Young J. Kim*
Ewha Womans University
Seoul, Korea

Stephane Redon
INRIA Rhone-Alpes
Montbonnot, France

Ming C. Lin
Dinesh Manocha
University of North Carolina
at Chapel Hill
Chapel Hill, NC

Jim Templeman
Naval Research Laboratory
Washington, DC

Interactive Continuous Collision Detection Using Swept Volume for Avatars

Abstract

We present an interactive algorithm for continuous collision detection between a moving avatar and its surrounding virtual environment. Our algorithm is able to compute the first time of contact between the avatar and the environment interactively, and also guarantees within a user-provided error threshold that no collision ever happens before the first contact occurs.

We model the avatar as an articulated body using line skeletons with constant offsets and the virtual environment as a collection of polygonized objects. Given the position and orientation of the avatar at discrete time steps, we use an arbitrary in-between motion to interpolate the path for each link between discrete instances. We bound the swept space of each link using interval arithmetic and dynamically compute a bounding volume hierarchy (BVH) to cull links that are not in close proximity to the objects in the virtual environment. The swept volumes (SVs) of the remaining links are used to check for possible interference and estimate the time of collision between the surface of the SV and the rest of the objects. Furthermore, we use graphics hardware to accelerate collision queries on the dynamically generated swept surfaces.

Our approach requires no precomputation and is applicable to general articulated bodies that do not contain a loop. We have implemented the algorithm on a 2.8 GHz Pentium IV PC with an NVIDIA GeForce 6800 Ultra graphics card and applied it to an avatar with 16 links, moving in a virtual environment composed of hundreds of thousands of polygons. Our prototype system is able to detect all contacts between the moving avatar and the environment in 10–30 ms.

1 Introduction

Collision detection is a fundamental geometric problem that arises in many applications such as virtual reality (VR), physically-based modeling, robotics, computer-aided design, and so on. Particularly in VR applications, simulating the physical presence of a virtual environment is crucial for enabling VR users to be immersed in the simulated environment. In order to simulate the virtual presence, fast and accurate collision detection is required so that the

Presence, *Vol. 16, No. 2, April 2007, 206–223*

*Correspondence to kimy@ewha.ac.kr.

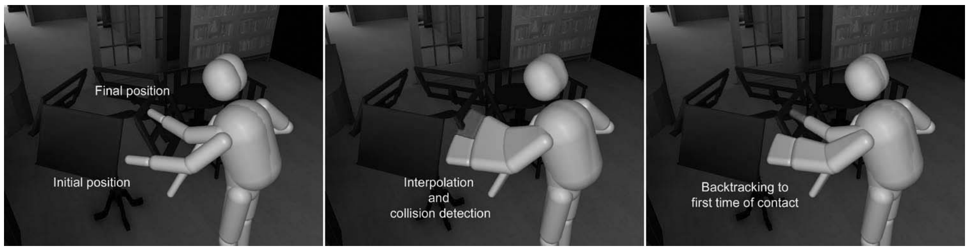
Final position
Initial position
Interpolation and collision detection
Backtracking to first time of contact

Figure 1. *This figure illustrates the benefits of our continuous collision detection algorithm over discrete methods. The left image shows two successive configurations of the avatar that highlight a fast arm motion. No collision is detected at these discrete time steps. The middle image shows the interpolating path used to detect a collision with the environment between these two configurations. The right image shows the time-slicing step used to compute the time of collision and the avatar position at that time. This image also highlights the time interval over which there is no collision with the virtual environment.*

behavior of the avatars and objects in the virtual environment can correctly mimic the real world. As we simulate the avatar's motion and behavior in a virtual environment, it is important to check for potential interference with the rest of the environment.

The problem of interference detection has been extensively studied in the literature. At a broad level, these algorithms can be categorized into specialized algorithms for convex primitives and general techniques for arbitrary polygonal models. However, there are two major limitations in using these algorithms for simulating the avatar's motion in the virtual environment. First of all, most of the algorithms check for interference only at discrete time intervals. As a result, the existing approaches can miss a collision between two sampled time instances. Such cases can arise frequently for fast moving avatars poking through thin objects or virtual objects moving at a high speed. The position and orientation of the avatar is typically measured at fixed time intervals using external tracking devices. For example, it is possible that some positional data arrives late at the client because of high latency and is therefore discarded. In these cases, it is possible that the avatar's arms or limbs have collided with the virtual environment in between time steps, as shown in Figure 1. To overcome this limitation, we need collision detection algorithms that model the avatar's motion as a continuous path and that check for interference along the path.

The second limitation of existing algorithms is the high preprocessing cost, such as constructing bounding volume hierarchies of complex objects. As we dynamically model the avatar's motion between successive instances, these techniques cannot be directly applied for real-time collision detection. Moreover, since our avatar model has many articulated links, we cannot completely precompute bounding volume hierarchies (BVHs) of the avatar as preprocessing and the runtime updates to the BVHs should be followed.

1.1 Main Results

We address the problem of continuously detecting collisions between a moving avatar and its surrounding virtual environment. In order to enable real-time continuous collision detection for an avatar in a virtual environment, we model the avatar using a relatively simple model, a line-based skeletal representation. More specifically, each body part (e.g., arm, limb) in the avatar is modeled by using a straight-line segment with some thickness or offset radius, that is, line swept sphere (LSS), and these line segments are linked together to form an articulated body representing the avatar as shown in Figure 2. The LSS is defined as the volume created by sweeping a sphere along a line segment (Larsen, Gottschalk, Lim, & Manocha, 2000). Even

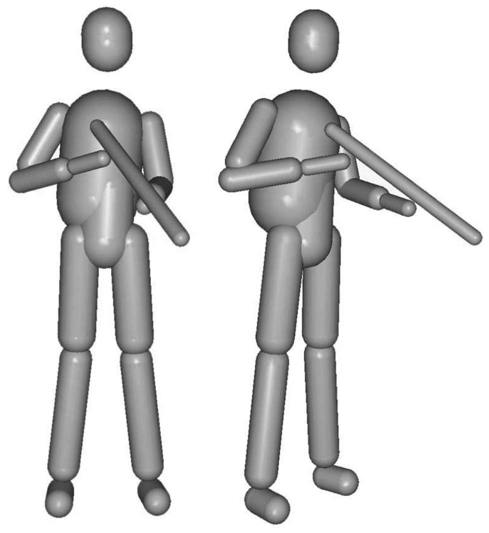

Figure 2. *Skeletal representation of a human avatar.*

though our chosen model for an avatar is simplistic, it is sufficiently effective for VR applications.

Furthermore, we assume that the configuration (i.e., position and orientation) of the line segment is intermittently available at discrete time steps and obtained by motion capture or tracking devices. Given the stream of data as a sequence of configurations, we interpolate the in between path for each link using an arbitrary in between motion. Thus, at a given time *t,* we analytically represent the configuration $C_i(t)$ of a line segment *i*. Given the continuous, interpolated stream of motion sequences of a human avatar, the collision detection problem reduces to checking whether the hierarchy of moving LSSs collides with the underlying environment and reporting the first estimated time of collision.

In order to check for collisions between a moving LSS and the environment, we compute a polygonal approximation of the swept volume (SV) of the LSS. We initially use the interpolated motion data stream as a sweeping trajectory, and check for collisions between the SV and the rest of the environment. Since the SV is dynamically generated, we use the graphics hardware to perform collision queries (Govindaraju, Redon, Lin, & Manochan, 2003). These queries are computed at an image space resolution. To accelerate the computations, we also generate a dynamic BVH of the swept volumes based on interval arithmetic. It is used to cull links that do not interfere with the environment. We have implemented the algorithm on a 2.8 GHz Pentium IV PC with an NVIDIA GeForce 6800 Ultra graphics card and applied it to an avatar model with 16 links. The virtual environment consists of hundreds of thousands of triangles and our algorithm is able to detect all contacts between the moving avatar and the environment in 10–30 ms and report the first time of contact, as shown in Figure 1.

1.2 Organization

In Section 2, we briefly review the earlier work on SV computation, collision detection and graphics hardware-based geometric computations. Section 3 gives an overview of our approach. In Section 4, we present our motion formulation for interpolating any two successive sets of links positions and orientations of the moving articulated figure, and Section 5 describes our algorithm to construct a BVH and perform culling using interval arithmetic. Section 6 presents our approximation algorithm to compute the SV of LSS, and Section 7 describes the graphics hardware-based collision detection algorithm. In Section 8, we describe its implementation and highlight its performance on a complex virtual scene.

