of magnitude faster than the *Min Sum* method, with only slightly worse collision detection times. Klosowski determines the split point by projecting the vertices of the object onto the split axis and calculates their mean, μ. The centroids, c, of the polygons are then compared with μ. If the object has n vertices, v_i, and the split axis is $axis$, then μ can be expressed as:

$$\mu = \frac{1}{n} \sum_{i=0}^{n-1} v_{i_{axis}} \tag{3.1}$$

If polygon P has m vertices, then the centroid c of P is the mean of all m vertices of P.

Zachmann [Zac00] has compared 14 different criterions to choose the split axis by comparing the average collision detection times for each criterion, but he has not taken the preprocessing time into account. The criterion which performed the best was:

- Choose the split axis, which minimizes the overlap of the BVs of the resulting two children.

RAPID [GLM96] uses the straightforward method *Longest Side*, to subdivide the hierarchy. If the longest side cannot be subdivided, the second longest is used, and so on down to the shortest. To obtain a balanced tree they also chose the centroid of a triangle to assign this triangle to one of the two children.

All these methods produce hierarchies of at least $O(\lg n)$ height, in the best case in which the hierarchy is balanced. However, it is extremely difficult to evaluate the average case behavior because it largely depends on the objects chosen and how the polygons are distributed.

Splitting Rules

This algorithm assumes that a single geometric object consists of a list of vertices and a list of vertex indices. Each vertex index points to a vertex. A polygon is described by a number of indices (at least three) followed by a -1 index, marking the end of a polygon. The geometric object holds these two lists. From these two lists, a list of faces (polygons) is created, with each face having a list of pointers to the vertices belonging to that face. Each node of the hierarchy gets a pointer to the list of faces. After creating the root of the hierarchy, the k-dop is built and for each face the centroid is calculated.

Object	**Height of BVH**			
	Optimal	*Long. Side*	*Splatter*	*Min Sum*
Al.wrl	13	17	18	17
Galleon.wrl	13	19	18	17
3000gt.wrl	14	22	20	20
Mustang.wrl	12	16	17	16
Eagle.wrl	15	20	19	22
Tree.wrl	16	22	22	22
Piping.wrl	16	21	21	21
Truck.wrl	16	99	34	56

Table 3.2: Overview of BVH's height of the splitting rules.

Splitting the root node is done recursively. The rule to choose the split axis, *Longest Side*, *Splatter* or *Min Sum*, can be chosen by the application. Klosowski [Klo98] assigns the indices of the triangles to the new children; this approach sorts the list of faces, by using the *select* algorithm instead [Sed90]. The *select* algorithm is basically *QuickSort* without recursion. It rearranges the a list $a_1, ..., a_n$ and returns a value i, so that $a_1, ..., a_{i-1}$ is smaller or equal and $a_{i+1}, ..., a_n$ is greater or equal than a_i. By using the mean, μ, the algorithm is able to rearrange the list of faces in $O(n)$ and return the split index i. Since this is done for each node, at the end each node and each leaf of the hierarchy just need a pointer to the list and two integers for the assigned interval. Therefore, the required memory can be reduced.

Discussion

Calculating the split axis takes $O(1)$ for *Longest Side* and $O(n)$ for *Splatter*. The time complexity for *Min Sum* is $O(n \lg n)$, it sorts the list for each axis and calculates the volume for the potential children. To rearrange and assign the polygons

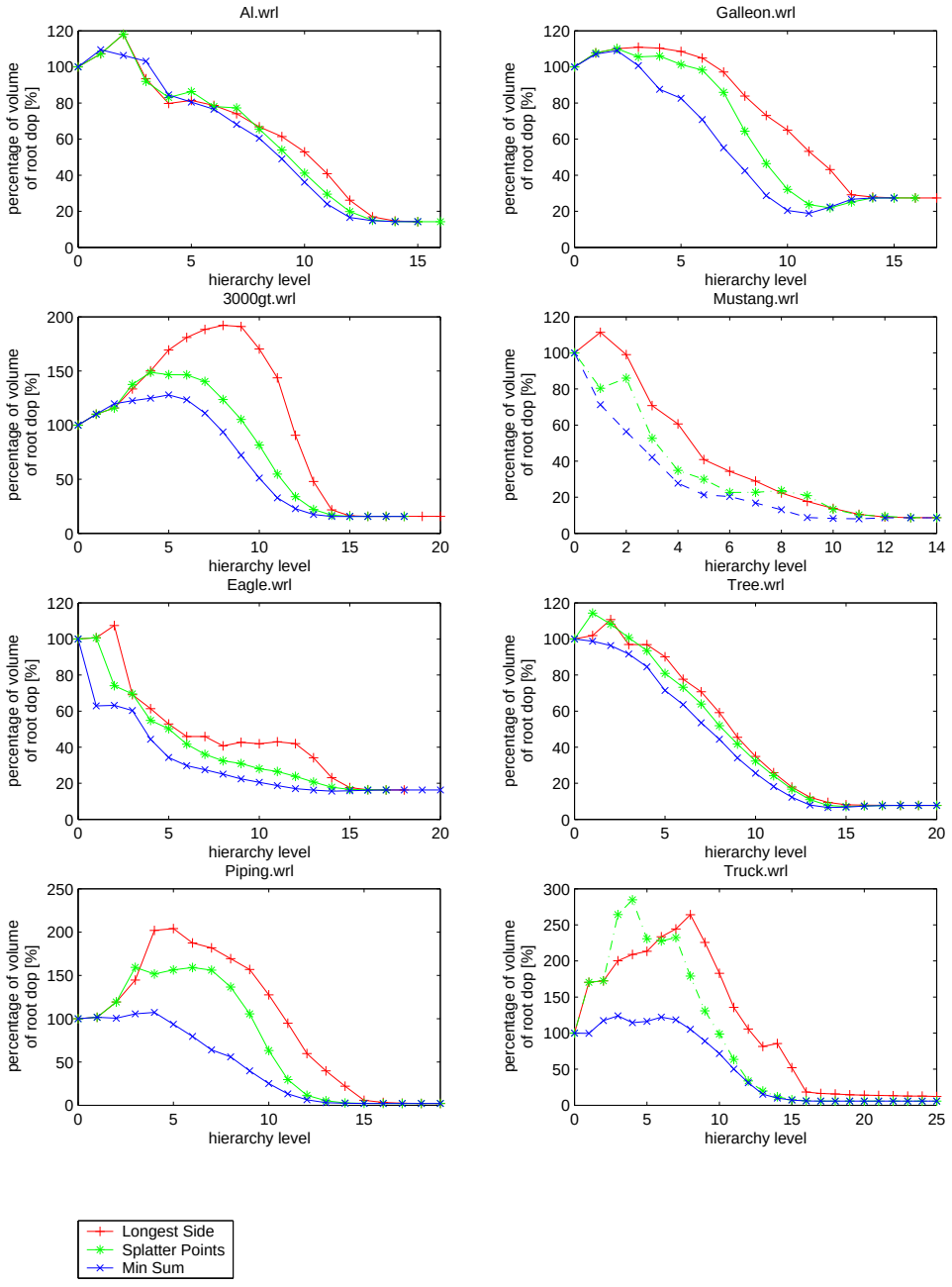


Figure 3.8: Volume comparison of splitting rules

Object	**Time to build hierarchy in seconds**		
	Longest Side	*Splatter*	*Min Sum*
Al.wrl	9.11	8.89	35.70
Galleon.wrl	4.72	4.72	16.64
3000gt.wrl	30.83	31.61	125.97
Mustang.wrl	2.22	2.27	7.49
Eagle.wrl	44.84	37.59	157.52
Tree.wrl	240.48	302.09	1141.98
Piping.wrl	212.00	197.77	819.52
Truck.wrl	270.42	285.91	946.31

Table 3.3: Comparison of preprocessing times for the different splitting rules.

to the children is done in $O(n)$ in the average case. If the resulting hierarchy has $O(\lg n)$ height (optimal case), the overall time is $O(n \lg n)$ for *Longest Side* and *Splatter* and $O(n \lg^2 n)$ for *Min Sum*. For the height of the hierarchies for the different splitting rules, see Table 3.2. The optimal hierarchy height is calculated by $h_{opt} = \lceil \lg n \rceil$, if n is the number of polygons of the object.

The table does not show great differences between the different methods and no splitting rule is the clear winner. But when comparing the resulting volumes of the methods (see Figure 3.8[4]) it shows clearly that *Splatter* converges faster than *Longest Side* in most cases. *Min Sum* converges even faster than both other methods, but the preprocessing time is more than three times longer for building the hierarchy, see Table 3.3[5].

Other splitting rules have been developed for this work, but the results have been unsatisfactory. The two most promising approaches were *Combined Longest Side* and *Combined Splatter*, which are a combination of *Longest Side* and *Min Sum* and *Splatter* and *Min Sum*, respectively. The idea behind them is to use *Min Sum* for the first levels and the other method for deeper levels, since intersections of hierarchy nodes are more likely for higher levels and therefore can benefit from earlier rejection. Of course the preprocessing times were much shorter than for *Min Sum*, but neither the volume comparison nor the collision detection times have shown a significant effect. Additionally, the BVH can be stored in a file to reuse it during subsequent session. Therefore, the preprocessing time for the different methods becomes the least important factor, since the time needed for

[4]The diagram of *Truck.wrl* has been cut of at $x = 25$, since the levels above 25 do not show much differences.

[5]All timings have been made on a computer with two Intel Xeon processors with 2 GHz clock rate, 1 GB RAM and a 3Dlabs Wildcat II 5110 graphics card.

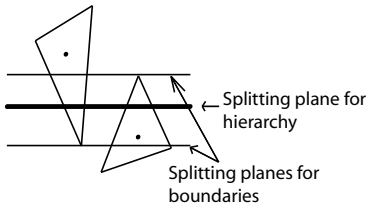


Figure 3.9: Splitting planes for boundary calculation.

loading a stored BVH is shorter the lower the hierarchies is. Furthermore, the collision detection times of *Min Sum* are significantly smaller (see Section 3.3.2). This implies using of a pre–stored *Min Sum* hierarchy for frequently used objects.

The effect that the volume of the nodes at level i is smaller than the volume at level $i+1$, means that the k-dops of the resulting children are heavily overlapping. This effect is easily spotted in the graph for *Piping.wrl* in Figure 3.8, where the volume increases up to level 4. Of course, this effect is not wanted because smaller BVs have a better chance of being rejected, but it cannot be avoided.

From the fact that the *Splatter* method builds some hierarchies faster than *Longest Side*, it can be concluded that the number of nodes in the hierarchy built by *Splatter* is smaller.

If the boundaries of the k-dop are calculated by cutting the faces of the boundary with planes (see Section 3.2.3), some extra work has to be done to use the boundaries for the hierarchy. The split axis together with the mean μ are not sufficient as a splitting plane for the boundary. To get the boundaries for the children, two splitting planes are required. Because the centroid of a polygon is used to decide to which sub tree it is assigned, a vertex of this polygon can be on the "other" side of the splitting plane (see Figure 3.9).

To get these two splitting planes, the minimum and the maximum of the vertices along the split axis have to be calculated. In Figure 3.9 the split axis is the y-axis, the bold line in the middle is the splitting plane for the hierarchy. The left triangle is assigned to the top and the right triangle to the bottom, the topmost vertex of the right triangle is above the splitting plane so the splitting plane for the hierarchy has to be at that vertex. With this extra information, the boundary of the parent node can be used to calculate the boundaries for the children. Instead of using 18 planes for an 18-dop this approach only needs one plane for each child.

The creation of the BVH can be a very time–consuming process, but even worse, the memory required for the hierarchy is immense for large geometric objects. A way to limit the creation time and the size of the hierarchy is to use a threshold. Up to now, a threshold of 1 has been implicitly used for the comparison of the different splitting methods. A threshold larger than 1 prevents a node of the hierarchy from

being split if the number of associated polygons is smaller than or equal to this threshold. The memory required can easily be reduced to a quarter or less of the original size by accepting slightly worse collision detection times. Of course, the time to create the BVH is closely related to the threshold as well and can be reduced also. For a comparison of threshold values, hierarchy sizes and collision detection times, see Section 3.3.2. Another way to speed up the preprocessing time is to store the hierarchy after it has been calculated and to reuse it during sequent sessions.

Hierarchies for Scene Graphs

Up to now only single geometric objects have been discussed, but complex virtual environments consist of a hierarchy of objects, referred to as *scene graph*. Geometric objects are aggregated by grouping nodes. A grouping node can have an arbitrary number of child nodes, which can also be grouping nodes. A virtual model of a car can be a hierarchy of nodes, the car being the root, with one of its children being the front left tire. The tire itself can be a hierarchy as well with geometric objects for the rim, the wheel and for each screw.

Definition 4. A bounding volume hierarchy of a grouping node G, BVH(G), is a tree. Each internal leaf v of BVH(G) corresponds to exactly one descendant of G. The leafs of BVH(G) correspond to a geometric object O with BVH(O).

Contrary to the creation of the BVH for geometric objects, the creation of a BVH for grouping nodes is done in a bottom–up manner. The k-dops associated with the root of the BVH of the geometric object are converted into an axis–aligned bounding box. The axis–aligned bounding box of the parent is the joint over all axis–aligned bounding boxes of its children. This is done recursively until the root grouping node is reached.

3.2.5 Object Transformations

For each geometric object in the scene graph a BVH is stored. Each node of the BVH stores a pointer to the sorted list of polygons. In addition, the node stores the interval for the list of polygons and the vertices of the boundaries.

Every dynamic geometric object in the scene graph will be transformed during the simulation. Since the k-dop is *axis–aligned*, it has to be realigned after each transformation. This recalculation has to be done in much less than a millisecond since a complex scene can consist of hundreds of dynamic objects transformed every frame. It is too costly to use the vertices of the object for the realignment, instead only the boundary's vertices of the k-dop are used. This approximation

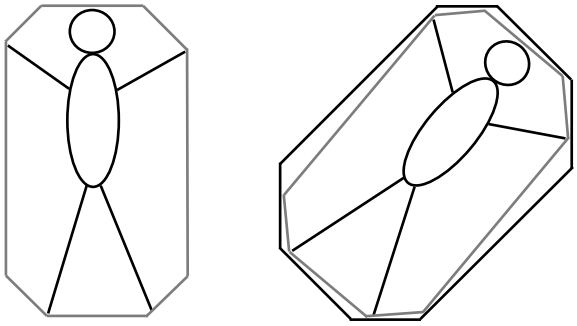


Figure 3.10: Approximation of a transformed object by an enclosing 8-dop (in 2D). On the left is the initial object and initial 8-dop, on the right the object is rotated, but only the rotated vertices of the 8-dop are used for the new enclosing 8-dop.

saves a lot of time and reduces the number of matrix–vector multiplications to a minimum (see Figure 3.10).

Usually, before a frame is rendered by an application, all objects are updated. During this update process the relative and absolute object transformation matrices (i.e. in object coordinates and in world coordinates, respectively) are calculated and the BVs are updated. The BVs in the scene graph are needed for the view frustum culling and the collision detection. The transformed k-dop, in world coordinates, of an object is calculated and then converted to an axis–aligned bounding box. This axis–aligned bounding box is passed to the parent node, which joins the axis-aligned bounding box of its children to one axis-aligned bounding box. I.e. each node of the scene graph has its own BV in world coordinates.

During the render process each node of the scene graph checks whether its BV intersects with the view frustum, if not this node and its children are not considered during this process.

Not every node of the BVH is used every frame. Therefore, a lot of time can be saved by updating the nodes of the BV as lazy as possible. The internal nodes of the BVH are only updated when required. I.e. a transformation of the geometric object is only applied to the root node immediately, to all other nodes upon request. In the case of a collision detection query, the nodes of the hierarchy are transformed while the algorithm steps through the hierarchy. Therefore, each node needs to know whether it has already been transformed. This can be achieved by using a frame counter, which is incremented each frame. Each node stores the frame number when it was transformed the last time. On transformation, the node checks if the frame counter is higher than the stored value, in this case, the transformation is performed and the absolute k-dop is returned, otherwise the stored k-dop is used. This approach is easy to implement and is very efficient,

since only an integer comparison is needed. The drawback with this approach is that if the objects are not transformed each frame, a lot of BV transformations have to be calculated even if the stored k-dop is still correct. Depending on the requirements of the application, the user should decide whether to use a frame counter — in the case of high update rates — or to store the last transformation matrix in each node and compare the transformation matrices instead of the frame number.

The approximation of the transformed k-dop can be very bad, depending on the transformation; therefore, the user can provide the convex hull of an object. If the convex hull is provided, the root of the hierarchy uses the vertices of the convex hull instead of the vertices of the boundary to calculate the transformed k-dop. One way to compute the convex hull is described in [BDH96].

3.3 Application Outline

The collision detection implementation presented in the previous section has been integrated into a multi–user virtual environment system, which has been designed and implemented for this work. This application allows the user to navigate through virtual environments and interact with objects in the scene. The objects can either be static or dynamic. Dynamic means they are able to change their location and rotation over time. This movement can either be predefined or emitted by user actions.

The virtual scenes, which can be used within this application, are Virtual Reality Modeling Language [VRM97] files. The VRML97 standard has been extended by application specific nodes. See Appendix C for a specification of these extensions. These extensions are necessary since VRML97 does not support every aspect of collision detection, e.g. the default collision detection VRML97 supports is between the viewpoint of the user or the avatar representing the user and geometry nodes, to prevent the avatar from "entering" the geometry of objects. *Collision* nodes are used to enable or disable collision detection for their children. Nodes have been added to support full collision detection between arbitrary *Group* and *Shape* nodes, without changing the behavior of existing VRML97 nodes. VRML97 has a hierarchical file structure, where geometry objects (*Shape* nodes) can be aggregated by *Group* nodes. Other aspects of VRML97, like *Lighting* nodes, are ignored, since they are irrelevant for collision detection. The aggregation of objects by *Group* nodes is a key factor, because the model of a car can be partitioned into distinct geometry objects, e.g. geometries for the wheels, the engine, the interior and the car body. Depending on the application, collision interest for the car can either be the car as a whole or each part of it. If the car drives over a curb, the collision detection query can return the car as a result, or the wheel, which

collided with the curb. If the application wants to calculate whether the wheel breaks when driving over a curb, it should do collision detection for each part.

Since collisions can only happen between static and dynamic objects and between two dynamic objects, objects have to be registered for collision detection queries. Of course, there is no restriction to do collision detection queries between two static objects, but the result will not change over time. On registration for the collision detection, an object has to decide on which basis the collision detection has to be done. There are three types of collision detection.

- **Bounding volume**
  Collision detection is done only on basis of BVs. If the BVs of two objects overlap, a collision is reported.

- **Witness**
  Collision detection is done on basis of polygon data. If one polygon is detected that intersects with one polygon of the other object, a collision is reported with the pair of polygons as the *witness*. No further collision detection between these two objects is processed.

