Interactive and Continuous Collision Detection for Avatars in Virtual Environments

http://gamma.cs.unc.edu/Avatar

Stephane Redon[1] Young J. Kim[2] Ming C. Lin[1] Dinesh Manocha[1] Jim Templeman[3]

[1] University of North Carolina at Chapel Hill {*redon,lin,dm*}*@cs.unc.edu*

[2] Ewha Womans University, Korea *kimy@ewha.ac.kr*

[3] Naval Research Laboratory *templema@itd.nrl.navy.mil*

Abstract

We present a fast algorithm for continuous collision detection between a moving avatar and its surrounding virtual environment. 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 a swept volume (SV) and compute a bounding volume hierarchy to cull away links that are not in close proximity to the objects in the virtual environment. We generate the SV's of the remaining links and use them to check for possible interferences and estimate the time of collision between the surface of the SV and the objects in the virtual environment. Furthermore, we use graphics hardware to perform collision queries on the dynamically generated swept surfaces. Our overall algorithm requires no precomputation and is applicable to general articulated bodies. We have implemented the algorithm on a 2.4 *GHz Pentium IV PC with NVIDIA GeForce FX 5800 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* $10 - 30$ *milliseconds.*

1 Introduction

Collision detection is a fundamental geometric problem that arises in virtual reality (VR), physically-based modeling, robotics, computer-aided design, etc. Fast and accurate collision detection is crucial for many VR applications where the behavior of the avatars and objects in the virtual environment should mimic that in the real world. As we simulate avatar motion and behavior in a virtual environment, it is important to check for potential interferences 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 based on spatial partitioning or bounding volume hierarchies. However, there are two major limitations in using these algorithms for simulating avatar motion in the virtual environments. First, most of the algorithms check for interferences only at fixed 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. It is possible that the avatar's arms or limbs have collided with the virtual environment in-between time steps, as shown in Fig. 5. To overcome this limitation, we need collision detection algorithms that model the avatar's motion as a continuous path and check for interferences 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.

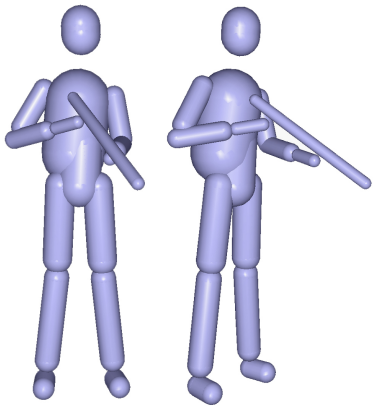

Figure 1. *Skeletal representation of a human avatar*

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 virtual environment, we model the avatar using a relatively simple model based on 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, i.e., line swept sphere (LSS), and these line segments are linked together to form an articulated body representing the avatar as shown in Fig. 1. The LSS is defined as the volume created by sweeping a sphere along a line segment [LGLM00]. Even though our chosen model for an avatar is simplistic, it is sufficiently effective for VR applications.

Furthermore, we assume that the configurations (i.e., po-

sition and orientation) of the line segment are 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 LSS's 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 swept 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 [GRLM03]. These queries are computed at an image-space resolution. To accelerate the computations, we also generate a dynamic bounding volume hierarchy (BVH) of the swept volumes based on interval arithmetic. It is used to cull away links that do not interfere with the environment. We have implemented the algorithm on a 2.4 GHz Pentium IV PC with NVIDIA GeForce FX 5800 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 continuously between the moving avatar and the environment in 10-30 milliseconds, as shown in Fig. 6 (color plate). Our algorithm is perhaps the first real-time continuous collision detection method for articulated avatars in virtual environments.

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

2 Previous Work

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

2.1 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 [AMBJ02]. Algorithms to compute and visualize the boundaries are presented in [KVLM03, RK00]. However, they are not fast enough for interactive applications.

A few algorithms have been proposed to use SV's for collision detection. [KL90] applied a SV-based interference detection to moving mechanical solids like gears, [Cam90] suggested a collision detection algorithm using four dimensional SV in the time and space domain, [Xav97] extended the GJK collision detection algorithm to handle a linear SV problem, and [FH94] also proposed a SV-based collision detection. However, none of these approaches address real-time collision detection for articulated bodies based on the SV.

2.2 Collision Detection

Most of the prior work on collision detection has focused on checking for collisions at discrete time instances. Check out [LM03] 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 bounding volume hierarchies.