2 Earlier Work

In this section, we give a brief survey of the earlier work related to continuous collision detection, SV computation, and geometric computation using graphics hardware.

2.1 Continuous Collision Detection

Most of the prior work on collision detection has focused on checking for collisions at discrete time instances. Check out Lin and Manocha (2003) for a recent survey. These include specialized algorithms for convex polytopes that exploit coherence between successive time steps, general algorithms for polygonal or spline models that precompute a spatial partitioning or BVHs.

A few algorithms have been proposed for continuous collision detection (CCD). These approaches model the trajectory of the object between successive discrete time instances and check the path for collisions. More specifically, there are four different approaches presented in the literature: The algebraic equation solving approach (Redon, Kheddar, & Coquillart, 2000; Canny, 1986; Kim & Rossignac, 2003), the swept volume (SV) approach (Abdel-Malek, Yang, Blackmore, & Joy, 2006), the adaptive bisection approach (Redon, Kheddar, & Coquillart, 2002; Schwarzer, Saha, & Latombe, 2002), and the kinetic data structures (KDS) approach (Agarwal, Basch, Guibas, Hershberger, & Zhang, 2000; Kirkpatrick, Snoeyink, & Speckmann, 2000). In practice, especially for 3D real time applications, the adaptive bisection approach has been shown to be useful.

Not much work has been reported on continuous collision detection for articulated models. The preliminary version of our work has appeared in Redon, Kim, Lin, and Manocha (2004a). Extending from Redon et al. (2004a), in this article, we give a more detailed explanation of how we interpolate the motion of an avatar (Section 4) and how we compute a dynamic AABB using interval arithmetic (Appendix). Moreover, we provide an improved solution to estimate the time of collision (Section 7) unlike the one presented in Redon et al. (2004a) as well as a method to compute the contact zone and penetration depth between a moving avatar and the surrounding environments. We also test our method with a new set of complex benchmarking models and use constrained dynamics to simulate the collision response from an avatar to the environment using the result of penetration depth computation (Section 8). Finally, in Section 6, we provide a better analysis on the error bound of our swept volume formulation.

The extension of Redon et al. (2004a) to general articulated models has been presented in Redon, Kim, Lin, and Manocha (2004b); however, its performance is too slow for interactive VR applications (e.g., Liu & Badler, 2003; Lok, Naik, Whitton, & Brooks, 2003).

2.2 Swept Volume Computation

SV has been widely investigated in various disciplines such as geometric modeling, computer graphics, computational geometry, and robotics.

The mathematical formulation of the SV problem has been studied using the singularity theory, sweep differential equation, Minkowski sums, envelope theory, implicit modeling and kinematics. A survey of these formulations is given in Abdel-Malek et al. (2006). Algorithms to compute and visualize the boundaries are presented in Kim, Varadhan, Lin, and Manocha (2003) and Rossignac and Kim (2000). However, they are not fast enough for interactive applications.

A few algorithms have been proposed to use SVs for collision detection. Kieffer and Litvin (1990) applied a SV-based interference detection to moving mechanical solids like gears; Cameron (1990) suggested a collision detection algorithm using four dimensional SV in the time and space domain, Xavier (1997) extended the GJK collision detection algorithm to handle a linear SV problem, and Foisy and Hayward (1994) also proposed a SV-based collision detection. Korein (1985) used a SV-based formulation to compute the reachability of a robot. However, none of these approaches address real-time collision detection for articulated bodies based on the SV.

2.3 Geometric Computations Using Graphics Hardware

Interpolation-based graphics hardware is increasingly being used for geometric applications (Manocha, 2002). This is mainly due to the recent advances in the performance of the graphics processors as well as support for programmability. They have been used for visibility and shadow computations, CSG rendering, and proximity queries including collision detection, morphing, object reconstruction, and so on. A recent survey on different applications is given in Harris (2003) and Theoharis, Papiannou, and Karabassi (2001). These include different algorithms for collision detection between closed objects (Hoff, Zaferakis, Lin, & Manocha, 2001; Rossignac, Megahed, & Schneider, 1992), as well as a recent algorithm for general and deformable objects that utilizes the visibility queries (Govindaraju et al., 2003; Govindaraju et al., 2005). All of these algorithms perform computations in the image space and their accuracy is governed by the underlying pixel resolution.

3 Algorithm Overview

In this section, we give an overview of our approach to perform collision detection between a moving avatar and the virtual environment. We initially describe the representation of the avatar model, discuss the complexity of performing continuous collision detection on the model, and present our pipeline that proceeds in five stages.

3.1 Notation

We begin this section by explaining the notation used throughout the paper. In the following section, we describe the representation for an articulated avatar model that we use in the paper.

We use boldface type to distinguish a vector from a scalar value (e.g., a vector for the origin $\mathbf{o}$). Let $\mathbf{u}_i^*$ denote the 3×3 skew-symmetric matrix such as $\mathbf{u}_i^*\mathbf{x} = \mathbf{u}_i \times \mathbf{x}$ for every 3D vector $\mathbf{x}$. If $\mathbf{u}_i = (u_i^x, u_i^y, u_i^z)^T$, then:

$$\mathbf{u}_i^* = \begin{pmatrix} 0 & -u_i^z & u_i^y \\ u_i^z & 0 & -u_i^x \\ -u_i^y & u_i^x & 0 \end{pmatrix} \quad (1)$$

We assume that there is no loop in the graph describing the articulated avatar. Consequently, each link has a *unique* parent link, except for the root node which has no parent. On the other hand, any link can have any number of children, as long as there is no loop induced. We first begin by expressing the motion of each link in the reference frame of its unique parent. The motion of the root node is similarly expressed in the global frame. For the sake of simplicity of notation, we assume that the index of link i's parent is $i - 1$. This can be easily modified when a parent has multiple children per link. Figure 3a illustrates our notation for a link i moving within the reference frame of its parent.

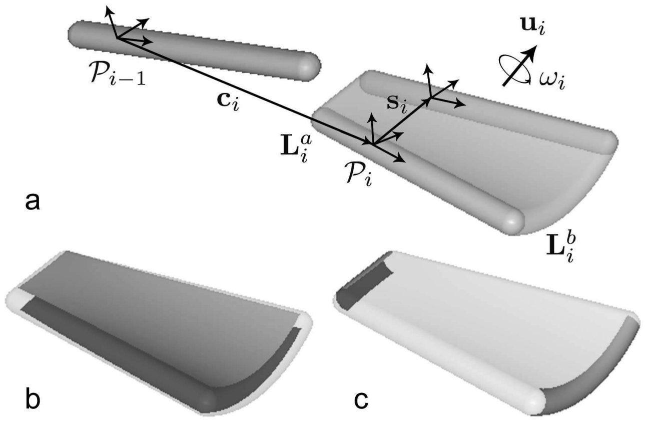

Figure 3. *(a) Link i is moving in the reference frame of its parent. The initial and final positions are outlined. (b) Offset of the rule surface. (c) Pipe surfaces.*

For a given node i, let P_i denote the reference frame associated with it. We assume that, in its local reference frame, the line segment is positioned along the x-axis between the two endpoints $\mathbf{L}_i^a$ and $\mathbf{L}_i^b$:

$$\mathbf{L}_i^a = \begin{pmatrix} l_i^a \\ 0 \\ 0 \end{pmatrix} \quad \text{and} \quad \mathbf{L}_i^b = \begin{pmatrix} l_i^b \\ 0 \\ 0 \end{pmatrix} \quad (2)$$

Moreover, we denote the position and orientation of P_i relative to P_{i-1} as $\mathbf{T}_i^{i-1}$ and $\mathbf{P}_i^{i-1}$, respectively. Also, the motion of P_i relative to P_{i-1} is described as $\mathbf{M}_i^{i-1}$.