- **Complete**
  Collision detection is done on basis of polygon data. The difference between this type of collision detection and "Witness" is that it reports all pairs of polygons which intersect.

```
# VRML V2.0 utf8

Transform {
    children [
        DEF AL Inline {
            url "Al.wrl"
        }
        DEF MUSTANG Inline {
            url "Mustang.wrl"
        }
    ]
}

CollisionInterest {
    objectPath          "AL"
    collisionInterests  [ "MUSTANG" ]
    collisionType       1            # CDT_WITNESS
}
```

Listing 3.3: Example for registering collision detection

These different types should be used as a trade–off between time spent on collision detection and exactness of the collision result. The more exact the collision report should be the more time will be spent, obviously.

The registration for the collision detection not only specifies the type of collision detection, but also the objects of interest, e.g. even if all parts of a car are registered for collision detection, maybe the application is not interested in collisions among them.

3.3.1 Collision Detection Queries

The registration of objects for collision detection is done by using the node *CollisionInterest* (see Appendix C.1). Each registered object, e.g. a *Transform*, a *Group* or a *Shape* node, has an interest list of objects for collision detection. Collisions are only reported among those objects and their interests. See Listing 3.3 for an example of a VRML97 file with enabled collision detection. In this example *AL* is interested in collision detection with *MUSTANG*, i.e. *AL* gets the collision detection results if collision between these two objects occurs.

```
bool DOP::collide( DOP& dop ) {
  if (intersect(dop)) {
    if (left && dop.left) {
      if (left.collide(dop.left))
        return true;
      if (left.collide(dop.right))
        return true;
      if (right.collide(dop.left))
        return true;
      return right.collide(dop.right);
    }
    else if (left) {
      if (left.collide(dop))
        return true;
      return right.collide(dop);
    }
    else {
      if (!intersectFaces(dop))
        return dop.intersectFaces(*this);
      else
        return true;
    }
  }
}
```

Listing 3.4: Collision detection for two hierarchies of k-dops, code is only for collision detection type Witness.

The scene graph manages the list of collision interests. Between the update process and the render process collision detection is performed. For each collision interest the collider, the object specified by *objectPath*, is checked for collision with each collidee[6], objects specified in *collisionInterests*. The collision detection executes, as follows, for the three collision detection types:

- **Bounding volume**
  Return **true** if the BV of the collider intersects with the BV of the collidee, **false** otherwise.

- **Witness**
  While the collider is a *Group*[7] node call collision detection for all children of this node with the collidee. If the collider is a *Shape* node and the collidee a *Group* node, call collision detection for the collider with all children of the collidee. Finally, collision detection is performed between pairs of *Shape* nodes (see. Listing 3.5). As long as no collision is reported, check collision between these pairs. This is done by using both hierarchies (see Listing 3.4).

- **Complete**
  Same as **Witness**, but collision detection reports all pairs of polygons which intersect and does not stop preemptively.

The *Transform* node has been extended by an exposed field *collisionResponse* and an exposed field *velocityVector* (see Appendix C). The *velocityVector* is the linear velocity of the node, used as a simple motion model. The *CollisionResponse* node specified in this field will be executed when a collision for this node is reported. The collision response is very limited, because it is not the subject of this work, for a detailed coverage of this subject compare [MW88, Bou02, Len02, AMH02]. Collision response is limited to two types of responses. The first response type sets the velocity of the node to zero; the second calculates a new velocity for the node by using the normal of the colliding polygon of the collidee to simulate a bouncing effect. In addition, it can be specified that the node should be *set back* to the last known translation where no collision occurred.

3.3.2 Collision Detection Times

As mentioned before, the creation of the BVH is very time consuming and the memory required for the hierarchy is immense for large geometric objects. Therefore, the use of split thresholds is important. To illustrate the dependency between

[6] Although the words *collider* and *collidee* do not exist, they are very useful for the distinction between the initiator of the collision query, the collider, and the object of interest, the collidee.

[7] *Transform* and *Inline* nodes are also *Group* nodes, see [VRM97].

```
bool VrmlGroup::collide( VrmlChild& collidee,
    CD_COLLISION_TYPE cdt ) {
  switch (cdt) {
    case CDT_BOX:
      return absBBox.intersect(collidee.absBBox);
    default:
      if (absBBox.intersect(collidee.absBBox)) {
        bool intersect = false;
        for all children do {
          intersect |= child->collide(collidee, cdt);
          if (intersect && cdt == CDT_WITNESS)
            return true;
        }
        return intersect;
      }
      else
        return false;
  }
}

bool VrmlShape::collide( VrmlChild& collidee,
    CD_COLLISION_TYPE cdt ) {
  switch (cdt) {
    case CDT_BOX:
      return absBBox.intersect(collidee.absBBox);
    default:
      if (absBBox.intersect(collidee.absBBox)) {
        if (collidee.isDerivedFrom(VRML_GROUP)) {
          bool intersect = false;
          for all children of collidee do {
            intersect |= collide(collidee.child, cdt);
            if (intersect && cdt == CDT_WITNESS)
              return true;
           }
          return intersect;
        }
        else
          return absDop.collide(collidee.absDop);
      }
      else
        return false;
  }
}
```

Listing 3.5: Code to determine collision pairs.

Threshold	Hierarchy Height	Hierarchy Size approx. (KByte)	Average CD times (msec)		
			LS	SP	MS
20	12	1790.16	3.0270	4.1532	3.2784
19	12	1893.47	2.9430	4.0333	3.0202
18	12	2028.84	2.8152	3.9305	2.8740
17	12	2153.53	2.7382	2.7350	2.5680
16	13	2306.72	2.6402	2.6908	2.2297
15	13	2442.10	2.2749	2.6367	2.0435
14	13	2581.03	2.2408	2.3500	1.6812
13	13	2759.16	2.0278	2.3096	1.6520
12	13	2972.91	1.8904	2.1469	1.6415
11	13	3247.22	1.7090	1.9447	1.5023
10	13	3564.28	1.44548	1.7389	1.4363
9	14	3952.59	1.25937	1.3628	1.2616
8	14	4486.97	1.17452	1.2365	1.1740
7	14	5074.78	1.10480	1.1470	1.0399
6	14	5805.09	1.07746	1.1463	0.9104
5	15	6973.59	0.93983	1.0450	0.8528
4	16	8526.84	0.82923	0.8871	0.7301
3	16	11091.84	0.76927	0.7800	0.6275
2	17	15527.16	0.70629	0.6980	0.5489
1	18	25377.47	0.60978	0.6281	0.5139

Table 3.4: Average collision detection times in close proximity in dependence of the split threshold. Sample scene is "ten_als.wrl".

threshold, hierarchy height, hierarchy size and collision detection times the Tables 3.4 to 3.6 compare the different values. Figure 3.11 shows a diagram for each of the tables.

All experiments are scenes with ten objects, Al.wrl, Eagle.wrl and Tree.wrl respectively, inside a box. Each of the ten objects is registered with all other objects and each side of the box. On colliding with another object they bounce off each other. The initial *velocityVector* has a random value, but is the same for all experiment runs. As the tables show clearly, the memory required for the hierarchy can be reduced easily, e.g. a threshold of 6 in scene "ten_eagles.wrl" reduces the required memory by the factor of 3 and decreases the average collision detection time only by 200 microseconds. Note that the memory required for a hierarchy means that this memory is required for each of the 10 objects in the scene, since it cannot be shared among same objects.

If collisions occur rarely, even large thresholds greater than 50 can be used. A collision detection time of 20 milliseconds is no problem, if collisions occur only every few seconds.

Table 3.6 also states, that a smaller hierarchy threshold does not imply faster collision detection times. In the scene "ten_trees.wrl", the minimal average collision detection time is for the threshold of 4, lower thresholds increase the detection times. This is very likely due to the higher required memory, if the overall memory usage is much higher than the available main memory size, page faults become more likely. The machine on which the timings have been made, as mentioned in Section 3.2.4, has 1 GByte RAM, for computers with less available main memory, this effect is likely to occur much earlier.

Threshold	Hierarchy Height	Hierarchy Size approx. (KByte)	Average CD times (msec)		
			LS	SP	MS
20	14	4501.22	1.8718	2.5958	1.4826
19	14	4700.72	1.7403	1.9155	1.3158
18	14	4918.03	1.6266	1.7169	1.2752
17	14	5249.34	1.5937	1.4591	1.2283
16	14	5537.91	1.5510	1.3399	1.2221
15	14	5890.59	1.4321	1.3218	0.9342
14	15	6314.53	1.2097	1.3058	0.9264
13	15	6806.16	1.1496	1.0777	0.8436
12	15	7361.91	0.9386	1.0702	0.5255
11	15	8053.03	0.8569	0.8646	0.4800
10	15	8818.97	0.8214	0.8572	0.3973
9	15	9691.78	0.7590	0.6998	0.3550
8	16	10867.41	0.6587	0.6697	0.3430
7	16	12399.28	0.5416	0.4823	0.3458
6	16	14433.47	0.4609	0.4861	0.3236
5	17	17012.72	0.4077	0.4078	0.3111
4	17	20824.59	0.4012	0.3069	0.2463
3	18	27137.34	0.3543	0.2583	0.2403
2	19	38138.34	0.2990	0.2365	0.2199
1	20	61725.66	0.2698	0.2279	0.2165

Table 3.5: Average collision detection times in close proximity in dependence of the split threshold. Sample object is "ten_eagles.wrl".

Threshold	Hierarchy Height	Hierarchy Size approx. (KByte)	Average CD times (msec)		
			LS	SP	MS
20	16	12794.72	3.5635	3.2552	3.0637
19	16	13396.78	3.5613	3.0589	2.9694
18	16	14034.47	2.8303	2.4743	2.9627
17	17	14679.28	2.8081	2.4672	2.7521
16	17	15605.53	2.6351	2.1416	2.6766
15	18	17230.03	2.6338	2.1476	2.3421
14	18	18480.47	2.6107	2.0567	2.3290
13	18	19527.84	2.1489	1.8147	2.2538
12	18	20764.03	2.0515	1.7757	2.1008
11	18	22370.72	1.9941	1.5686	1.8273
10	18	24169.78	1.8805	1.4740	1.7883
9	18	26485.41	1.6489	1.3713	1.7153
8	19	30022.97	1.3408	1.3134	1.6692
7	19	37130.16	1.3415	1.2827	1.6298
6	19	42470.34	1.3123	1.1558	1.6096
5	20	47443.59	1.2552	1.1289	1.5979
4	20	54842.91	1.2409	1.1081	0.9487
3	20	75883.03	1.5086	1.2747	1.1973
2	21	98686.59	1.9332	1.3881	1.5650
1	22	165169.97	2.1097	1.2887	2.5305

Table 3.6: Average collision detection times in close proximity in dependence of the split threshold. Sample object is "ten_trees.wrl".

3.4 Conclusions

In this chapter, all important issues of BVs and collision detection have been discussed. The implementation is able to meet the requirements specified in Section 3.2.1. Even large objects, like "Tree.wrl", can be checked for collisions in real–time. By the extensions of the VRML97 specification, the collision detection can be handled on scene basis and the user is able to specify the resolution level of the collision detection queries.

The intensive examinations of bounding volumes and the effectiveness of algorithms for collision detection have provided an in depth look into the subject of real–time collision detection. Furthermore, the simulations and comparisons done with the existing and new approaches helped to combine the best approaches for this work. The implementation of this chapter is the basis for the approach to

collision detection in distributed virtual environments, presented in the following chapter. Since *RAPID* and *QuickCD* are focused on reporting collisions between two dynamic objects and not the integration of collision detection into a scene graph hierarchy the usage of one of these collision detection systems was not an issue. However, they could have been used for this task as well, but not without extra work to integrate these systems into the scene graph hierarchy.

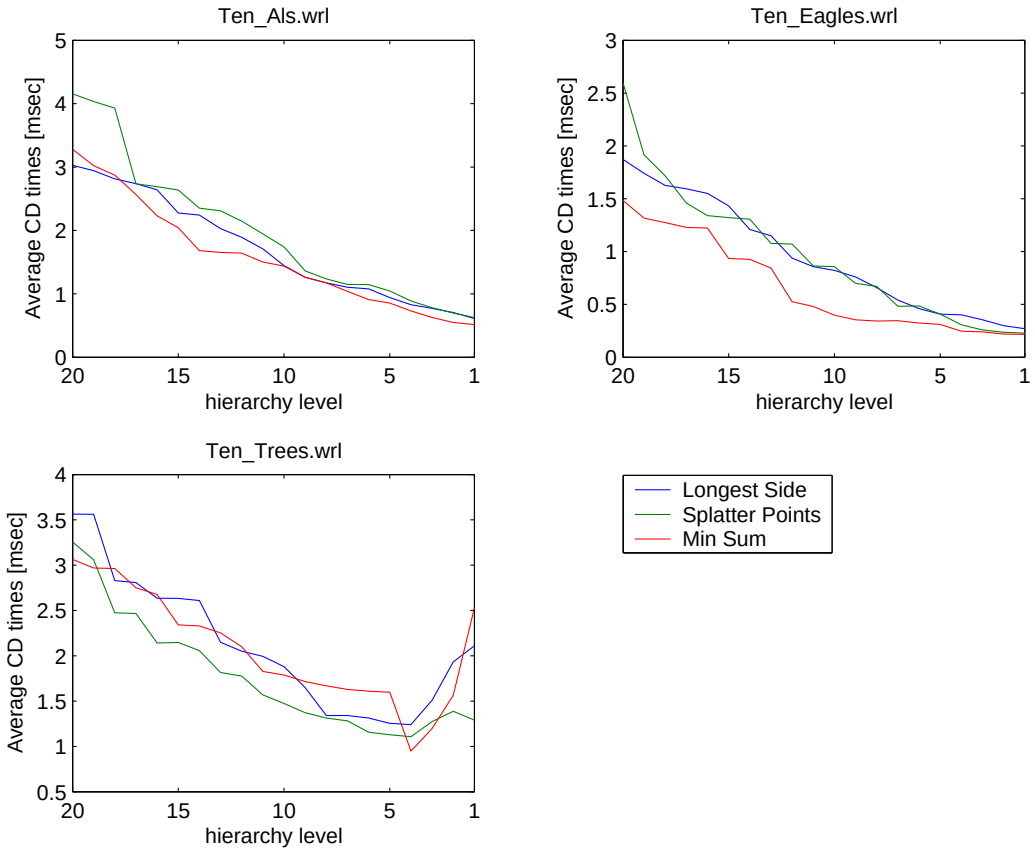


Figure 3.11: Average collision detection times in close proximity in dependence of the split threshold.

Chapter 4

Collision Detection in Distributed Virtual Environments

In distributed virtual environments multiple users interact with each other or with world objects in real–time. Usually, these users are represented by an avatar in the virtual world. When navigating through the world or interacting with objects the other users are notified of the user action by visual, sensory or audio feedback, e.g. they see how the avatar of that user walks or interacts with objects. To ensure this awareness, messages have to be exchanged between all users. The awareness in distributed virtual environments is a key factor, for more information see [BP99, MBP99, PPB01]. How these messages are distributed among the users is an important factor and depends on the underlying framework of the distributed virtual environment. These frameworks are different in the terms of scalability, network architecture and network communication.

The properties of the frameworks have great impact on how and when messages are distributed among the users. The most important factor for collision detection is network latency. As mentioned in Chapter 3, collision detection is performed between a pair of virtual objects. In non–distributed virtual environments the position and the orientation of both objects is known at the time of the collision query. This is not the case in distributed virtual environments. Collision detection between two moving avatars can hardly be performed with the exact positions and the exact orientations of both objects at the time of the collision query. Due to network latency, all update messages arrive some time after the message has been sent. Therefore, at the time of the collision query the position and the orientation could have changed, since another update message has not arrived yet. A fundamental rule about distributed virtual environment's shared state describes this drawback. It is known as the *Consistency–Throughput Tradeoff* [SZ99]:

> "It is impossible to allow dynamic shared state to change frequently and guarantee that all hosts simultaneously access identical versions of that state."

The *Consistency–Throughput Tradeoff* states that a distributed virtual environment cannot support both dynamic behavior and absolute consistency. To proof this statement, two scenarios are described, the first consistent and the other highly dynamic.

In a consistent distributed virtual environment no client may change the state of the environment unless all other clients have agreed to the new state. Suppose, a user wants to change his avatar's position and suffers from 100 ms network latency. It takes at least 100 ms until all other clients have received this request and another 100 ms to send an acknowledgement back to user. 200 ms of round–trip time to change the state of the environment means that no more than 5 consecutive state changes can be processed per second. Interaction is therefore limited to 5 frames per second, which is far away from supporting dynamic behavior.

In a highly dynamic distributed virtual environment users may change the state of objects without the agreement of other users. Inconsistencies are handled after they have occurred. Suppose, a user is walking around in the environment, at each frame he sends update messages to all other users about his current position, e.g. current position is p. At the time this update message arrives at the other clients, these clients only know that the user is somewhere near p, since in the mean time his position could have been changed again. As long as users are changing the state of the environment, no consistent state will be reached.

These two scenarios show clearly, to provide highly dynamic object behavior, absolute consistency cannot be supported.

The following section provides a short introduction about network architectures and how messages are distributed among users. This overview is important for the later sections, since they imply basic knowledge of this subject. The experienced reader may skip this section. Section 4.2 examines methods how to deal with network latency and especially prediction of remote object movement. In Section 4.3 it is explained how to keep a consistent state for distributed virtual environments. Section 4.4 provides new approaches to significantly reduce the aggravation of dead reckoning techniques. Additionally, experiments with different types of object motion show the efficiency of these approaches and a solution is provided to decide whether a detected collision has actually occurred.

4.1 Network Architectures

4.1.1 Reliable vs. Unreliable Connections

Network applications usually use one of the two network layer protocols, TCP and UDP, both sitting on top of the transport layer protocol IP. The difference between these two network layer protocols is the type of connection. TCP/IP is a connection–oriented reliable transfer protocol and UDP/IP is not. TCP/IP ensures that no transferred data is corrupted or lost and all data will arrive in order. In addition, at the beginning of a TCP/IP connection between two hosts a *hand–shake* is performed, where both establish parameters for the following data transfer. UDP/IP does not guarantee anything, neither that received data is not corrupted nor that data will arrive in order or at all. UDP/IP is connection–less, i.e. no handshake between the two hosts is performed [KR02].

By ensuring reliable data transfer, TCP/IP induces longer latencies and slower data transfer than UDP/IP. In addition, TCP/IP utilizes congestion control, which stops sending packages to the remote host if the connection is congested.

Another transport layer protocol is the multicast protocol, which enables a host to send and receive data to and from a group of hosts without having a direct connection to them. The multicast protocol is unreliable as well, but since it is able to support bulk data transfer to a group of receivers while limiting required bandwidth, a lot of effort is done to implement reliable multicast protocol. No standard has been proposed yet, due to the great challenges of this problem, also reliable multicast is widely used in frameworks for distributed virtual environments.