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: algebraic equation solving approach [RKC00, Can86, KR03], swept volume (SV) approach [AMBJ02], adaptive bisection approach [RKC02, SSL02], and kinetic data structures (KDS) approach [ABG$^+$00, KSS00]. In practice, especially for 3D real time applications, the adaptive bisection approach has been shown to be useful.

2.3 Graphics Hardware-based Geometric Computations

Interpolation-based graphics hardware is increasingly being used for geometric applications [Man02]. 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, proximity queries including collision detection, morphing, object reconstruction, etc. A recent survey on different applications is given in [Har03, TPK01]. These include different algorithms for collision detection between closed objects [HZLM01, RMS92] as well as a recent algorithm for general and deformable objects that utilizes the visibility queries [GRLM03]. All of these algorithms perform computations in the image space and their accuracy is governed by the underlying pixel resolution.

3 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 mathematical formulation of performing continuous collision detection using SV's and present our pipeline that proceeds in five stages.

3.1 Swept Volume-based Collision Detection

Our goal is to check collisions between moving articulated human figures and its surrounding environment. We use a simple model of the avatar for collision detection and formulate each joint in the articulated figure as LSS (as shown in Fig. 1). As a result, the collision detection problem reduces to checking for collision between each moving LSS and the environment. We pose this problem as a swept volume (SV) problem; i.e. 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, $\mathbf{\Gamma}$, under rigid motions ($\boldsymbol{\Psi}(t)$ and $\boldsymbol{R}(t)$) can be expressed as the following equation:

$$\mathbf{\Gamma}(t) = \boldsymbol{\Psi}(t) + \boldsymbol{R}(t)\mathbf{\Gamma} \tag{1}$$

Here, $\boldsymbol{\Psi}(t)$ and $\boldsymbol{R}(t)$ 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 $\boldsymbol{R}(t)$, may contain general, non-rational functions such as a high order of trigonometric functions. Finally, the SV is defined as follows:

$$SV(\mathbf{\Gamma}) = \{ \cup \mathbf{\Gamma}(t) \mid t \in [0,1] \}, \tag{2}$$

where we assume that the time parameter t, is normalized to a unit time interval. In our formulation, the generator $\mathbf{\Gamma}$ is LSS. Notice that the medial axis of LSS corresponds to a line segment, and conversely, the offset surface of the line segment reconstructs the LSS. Therefore, the SV of 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 [PW01].

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

$$\boldsymbol{x}(t,s) = \boldsymbol{b}(t) + s\boldsymbol{\delta}(t) \tag{3}$$

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:

$$\boldsymbol{x}_d(t,s) = \boldsymbol{x}(t,s) \pm d\,\boldsymbol{n}(t,s), \tag{4}$$

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; i.e., each $\boldsymbol{n}(t,s)$ is uniquely defined.

We assume that LSS with radius d follows the sweep equation given in Eq. 1, and its medial axis at time t is thus parameterized as $\boldsymbol{x}(t,s)$. Then, the SV of the LSS following Eq. 1 is $\boldsymbol{x}_d(t,s)$ in Eq. 4. Mathematically speaking, our goal is to check intersections of $\boldsymbol{x}_d(t,s)$ with other objects in the environment.

3.2 Our Approach

The main challenge is to compute the offset surface of a ruled surface and quickly check whether the offset surface intersects with other objects. However, it is hard to compute the exact offset surface and check for interferences. This is due to the following reasons:

- It is challenging and still an open problem to compute the exact offset surface where the progenitor surface includes non-rational functions. Even when the progenitor can be described using regular NURBS, the offset surface can have self-intersections and singularities [Hof89]. As a result, computing an explicit representation of the offset surface is non-trivial.

- The problem of performing exact collision detection between high order or non-linear surfaces is considered hard in practice [LM03]. The underlying algorithms suffer from robustness and accuracy problems.

- The VR application demand interactive performance, i.e. 30 Hz or higher update rate. It is a major challenge to perform exact collision detection between curved primitives at such rates.

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 Fig. 2):

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

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

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

5. 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.

Figure 2. The overall pipeline of our collision detection algorithm. Different stages are performed on the CPU and the graphics processor.

4 Motion Formulation

In this section, we describe the motion formulation used 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 [RKC00, RKC02]. As is the case in many applications, the actual motion of the avatar is not known and we are only given its positions and orientations at discrete time instances. This is mainly due to the fact that the tracking device can only sample the position and orientation 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 avatar's motion.