3.2 Complexity of Continuous Collision Detection

The main challenge in our continuous collision detection algorithm is to compute the swept volume of an avatar consisting of many LSSs and quickly check for its intersection with other objects in the environment. As will be explained in detail in Section 6, computing swept volume of an LSS requires calculating the offset surface of a ruled surface. However, it is difficult to compute the exact offset surface and check for interference. This is due to the following reasons:

- It is challenging and still an open problem to compute the exact offset surface where the progenitor

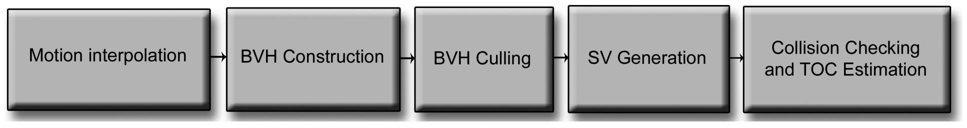

Figure 4. *The overall pipeline of our collision detection algorithm. Different stages are performed on the CPU and the graphics processor.*

surface includes nonrational functions. Even when the progenitor can be described using regular NURBS, the offset surface can have *holes* for convex edges and vertices and *loops* for concave edges and vertices (Farouki, 1986). Therefore, the offset surface can contain self-intersections and singularities (Hoffmann, 1989) and thus require costly *trimming* or arrangement computations to obtain a final surface representation. As a result, computing an explicit representation of the offset surface is nontrivial.

- The problem of performing exact collision detection between high order or nonlinear surfaces is considered hard in practice (Lin & Manocha, 2003). The underlying algorithms suffer from robustness and accuracy problems.
- The VR application demands interactive performance, which calls for a 30 Hz or higher update rate. It is a major challenge to perform exact collision detection between curved primitives at such rates.

3.3 Our Approach

In order to meet the above challenges, we present an approximate but fast solution to the problem. The main idea is to approximate the SV of LSS and use the graphics processors to perform the collision queries. To accelerate this process, we also build a dynamic BVH based on interval arithmetic. We apply it to the motion of each link in the articulated figure, and prune away some links that do not collide with the environment. Moreover, we simplify the motion trajectory by using an arbitrary in between motion, and this reduces the computation time for both approximating the offset surface and the BVH construction. Our overall algorithm uses a five-stage pipeline (shown in Figure 4 and discussed below).

3.3.1 Motion Interpolation. Given two successive available configurations of the avatar, we determine an interpolating path from the initial to the final configuration.

3.3.2 BVH Construction. For each link in the articulated model, we use interval arithmetic to compute an enclosing bounding box, and recursively construct a dynamic BVH around the entire avatar.

3.3.3 BVH Culling. Based on the BVH, we use conservative tests to cull some of the links that do not collide with objects in the environment.

3.3.4 SV Generation. For the remaining links, we compute a polygonal approximation of their SV by tessellating the offset surface.

3.3.5 Collision Checking and TOC Estimation. We use graphics hardware to check whether the approximate SV collides with objects in the environment. These queries are performed at an image-space resolution. Our algorithm also estimates the time for each collision.

4 Motion Interpolation

In this section, we describe a motion formulation to compute a continuous path for each LSS between discrete time instances. In particular, we use an arbitrary in between motion to interpolate successive configurations of the avatar (Redon et al., 2000; Redon et al., 2002). As is the case in many applications, the actual motion of the avatar is not known and we are only given its position and orientation at discrete time instances. This is mainly due to the fact that the tracking device can sample positions and orientations only at discrete time steps. Moreover, the avatar is simulated as a virtual object. It is modeled as part of a constraint-based multibody dynamics simulation system. In most cases, the differential equations governing the system's dynamics are solved using discretized techniques (e.g., Euler or Runge-Kutta methods). As a result, we do not have a closed-form expression of the avatar's motion.

Given these constraints, we choose a motion formulation to interpolate between different avatar configurations. The goal is to use a formulation that is general enough to interpolate between any two successive configurations and preserves the rigidity of the links, yet is simple enough to allow us to perform the various steps of our collision detection algorithm. We assume that the links are not deformed during the interpolation, as is the case for linear interpolation between the endpoints' initial and final positions. Note that the arbitrary in between motion (Redon et al., 2002) used to detect collisions is also used to compute a position of the object at the time of collision. This ensures that all the objects in the scene are maintained in a consistent state and there are no interpenetrations. Next, we give details of the specific arbitrary in between motion (Redon et al., 2002) used to compute the path between successive instances.

Let's now describe the motion of the reference frame of link i, P_i, relative to that of link $i-1$, P_{i-1}. We use the 3D vector $\mathbf{c}_i$ and the 3×3 matrix $\mathbf{R}_i$ to denote the position and orientation of P_i relative to P_{i-1} at the beginning of the time interval $[0, 1]$. We assume that the motion of P_i relative to P_{i-1} is composed of a rotation of angle ω_i around an axis $\mathbf{u}_i$, and of a translation $\boldsymbol{s}_i$. The parameters $\mathbf{c}_i$, $\mathbf{R}_i$, $\mathbf{u}_i$, and $\mathbf{s}_i$ are constants for a given time step and are expressed in P_{i-1}. Moreover, we assume that P_i moves with constant translation and rotation velocities, as shown in Figure 3a.

The position of P_i relative to P_{i-1} for a given time t in $[0, 1]$ is thus:

$$\mathbf{T}_i^{i-1}(t) = \boldsymbol{c}_i + t\boldsymbol{s}_i \tag{3}$$

The orientation of P_i relative to P_{i-1} is given as:

$$\mathbf{P}_i^{i-1}(t) = \cos(\omega_i t) \cdot \mathbf{A}_i + \sin(\omega_i t) \cdot \mathbf{B}_i + \mathbf{C}_i \tag{4}$$

where $\mathbf{A}_i$, $\mathbf{B}_i$, and $\mathbf{C}_i$ are 3×3 constant matrices that can be computed at the beginning of the time step:

$$\begin{aligned} \mathbf{A}_i &= \mathbf{R}_i - \mathbf{u}_i \cdot \mathbf{u}_i^T \cdot \mathbf{R}_i \\ \mathbf{B}_i &= \mathbf{u}_i^* \cdot \mathbf{R}_i \\ \mathbf{C}_i &= \mathbf{u}_i \cdot \mathbf{u}_i^T \cdot \mathbf{R}_i \end{aligned} \tag{5}$$

Consequently, the motion of P_i relative to P_{i-1} is described by the following 4×4 homogeneous matrix:

$$\mathbf{M}_i^{i-1}(t) = \begin{pmatrix} \mathbf{P}_i^{i-1}(t) & \mathbf{T}_i^{i-1}(t) \\ (0, 0, 0) & 1 \end{pmatrix} \tag{6}$$

resulting in coordinates in the reference frame of the parent link P_{i-1}. Consequently the matrix:

$$\mathbf{M}_i^0(t) = \mathbf{M}_1^0(t) \cdot \mathbf{M}_2^1(t) \ldots \mathbf{M}_i^{i-1}(t) \tag{7}$$

describes the motion of link i in the world frame.

Note that our formulation makes it extremely simple to compute all the motion parameters $\boldsymbol{s}_i$, $\boldsymbol{u}_i$, and ω_i for a given time step. For a given link i, assume that $\boldsymbol{c}_i^0$ and $\boldsymbol{c}_i^1$ (respectively $\mathbf{R}_i^0$ and $\mathbf{R}_i^1$) are the initial and final positions (respective orientations) of P_i relative to P_{i-1}. Then $\boldsymbol{s}_i = \boldsymbol{c}_i^1 - \boldsymbol{c}_i^0$, and $(\mathbf{u}_i, \omega_i)$ is the rotation extracted from the rotation matrix $\mathbf{R}_i^1(\mathbf{R}_i^0)^T$. An example of motion interpolation between successive instances of an avatar is illustrated in Figure 5.

5 Dynamic BVH Generation and Culling

Given the motion formulation between successive links, the next step in the collision detection algorithm is to compute a BVH around the avatar. In this section, we describe the dynamic BVH generation algorithm and use it to cull some of the links that do not collide with the environment.