4.1.2 Client/Server

Network applications typically have two parts, the client and the server part. The server *serves* the client and usually both parts reside on different computers (hosts) [KR02]. If a distributed virtual environment uses this network architecture, the user's system provides the client which communicates with the server. Without loss of generality, we assume that the server is a stand–alone system not hosted by a user of the distributed virtual environment. A user connects from the client host to the server host to *enter* the distributed virtual environment. The server responds by sending the world's scene information to the client and adds the user to the list of active users. If the user is represented by an avatar, the client sends the information of the user's avatar to the server. The server distributes this avatar information to all other users. Each user action is sent to the server and then distributed to the community. See Figure 4.1 for a diagram of this architecture.

Usually the server does not forward all messages to every user. To keep network traffic low, only messages are forwarded which are useful for the user, e.g. if user A

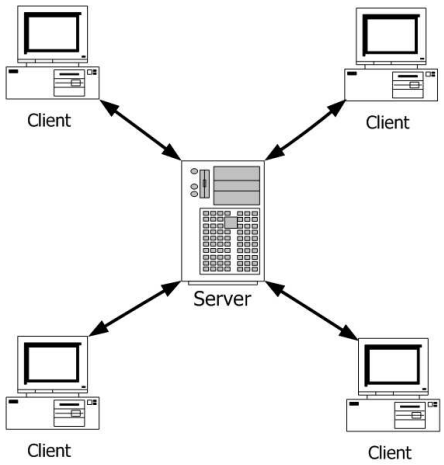


Figure 4.1: Diagram of a Client/Server network architecture.

is behind a wall and cannot be seen by user B, user B is not interested in position updates from user A until he moves into his field of vision.

The advantage of this network architecture is the simplicity of the centralized approach. Users need only one connection between their host and the server. The disadvantage of the Client/Server architecture is that they do not scale. While the number of users increases, the server becomes more and more the bottleneck of the system, since the amount of work increases quadratically for the server.

The server has to support that users logging–on and –off and provide the persistency and consistency of the virtual environment, i.e. to store the state of the virtual environment, even if no user is active and make sure that no inconsistent state can occur.

4.1.3 Peer–to–Peer

In a peer–to–peer network architecture each user's host is client and server at the same time. A user's host has to establish connections with all clients of the distributed system. This is done by connecting to one host of the system, which responds by sending the world's scene information and the list of all active users. Then the host connects to all users and sends the information of his avatar.

Each user action is sent directly to all other users. Filtering messages for specific users to lower network traffic also applies, but has to be done by each client.

The advantage of peer–to–peer architectures is that it is decentralized and extremely failure tolerant. If one host crashes or suffers from network congestion, no other host suffers from this defect. The disadvantage is that each host has to

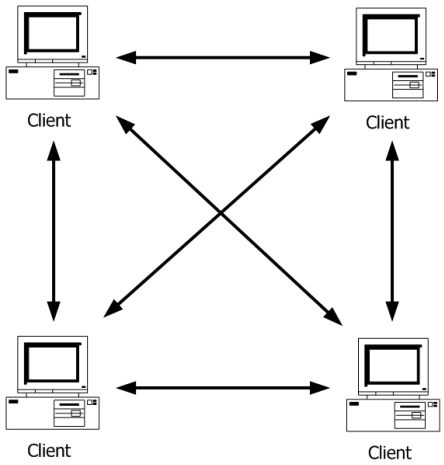


Figure 4.2: Diagram of a Peer–to–Peer network architecture.

establish connections to every host of the community. Each host has to provide server functionalities, like log–on, log–off, persistency and consistency control. To ensure persistency the last active user of the environment usually has to store the world's information on a server.

4.1.4 Multicast

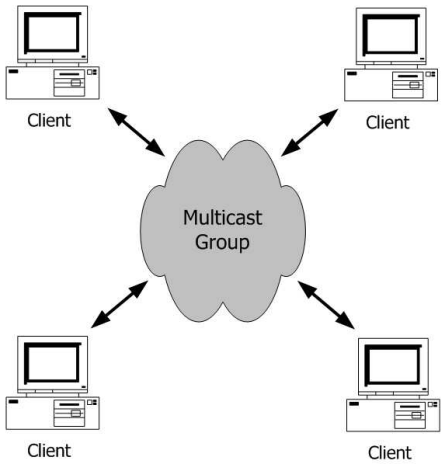


Figure 4.3: Diagram of a Multicast network architecture.

Multicast networks, see Figure 4.3, allow arbitrarily sized groups to communicate on a network via a single connection. Each user message is sent to the

multicast group once and is distributed to all users of the group. The sender can specify how far a message is sent by using the time–to–life (ttl) field, which is decremented when it passes through a router. A ttl value of 16 will address all members of the group within the same local site [McC98]. Distributed virtual environments utilizing this architecture are usually combined with a Client/Server architecture for the initialization phase. The user connects the server to receive the current world's scene information and information about the multicast group addresses used by the environment. After the initialization phase the user joins the multicast group to receive all update messages.

The server, responsible for providing the world's scene information, must also ensure persistency. Consistency control has to be performed by each host of the group.

Filtering messages for specific users to lower network traffic cannot be supported, since no direct connection to other users is used. In DIVE [FS98] multiple multicast groups are used for subdivisions of the scene graph to allow users to get only the relevant update messages, by joining and leaving multicast groups.

The advantage of multicast architectures is that they are scalable and the network traffic is reduced to a minimum. The disadvantage is that the underlying protocol is unreliable.

4.2 Latency Compensation

Movement of objects in distributed virtual environments is often subject to approximation in order to reduce network traffic and to animate remote objects smoothly. In Client/Server architectures even the movement of the local objects is approximated, see Section 4.2.1. Since available bandwidth is still limited, updates for the position and orientation of remote objects are sent as lazy as possible; update messages are usually sent only when the approximated position has a minimum error. Even so, these latency compensation methods aggravate exact collision detection between remote objects, due to a requirement for high update rates of remote object positions to have most accurate positions for the collision detection; they are also helpful, since they are able to predict object position for the near future. In this section the most relevant latency compensation methods are outlined.

4.2.1 Multiplayer Network Games

In the early '90s the first multiplayer network games came out which were using peer–to–peer networks like Doom[1]. Each host was running the same game and the same game state logic, by synchronizing all input and timings from other hosts.

[1]Doom product of *id software*; http://www.idsoftware.com.

The disadvantages were that each host had to run at the same fixed frame rate, and users could only join at the beginning of the game[2].

The next generation of multiplayer network games utilized Client/Server architectures; this generation was pioneered by Quake[3]. Even today's multiplayer games like Half–Life[4] and the Quake3 engine derivatives[5] still use this network topology. In these type of games one authoritative server is responsible for all game state logic, connected to the server are a number of "dumb" clients. These clients send all input messages, like mouse and keyboard input, directly to the server which updates the game state and sends back a list of objects to render. The problem was that until the user got visual feedback of his actions it took a complete round–trip.

Therefore, more and more game state logic has been put into the clients, e.g. in Half–Life [Ber01], QuakeWorld and QuakeFinal. The clients still send all input data to the server, but they predict the result from the server until the response from the server arrives. The last acknowledged movement from the server is used as a starting point for the prediction. Then all inputs from this time on are used to predict the current position of the client's user. To minimize discrepancies between client and server the same code is used in both the client and in the server part. This works well, but the positions of other players are still subject to latency. To shoot at another player the user has to predict the movement and the latency. In other words, to aim at a point in front of the target, the distance of this point depends on the user's latency. To eliminate this effect, the positions of other players are predicted by a position history for these players. To enable the client to directly aim at other players, the server takes into account that the player was aiming at a predicted position. I.e. even if the prediction was wrong, the user is able to score a hit by aiming correctly. This can lead to paradox situations. In the user community of Half–Life an inconsistency was described: users with high latencies were "able to shoot around the corner" [Ber01]. Since the user suffering from high latency still had a direct line of fire, while the other user already had taken cover, but this update message had not arrived at the highly lagged user.

The reason why multiplayer network games do not use multicast or peer–to–peer architectures, is that they do not trust their clients. They fear that they would raise the motivation to hack the client, if there is no authoritative server [Ber01].

---

[2]See online document: Unreal Networking Architecture, by Tim Sweeney; http://unreal.epicgames.com/network.htm.

[3]Quake product of *id software*, see [1].

[4]Half–Life product of Valve; http://www.valvesoftware.com/.

[5]Quake3 engine produced by *id software*. QuakeWorld and QuakeFinal use the Quake3 engine, see [1].

4.2.2 Dead Reckoning

Dead reckoning[6], used by NPSNET [Pra93], predicts the position and orientation of remote objects by using the last known position and orientation, X', together with the positional velocity and angular velocity, $\dot{X}$, to calculate the actual position and orientation, X. Δ_t is the time since the last update message arrived, see Equation 4.1. Second–order dead reckoning, Equation 4.2, also uses the positional and angular accelerations, $\ddot{X}$.

$$X = X' + \Delta_t \, \dot{X} \tag{4.1}$$

$$X = X' + \Delta_t \, \dot{X} + \frac{1}{2} \, {\Delta_t}^2 \, \ddot{X} \tag{4.2}$$

The same dead reckoning algorithm is executed at the local and the remote host for each dead reckoned object. Therefore, the local host knows the approximation error of the remote host. Two thresholds are used to decide whether to send a new update message to the remote host: an error–based threshold and a time–based threshold. If the approximation error exceeds the error–based threshold, a new update message is sent to the remote host, while the time–based threshold specifies the maximal time between two consecutive update messages. Usually both thresholds are used at the same time.

On the arrival of a new update message, the dead reckoning process has to decide whether to *jump* to the actual position, or to calculate a smooth path to a predicted position in the near future. The *smooth back* method reduces visual inconsistencies, but this can result in the object chasing its correct location.

By utilizing dead reckoning the network traffic can be reduced by two orders of magnitude, while still having good approximations of remote objects.

4.2.3 Position History–Based Protocol

A more complex solution to extrapolating the positions and orientations of remote objects is the *position history–based protocol* proposed by Singhal and Cheriton [SC94]. Like dead reckoning, the local host executes the tracking algorithm as well to be aware of the approximation error. Instead of using the object's positional and angular velocities and accelerations, the tracking algorithm uses the last three positions and orientation of an object.

The algorithm adapts to the object's behavior by differentiating between sharp turns and smooth motion. A sharp turn in the object's movement is recognized by calculating the angle between the three most recent update positions. If the

[6]According to the Oxford Advanced Learner's Dictionary of Current English [ODE89], dead reckoning is the "calculation of one's position by log or compass (when visibility is bad)".

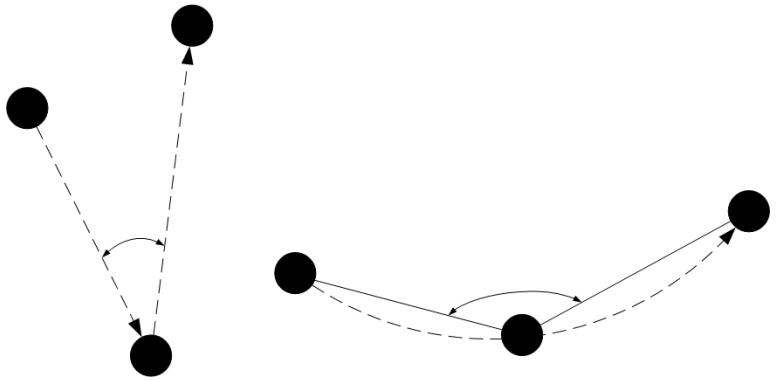


Figure 4.4: Adaptation to object's behavior, small angle on the left indicating a sharp turn, a large angle on the right indicating smooth motion

angle is small, a sharp turn is assumed; otherwise smooth motion is assumed, see Figure 4.4. In the case of a sharp turn, the two most recent positions are used to extrapolate subsequent positions. In the case of a smooth motion, second–order tracking is applied. If the three most recent positions are x_0, x_{-1} and x_{-2} at times $0, -\delta_{-1}$ and $-\delta_{-1}-\delta_{-2}$, then the initial position, the initial velocity and the initial acceleration at time 0 are determined by Equation 4.3 to Equation 4.5.

$$x(\tau)\mid_{\tau=0} = x_0 \tag{4.3}$$

$$x'(\tau)\left|_{\tau=0} = \frac{\delta_{-1}}{(\delta_{-1}+\delta_{-2})\delta_{-2}}x_{-2} + \left(\frac{1}{\delta_{-1}} + \frac{1}{\delta_{-2}}\right)x_{-1} + \left(\frac{1}{\delta_{-1}} + \frac{1}{\delta_{-1}+\delta_{-2}}\right)x_0\right. \tag{4.4}$$

$$x''(\tau)\left|_{\tau=0} = \frac{2}{(\delta_{-1}+\delta_{-2})\delta_{-2}}\ x_{-2} - \frac{2}{\delta_{-1}\ \delta_{-2}}\ x_{-1} + \frac{2}{\delta_{-1}\ (\delta_{-1}+\delta_{-2})}\ x_0\right. \tag{4.5}$$

$$x(\tau)\left|_{\tau=\delta_{-1}} = \frac{2{\delta_{-1}}^2}{\delta_{-2}\ (\delta_{-1}+\delta_{-2})}\ x_{-2} - \left(\frac{2\delta_{-1}}{\delta_{-2}} + 1\right)x_{-1} + \left(\frac{2\delta_{-1}}{\delta_{-1}+\delta_{-2}}\right)x_0\right. \tag{4.6}$$

The convergence point $x(\tau)$ (Equation 4.6) at time $\tau = \delta_{-1}$ is the point where the displayed position converges to. The convergence point for first–order tracking is straightforward derived by the two most recent positions, the algorithms can be found in [SC94].

Not only the convergence point is adaptively determined depending on the movement, but also the convergence path, which is either parabolic (second–order) or linear (first–order). First–order convergence is applied if the angle between the three most recent positions is large, indicating almost linear movement; otherwise second–order convergence is utilized.

Singhal and Cheriton have compared their approach with dead reckoning used by the DIS protocols, see Section 4.2.2. They observed that the position history–based protocol requires less bandwidth in performing at least as well as dead reckoning for smooth object motion and potentially better for non–smooth motion.

4.2.4 Special Case Approaches

As already mentioned in Section 2.3, special case approaches exist which are able to eliminate the side effects of network latency, e.g. the approach of Sandoz and Sharkey [SS96]. While these approaches do not apply to common object movement, some of these approaches are outlined to demonstrate that collision detection in distributed virtual environments has to be designed for a specific application as no general approach will meet all requirements of all applications.

In a virtual tennis match [NPC+96, NSST00], the above approaches would fail, due to the high velocity of the ball and the racket. Collision detection and response between these two objects would probably lead to an unrealistic simulation. Since the movement of the ball follows physical laws, the ball's position can be predicted exactly in spite of high latencies, e.g. world–class tennis players are able to serve the ball with velocities of 180 km/h which is 50 m/s. The tennis court is 23.77 m long, for the ball to travel from base line to base line nearly 500 ms of network latency are acceptable, at the time of the message–arrival the user, who has to return the ball, can determine the current ball position exactly. In addition, this user also knows exactly his racket's movement, therefore time exact collision detection is possible. Time–step fixed collision detection would probably miss most of the collisions, since in an animation running at 20 frames/s, the ball travels 1 meter per frame, therefore, collision between the net and the ball is likely to be missed.

This approach applies to all object movement following strict rules (usually physical laws). In a distributed snooker game (see Section 2.3) movement of all balls on the table can be exactly predicted, the same is true for the movement of projectiles and particle systems.

4.2.5 Discussion

While the special case approaches only apply to a small subset of movements which can be described by a closed formula, for random movement prediction, either dead

reckoning or the position history–based protocol has to be used. There is no limitation to combining closed formula movement and prediction of movement. In a battlefield simulation movement of troupes, helicopters and tanks can be predicted while the positions of missiles are calculated by a closed formula after they have been launched. Different collision detection approaches are easily combined by meta information in the scene graph. The application running the battlefield simulation knows the types of the objects and therefore can easily provide different collision queries for missile–missile, tank–tank or missile–tank collisions. At the core of each of these approaches, efficient collision detection for a pair of objects at certain positions is needed.

As already mentioned, the prediction of movement is usually utilized to reduce network traffic and to reduce visual inconsistencies[7]. The reduction of network traffic is achieved by larger intervals between update messages. Although the collision detection process is interested in accurate prediction of remote object positions to reduce the effects of latency, usually it demands a high update frequency to have mostly correct positions. However, prediction means that the position of the remote object is usually more error prone than without it.

4.3 A Centralized Approach

If the underlying network architecture is centralized, no inconsistencies can occur, due to collision detection. As mentioned before, in a distributed virtual environment, utilizing a Client/Server architecture, all user input is sent to the server, where it is processed and the result is sent back to the user. If no client side prediction is performed, the system is in a consistent state all the time. Collision detection can be used as in the non–distributed case, since the server has all required information, and the results are correct. I.e. if the server receives an input message from one of the users, a new position for the avatar of that user is calculated. Unless a collision for that avatar occurs at this new position, the new position is sent to the user. Otherwise a the collision response has to be calculated and the result has to be sent to the user.

The problem of this approach is that the consistency is achieved by delaying the user action. Until the user becomes aware of his action, a complete round–trip time is needed. Depending on network latency this can lead to unsatisfying reaction times.

As mentioned in Section 4.2.1, owing to this drawback, client–side prediction is utilized. Nonetheless, the centralized approach is still able to keep a consistent state at all time, only the clients have temporarily inconsistent states. Since the

[7]Visual inconsistencies means that inconsistent states are temporarily accepted to provide smooth animation of remote objects. They do not "jump" to their current positions.

server is distributing the results of the collision detection to the clients, these inconsistencies can be eliminated after a short time. Although, this is a acceptable from a theoretical point of view, in multiplayer network games the requirement for consistent states is weakened to reduce visual inconsistencies for each user. That this can lead to paradox situations has been already been discussed.

4.4 New Approaches for Distributed Virtual Environments

The following approaches apply mainly to non-centralized network architectures. They apply to Client/Server network architectures as well, if the server is used to distribute the messages of the clients only, but does not perform collision detection.

Section 3.3 stated that collisions can occur between two static objects, one static and one dynamic object and between two dynamic objects. For distributed virtual environments the types of collision pairs have to be diversified.

Definition 5. If an object's movement is due to the action of user A, e.g. user A moves his avatar, this object is marked **enslaved**. Each object of the scene can only be enslaved by one user or not at all. An enslaved object is marked enslaved only at the client of user A, at all other clients it is marked **remote**. If an object not enslaved by a user is marked **scene**.

With this definition the above statement will changed to: Collisions can occur between the following pairs of objects

- scene–scene objects,
- scene–remote objects,
- scene–enslaved objects,
- remote–remote objects,
- remote–enslaved objects and
- enslaved–enslaved objects.

The important object pairs are scene–remote, remote–remote and remote–enslaved, since the position of at least one object is error prone. The other object pairs are collisions among objects, whose positions are exactly know at the time of contact and therefore no special approach has to be applied to improve the collision detection result.

Object	**scene**	**remote**	**enslaved**
scene	none	Remote Collision Prediction	none
remote	Remote Collision Prediction	Remote Collision Prediction	Adaptive Collision Prediction Tracking
enslaved	none	Adaptive Collision Prediction Tracking	none

Table 4.1: Overview of applied approaches for distributed collision detection. *none* means that no approach is applied, only collision detection is executed.