Given these constraints, we *arbitrarily* 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[1], yet is simple enough to allow us to perform the various steps of our collision detection algorithm. Note that the arbitrary in-between motion used to detect collisions is also used to compute a position 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 inter-penetrations. Next, we give details of the specific arbitrary in-between motion used to compute the path between successive instances.

We assume that there is no loop in the graph describing the articulated figure. Consequently, each link has a *unique* parent link, except for the root node which has no parent. On the opposite, 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 3.(a) illustrates our notation for a link i moving within the reference frame of its parent.

For a given node i, let $\mathcal{P}_i$ 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 $\mathbf{L}_i^b$:

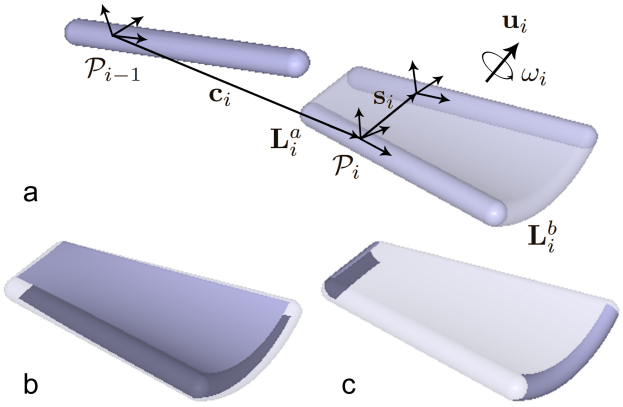

Figure 3. *a. Link i is moving in the reference frame of its parent. The initial and final positions are outlined. b. Offset of the rule surface. c. Pipe surfaces.*

$$\mathbf{L}_i^a = \begin{pmatrix} l_i^a \\ 0 \\ 0 \end{pmatrix} \quad \text{and} \quad \mathbf{L}_i^b = \begin{pmatrix} l_i^b \\ 0 \\ 0 \end{pmatrix} \tag{5}$$

Let's now describe the motion of $\mathcal{P}_i$ relatively to $\mathcal{P}_{i-1}$. We use the 3-dimensional vector $\mathbf{c}_i$ and the 3×3 matrix $\mathbf{R}_i$ to denote the position and orientation of $\mathcal{P}_i$ relative to $\mathcal{P}_{i-1}$ at the beginning of the time interval $[0, 1]$, respectively. We assume that the motion of $\mathcal{P}_i$ relatively to $\mathcal{P}_{i-1}$ is composed of a rotation of angle ω_i around an axis $\mathbf{u}_i$, and of a translation $\mathbf{s}_i$. 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 $\mathcal{P}_{i-1}$. Moreover, we assume that $\mathcal{P}_i$ moves with constant translation and rotation velocities.

The position of $\mathcal{P}_i$ relatively to $\mathcal{P}_{i-1}$ for a given time t in $[0, 1]$ is thus:

$$\mathbf{T}_i^{i-1}(t) = \mathbf{c}_i + t\mathbf{s}_i, \tag{6}$$

Let $\mathbf{u}_i^*$ denote the 3×3 matrix such as $\mathbf{u}_i^*\mathbf{x} = \mathbf{u}_i \times \mathbf{x}$ for every three-dimensional vector $\mathbf{x}$. If $\mathbf{u}_i = (u_i^x, u_i^y, u_i^z)^T$, then:

$$\mathbf{u}_i^* = \begin{pmatrix} 0 & -u_i^z & u_i^y \\ u_i^z & 0 & -u_i^x \\ -u_i^y & u_i^x & 0 \end{pmatrix} \tag{7}$$

The orientation of $\mathcal{P}_i$ relative to $\mathcal{P}_{i-1}$ is given as :

$$\mathbf{P}_i^{i-1}(t) = \cos(\omega_i t).\mathbf{A}_i + \sin(\omega_i t).\mathbf{B}_i + \mathbf{C}_i, \tag{8}$$

[1] 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.