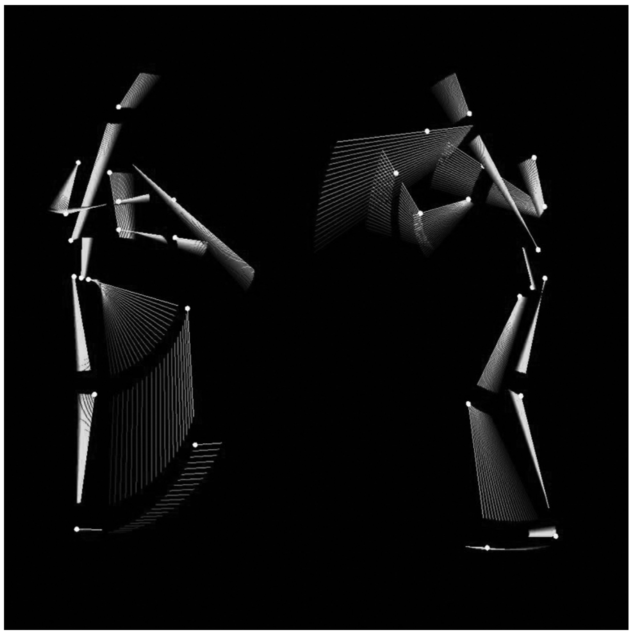

Figure 5. *Interpolation of successive instances of avatar motions.*

Each bounding volume (BV) in the BVH corresponds to an axis-aligned bounding box (AABB). We compute an AABB for each link that encloses its complete trajectory over the time step. These leaf boxes are then used to efficiently compute a complete hierarchy of AABBs used to quickly cull links that are far from the environment.

The leaf boxes are computed using interval arithmetic (Moore, 1979). For more information about interval arithmetic, we refer the reader to the appendix at the end of this paper. We use interval arithmetic to bound the functions describing the trajectories of the links that produce the AABBs that enclose these trajectories. To bound the trajectory, we perform elementary interval arithmetic operations (Moore, 1979) recursively on their expressions. We begin by bounding the sine and cosine functions from Eq. 4 over the time interval [0, 1]. Using elementary interval operations, we bound each component of the orientation matrices $\mathbf{P}_i^{i-1}(t)$ over the entire time interval [0, 1]. Similarly, we use elementary interval operations to bound the translation components $\mathbf{T}_i^{i-1}(t)$.

Eventually, we obtain 4×4 homogeneous interval matrices $\tilde{\mathbf{M}}_i^{i-1}$ whose interval components bound the corresponding components of $\mathbf{M}_i^{i-1}$ over the time interval [0, 1]. These interval matrices are concatenated by again performing elementary interval operations to compute the interval version $\tilde{\mathbf{M}}_i^0$ of the matrix $\mathbf{M}_i^0$.

By applying this interval matrix to the both endpoints of a link, $\mathbf{L}_i^a$ and $\mathbf{L}_i^b$, we obtain two 3D interval vectors that bound the coordinates of the endpoints of the links over the time interval [0, 1]. In other words, we obtain two AABBs that bound the endpoints' trajectories over the time interval. By using a convexity argument, it can be seen that the AABB that encloses these two boxes bounds the entire link over the time interval. Next, we enlarge the box by an offset equal to the radius of the corresponding LSS to make sure that the AABB bounds the LSS and its whole trajectory, as illustrated in the left figure of Figure 6. Given the AABBs around the leaf nodes, we compute the BVH in a bottom-up manner around the entire avatar. After computing the BVH, we recursively check for overlaps with the environment and cull a subset of the links that do not collide with the virtual environment, as illustrated in the right figure of Figure 6.

6 Swept Volume Generation

In the previous section, we described an algorithm to cull some of the links that do not collide with the environment. In this section, we present an algorithm to compute the SV of the LSS for the remaining links. We compute a polygonal approximation of the SV and use it for collision detection with the environment.

6.1 Swept Volume-based Collision Detection

Our goal is to check collisions between moving articulated human figures and the surrounding environment. We use a simple model of the avatar for collision detection and formulate each joint in the articulated figure as an LSS. As a result, the collision detection problem reduces to checking for collision between each moving LSS and the environment. We pose this prob-

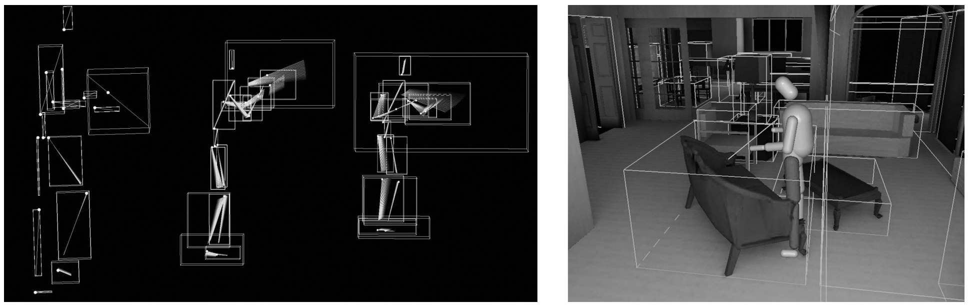

Figure 6. *Dynamic AABB hierarchy construction and culling. The left figure illustrates an example of AABBs that enclose the links of an avatar under motion. The right figure shows a result of the AABB culling of an avatar with the environment; only the AABB of a couch collides with that of the avatar.*

lem as a swept volume (SV) problem; that is, we generate the SV of each moving LSS and check the SV for interference with the environment.

The SV is the volume created by sweeping a solid (or surfaces) in space along some continuous trajectory. Mathematically, the sweep equation of an object, $\boldsymbol{\Gamma}$, under rigid motions [$\boldsymbol{\Psi}(t)$ and $\mathbf{R}(t)$] can be expressed as the following equation:

$$\boldsymbol{\Gamma}(t) = \boldsymbol{\Psi}(t) + \mathbf{R}(t)\boldsymbol{\Gamma} \tag{8}$$

Here, $\boldsymbol{\Psi}(t)$ and $\mathbf{R}(t)$ are translation and rotation matrices, respectively, at time t during the sweep. Notice that, since we are dealing with articulated bodies, the transformation matrices, $\boldsymbol{\Psi}(t)$ and $\mathbf{R}(t)$, may contain general, nonrational functions such as a high order of trigonometric functions. Finally, the SV is defined as follows:

$$SV(\boldsymbol{\Gamma}) = \{\cup\boldsymbol{\Gamma}(t) | t \in [0, 1]\} \tag{9}$$

where we assume that the time parameter t, is normalized to a unit time interval. In our formulation, the generator $\boldsymbol{\Gamma}$ is an LSS. Notice that the medial axis of an LSS corresponds to a line segment, and conversely, the offset surface of the line segment reconstructs the LSS. Therefore, the SV of an LSS is equivalent to the offset surface of the swept surface of the medial line segment. In general, the swept surface of a line segment creates a ruled surface (Pottmann & Wallner, 2001).

A ruled surface $\boldsymbol{x}(t, s)$ has the following form:

$$\boldsymbol{x}(t, s) = \boldsymbol{b}(t) + s\boldsymbol{\delta}(t) \tag{10}$$

Here, $\boldsymbol{b}(t)$ is a directrix and $\boldsymbol{\delta}(t)$ is the direction of a ruling line. In the case of sweeping a line segment, the directrix curve is computed by the endpoints of the line segment at time t, and the direction of a ruling line by the direction of the line segment at t. Therefore, given rigid motions, we can easily determine the SV (i.e., ruled surfaces) of line segments.

The definition of the offset surface $\boldsymbol{x}_d(t, s)$ of a given ruled surface $\boldsymbol{x}(t, s)$ with offset distance d is expressed as follows:

$$\boldsymbol{x}_d(t, s) = \boldsymbol{x}(t, s) \pm d\boldsymbol{n}(t, s) \tag{11}$$

where $\boldsymbol{n}(t, s)$ is the unit normal vector field defined on the surface of $\boldsymbol{x}(t, s)$, and $\boldsymbol{x}(t, s)$ is assumed to be regular; that is, each $\boldsymbol{n}(t, s)$ is uniquely defined.

We assume that LSS with radius d follows the sweep equation given in Eq. 8, and its medial axis at time t is thus parameterized as $\boldsymbol{x}(t, s)$. Then, the SV of the LSS following Eq. 8 is $\boldsymbol{x}_d(t, s)$ in Eq. 11. Mathematically speaking, our goal is to check intersections of $\boldsymbol{x}_d(t, s)$ with other objects in the environment.