Table 4.1 shows the applied approaches for the different object pairs.

If collisions have to be detected between scene and remote objects or between two remote objects, the local client is not included in the decision whether a collision has occurred or not. Until the message arrives indicating a collision or a change of direction, collisions of remote objects should be predicted. This is done by the method of *remote collision prediction* (see Section 4.4.1).

The hardest challenge is the detection of collisions between remote and enslaved objects. No client has exact information about all objects involved. To reduce the approximation errors induced by dead reckoning, the *approach adaptive collision prediction tracking* is utilized (see Section 4.4.2).

For the following experiments the dead reckoning technique has been used. The results can be applied to the position history–based protocol as well, since they basically work the same way. The main difference would be that the position history–based protocol approximates smooth motions better than dead reckoning.

The Test Scenario

To demonstrate the effects and the efficiency of the two approaches described below, the following test scenario has been chosen, see Figure 4.5.

An enslaved object is moved from $z = 0$ to $z = -40\pi$ in 15 seconds. The value for x is determined by Equation 4.7 and $y = 0$.

$$x = \begin{cases} \frac{-z}{6} & ; \quad |z| < 10\pi \\ \frac{z+20\pi}{6} & ; \quad 10\pi < |z| \leq 20\pi \\ |5\sin(\frac{z}{5})| & ; \quad 20\pi < |z| \end{cases} \tag{4.7}$$

Collision with scene objects occur at $z = 10\pi$ and $z = (20 + 5i)\pi$, where $i = 0 \ldots 4$. Remote objects are located at $(8, 0, 27.5\pi)$, $(8, 0, 32.5\pi)$ and $(8, 0, 37.5\pi)$, but no collisions with these remote objects occur. The enslaved object is 5.42m wide.

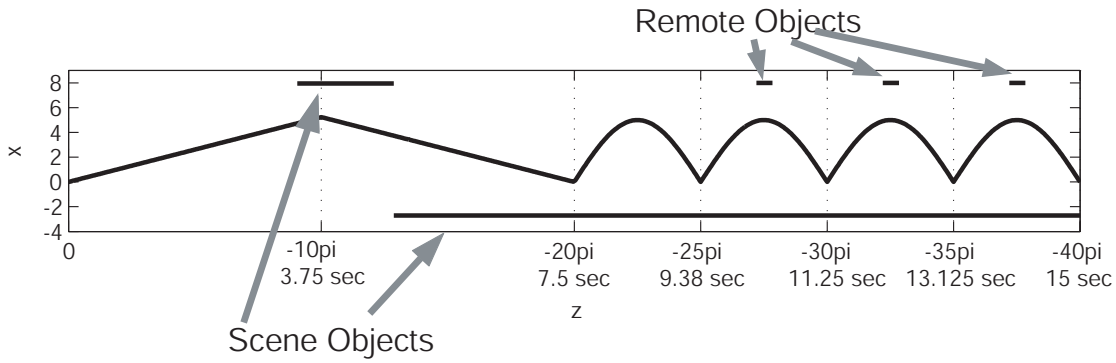


Figure 4.5: Test scenario

The test scenario combines smooth motion and sharp turns, which are two common motion classes [SC94]. The third motion class is random motion. Since tracking and convergence algorithms are of little benefit for this kind of motion, this motion class is not tested with this scenario, see Section 4.4.3 for a discussion of this topic.

At the end of this section, the problem of decision resolution is outlined and exemplary solutions are provided.

All diagrams of the experiments for this test scenario can be found in Appendix A. See Figure B.13 for a screen shot from one of the experiments.

4.4.1 Remote Collision Prediction

Dead reckoning has its weaknesses, after a collision has occurred, it takes some time for this update message to arrive at remote hosts. Sharp turns, e.g. as a result of a collision, cannot be predicted by the dead reckoning technique. Therefore, a client has to predict collisions of a remote object on bases of the current approximated object position, see Figures A.1 to A.4.

By applying remote collision predicting, the worst approximation errors could have been eliminated. The collision at time $t = 3.75$ is predicted correctly and the approximation error is independent of network latency. The remaining approximation error is due to the different frame rates of the two clients, collisions are detected at different times and therefore with different object positions. The error is negligible, since the predicted path and the visual appearance is correct. Even though the collision response for the collision at time $t = 7.5$ is wrong, remote collision prediction is able to reduce the approximation error by more than 20% and at the time the next update message arrives, the predicted position is less error prone. See Figures A.5 to A.8 for the results of the experiments, while remote

collision prediction was being applied.

The first two collisions could have been detected with correct remote object positions, because the remote object's movement had been straight. This is not the case for the collisions at time $t = 9.38$, $t = 11.25$ and $t = 13.125$. As the diagrams show, the remote object's position is error prone at the time of collision. In the test scenario remote collision prediction is able to reduce the approximation error to a minimum because the collision response is the same on both clients. Of course, this approach would fail if the no collision would have been detected at the remote host, in this case the path would have stayed the same until the next update message arrived.

The most important reason for remote collision prediction is that it prevents remote objects from entering other objects.

4.4.2 Adaptive Collision Prediction Tracking

The reduction of network traffic is achieved by using an error–based threshold, as mentioned before, update messages for enslaved objects are only sent if the error of the predicted position exceeds the error–based threshold. This mechanism implies larger error of remote object positions; the minimal error only depends on network latency. Since the network bandwidth is a limited resource, a trade–off has to be found between high update rates to support collision detection and low network traffic to save bandwidth.

The following approach, which has been developed for this work, is able to compensate low update rates at times where collisions between remote and enslaved objects are very unlikely for the near future. This is done by calculating the time of collision for two moving objects. If this time is below a threshold, the parameters for the dead reckoning protocol are adapted to achieve higher update rates, i.e. the error–based threshold is reduced to a small value.

Definition 6. Two objects are in close proximity if their enclosing bounding spheres intersect, or if they will intersect or have intersected within a certain time Δ_τ. Δ_τ is called closed proximity threshold.

To test whether two objects, P and Q, are in close proximity, it is checked whether their bounding spheres intersect at time $t = 0$. If they do, no further calculations have to be performed. Otherwise, the time at which the bounding spheres have contact, is calculated with their current position and current velocity. The position of object P and Q at time $t = \delta$ is calculated by Equation 4.8 and Equation 4.9 respectively. The distance of both objects at time $t = \delta$ is the length of the vector $\overrightarrow{d_\delta}$ (Equation 4.10), which is calculated by Equation 4.11. The bounding sphere is used for this approach, since it is the only bounding volume, which is rotation invariant. Due to this property, the distance of two bounding

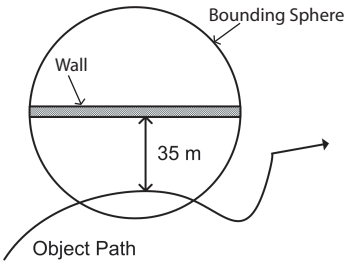


Figure 4.6: Close proximity example. Although the object path is far away from the wall, this is a close proximity situation, due to the poor approximation of the bounding sphere.

spheres can be described by a closed formula. All other bounding volumes cannot be used for this approach. The disadvantage of the bounding sphere that it fits objects very poorly is also an issue for this approach, e.g. a wall 10 m high, 1 m thick and 100 m long has a bounding sphere of 50.25 m. No matter which direction an object is moving towards this wall the close proximity starts at latest when entering the sphere, see Figure 4.6.

$$\overrightarrow{P_\delta} = \overrightarrow{P_0} + \overrightarrow{v_P} \cdot \delta \tag{4.8}$$

$$\overrightarrow{Q_\delta} = \overrightarrow{Q_0} + \overrightarrow{v_Q} \cdot \delta \tag{4.9}$$

$$\overrightarrow{d_\delta} = \overrightarrow{Q_0} - \overrightarrow{P_0} \tag{4.10}$$

$$distance_\delta = \sqrt{\overrightarrow{d_\delta} \cdot \overrightarrow{d_\delta}} \tag{4.11}$$

$$\begin{aligned} distance_\delta = r \quad &\Leftrightarrow \\ {distance_\delta}^2 = r^2 \quad &\Leftrightarrow \\ \overrightarrow{d_\delta} \cdot \overrightarrow{d_\delta} = r^2 & \end{aligned} \tag{4.12}$$

To determine the time where the bounding spheres of object P and Q have contact, Equation 4.11 is set equal to the sum of both radii, r. The calculation is simplified by squaring both sides of the equation, i.e. squared distance equal to r^2.

Solving Equation 4.12 yields either one, two or no result for δ. No result means, the bounding spheres do not intersect at their current velocities, neither in the past nor in the future. Otherwise, there is at least one result. Let δ_- be the minimum

of the two results and δ_+ the maximum, in the case of one result $\delta_- = \delta_+$. δ_- and δ_+ can both be negative, a negative value indicates that the time of contact was in the past. If δ_- or δ_+ is negative the double of the absolute value is assigned to them. This approach declares two objects within close proximity even if the time of contact was in the past. The reason for that is explained by an example of the test scenario. In the test scenario, the enslaved object avoids collisions with objects at time $t = 10.315$, $t = 12.185$ and $t = 14.06$, after the enslaved object has reached its turning–point and moves into the opposite direction the collision has been avoided. Depending on network latency, the object position at the remote client has not reached its turning–point yet. To ensure that the remote object follows the path of the enslaved object smoothly, the adaptive collision prediction tracking requires for high update rate some time after a collision had been avoided. Since the movement towards another object is more critical than the movement away, negative values are multiplied by -2, i.e. half of the close proximity threshold.

The two objects are in close proximity if the value of $min(\delta_-, \delta_+)$ is less than the close proximity threshold, τ.

The efficiency of this approach can be spotted in Figures A.9 to A.12, where the approach of remote collision prediction as well as adaptive collision predicting tracking have been applied. At time $t = 8.44$ no obstacle has to be avoided, therefore this approach does not have to be applied. At times $t = 10.315$, $t = 12.185$ and $t = 14.06$ the approach notices that the enslaved object comes into close proximity and uses a different error–based threshold. The resulting average error is significantly smaller than without applying adaptive collision prediction tracking. Of course the network traffic is increased during close proximity situations, but the results prove the great influence of this approach.

Since an application can not only choose the error–based and time–based threshold to find a trade–off between network traffic and exact remote object visualization for each dead reckoned object, but also an error–based threshold for close proximity situations, it is now possible that this trade–off does not aggravate the collision detection results.

4.4.3 Random Motion

The reason, why random motion is not examined here, is that this type of motion is usually not the result of a collision. More likely, it is the result of a collision avoidance, e.g. while someone walks through a crowd of people he will try to avoid running into someone else by changing direction, velocity and acceleration very often. Nonetheless, the above two approaches apply to random motion as well, since they do not make assumptions about the type of motion. While remote collision prediction will be of little use for random motion, the adaptive collision

prediction tracking will have comparable results, since the area of interest can be made large enough so that more update messages are able to approximate the motion.

4.4.4 Decision Resolution

When two enslaved objects collide, it is very likely that the two hosts compute different collision results. As examined in the test scenario, only if one of the objects' movement is straight or does not move at all, the remote side is able to predict its position exactly. The approximation error for turns becomes larger the faster an object moves.

Unless the distributed virtual environment uses a client/server architecture, where the collision detection is only performed on the server, there has to be a strategy to decide, which of the hosts' result is chosen to be correct (even if it is not).

This decision has to be made independently and the result must be the same. On detecting a collision a host needs to know whether it may distribute the result or not, to prevent the distribution of different messages for the same event. E.g. host A detects a collision of its enslaved object O_A with the remote object O_B enslaved by host B, host A calculates the collision response and distributes the results. Host B has concluded for the same situation that no collision has occur and therefore will send a message of its new position. Other hosts will not be able to decide which message is correct and will end up in an inconsistent state. The process of choosing a host to be responsible for the collision detection will be named *decision resolution*.

The easiest way to make such a decision is to use the hosts' unique identification number. The host with the lower identification number, A, will be responsible for detection collisions and distributing the results. The other host, B, knows that it has the higher identification number and therefore will not perform collision detection, even if this means that the enslaved object enters the other object's geometry. Since if host A does not detect a collision it will only send update messages of its object and not a message that no collision has occurred. Otherwise, if host B would perform e.g. remote collision prediction this could lead to an inconsistent state, because no resynchronization message for its object is sent. Owing to the simplicity of this method any other criterion will probably produce better results.

Velocity–Based Decision Resolution

To produce better collision detection results more complex solutions have to be considered. First of all, the maximal approximation error is derived. If no dead

reckoning is used or used with an error–based threshold of 0, the maximal approximation error is $dist_{max} = |velocity \cdot latency|$. $dist_{max}$ is the distance an object can move until the message arrives at the remote host at the current velocity. Of course, this implies that the velocity is constant, otherwise the maximal error is $dist_{max} = |velocity \cdot latency + 0.5 \cdot acceleration \cdot latency^2|$. I.e. the higher the latency and/or the velocity is, the bigger will be the approximation error.

This fact can be utilized for deciding which host is responsible for collision detection, since the velocity of remote objects is known. If the host of the object with the higher velocity is chosen the collision detection result will be in general more exact, since the other host will have a higher approximation error for the object. There are some drawbacks of this method, the type of movement is not taken into account and due to network latency, different velocities for the same object can exist at the two hosts. Therefore, if the velocity is chosen as the criterion, a velocity at time $t = -\delta$ has to be used, and δ has to be greater than any possible network latency. Thus this criterion can only be used for reasonable values of δ. To take the type of motion into account, a position history has to be used, such as used in the position history–based protocol. The usage of δ applies as well.

Field of View–Based Decision Resolution

One reason, why better collision detection results are desirable, is that visual inconsistencies should be limited. In a distributed virtual environment with low latencies (less than 10 ms) and slow moving objects, e.g. with maximal velocity of 10 m/s then $dist_{max} = 10cm$ and the environment will be resynchronized after a collision within 10 ms. In this scenario the simple approach from above will have the desired effect, tolerable inconsistencies and no noticeable visual inconsistencies. This is not the common case, therefore a criterion for decision resolution can be the reduction of visual inconsistencies. Since collision detection is performed with the information about the enslaved and the remote object of the deciding host, the user will not suffer from visual inconsistencies. Therefore, the field of view of the users can be used. E.g. in a distributed racing game, the player in the car in front, A, only sees the other car of player B through his mirrors. Collisions with his rear–side and the front–side of the following car should be performed on the basis of the information residing at the host of player B. It also has to be ensured that both host are doing the decision resolution on the same data, therefore the field of view at time $t = -\delta$ has to be used.

Summary

This subsection has shown that decision resolution can be simple and easy if the identification number is used. Other criteria will result in better collision detection results or will reduce visual inconsistencies. The usefulness of the different criteria will not be discussed here, since this is an application specific question. More feasible criteria should be derived from an application's requirements with no problem. Even the combination of several criteria for different situation can be applicable if they can guarantee the unambiguity of the result. Additionally, if no decision can be resolved by a criterion, e.g. both objects have the same velocity, at last the identification number can still be used to provide the unambiguity.

4.5 Conclusions

In this chapter all aspects of collision detection in distributed virtual environments have been explained and discussed. Especially with the focus on network latency and bandwidth limitation. The *Consistency–Throughput Tradeoff* has been stated and proven.

The problems of network latency prevent exact collision detection in interactive distributed virtual environments and the effect is aggravated by techniques to reduce network traffic. Therefore two approaches have been provided along with experimental results showing how to find a trade–off between high update rates to support collision detection and the reduction of network traffic. The approaches can be easily integrated into methods for remote object visualization, including dead reckoning and the position history–based protocol. By distributing update messages at higher rates only if two enslaved objects are in close proximity and a collision becomes more likely, the approach of adaptive collision prediction tracking is able to reduce the approximation error to a minimum when required.

Additionally, the problem of decision resolution has been outlined and criteria have been provided to solve this problem.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The design and the implementation of the process of collision detection in virtual environments have combined the best procedures of existing as well as new approaches for this work. To achieve that, simulations and experiments have been performed and analyzed. The design of the collision detection process has been focused on the integration into a larger application, i.e. all design decisions have been made for a wider context and not solely for collision detection.

The resulting implementation has been integrated into a multi–user virtual environment system in an exemplary way. Experiments have shown that the implementation meets the demanded requirements, namely no restriction of the input data, fast collision detection queries and exact collision detection results. Collision detection queries of complex objects within close proximity take less than 1 ms on average. The robustness of the design and the implementation are important for the purpose of collision detection in distributed virtual environments.

The problems of collision detection in distributed virtual environments have been pinpointed, namely network latency and limitation of available bandwidth. Most frameworks for distributed virtual environments utilize techniques for remote object visualization to reduce the network traffic, which aggravate the effect of network latency. Two new approaches have been provided to collaborate with these techniques. The results show that a trade–off can be found between low bandwidth usage and high update rates to support collision detection in distributed virtual environments. The approaches *remote collision prediction* and *adaptive collision prediction tracking* are able to reduce the approximation error of remote objects significantly after collisions have occurred and when two enslaved objects come into close proximity, respectively.

Additionally, the problem of decision resolution has been outlined and exem-

plary solutions have been provided to produce better collision detection results. The provided solutions are different in the terms of reducing visual inconsistencies or providing the collision detection result with the smaller approximation error.

5.2 Future Work

The dependence on long preprocessing times limits the ability of an application to insert new objects into the scene at real–time and to change the object's geometry. The creation of a bounding volume hierarchy upon request can possibly reduce this drawback. The hierarchy would then only be created as deeply as needed. The aspects of flexible objects and dynamic scenes are separate research topics. Therefore, additional effort has to be made to integrate these features into the approach.

The collision detection in distributed virtual environments determines the close proximity by utilizing bounding spheres. Poorly approximated objects have a large close proximity area and therefore raise the required bandwidth usage unnecessarily. Subdivision of the object to provide a better approximation of an object to reduce the close proximity area should be very helpful.

Also, collisions between remote objects, which are not in the field of view, do not have to be detected by each client. A client becomes aware of the collision by update messages. Therefore, a strategy to decide for which objects collision detection has to be performed should be added to the system.

Quality of Service (QoS), which is part of the new internet standard IPv6, is able to provide a known upper–bound on latencies and guaranteed bandwidth [Hin96]. With this new service completely new approaches are possible, since these approaches can rely on fixed circumstances and therefore are able to ignore possible network congestions.

Appendix A

Experiments

This appendix contains the diagrams for the experiments and simulations described in Section 4.4. Figures A.1 to A.4 show the diagrams for the test scenario without applying the new approaches remote collision prediction and adaptive collision prediction tracking, only dead reckoning is used. In each topmost diagram of the figures, the center's x-component of the enslaved and the remote object is shown at time t, t corresponds to a unique z value. z moves from 0 to -40π in 15 seconds.

Figures A.5 to A.8 show the diagrams for the test scenario, while the remote collision prediction was being applied.