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

$$\begin{aligned} \mathbf{A}_i &= \mathbf{R}_i - \mathbf{u}_i.\mathbf{u}_i^T.\mathbf{R}_i \\ \mathbf{B}_i &= \mathbf{u}_i^*.\mathbf{R}_i \\ \mathbf{C}_i &= \mathbf{u}_i.\mathbf{u}_i^T.\mathbf{R}_i \end{aligned} \tag{9}$$

Consequently, the motion of $\mathcal{P}_i$ relatively to $\mathcal{P}_{i-1}$ is described by the following 4×4 homogeneous matrix:

$$\mathbf{M}_i^{i-1}(t) = \begin{pmatrix} \mathbf{P}_i^{i-1}(t) & \mathbf{T}_i^{i-1}(t) \\ (0,0,0) & 1 \end{pmatrix} \tag{10}$$

resulting coordinates in the reference frame of the parent link $\mathcal{P}_{i-1}$. Consequently the matrix:

$$\mathbf{M}_i^0(t) = \mathbf{M}_1^0(t).\mathbf{M}_2^1(t)...\mathbf{M}_i^{i-1}(t) \tag{11}$$

describes the motion of link i in the world frame.

Note that our formulation makes it extremely simple to compute all the motion parameters $\mathbf{s}_i$, $\mathbf{u}_i$ and ω_i for a given timestep. For a given link i, assume that $\mathbf{c}_i^0$ and $\mathbf{c}_i^1$ (resp. $\mathbf{R}_i^0$ and $\mathbf{R}_i^1$) are the initial and final positions (resp. orientations) of $\mathcal{P}_i$ relatively to $\mathcal{P}_{i-1}$. Then $\mathbf{s}_i = \mathbf{c}_i^1 - \mathbf{c}_i^0$, and $(\mathbf{u}_i, \omega_i)$ is the rotation extracted from the rotation matrix $\mathbf{R}_i^1 (\mathbf{R}_i^0)^T$.

5 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 computation algorithm and use it to cull away some of the links that do not collide with the environment.

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 AABB's used to quickly cull away links which are far from the environment.

The leaf-boxes are computed using *interval arithmetic* [Moo79]. We only use closed intervals. For operations in $\mathbb{IR}^n$, the interval computations are performed for each coordinate. Note that an interval vector in $\mathbb{IR}^n$ is an AABB. Consequently, using interval arithmetic to bound the functions describing the trajectories of the links produces AABB's that enclose these trajectories. To bound the trajectory, we perform elementary interval arithmetic operations [Moo79] recursively on their expressions. We begin by bounding the sine and cosine functions from equation (8) 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 both $\mathbf{L}_i^a$ and $\mathbf{L}_i^b$, we obtain two 3-dimensional interval vectors that bound the coordinates of the endpoints of the links over the time interval $[0, 1]$. In other words, we obtain two AABB's that bound the endpoints' trajectories over the time interval. By using the 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. Given the AABB's 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 away a subset of the links that do not collide with the virtual environment.

6 Swept Volume Generation

In the previous section, we described an algorithm to cull away 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 of Line Swept Sphere

It is well known that the envelope (or swept volume) of a moving cylinder following continuous trajectory is equivalent to the offset surface of a ruled surface. 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. 4. Also notice that, in Eq. 4, $\boldsymbol{x}_d(u, v)$ is defined as a two-sided offset surface suited for our application. It is possible that $\boldsymbol{x}(u, v)$ may contain non-regular points. Some of the conventional techniques to handle such cases is to bound $\boldsymbol{n}(u, v)$ with a spherical polygon [PW01].

We extend the relationship between the offset of a ruled surface and the SV of a cylinder by computing the SV of LSS. This 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., SV of LSS). The SV generated by the caps of LSS is a *pipe surface*. As a matter of 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. A pipe surface is a special case of the canal surface where the radius is fixed. The parametric equation of a pipe surface can be given as follows [KL03]:

$$\begin{aligned} \boldsymbol{K}(t, \theta) &= \boldsymbol{C}(t) + R(\cos\theta \boldsymbol{b_1}(t) + \sin\theta \boldsymbol{b_2}(t)) \\ \boldsymbol{b_1}(t) &= \frac{\boldsymbol{C'}(t) \times \boldsymbol{C''}(t)}{\|\boldsymbol{C'}(t) \times \boldsymbol{C''}(t)\|} \\ \boldsymbol{b_2}(t) &= \frac{\boldsymbol{C'}(t) \times \boldsymbol{b_1}(t)}{\|\boldsymbol{C'}(t) \times \boldsymbol{b_1}(t)\|} \end{aligned} \tag{12}$$

Once we have computed the offset of ruled surface and pipe surface, 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.