6.2 Swept Volume of Line Swept Sphere

It is well known that the envelope (or swept volume) of a moving cylinder following a continuous trajectory is equivalent to the offset surface of a ruled surface as illustrated in Figure 3b. Moreover, the axis and radius of the moving cylinder correspond to the ruling line and offset radius of the ruled surface, respectively. As a result, we can calculate the SV of a moving cylinder with radius d by computing the offset surface of a ruled surface with the offset distance d.

The mathematical formulation of an offset surface is given in Eq. 11. Also notice that, in Eq. 11, $\boldsymbol{x}_d(u, v)$ is defined as a two-sided offset surface suited for our application. It is possible that $\boldsymbol{x}(u, v)$ may contain nonregular points. One of the conventional techniques to handle such cases is to bound $\boldsymbol{n}(u, v)$ with a spherical polygon (Pottmann & Wallner, 2001).

We extend the relationship between the offset of a ruled surface and the SV of a cylinder by computing the SV of LSS. This volume is obtained by independently computing the SV of the cap portion of LSS and computing the union with the remaining portion of LSS (i.e., the SV of the LSS). The SV generated by the caps of LSS is a *pipe surface* as illustrated in Fig. 3c. In fact, the pipe surface is a special case of a *canal surface*. A canal surface is generated by sweeping a sphere of varying radii along some continuous trajectory, $\boldsymbol{C}(t)$. A pipe surface is a special case of the canal surface where the radius is fixed. The parametric equation $\boldsymbol{K}(t, \theta)$ of a pipe surface can be given in terms of t and θ as follows (Kim & Lee, 2003):

$$\boldsymbol{K}(t, \theta) = \boldsymbol{C}(t) + R(\cos\theta \boldsymbol{b}_1(t) + \sin\theta \boldsymbol{b}_2(t)) \quad (12)$$

$$\boldsymbol{b}_1(t) = \frac{\boldsymbol{C}'(t) \times \boldsymbol{C}''(t)}{\|\boldsymbol{C}'(t) \times \boldsymbol{C}''(t)\|}$$

$$\boldsymbol{b}_2(t) = \frac{\boldsymbol{C}'(t) \times \boldsymbol{b}_1(t)}{\|\boldsymbol{C}'(t) \times \boldsymbol{b}_1(t)\|}$$

Here, $\mathbf{C}(t)$ is defined as a trajectory of the center of a swept sphere whose radius is R, and $\boldsymbol{b}_1(t)$, $\boldsymbol{b}_2(t)$ form the basis vectors of the normal plane of the spine curve $\boldsymbol{C}(t)$. The above equation is obtained by computing the envelope of a moving sphere centered at $\boldsymbol{C}(t)$ with a fixed radius, R.

Once we have computed the offset of ruled surface and pipe surface (see Figure 7), we compute the SV of LSS by taking the union of them. In the next section, we explain how to approximate the offset of the ruled surface and pipe surface.

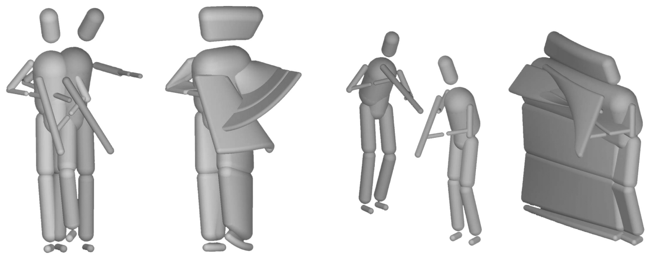

Figure 7. *Computing swept volume for moving avatars. In each figure, two instances of the same avatar under motion are shown at the left, and the composit of the swept volume of the moving avatar is shown at the right. For a color version of this figure, see http://gamma.cs.unc.edu/Avatar/.*

6.3 Tessellation of Swept Volume

Our goal is to approximate the offset and pipe surfaces with piecewise planar surface patches. More specifically, we want to tessellate these surfaces and analyze the maximum deviation error from the exact surfaces.

The earlier algorithms for approximating an offset surface assume that the underlying progenitor surface is a freeform surface such as a Bézier or NURBS surface. Under this assumption, there are three typical approaches to approximate an offset surface (Elber, Lee, & Kim, 1997); control polygon-based, interpolation-based, and circle-approximation approach. In particular, the interpolation-based approach is based on directly sampling the positions and derivatives of the exact offset surface and attempts to optimize the approximated offset surfaces (Farouki, 1986; Hoschek, 1988; Klass, 1983). We adopt this technique in our application because of its simplicity and its suitability for interactive applications. In particular, we uniformly sample the offset of the ruled surface in the u and v parameter domain, as given in Eq. 11, and create strips of triangles by varying one of the parameters while fixing the other one. The tessellation of a pipe surface is performed using a similar approach. Given the formulation in Eq. 12, we uniformly sample the pipe surface along the t and θ parameters.

6.4 Tessellation Error Bounds

The deviation error of an approximated offset surface is calculated by computing $\|x_d(u, v) - x(u, v)\| - d$ or squared distance $\|x_d(u, v) - x(u, v)\|^2 - d^2$ (Elber et al., 1997). The error is relatively easy to compute when the progenitor surface is represented as a Bézier or NURBS surface. However, the progenitor surface in our case is a non-rational surface described using trigonometric function. As a result, error calculation becomes nontrivial.

Our method to derive an error bound is based on a well-known result by Filip, Magedson, and Markot (1986) stated as follows: Given a C^2 surface $f: [0, 1] \times [0, 1] \rightarrow \mathrm{IR}^3$ and a error tolerance ε, a piecewise linear surface $l: [0, 1] \times [0, 1] \rightarrow \mathrm{IR}^3$ with n and m uniform subdivision along each $[0, 1]$ satisfies $\sup \|f(t, s) - l(t, s)\| \leq \varepsilon$ when

$$\frac{1}{8}\left(\frac{1}{n^2} M_1 + \frac{2}{nm} M_2 + \frac{1}{m^2} M_3\right) = \varepsilon \qquad (13)$$

where

$$M_1 = \sup_{(t, s) \in [0, 1] \times [0, 1]} \left\| \frac{\partial^2 f(t, s)}{\partial u^2} \right\|$$

$$M_2 = \sup_{(t, s) \in [0, 1] \times [0, 1]} \left\| \frac{\partial^2 f(t, s)}{\partial u \partial v} \right\|$$

$$M_3 = \sup_{(t, s) \in [0, 1] \times [0, 1]} \left\| \frac{\partial^2 f(t, s)}{\partial v^2} \right\|$$

In our case, $f(t, s)$ corresponds to the offset surface $x_d(t, s)$ of a ruled surface $x(t, s)$ in Eq. 10 and Eq. 11. The relationship between the derivatives of $x_d(t, s)$ and $x(t, s)$ can be algebraically expressed (Farouki, 1986). Therefore, we first bound the derivatives of $x(t, s)$ using interval arithmetic, followed by bounding the derivatives of $x_d(t, s)$. As a result, given error tolerance ε, we can determine the required subdivision step sizes (i.e., n, m in Eq. 13) to tessellate the offset surface.

Another possibility to compute an error bound is to analyze the screen space error when the approximated surface is projected onto the screen space (Kumar & Manocha, 1995). This projection is performed as part of the graphics hardware based collision detection algorithm. In this case, we need bounds on the derivatives of the projected surface function. These bounds are computed by applying interval arithmetic techniques to the derivatives.

7 Collision Detection

In this section, we describe the final stage of our algorithm that performs the collision queries using the graphics hardware. We also show how we determine an estimate of the time of impact and an approximate contact information.

There are two main challenges in performing collision detection using the SV. These include computing an accurate, explicit representation of the SV and checking its interference with the environment. We have described an algorithm to compute a polygonal approximation of the SV of each LSS in the previous section. Given the polygonal approximation, we use the graphics processor to check for collisions with the environment.