Figures A.9 to A.12 show the diagrams for the test scenario, while the remote collision prediction and the adaptive collision prediction tracking were being applied.

The scales of the y-axes of the diagrams showing the approximation error at time t, are chosen to be unique for the identical latencies. Different scales for different latencies have been chosen to clarify the provided information.

The diagrams at the bottom of each figure show the output rate of the simulation at time t. Since it is measured in KByte/sec, block diagrams are used. The values at time $t = 0$ and $t = 15$ seem to be incorrect in some diagrams, e.g. in Figure A.8 the output rate is 0.64 KByte/sec at $t = 0$, but only one package was sent. This is due to the fact that the simulation had been run by a loop. Therefore some packages which have been sent at times $t < 0$ and at times $t > 15$, have, however, been counted for this simulation loop. Since the important sections of these diagrams are not at time $t = 0$ and at time $t = 15$, no effort has been made to eliminate this effect.

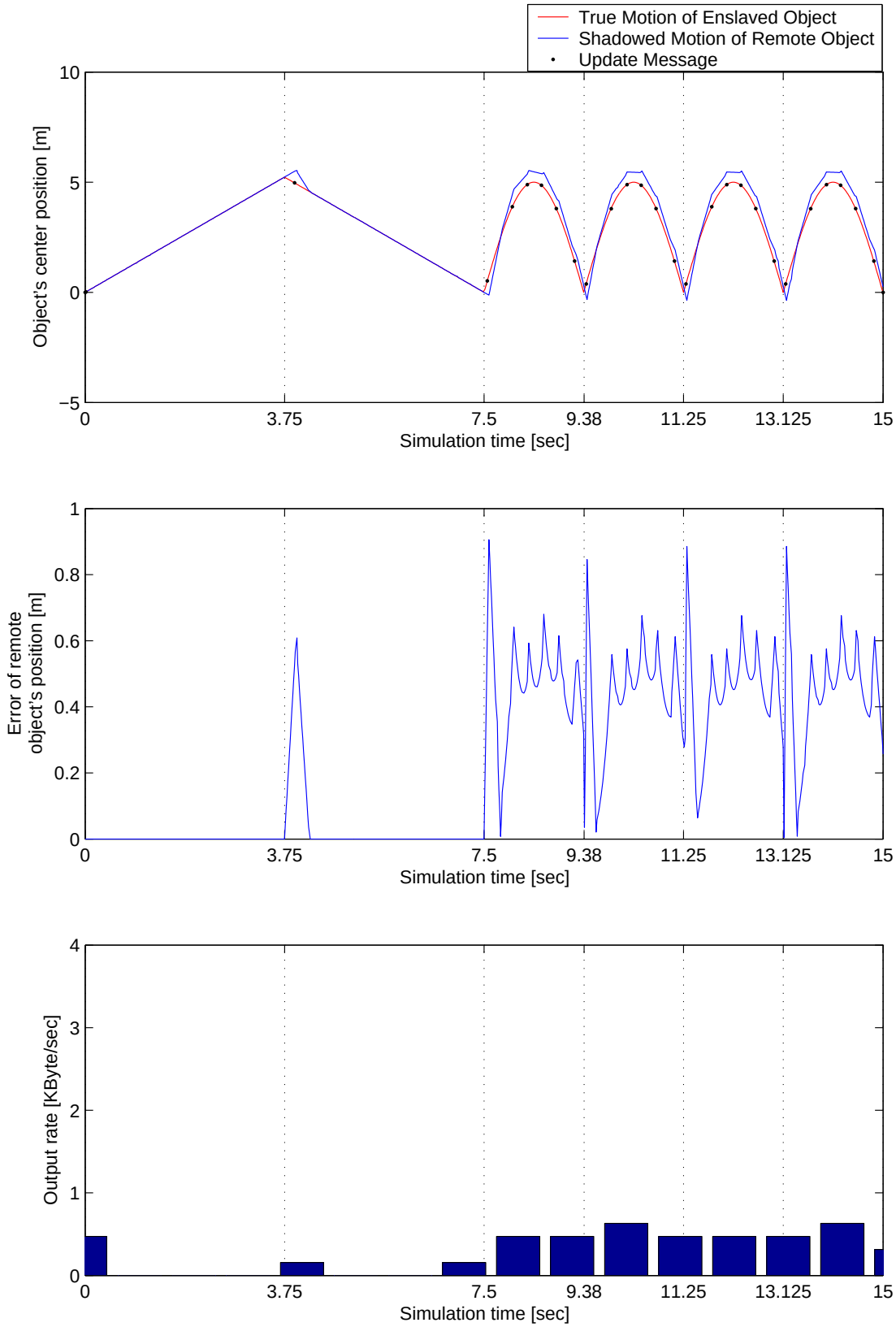


Figure A.1: Test scenario 1 with less than 10 ms network (error–based threshold 0.5 m, time–based threshold 5 sec).

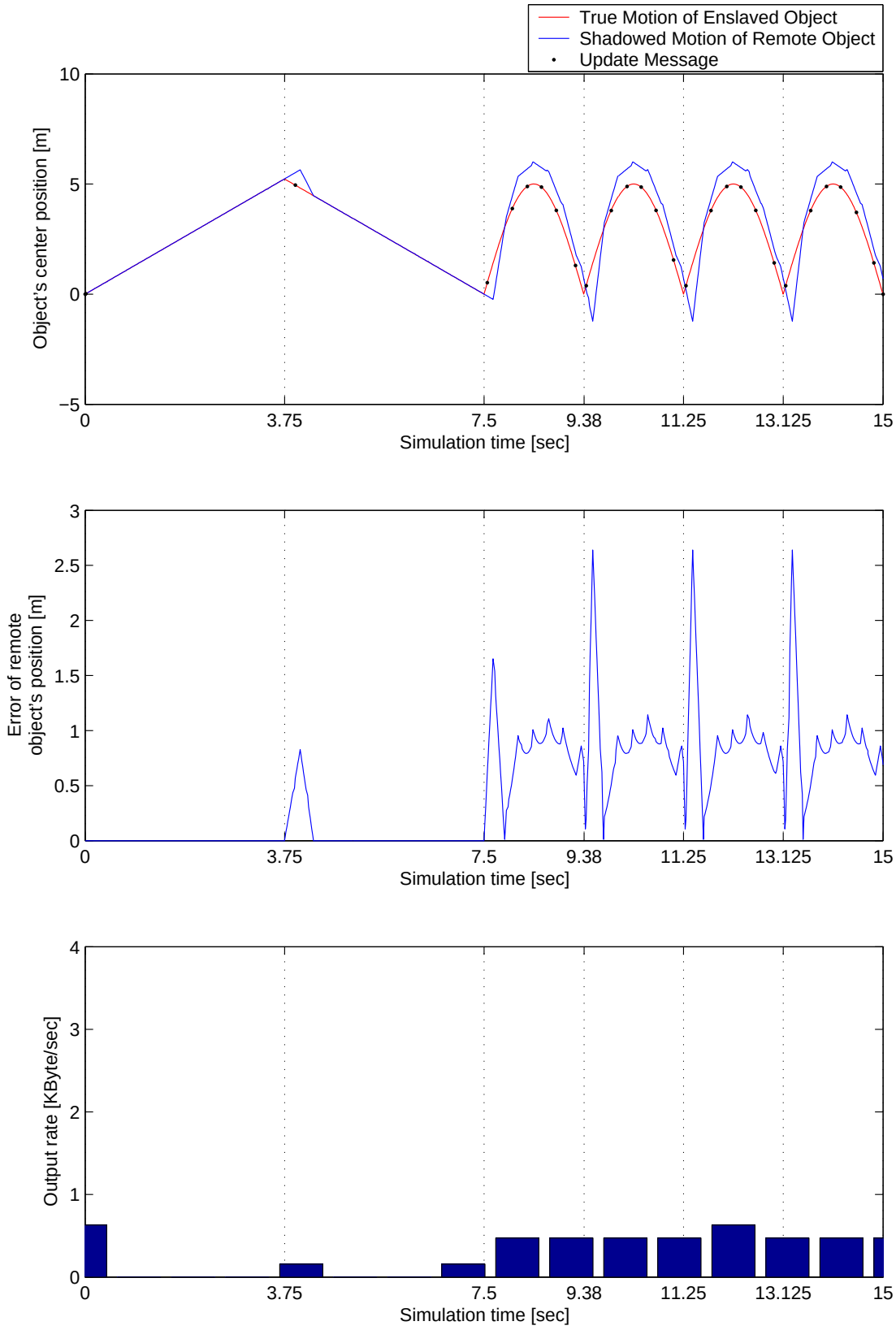


Figure A.2: Test scenario 1 with 100 ms network (error–based threshold 0.5 m, time–based threshold 5 sec).

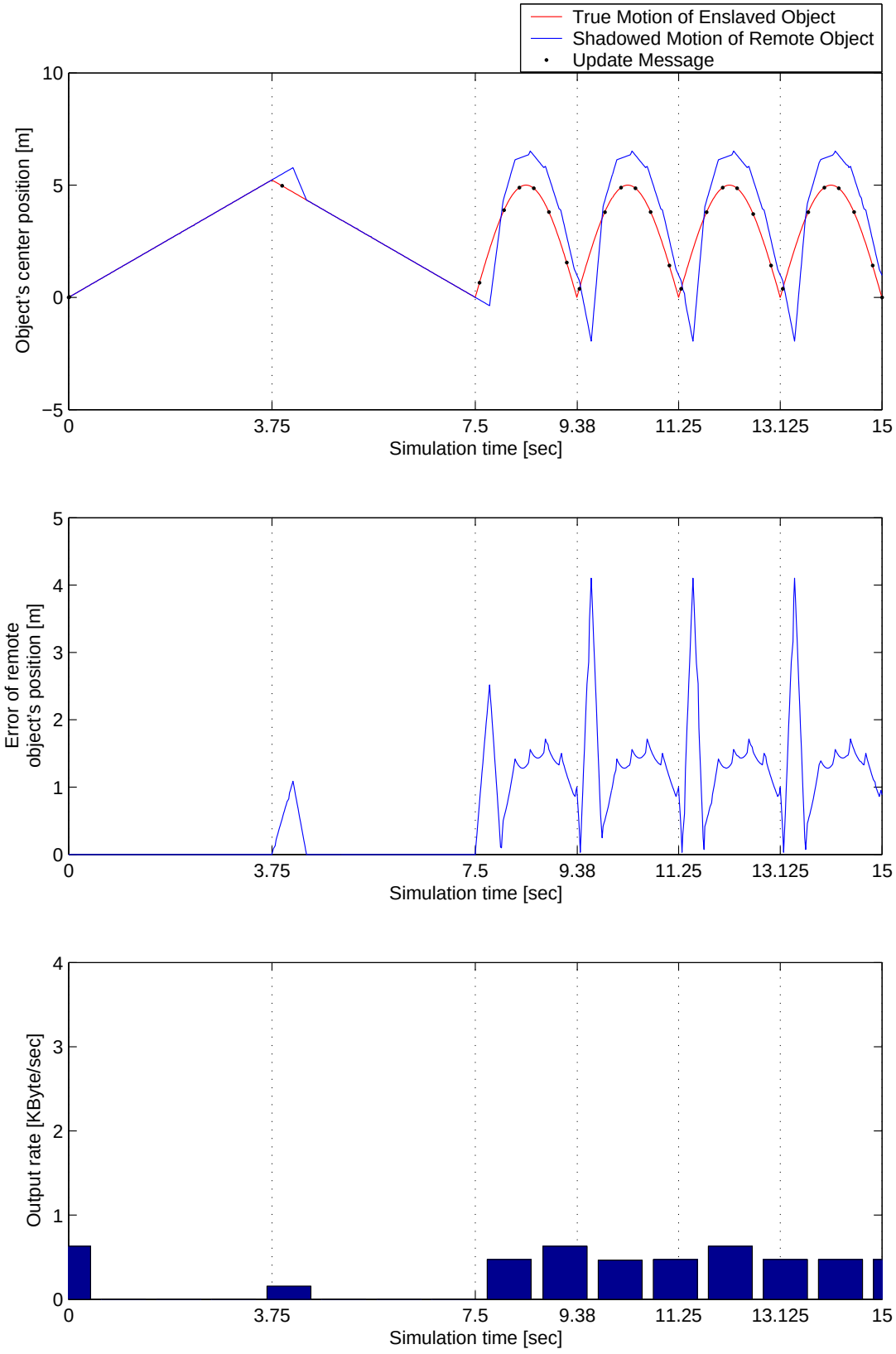


Figure A.3: Test scenario 1 with 200 ms network (error–based threshold 0.5 m, time–based threshold 5 sec).

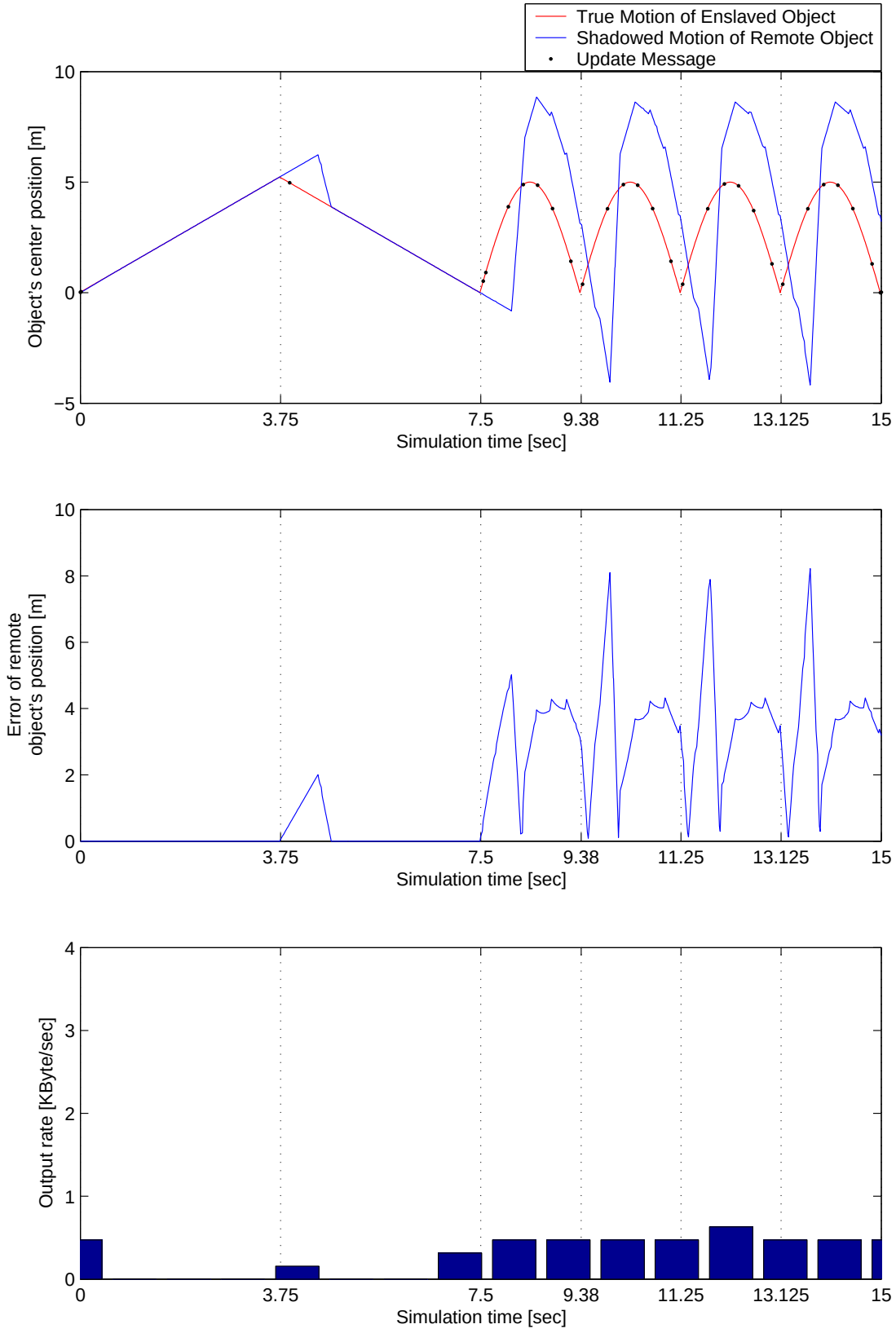


Figure A.4: Test scenario 1 with 500 ms network (error–based threshold 0.5 m, time–based threshold 5 sec).

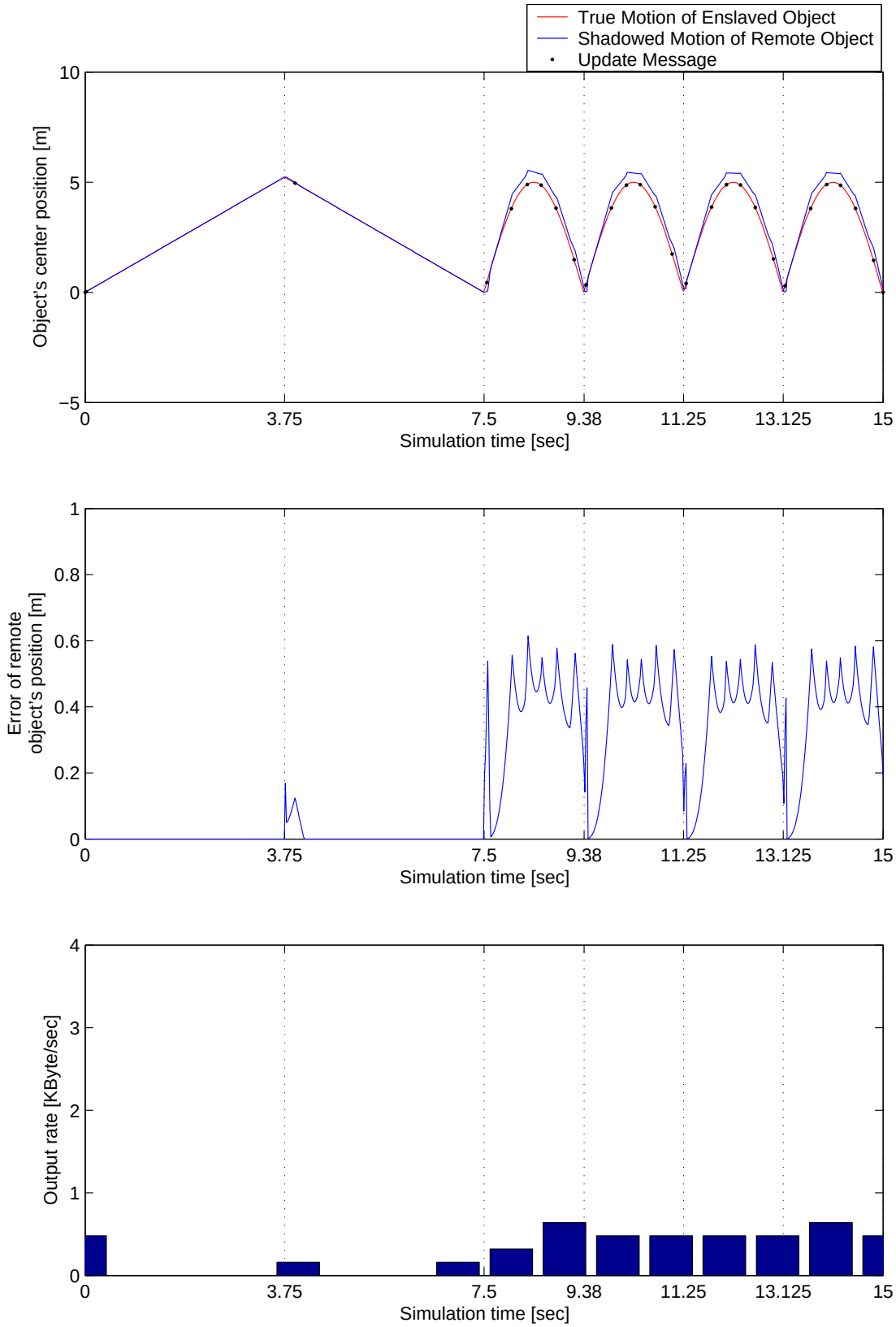


Figure A.5: Test scenario 1 with remote collision prediction (< 10 ms latency, error–based threshold 0.5 m, time–based threshold 5 sec).