6.2 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 free-form surface such as Bézier or NURBS surface. Under this assumption, there are three typical approaches to approximate an offset surface [ELK97]; 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 [Far86, Hos88, Kla83]. We adopt this technique in our application because of its simplicity and therefore, it is better suited 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. 4, 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.

6.3 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 $\|x_d(u,v) - x(u,v)\|^2 - d^2$ [ELK97]. The error is relatively easy to compute when the progenitor surface is represented as Bézier or NURBS surface. However, when the progenitor surface in our case is a non-rational surface described using trigonometric function. As a result, error calculation becomes non-trivial. We used iterative numerical techniques like the Newton-Raphson method to derive the error bound.

Another possibility to compute an error bound is to analyze the screen space error when the approximated surface is projected onto the screen space [KM95]. 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 estimate the time of collision.

There are two main challenges in performing collision detection using the SV. These include computing an accurate, explicit representation of the SV and checking it for 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 that precompute hierarchies to speed up the runtime queries. Instead, we choose the CULLIDE algorithm [GRLM03] 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. In particular, we uses the NVIDIA OpenGL extension GL_NV_occlusion_query[GL-02] 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 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.

We apply CULLIDE separately to the tessellated SV for each link that potentially overlaps with the environment.

7.2 Estimating the time of collision

In many VR applications we need to know more than whether an avatar is colliding with the environment. In particular, it is important to know the time and position when the first collision occurs. The simulation uses this information to compute an appropriate response. To estimate the time of collision, we present an extension to the CULLIDE algorithm. Specifically we estimate the time of collision (TOC) in two steps:

- We first estimate the upper bound on TOC from the time-parameterized SV.
- We refine the upper bound by backtracking.

When we tessellate the surface of SV (as explained in Sec. 6.2), we group the triangles that have the same time parameter, and consider each group as an input unit (e.g. triangle strip) used by CULLIDE. Next, CULLIDE computes which group is the first group (in terms of the time parameter) that is colliding with the environment. The time value associated with that group gives an upper bound to the TOC. Note that this upper bound does not usually match the actual TOC as the SV surface approximates only the envelope of a moving LSS, not the internal region as illustrated in Fig. 4.

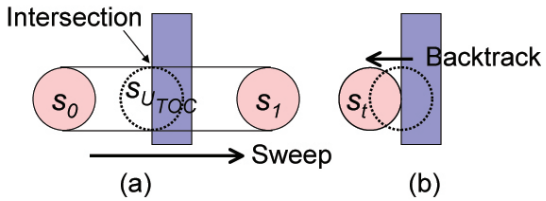

Figure 4. *Estimating the time of collision. (a) shows 2D cross sections of moving LSS (red object) and its SV as well as environment (blue object). The LSS is swept from the initial (S_0) position to the final position (S_1), and the collision detection routine will report the first time of collision of the SV against the environment at $S_{U_{TOC}}$. However, the internal region of the LSS at $S_{U_{TOC}}$ already penetrated the environment earlier, and $S_{U_{TOC}}$ is used as an initial position of backtracking to find a correct, first time of collision. (b) shows the result of the backtracking from the position at $S_{U_{TOC}}$ to S_t, where the first time of collision is correctly found.*

Given an upper bound to the TOC, say U_{TOC}, we refine U_{TOC} by backtracking the position of LSS in time until there is no collision between the LSS and environment. The unit for backtracking the time is set to the same value as the one used for tessellating the SV. We also use CULLIDE to check for interference each time the LSS is backtracked.

8 Implementation and Results

We have implemented our collision detection algorithm on a 2.4 GHz Pentium IV PC with NVIDIA GeForce FX 5800 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 collisions between the moving avatar and the environment in $10 - 30$ milliseconds, resulting in a refresh rate of $30 - 100$ frames per second. The collision queries are performed at an image-space resolution of 1024×768.

Our benchmark includes a client-server based application. The server updates the avatar's position every 10 milliseconds. The collision detection module is included as part of the client that requests new configurations when desired. Figure 6 shows our test environment along with some of the avatar trajectories and interactions. 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, which collides with a sofa shown in the last image.

The sequence in Fig. 5 highlights the benefit of our continuous collision detection method over traditional discrete methods. The left image shows two successive configurations of the avatar revealing 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 to stop the avatar and to determine a time interval $[0, t_c]$, $t_c < U_{TOC}$, over which there exists a collision-free path for all its links.