7.1 Graphics Hardware-based Computation

The real-time constraints for collision detection imply that all the computations need to be performed on the fly. As a result, we are unable to use earlier techniques referenced in Section 2 that precompute hierarchies to speed up the runtime queries. Instead, we choose the CULLIDE algorithm (Govindaraju et al., 2003) that uses graphics hardware to perform interactive collision detection. The basic idea of CULLIDE is to pose the collision detection problem in terms of performing a sequence of visibility queries. If an object is classified as fully visible with respect to the rest of the environment, it is a sufficient condition that the object does not overlap with the environment. For those objects that are classified as partially visible, the algorithm performs exact triangle-level intersection tests. CULLIDE performs the visibility queries using the graphics processors and the exact triangle-level intersection tests on the CPUs. More precisely, we perform 2.5D overlap tests between the objects on the GPU by performing orthographic projections along the X, Y, and Z directions. The graphics hardware is very well optimized to perform these transformations, scan converting the primitives and performing these pixel level comparisons by using the multiple pixel processing engines in parallel. In particular, we used the NVIDIA OpenGL extension GL_NV_occlusion_query (NVIDIA Occlusion Query, 2002) to perform the visibility queries. This query is available on the commodity graphics processors.

The main benefits of this approach include:

- The algorithm does not require any preprocessing and can handle dynamically generated polygonal objects obtained from the tessellation of the SV.
- The algorithm computes all overlapping objects and triangles up to screen-space precision and does not report any false negatives.

7.2 Estimating the Time of Collision

We first define *slices* of the swept-volume of the LSS as a restriction of the swept volume to a subinterval of the current time step. The number of slices is independent of the tessellation parameters used to determine the approximate swept volume of the LSS. For ease of implementation, the algorithm described in the previous section to compute an approximate swept volume of the LSS can be used for each time slice. Assuming for example that the time interval is divided into four time subintervals, $[t_0, t_1], \ldots, [t_3, t_4]$, the geometric model of the third slice includes the LSS at times t_2 and t_3, as well as the offsets of the ruled surface and the pipe surfaces over the time subinterval $[t_2, t_3]$.

We apply CULLIDE separately for each potentially colliding link. Precisely, for each potentially colliding link, we first render in the depth-buffer the corresponding potentially colliding environment obstacles (determined with AABBs during the previous step), and then perform occlusion queries for each slice of the link. We thus obtain an approximate collision time by determining the first colliding slice.

7.3 Approximate Contact Determination

Once we have determined the first colliding slice, we find an approximate contact zone by determining the intersection between this colliding slice and the environment. As both the colliding slice and the environment are composed of triangles, this intersection is a list of intersection segments, that is, a list of intersections between pairs of triangles, where each pair contains one triangle from the slice and one triangle from an environment obstacle. Figure 8 shows an example of approximate contact determination after the avatar legs have collided with a table and a chair. The top left part of Figure 8 shows collision between the avatar and the table, as well as the colliding slices. The top middle part of Figure 8 shows the valid, nonpenetrating slices only. The top right part shows a resulting intersection between the table and the first colliding slice (the bright white polygonal curve on the table). The bottom row in Figure 8 shows a case of multiple links of an avatar colliding with a chair. When such a multiple-link collision occurs, the performance of contact determination decreases linearly proportional to the number of colliding links.

Using the intersection segments, we then determine an approximate penetration depth and direction using graphics hardware based computations (Redon & Lin, 2006), which can be used to perform collision response and simulate objects' dynamics.

8 Results and Analysis

We have implemented our collision detection algorithm on a 2.8 GHz Pentium IV PC with an NVIDIA GeForce FX 6800 Ultra graphics card. We have applied it to an avatar with 16 links, moving in a virtual environment composed of several hundreds of thousands of triangles. Our method is able to detect all

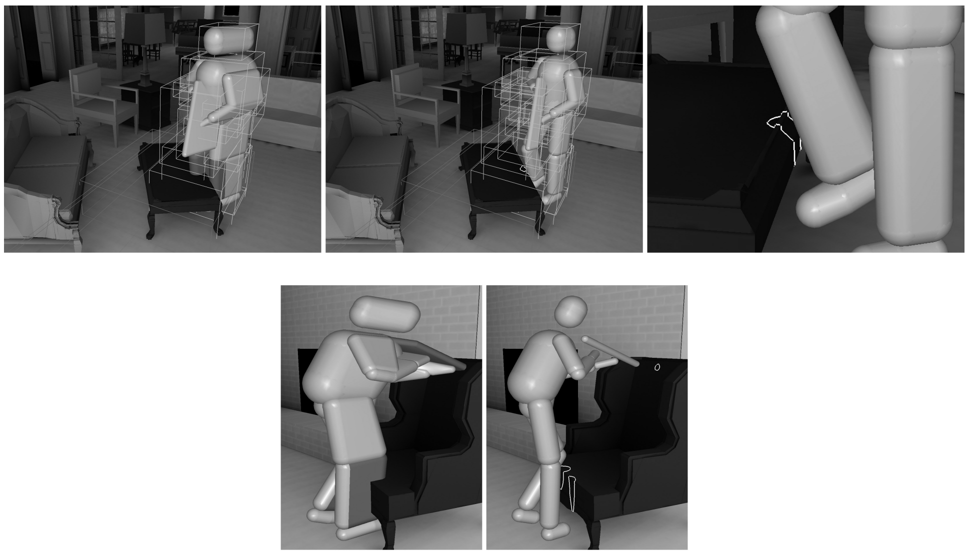

Figure 8. *Approximate contact determination. Top left: the legs of the avatar have just collided with the table. Top middle: the first colliding slice is determined, and only the nonpenetrating slices are kept. Top right: the intersection between the first colliding slice and the environment obstacles is found, and used to determine an approximate penetration depth and direction. Bottom: multiple links of an avatar colliding with a chair.*

collisions between the moving avatar and the environment in 10–30 ms, resulting in a refresh rate of 30–100 frames per second. All collision queries are performed at an image-space resolution of 1024×1024.

Our benchmark includes a client-server based application. The server updates the avatar's position every 10 ms. The collision detection module is included as part of the client that requests new configurations when desired. Figures 9 and 10 show our test environments along with some of the avatar trajectories and interactions. In Figure 9, the top row shows the avatar visiting a room in the house model. In the middle image, the lower right arm of the avatar collides with a music stand. The bottom row shows the avatar in the other room; the avatar collides with a sofa, as shown in the last image. In Figure 10, an avatar collides with a chessman and as a result of collision, the chessman tumbles down. The collision response is implemented based on constraint-based dynamics. We use Open Dynamics Engine (ODE; Smoth, 2005) to implement the dynamics. The ODE requires penetration depth information to implement constrained dynamics for articulated bodies and we use the penetration depth information as computed in Section 7.3.

The sequence shown back in Figure 1 highlights the benefit of our continuous collision detection method over traditional discrete methods. The left image shows two successive configurations of the avatar indicating a fast arm motion. The middle image shows the SV following an arbitrary in-between motion specified in Section 4. A collision is detected during the interpolation at time U_{TOC}. The right image shows that the backtracking step allows the algorithm to stop the avatar and to determine a time interval $[0, t_c]$, $t_c < U_{\mathrm{TOC}}$, over which there exists a collision-free path for all its links.

Table 1 shows the average computation time required for different steps in the house and chess models. It highlights the time for different stages of the algorithm. Various steps of the algorithm include updating the position of the avatar, computing its motion parameters from two successive configurations, and determining the swept AABBs which bound the entire trajectories of the

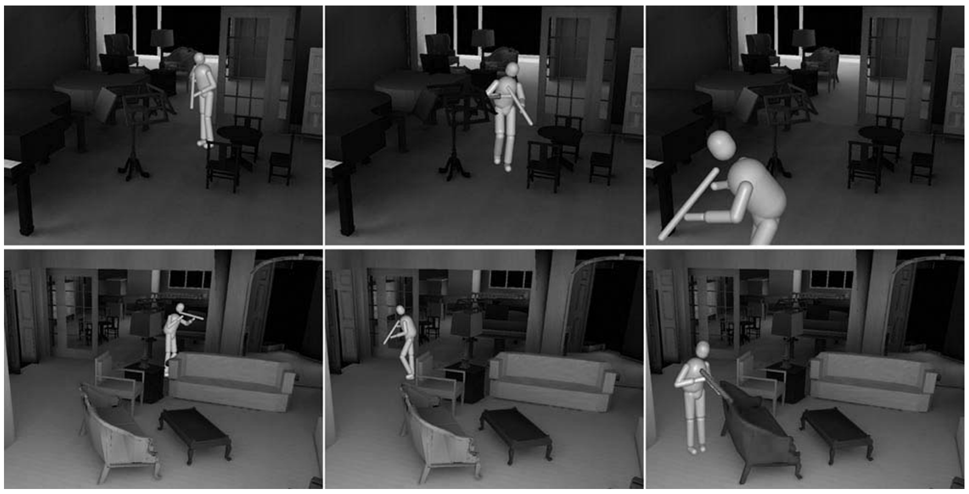

Figure 9. *The benchmark environment and the avatar model used to test the performance of our algorithm. Top row: the avatar visiting the music room. In the middle image, the avatar's lower right arm collides with the music stand. Bottom row: the avatar in the living room colliding with the sofa in the rightmost image.*