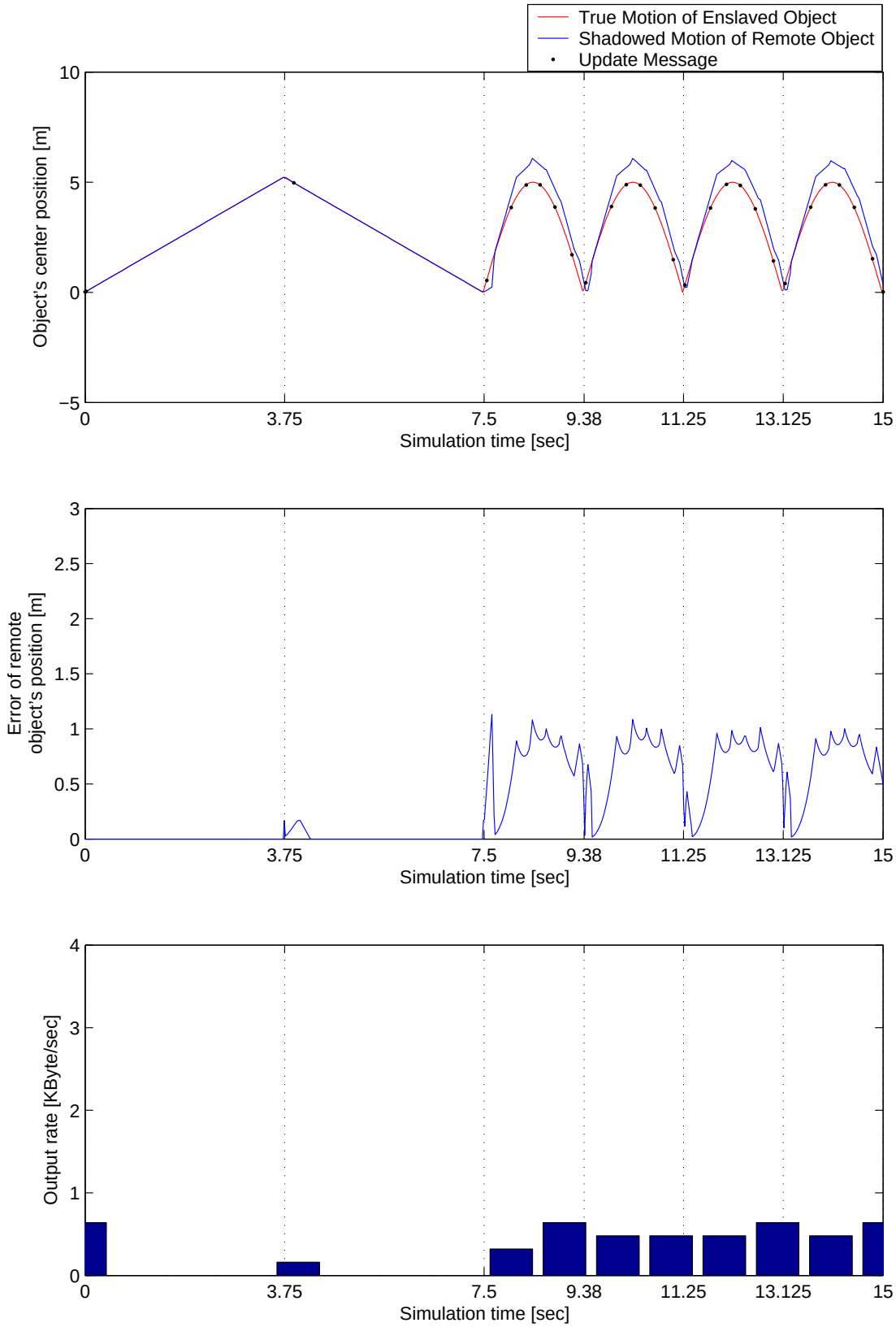


Figure A.6: Test scenario 1 with remote collision prediction (100 ms latency, error–based threshold 0.5 m, time–based threshold 5 sec).

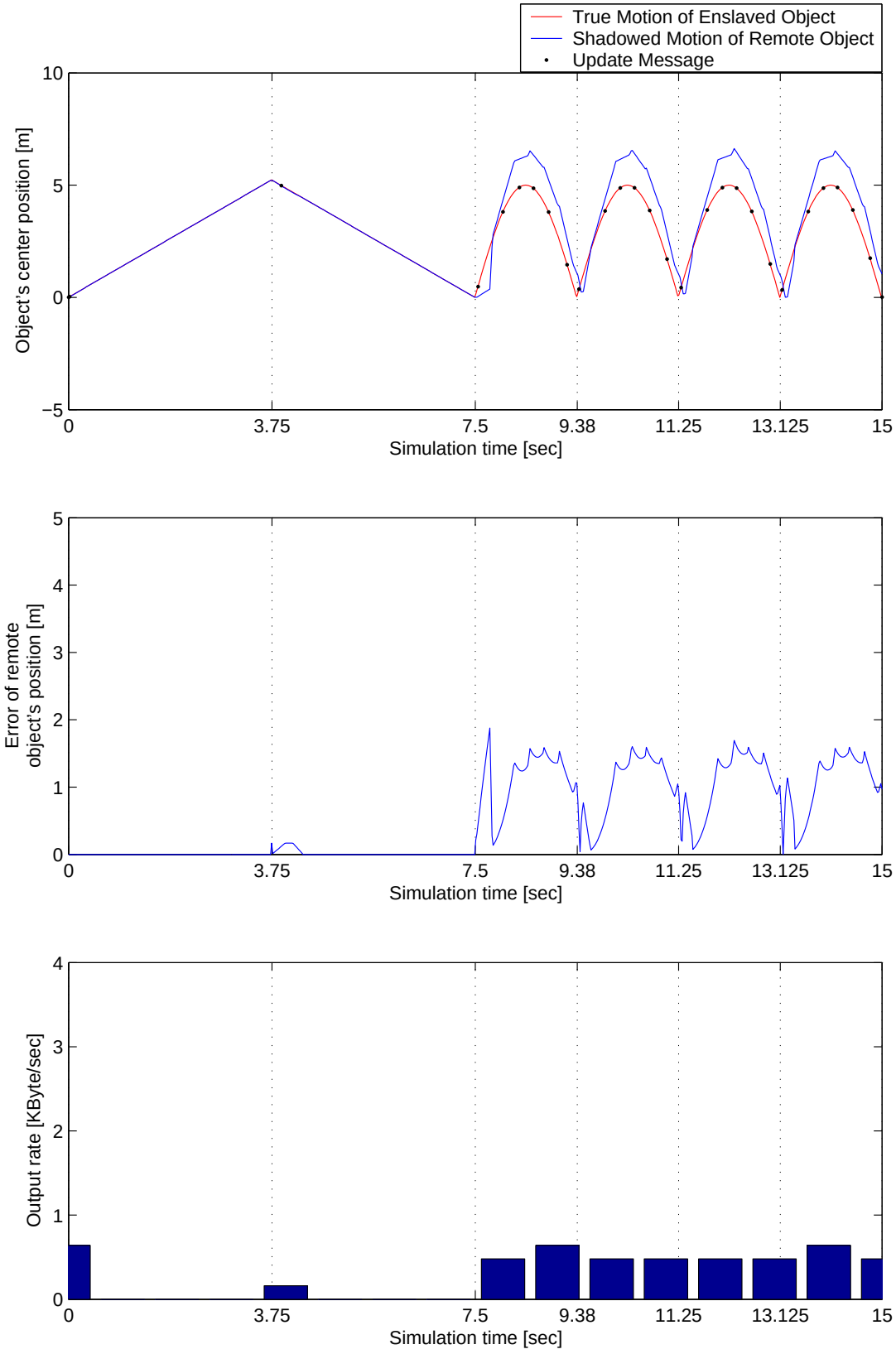


Figure A.7: Test scenario 1 with remote collision prediction (200 ms latency, error–based threshold 0.5 m, time–based threshold 5 sec).

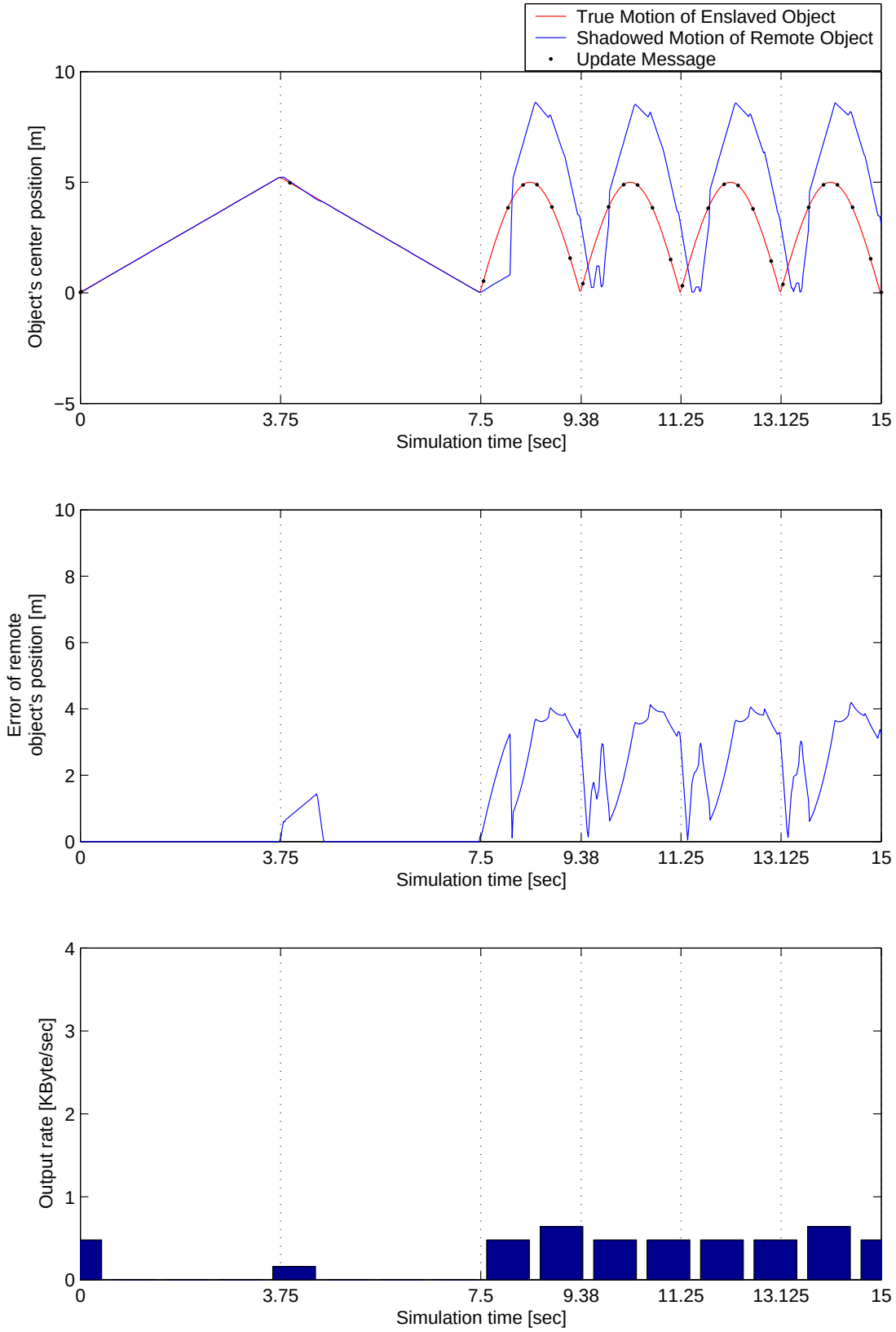


Figure A.8: Test scenario 1 with remote collision prediction (500 ms latency, error–based threshold 0.5 m, time–based threshold 5 sec).

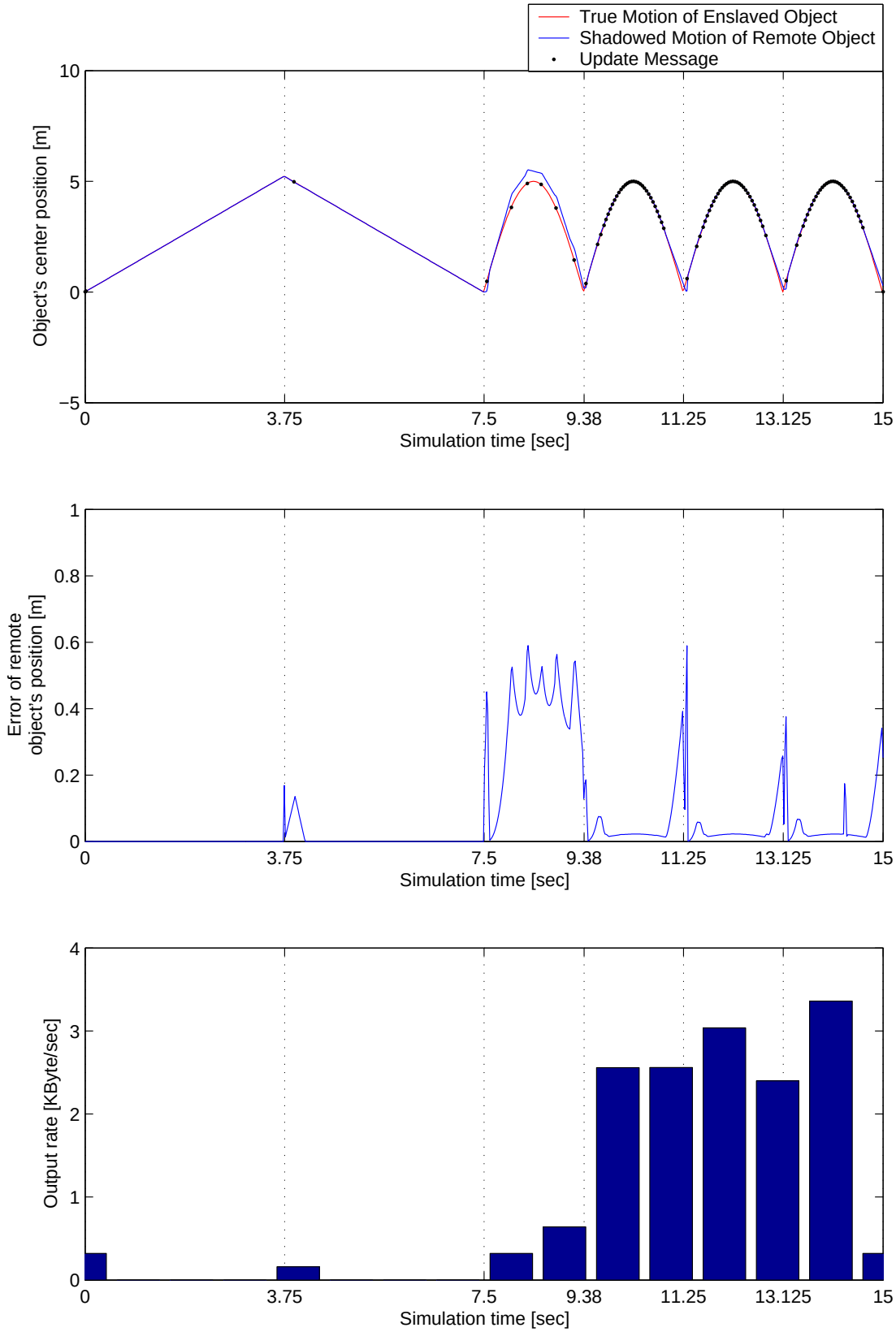


Figure A.9: Test scenario 1 with adaptive collision prediction tracking ($<$ 10 ms latency, error–based threshold 0.5 m, close proximity threshold 0.01 m, time–based threshold 5 sec).

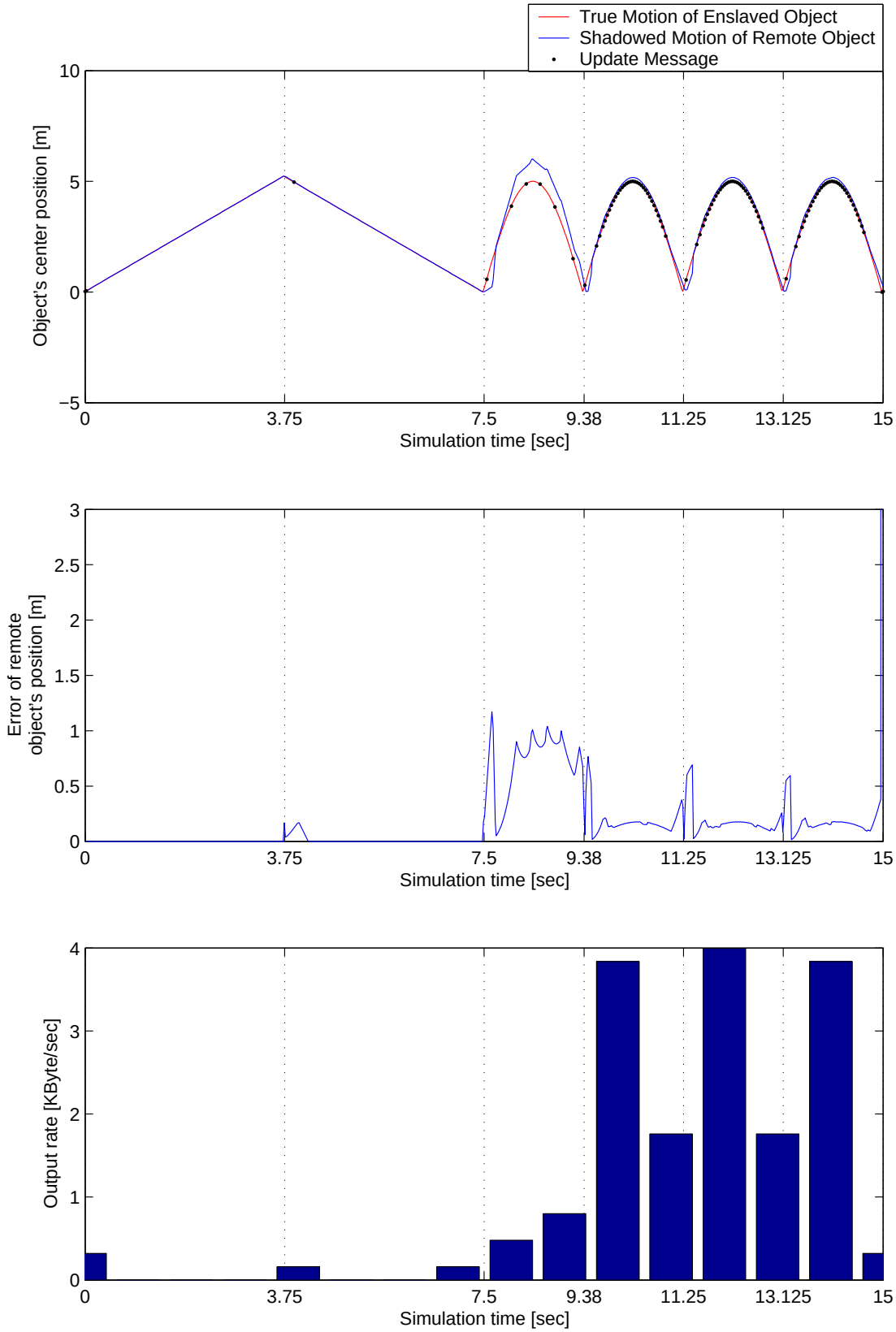


Figure A.10: Test scenario 1 with remote collision prediction (100 ms latency, error–based threshold 0.5 m, close proximity threshold 0.01 m, time–based threshold 5 sec).