Table 1 shows an average computation time required for different steps within this environment. It highlights the time for different stages of the algorithm. That includes updating the position of the avatar, computing its motion parameters from two successive configurations, and determining the swept AABB's which bound the entire trajectories of the links using interval arithmetic. We obtain considerable speed-up by using a dynamically generated BVH of AABB's. 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 CULLIDE algorithm is given in [GRLM03].

Step	Average cost
Position update	15 μs
Motion parameters update	20 μs
Swept AABB's update (all links)	50 μs
AABB culling (all links)	20 μs
SV computation (one link)	300 μs
Interference detection (one link)	600 μs
Backtracking step (one link)	500 μs
Rendering the scene	10 ms

Table 1. *Average performance of the various steps of our algorithm for a 16-link avatar moving in the virtual environment composed of hundreds of thousands of polygons*

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 because of 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 results into a consistent state of the simulation with no interpenetration between the objects at any time.

9 Conclusions and Limitations

In this paper, 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 compute 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 moderately 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 $10 - 30$ milliseconds.

Our approach presented in this paper has some 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 errors are the tessellation error that arises during polygonization of the SV as well as the image-space resolution used to perform visibility queries. A technique to overcome the image-space resolution errors has been described in [GLM04].

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. Gaiter [Tem98]) for training and other applications. Currently we are investigating a more exact method to perform continuous collision detections for articulated, general polygonal models for applications with no stringent real-time performance constraints [RKLM03].

Acknowledgment

We would like to thank researchers at Naval Research Laboratory, especially Justin McCune, for providing the avatar's locomotion data and 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 the Ewha Womans University research grant of 2003.

References

[ABG+00] Pankaj K. Agarwal, J. Basch, L. J. Guibas, J. Hershberger, and L. Zhang. Deformable free space tiling for kinetic collision detection. In *Proc. 4th Workshop Algorithmic Found. Robot.*, 2000. To appear.

[AMBJ02] K. Abdel-Malekl, D. Blackmore, and K. Joy. Swept volumes: Foundations, perspectives, and applications. *International Journal of Shape Modeling*, 2002.

[Cam90] S. Cameron. Collision detection by four-dimensional intersection testing. *IEEE Transactions on Robotics and Automation*, 6:291–302, 1990.

[Can86] J. Canny. Collision detection for moving polyhedra. *IEEE Trans. Pattern and Mach. Intell.*, 8:200–209, 1986.

[ELK97] G. Elber, I.-K. Lee, and M.-S. Kim. Comparing offset curve approximation methods. *IEEE Computer Graphics and Applications*, 17(3):62–71, 1997.

[Far86] R. Farouki. The approximation of non-degenerate offset surfaces. *Computer Aided Geometric Design*, 3:15–43, 1986.

[FH94] A. Foisy and V. Hayward. A safe swept volume method for robust collision detection. In *Robotics Research, Sixth International Symposium*, 1994.

[GL-02] Nvidia occlusion query. http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt, 2002.

[GLM04] N. Govindaraju, M. Lin, and D. Manocha. Fast and reliable collision detection using graphics hardware. Technical report, University of North Carolina, Department of Computer Science, 2004.

[GRLM03] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. *Proc. of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 25–32, 2003.

[Har03] M. Harris. General Purpose Programming on GPUs. http://www.gpgpu.org, 2003.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, California, 1989.

[Hos88] J. Hoschek. Spline approximation offset curves. *Computer Aided Geometric Design*, 5(1), 1988.

[HZLM01] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. Fast and simple 2d geometric proximity queries using graphics hardware. *Proc. of ACM Symposium on Interactive 3D Graphics*, pages 145–148, 2001.

[KL90] J. Kieffer and F. Litvin. Swept volume determination and interference detection for moving 3-D solids. *ASME Journal of Mechanical Design*, 113:456–463, 1990.

[KL03] K.-J. Kim and I.-K. Lee. The perspective silhouette of a canal surface. In *Eurographics (Graphics Forum)*, volume 22, pages 15–22, 2003.

[Kla83] R. Klass. An offset spline approximation for plane cubic splines. *Computer-Aided Design*, 15(5):297–299, 1983.

[KM95] S. Kumar and D. Manocha. Efficient rendering of trimmed nurbs surfaces. *Computer-Aided Design*, pages 509–521, 1995.

[KR03] B. Kim and J. Rossignac. Collision prediction for polyhedra under screw motions. In *ACM Conference on Solid Modeling and Applications*, June 2003.