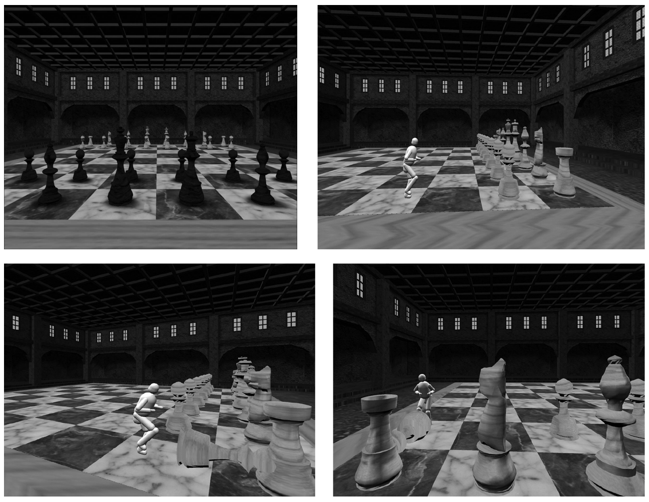

Figure 10. *The chessboard benchmark environment (50K triangles) and the avatar model. Top left: the chessboard environment. Top right: the avatar running toward a chessman. Bottom row: the avatar collides with a chessman and the chessman tumbles down due to the impact force.*

links using interval arithmetic. We obtain a considerable speed-up by using a dynamically generated BVH of AABBs. The cost of generating the BVH and performing culling with it is much smaller as compared to swept surface computation and collision detection using graphics hardware. A detailed analysis of the performance of the CULLIDE algorithm is given in Govindaraju et al. (2003).

There is an additional benefit of a continuous collision detection framework in a client-server model. It can easily handle the variable latency that arises due to the underlying application or networking delays. For example, it is possible that some positional data arrives late at the client because of high latency and is therefore discarded. In such cases, the continuous collision detection algorithm ensures that the received configurations have been interpolated. This approach results in a consistent state of the simulation with no interpenetration between the objects at any time.

8.1 Error Analysis

Our algorithm is not exact. In fact, the algorithm sacrifices exactness in order to achieve a real-time performance. The errors are due to the following sources in the algorithm:

Table 1. *Average Performance (in Milliseconds) of the Various Steps of Our Algorithm for a 16-Link Avatar Moving in a Virtual Environment Composed of Hundreds of Thousands of Polygons*

Step	Timing (house, ms)	Timing (chessboard, ms)
AABB computation (all links)	0.034	0.064
AABB culling (all links)	0.026	0.013
Offset computation (per link)	0.098	0.092
Link hierarchy update (per link)	0.6	0.2
Occlusion queries (collision, per link)	4.4	2.7
Occlusion queries (no collision, per link)	1.3	1.6
Contact information (per colliding link)	4.9	3.3

- **Surface Tessellation Error.** In Section 6.3, we approximate the SV of LSS using planar surface patches. As a result, we tessellate the pipe and offset surface within some error deviation, ε. Thus, if the articulated object moves closer to some of the objects in the environment within ε or penetrates the objects by ε, these collisions can be missed.
- **Image Space Precision Error.** We use a graphics-hardware based collision checking algorithm to check for an collision of the tessellated SV. As a result, the precision of the algorithm is limited by the underlying hardware precision, such as frame-buffer and depth-buffer resolution. However, recent results show that GPU-based interference checking can be made conservative, so that no collision is ever missed (Govindaraju et al., 2004).
- **Floating Point Error.** Essentially, the precision of the interval arithmetic is limited by underlying floating point precision. However, with a careful implementation, the interval arithmetic computations can be made conservative (Snyder, Woodburg, Fleischer, Currin, & Barr, 1993).

9 Conclusions and Limitations

In this article, we have presented a novel algorithm for continuous collision detection between a moving avatar and the virtual environment. Given discrete positions of the avatar, it uses an arbitrary in between motion to compute an interpolated path between the instances, dynamically computes a BVH around the links of the avatar, generates the SV of each potentially colliding link, and finally uses the graphics hardware to check for collisions with the environment. We have applied the algorithm to an avatar moving in a complex virtual environment composed of hundreds of thousands of polygons. Our initial results are quite promising and the algorithm is able to compute all the contacts, as well as the time of first possible collision within 10–30 ms. Moreover, our algorithm has been successfully integrated into an existing, fully immersive virtual environment, the GAITER system (Sibert et al., 2004) of the U.S. Naval Research Lab, where they utilize our algorithm to assess human performance in a training environment.

Our approach presented in this paper has a few limitations. These include:

- We use a relatively simple model for each link of the avatar using LSS. Furthermore, we assume that each link undergoes rigid motion.
- Our algorithm assumes that there are no loops in the articulated model.
- Our overall collision detection algorithm is approximate. The two main sources of error are the tessellation error that arises during polygonization of the SV and the image-space resolution used to perform

visibility queries. A technique to overcome the image-space resolution error has been described in (Govindaraju, Lin, & Manocha, 2004).

There are many avenues for future work. We would like to work on each of these limitations to improve the performance and applicability of our algorithm. We would like to apply it to more complex virtual environments and interface with virtual locomotion techniques (e.g., the Gaiter technique, see Templeman, Denbrook, & Sibert, 1999) for training and other applications. Moreover, we need to model the avatar more realistically to avoid collision artifacts such as "clotheslines" passing through the avatar neck. Recently, we proposed a new method to perform continuous collision detection for articulated, general polygonal models (Redon et al., 2004b); however, the algorithm does not guarantee interactive performance.

Acknowledgments

We would like to thank researchers at Naval Research Laboratory, especially Justin McCune, for providing the avatar's locomotion data and the UNC Walkthrough group for the model of Brook's house. This project is supported in part by the Office of Naval Research VIRTE Program, National Science Foundation, U.S. Army Research Office, Intel Corporation, and RDECOM. Young J. Kim was supported in part by the grant 2004-205-D00168 of KRF, the STAR program of MOST and the ITRC program in Korea. We are thankful to Naga Govindaraju for many useful discussions on GPU-based collision detection.

References

Abdel-Malek, K., Yang, J., Blackmore, D., & Joy, K. (2006). Swept volumes: Foundations, perspectives, and applications. *International Journal of Shape Modeling, 12*(1), 87–127.

Agarwal, P. K., Basch, J., Guibas, L. J., Hershberger, J., & Zhang, L. (2000). Deformable free space tiling for kinetic collision detection. *International Journal of Robotics Research, 21,* 179–197.

Cameron, S. (1990). Collision detection by four-dimensional intersection testing. *IEEE Transactions on Robotics and Automation, 6,* 291–302.

Canny, J. (1986). Collision detection for moving polyhedra. *IEEE Transactions Pattern Analysis and Machine Intelligence, 8*(2), 200–209.

Elber, G., Lee, I.-K., & Kim, M.-S. (1997). Comparing offset curve approximation methods. *IEEE Computer Graphics and Applications, 17*(3), 62–71.

Farouki, R. (1986). The approximation of non-degenerate offset surfaces. *Computer Aided Geometric Design, 3*(11), 15–43.

Filip, D., Magedson, R., & Markot, R. (1986). Surface algorithms using bounds on derivatives. *Computer-Aided Geometric Design, 3*(4), 295–311.

Foisy, A., & Hayward, V. (1994). A safe swept volume method for robust collision detection. *Sixth International Symposium on Robotics Research,* 61–68.