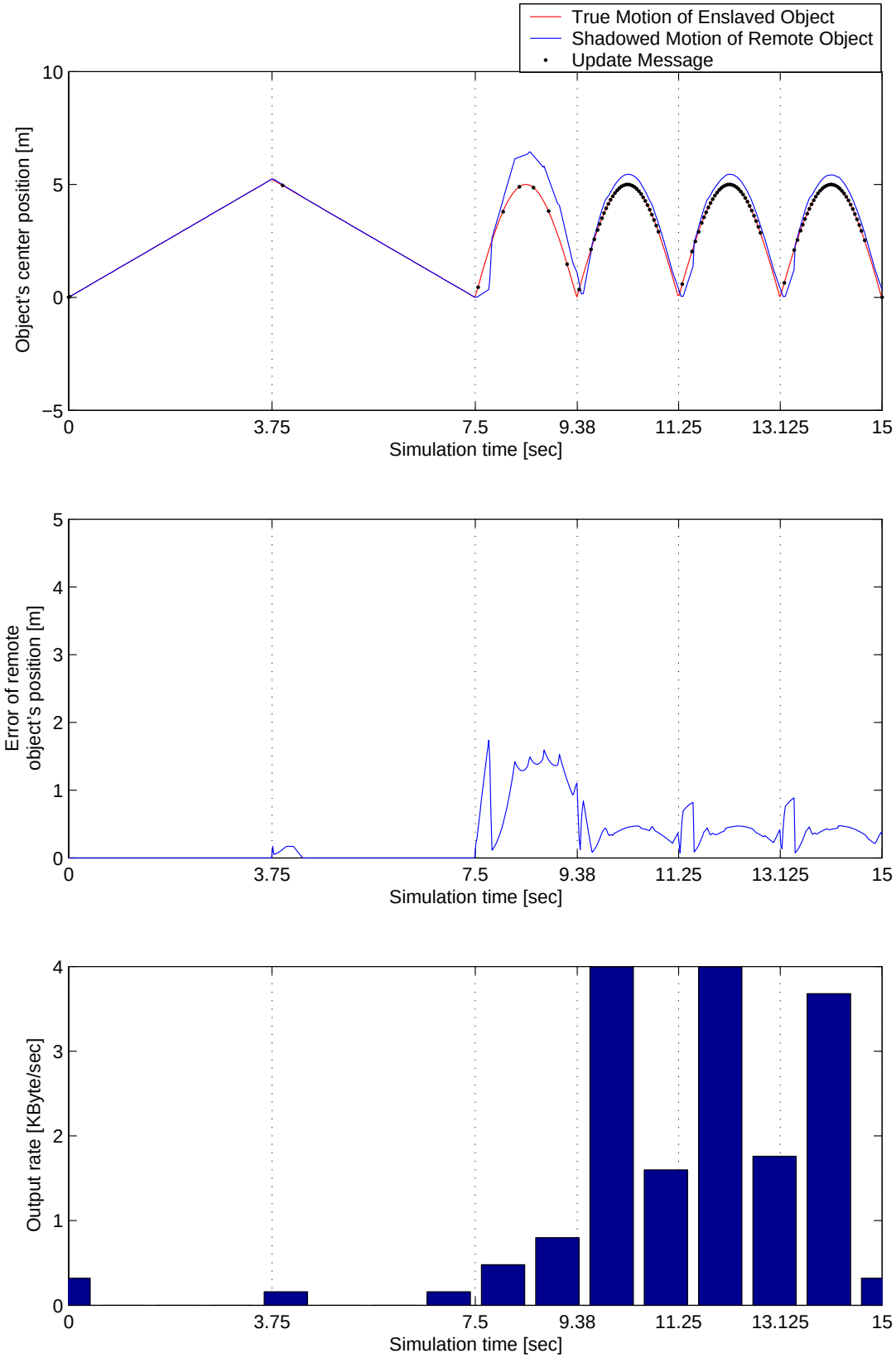


Figure A.11: Test scenario 1 with remote collision prediction (200 ms latency, error–based threshold 0.5 m, close proximity threshold 0.01 m, time–based threshold 5 sec).

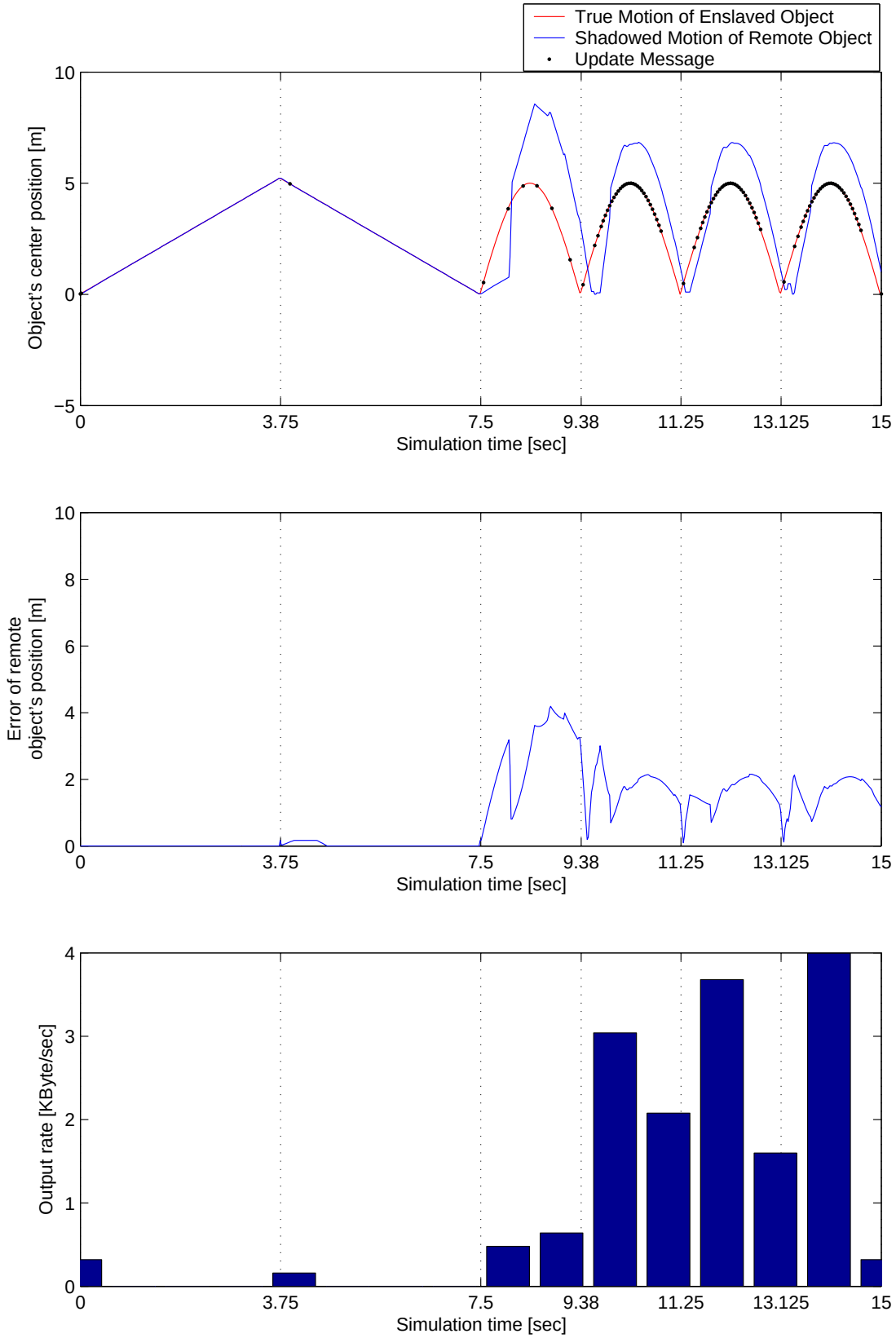
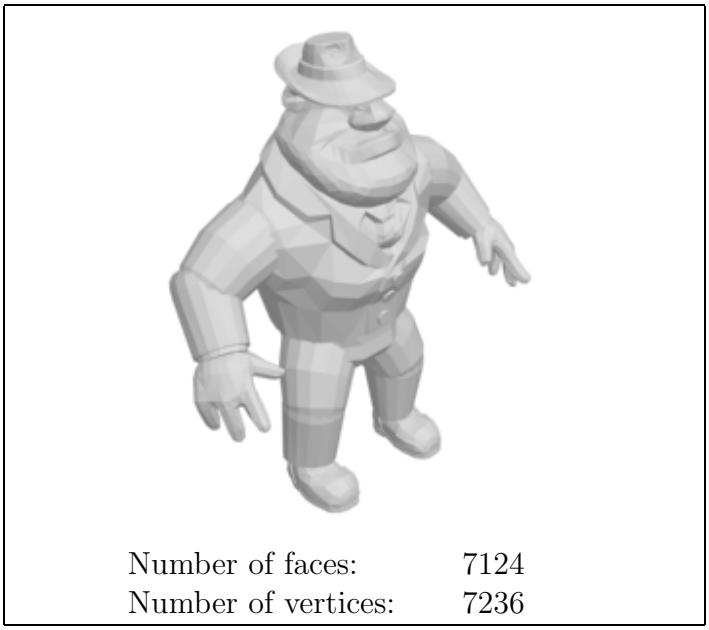


Figure A.12: Test scenario 1 with remote collision prediction (500 ms latency, error–based threshold 0.5 m, close proximity threshold 0.01 m, time–based threshold 5 sec).

Appendix B

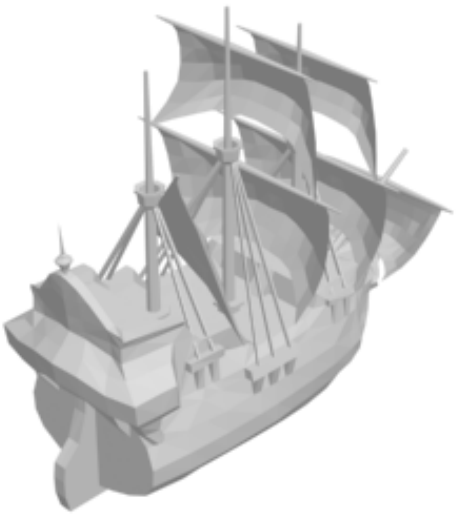
Sample Objects and Scenes

The following eight images are the sample objects used for the experiments in Chapter 3, at the bottom of each picture the number of faces and the number of vertices is provided. At the end of this chapter, screenshots are provided, which are referenced in Chapter 3 and Chapter 4.



Number of faces:	7124
Number of vertices:	7236

Figure B.1: Al.wrl



Number of faces:	4698
Number of vertices:	2372

Figure B.2: Galleon.wrl

Number of faces:	14978
Number of vertices:	9213

Figure B.3: 3000gt.wrl

Number of faces:	2614
Number of vertices:	7842

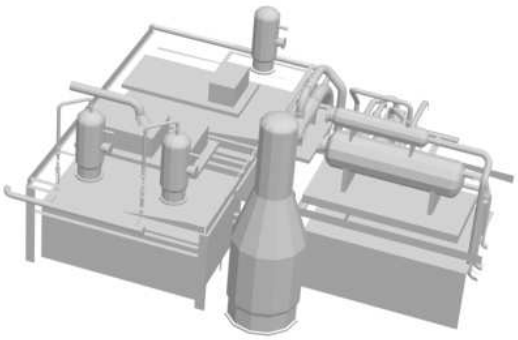
Figure B.4: Mustang.wrl

Number of faces:	17327
Number of vertices:	8801

Figure B.5: Eagle.wrl

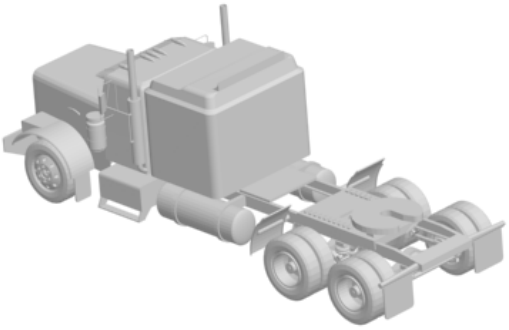
Number of faces:	46364
Number of vertices:	116738

Figure B.6: Tree.wrl



Number of faces:	38146
Number of vertices:	20634

Figure B.7: Piping.wrl



Number of faces:	47805
Number of vertices:	24862

Figure B.8: Truck.wrl

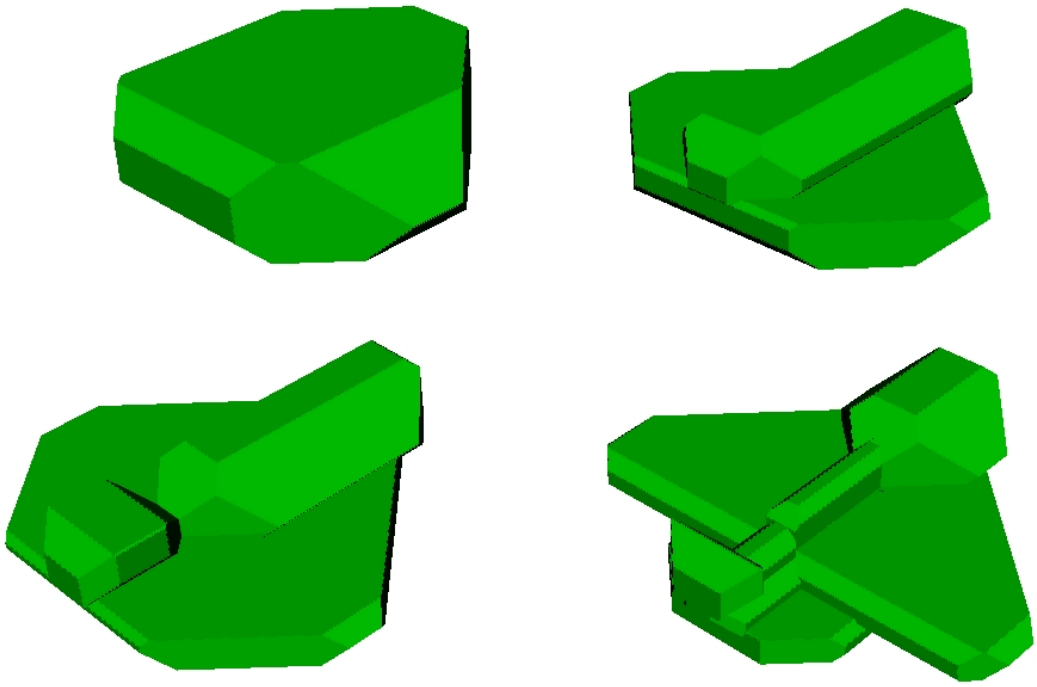


Figure B.9: Bounding volume for Mustang.wrl. The top level figure shows the BVH of Mustang.wrl at level 0 (the root), the top right, the bottom left and the bottom right figures show the hierarchy at levels 1, 2 and 3, respectively.