Govindaraju, N., Knott, D., Jain, N., Kabal, I., Tamstorf, R., Gayle, R., et al. (2005). Interactive collision detection between deformable models using chromatic decomposition. *ACM Transactions on Graphics, ACM SIGGRAPH,* 991–999.

Govindaraju, N., Lin, M., & Manocha, D. 2004. Fast and reliable collision culling using graphics processors. *Proceedings of ACM Symposium on Virtual Reality Software and Technology,* 2–9.

Govindaraju, N., Redon, S., Lin, M., & Manocha, D. (2003). CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware,* 25–32.

Harris, M. (2003). General purpose programming on GPUs. Available at http://www.gpgpu.org.

Hoff, K., Zaferakis, A., Lin, M., & Manocha, D. (2001). Fast and simple 2D geometric proximity queries using graphics hardware. *ACM Symposium on Interactive 3D Graphics,* 145–148.

Hoffmann, C. (1989). *Geometric and solid modeling.* San Mateo, CA: Morgan Kaufmann.

Hoschek, J. (1988). Spline approximation offset curves. *Computer Aided Geometric Design, 5*(1), 33–40.

Kearfott, R. B. (1996). Interval extensions of non-smooth functions for global optimization and nonlinear systems solvers. *Computing, 57*(2), 149–162.

Kieffer, J., & Litvin, F. (1990). Swept volume determination and interference detection for moving 3-D solids. *ASME Journal of Mechanical Design, 113,* 456–463.

Kim, K.-J., & Lee, I.-K. (2003). The perspective silhouette of a canal surface. *Computer Graphics Forum, 22,*(1), 15–22.

Kim, B., & Rossignac, J. (2003). Collision prediction for polyhedra under screw motions. *The Eight ACM Conference on Solid Modeling and Applications,* 4–10.

Kim, Y., Varadhan, G., Lin, M., & Manocha, D. (2003). Efficient swept volume approximation of complex polyhedral models. *ACM Symposium on Solid Modeling and Applications,* 11–22.

Kirkpatrick, D., Snoeyink, J., & Speckmann, B. (2000). Kinetic collision detection for simple polygons. *ACM Symposium on Computational Geometry,* 322–330.

Klass, R. (1983). An offset spline approximation for plane cubic splines. *Computer-Aided Design, 15*(5), 297–299.

Korein, J. U. (1985). *Geometric investigation of reach.* Cambridge, MA: MIT Press.

Kumar, S., & Manocha, D. (1995). Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design, 27*(7), 509–521.

Larsen, E., Gottschalk, S., Lin, M., & Manocha, D. (2000). Fast distance queries with rectangular swept sphere volumes. *Proceedings of the IEEE International Conference on Robotics and Automation,* 3719–3726.

Lin, M., & Manocha, D. (2003). Collision and proximity queries. In J. E. Goodman and J. O'Rourke, (Eds.), *Handbook of discrete and computational geometry* (pp. 787–807). Boca Raton, FL: CRC Press.

Liu, Y., & Badler, N. (2003). Real-time reach planning for animated characters using hardware acceleration. *Sixteenth International Conference on Computer Animation and Social Agents (CASA).*

Lok, B., Naik, S., Whitton, M., & Brooks, F. (2003). Incorporating dynamic real objects into immersive virtual environments. *Proceedings of ACM Symposium on Interactive 3D Graphics,* 701.

Manocha, D. (2002). *Interactive geometric computations using graphics hardware.* SIGGRAPH Course Notes, 31.

Moore, R. (1979). *Methods and applications of interval analysis.* SIAM studies in applied mathematics. Philadelphia: Society for Industrial and Applied Mathematics.

NVIDIA occlusion query. (2002). Available at http://oss.sgi.com/projects/ogi-sample/registry/NV/occlusion_query.txt.

Pottmann, H., & Wallner, J. (2001). *Computational line geometry.* Berlin: Springer.

Redon, S., Kheddar, A., & Coquillart, S. (2000). An algebraic solution to the problem of collision detection for rigid polyhedral objects. *Proceedings of IEEE Conference on Robotics and Automation,* 3733–3738.

Redon, S., Kheddar, A., & Coquillart, S. (2002). Fast continuous collision detection between rigid bodies. *Proceedings of Eurographics (Computer Graphics Forum).*

Redon, S., Kim, Y. J., Lin, M. C., & Manocha, D. (2004a). Interactive and continuous collision detection for avatars in virtual environments. *Proceedings of IEEE Virtual Reality.*

Redon, S., Kim, Y. J., Lin, M. C., & Manocha, D. (2004b). Fast continuous collision detection for articulated models. *Proceedings of ACM Symposium on Solid Modeling and Applications.*

Redon, S., & Lin, M. C. (2006). A fast method for local penetration depth computation. *Journal of Graphics Tools, 11*(2), 37–50.

Rossignac, J., & Kim, J. (2000). Computing and visualizing pose-interpolating 3-D motions. *Computer-Aided Design, 33*(4), 279–291.

Rossignac, J., Megahed, A., & Schneider, B. (1992). Interactive inspection of solids: Cross-sections and interferences. *Proceedings of ACM SIGGRAPH,* 353–360.

Schwarzer, F., Saha, M., & Latombe, J.-C. (2002). Exact collision checking of robot paths. *Workshop on Algorithmic Foundations of Robotics (WAFR).*

Sibert, L., Templeman, J., Page, R., Barron, J., McCune, J., & Denbrook, P. (2004). Initial assessment of human performance using the gaiter interaction technique to control locomotion in fully immersive virtual environments. Technical report, Washington, DC: Naval Research Laboratory.

Smoth, R. (2005). Open dynamics engine. Available at http://www.ode.org.

Snyder, J. (1992). Interval arithmetic for computer graphics. *Proceedings of ACM SIGGRAPH,* 121–130.

Snyder, J. M., Woodbury, A. R., Fleischer, K., Currin, B., & Barr, A. H. (1993). Interval method for multi-point collision between time-dependent curved surfaces. *Computer Graphics (SIGGRAPH '93), 27,* 321–334.

Templeman, J. N., Denbrook, P. S., & Sibert, L. E. (1999). Virtual locomotion: walking in place through virtual environments. *Presence: Teleoperators and Virtual Environments, 8*(6), 598–617.

Theoharis, T., Papaiannou, G., & Karabassi, E. (2001). The magic of the z-buffer: A survey. *The Ninth International Conference on Computer Graphics, Visualization and Computer Vision, WSCG.*

Xavier, P. (1997). Fast swept-volume distance for robust collision detection. *Proceedings of the International Conference on Robotics and Automation*.

Appendix: Interval Arithmetic

A good, general introduction to interval arithmetic can be found in the work by Kearfott (1996). Snyder (1992) has presented some applications of interval arithmetic to computer graphics, for example to obtain triangulations of implicit surfaces.

In our algorithm, we only use closed intervals. By definition, a closed real interval $[a, b]$ is:

$$I = [a, b] = \{x \in \mathrm{IR},\ a \leq x \leq b\}$$

This definition can be generalized to vectors in IR^n:

$$\begin{aligned} I_n &= [a_1, b_1] \times \ \dots \ \times [a_n, b_n] \\ &= \{\mathbf{x} = (x_1, \dots, x_n) \\ &\in \mathrm{IR}^n,\ a_i \leq x_i \leq b_i \forall_i,\ 1 \leq i \leq n\} \end{aligned}$$

The set of real intervals is denoted IIR, while the set of real vector intervals is denoted IIR^n. Elementary operations on real numbers can be transposed on real intervals:

$$\begin{aligned} [a, b] + [c, d] &= [a + c, b + d] \\ [a, b] - [c, d] &= [a - d, b - c] \\ [a, b] \times [c, d] &= [min(ac, ad, bc, bd), \\ &\qquad max(ac, ad, bc, bd)] \\ 1/[a, b] &= [1/b, 1/a] \\ &\qquad \text{if } a > 0 \text{ or } b < 0 \\ [a, b]/[c, d] &= [a, b] \times (1/[c, d]) \\ &\qquad \text{if } c > 0 \text{ or } d < 0 \\ [a, b] \leq [c, d] &\qquad \text{if } b \leq c \end{aligned} \tag{14}$$

For operations in IIR^n, the interval computations are performed for each coordinate. Note that an interval vector in IIR^n is an AABB.