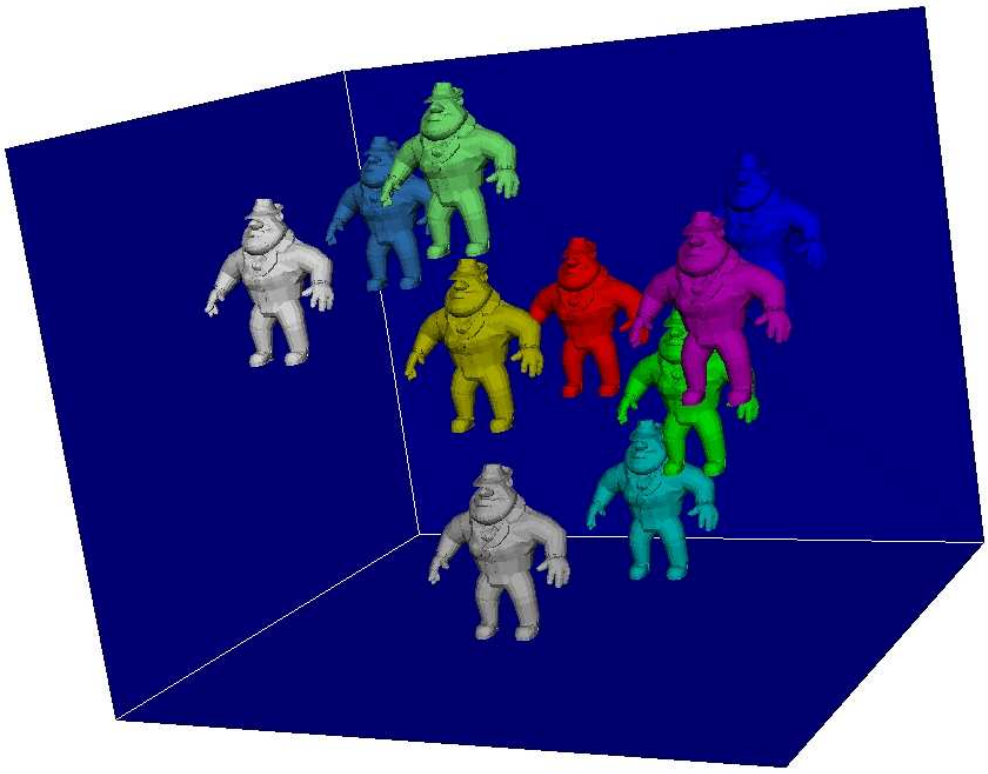


Figure B.10: Ten_Als.wrl

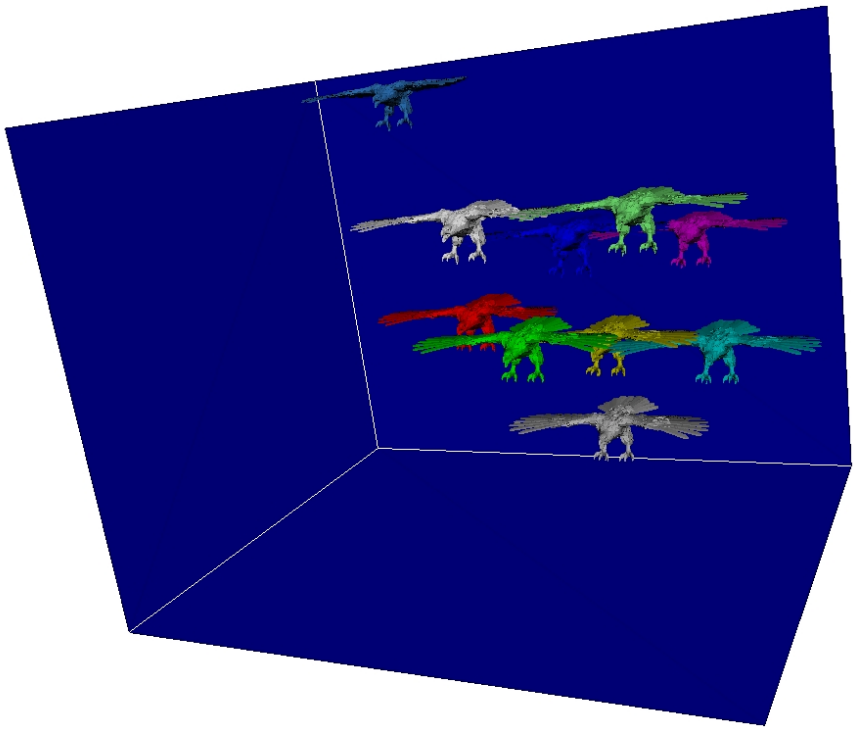


Figure B.11: Ten_Eagles.wrl

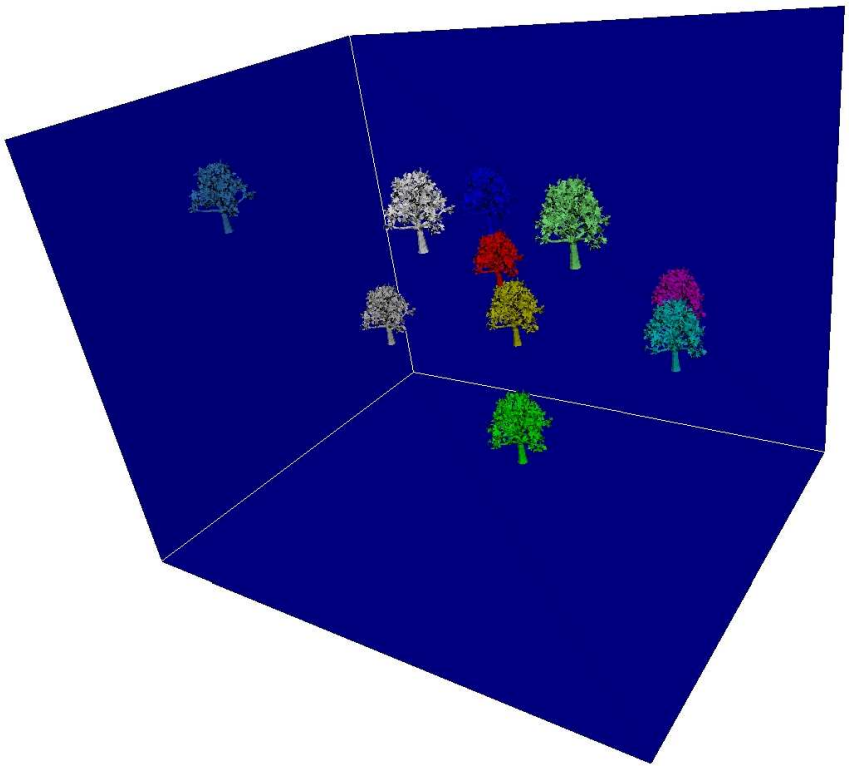


Figure B.12: Ten_Trees.wrl

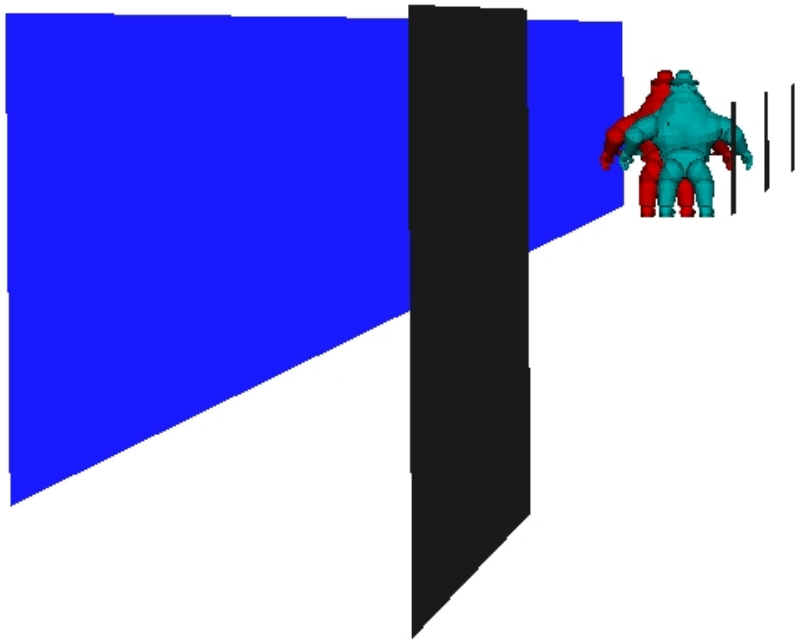


Figure B.13: Screen shot of the test scenario, the red object is enslaved and the cyan object is remote.

Appendix C

VRML97 Extensions

The following extensions to the VRML 97 specification have been implemented.

C.1 CollisionInterest

CollisionInterest {
    field SFString **objectPath** ""
    field MFString **interests** [ ]
    field SFInt32 **collisionType** **0** # CDT_BOX
}

The CollisionInterest node registers a node, specified by *objectPath*, for collision detection with the nodes specified by *interests*.

The *collisionType* defines the type of collision detection, which is performed. The following values are supported:

0:	CDT_BOX	Collisions are detected by intersecting the bounding boxes of a pair of nodes.
1:	CDT_WITNESS	Collisions are detected by testing the polygons for intersection of a pair of nodes. Reports only the first pair of intersecting polygons.
2:	CDT_ALLPAIRS	Collisions are detected by testing the polygons for intersection of a pair of nodes. Reports all pairs of intersecting polygons.

A node can be registered for collision detection by a CollisionInterest node more than once. E.g. a node can be interested in collision detection with objects A and B with *collisionType* CDT_WITNESS and with object C with *collisionType*

CDT_BOX.

C.2 CollisionResponse

CollisionResponse {
    field SFInt32 **responseType** **0** # {CRT_INVERSE, CRT_ZERO}
    field SFBool **setBack** **TRUE**
}

The CollisionResponse node specifies the type of action in the case of a collision for its parent node.

responseType defines the type of action and can be any of the following values:

0:	CRT_INVERSE	The velocity vector of the parent Transform node and the face normal of the polygon of the collidee is used to simulate a bouncing effect. The collision response is undefined for collision type CDT_BOX.
1:	CRT_ZERO	The velocity vector of the parent Transform node is set to zero.

If *setBack* is TRUE (the default), the translation of the parent Transform node is set back to the last translation at which no collision occurred.

C.3 KeySensor

KeySensor {

field	SFBool	**absoluteValue**	**FALSE**
field	SFBool	**consume**	**TRUE**
exposedField	SFBool	**enabled**	**TRUE**
field	SFInt32	**keyState**	**0**
field	SFInt32	**keyValue**	**0**
field	SFString	**recipient**	**""**
field	SFVec3f	**value**	**0 0 0**
eventIn	SFBool	**disable**	
eventOut	SFBool	**keyHit**	

}

The KeySensor node processes keyboard events. If *keyValue* and *keyState* are equal to the keyboard codes sent to this node, the velocity vector of the recipient

is changed by *value*. The KeySensor can be enabled or disabled by setting *enabled* to TRUE or FALSE, respectively.

If *absoluteValue* is TRUE, the velocity value of *recipient* is set to *value*. Otherwise, *value* is added to the velocity vector.

If *consume* is TRUE, the keyboard event is consumed and no any other KeySensor node gets this keyboard event nor does the application. Otherwise, other nodes can process the keyboard event.

keyState and *keyValue* define the keyboard event. The values are identical to the keyboard codes of Trolltech Qt's keyboard codes, namely Qt::Key and Qt::ButtonState.

recipient specifies the address of the recipient node.

C.4 Shape

Shape {

exposedField	SFNode	**appearance**	**NULL**
field	MFVec3f	**convexHull**	[ ]
field	SFNode	**geometry**	**NULL**
field	SFString	**hierarchyFile**	**""**
field	SFInt32	**splitThreshold**	**10**

}

For *appearance* and *geometry*, refer to VRML97 specification.

convexHull defines a set of 3D coordinates, which form the convex hull of the *geometry*. If specified, the *convexHull* is used to calculate a new bounding k-dop for this node after a transformation had been applied. Otherwise, the boundaries of the original k-dop are used for this calculation.

If *hierarchyFile* is specified and the file exists, the hierarchy for the geometry is read from the file. If the file does not exit, the hierarchy is created and written to the file.

splitThreshold defines the threshold, which is used during the creation of the k-dop hierarchy for *geometry*. A node of this hierarchy is split if the number of polygons is larger than the *splitThreshold*. In the case the file is successfully read from the file specified by *hierarchyFile*, the value of *splitThreshold* is ignored.

C.5 Transform

Transform {

eventIn	MFNode	addChildren		
eventIn	MFNode	removeChildren		
exposedField	SFVec3f	**center**	**0 0 0**	# ($-\infty$,∞)
exposedField	MFNode	**children**	**NULL**	
exposedField	SFNode	**collisionResponse**	**NULL**	
exposedField	SFFloat	**distanceThreshold**	**0.5**	# [0, ∞)
exposedField	SFFloat	**enslaved**	**FALSE**	
exposedField	SFFloat	**remote**	**FALSE**	
exposedField	SFRotation	**rotation**	**0 0 1 0**	# [-1,1],($-\infty$,∞)
exposedField	SFVec3f	**scale**	**1 1 1**	# (0,∞)
exposedField	SFRotation	**scaleOrientation**	**0 0 1 0**	# [-1,1],($-\infty$,∞)
exposedField	SFVec3f	**translation**	**0 0 0**	# (($-\infty$,∞)
exposedField	SFVec3f	**velocityVector**	**0 0 0**	# (($-\infty$,∞)
field	SFVec3f	**bboxCenter**	**0 0 0**	# ($-\infty$,∞)
field	SFVec3f	**bboxSize**	**-1 -1 -1**	# (0,∞)
				# or -1, -1, -1
eventOut	SFBool	**notColliding**		

}

For *addChildren*, *removeChildren*, *center*, *children*, *rotation*, *scale*, *scaleOrientation*, *translation*, *bboxCenter* and *bboxSize*, refer to VRML97 specification.

If *collisionResponse* is not NULL, it shall contain a CollisionResponse node (Section C.2), which specifies how the node should behave in the case of collisions. The collisionResponse is only used, if the object is registered for collision detection.

If *enslaved* is TRUE, the exposedField *distanceThreshold* defines the error–based threshold for the dead reckoning technique.

If *remote* is TRUE, the dead reckoning technique converges to a point in the future to allow smooth animation of remote objects.

If the *velocityVector* is not (0, 0, 0), it specifies the linear velocity of the object in meters per seconds. If Δ is the time since the last frame than the new translation is:

$$translation = translation + velocityVector * \Delta \tag{C.1}$$

notColliding events output a boolean flag indicating if the Transform node is colliding with an object.

References

[AMH02] T. Akenine-Möller and E. Haines. *Real–Time Rendering*. A K Peters, Ltd., 3^{rd} edition, 2002.

[BDH96] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[Ber01] Y. W. Bernier. Latency Compensating Methods in Client/Server Ingame Protocol Design and Optimization. In *Proceedings of the 15th Games Developers Conference*, 2001.

[BKS01] D. Bartz, J. T. Klosowski, and D. Staneker. k-DOPs as Tighter Bounding Volumes for Better Occlusion Performance. *In Proc. of ACM Siggraph*, page 213, 2001.

[Bou02] D. M. Bourg. *Physics for Game Developers*. O'Reilly, 2002.

[BP99] W. Broll and W. Prinz. Using 3D to Support Awareness in Virtual Teams on the Web. *Proceedings of WebNet 99 – World Conference on the WWW and the Internet*, 1:137–142, 1999.

[Bro98] W. Broll. Smalltool – a toolkit for realizing shared virtual environments on the Internet. *Distributed Systems Engineering*, 5(3):118–128, 1998.

[CLMP95] J. Cohen, M. Lin, D. Manocha, and K. Ponamgi. I–COLLIDE: An Interactive and Exact Collision Detection System for Large–Scaled Environments. *In Proc. of ACM Int. 3D Graphics Conference*, pages 189–196, 1995.

[Ebe01] D. H. Eberly. *3D Game Engine Design*. Morgan Kaufmann Publishers, 2001.

[EGJ93] R. A. Earnshaw, M. A. Gigante, and H. Jones. *Virtual Reality Systems*. Academic Press Ltd, 1993.

[FS98] E. Frécon and M. Stenius. DIVE: a scalable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(4):91–100, 1998.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: a hierarchical structure for rapid interference detection. *In Proc. of ACM Siggraph*, pages 171–180, 1996.

[Hai94] E. Haines. *Graphics Gems IV*. Academic Press, 1994.

[Hin96] Robert M. Hinden. IP next generation overview. *Communications of the ACM*, 39(6):61–71, 1996.

[HLC+97] T. C. Hudson, M. C. Lin, J. Cohen, S. Gottschalk, and D. Manocha. V–COLLIDE: Accelerated Collision Detection for VRML. *In Proc.* 2^{nd} *Annual Sympos. on Virtual Reality Modeling Language*, 1997.

[Hub95] P. M. Hubbard. Collision Detection for Interactive Graphics Applications. *In IEEE Transactions on Visualization and Computer Graphics*, pages 218–230, 1995.

[Kal93] R. S. Kalawsky. *The Science of Virtual Reality and Virtual*. Addison–Wesley, 1993.

[KK86] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. *In Proc. of Computer Graphics (SIGGRAPH)*, 20(4):269–278, 1986.

[Klo98] J. T. Klosowski. *Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments*. PhD thesis, State University of New York at Stony Brook, 1998.

[KR02] J. F. Kurose and K. Ross. *Computer Networking: A Top–Down Approach Featuring the Internet*. Addison–Wesley, 2^{nd} edition, 2002.

[Len02] E. Lengyel. *Mathematics for 3D Game Programming & Computer Graphics*. Charles River Media, 2002.

[LMCG96] M. C. Lin, D. Manocha, J. Cohen, and S. Gottschalk. Collision Detection: Algorithms and Applications. *In Proc. of Algorithms for Robotics Motion and Manipulation*, pages 129–142, 1996.

[MBP99] E. Meier, W. Broll, and W. Prinz. Augmenting Cooperative Settings by Shared Awareness Spaces. *Proceedings of the IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 84–89, 1999.

[McC98] S. R. McCanne. Scalable Multimedia Communication with Internet Multicast, Light-weight Sessions, and the MBone. Technical Report CSD-98-1002, University of California, Berkeley, 1998.

[MW88] M. Moore and J. Wilhelms. Collision Detection and Response for Computer Animation. *In Proc. of Computer Graphics (SIGGRAPH)*, 22(4):289–298, 1988.

[MZP+94] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz. NPSNET: A Network Software Architecture for Large-Scale Virtual Environment. *Presence*, 3(4):265–287, 1994.

[NAT90] B. Naylorm, J. Amanatides, and W. Thibault. Merging BSP Trees Yields Polyhedral Set Operations. *In Proc. of Computer Graphics (SIGGRAPH)*, 24(4):115–124, 1990.

[NPC+96] H. Noser, I. S. Pandzic, T. K. Capin, N. M. Thalmann, and D. Thalmann. Playing Games through the Virtual Life Network. In *ALIFE V*, pages 114–121, 1996.

[NSST00] H. Noser, C. Stern, P. Stucki, and D. Thalmann. Generic 3D Ball Animation Model for Networked Interactive VR Environments. In *Virtual Worlds*, pages 77–90, 2000.

[ODE89] *Oxford Advanced Learner's Dictionary of Current English*. Oxford University Press, 4^{th} edition, 1989.

[PHM98] S. Powers, M. Hinds, and J. Morphett. DEE: an architecture for distributed virtual environment gaming. *Distributed Systems Engineering*, 5(3):107–117, 1998.

[PPB01] W. Prinz and U. Pankoke-Babatz. Awareness of Cooperative Actvities in Mixed Realities. *Living in mixed realities*, pages 231–234, 2001.

[Pra93] D. R. Pratt. *A Software Architecture for the Construction and Management of Real–Time Virtual Worlds*. PhD thesis, Naval Postgraduate School, Monterey, CA, 1993.

[SC94] S. K. Singhal and D. R. Cheriton. Using a Position History–Based Protocol for Distributed Object Visualization. Technical Report CS-TR-94-1505, Stanford University, 1994.

[Sed90] R. Sedgewick. *Algorithms in C*. Addison–Wesley, 1990.

[SS96] P. D. Sandoz and P. M. Sharkey. Collision detection in distributed virtual worlds. 3^{rd} *UK Virtual Reality Special Interest Group Conference*, 1996.

[SZ99] S. K. Singhal and M. J. Zyda. *Networked Virtual Environments: Design and Implementation*. ACM Press, 1999.

[VRM97] Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML) – Part 1: Functional specification and UTF-8 encoding. Technical Report ISO/IEC 14772-1:1997, ISO/IEC, 1997.

[WNDS00] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide*. Addison–Wesley, 2000.

[Zac00] G. Zachmann. *Virtual Reality in Assembly Simulation – Collision Detection, Simulation Algorithms, and Interaction Techniques*. PhD thesis, Technische Universität Darmstadt, 2000.

[ZOMP93] M. J. Zyda, W. D. Osborne, J. G. Monahan, and D. R. Pratt. NPSNET: Real–Time Vehicle Collisions, Explosions and Terrain Modifications. *The Journal of Visualization and Computer Animation*, 4(1):13–24, 1993.