[KSS00] David Kirkpatrick, Jack Snoeyink, and Bettina Speckmann. Kinetic collision detection for simple polygons. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 322–330, 2000.

[KVLM03] Y. Kim, G. Varadhan, M. Lin, and D. Manocha. Efficient swept volume approximation of complex polyhedral models. *Proc. of ACM Symposium on Solid Modeling and Applications*, pages 11–22, 2003.

[LGLM00] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Distance queries with rectangular swept sphere volumes. *Proc. of IEEE Int. Conference on Robotics and Automation*, 2000.

[LM03] M. Lin and D. Manocha. Collision and proximity queries. In *Handbook of Discrete and Computational Geometry*, 2003.

[Man02] D. Manocha. *Interactive Geometric Computations using Graphics Hardware*. SIGGRAPH Course Notes # 31, 2002.

[Moo79] R.E. Moore. *Methods and applications of interval analysis*. SIAM studies in applied mathematics. Siam, 1979.

[PW01] H. Pottmann and J. Wallner. *Computational Line Geometry*. Springer, 2001.

[RK00] J. Rossignac and J. Kim. Computing and visualizing pose-interpolating 3-D motions. *Computer-Aided Design*, 2000.

[RKC00] S. Redon, A. Kheddar, and S. Coquillart. An algebraic solution to the problem of collision detection for rigid polyhedral objects. *Proc. of IEEE Conference on Robotics and Automation*, 2000.

[RKC02] S. Redon, A. Kheddar, and S. Coquillart. Fast continuous collision detection between rigid bodies. *Proc. of Eurographics (Computer Graphics Forum)*, 2002.

[RKLM03] S. Redon, Y. Kim, M. Lin, and D. Manocha. Fast continuous collision detection for articulated models. Technical Report TR03-038, University of North Carolina, Department of Computer Science, 2003.

[RMS92] J. Rossignac, A. Megahed, and B.D. Schneider. Interactive inspection of solids: cross-sections and interferences. In *Proceedings of ACM Siggraph*, pages 353–60, 1992.

[SSL02] F. Schwarzer, M. Saha, and J.-C. Latombe. Exact collision checking of robot paths. In *Workshop on Algorithmic Foundations of Robotics (WAFR)*, Dec. 2002.

[Tem98] J. Templeman. Performance based design of a new virtual locomotion control. In *NATO RSG-28 Human Factors Issues in Virtual Reality Technologies Conference Proceedings*, 1998.

[TPK01] T. Theoharis, G. Papaiannou, and E. Karabassi. The magic of the z-buffer: A survey. *Proc. of 9th International Conference on Computer Graphics, Visualization and Computer Vision, WSCG*, 2001.

[Xav97] P. Xavier. Fast swept-volume distance for robust collision detection. In *Proceedings of International Conference on Robotics and Automation*, 1997.

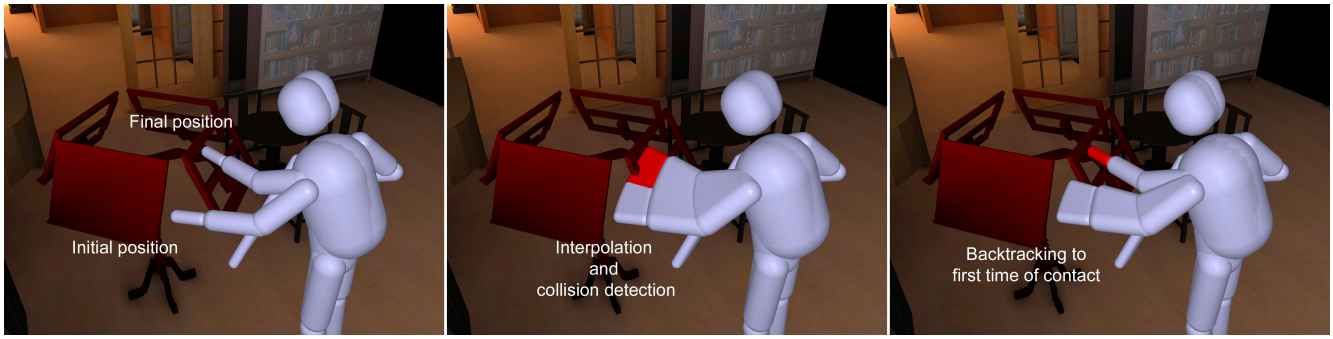

Figure 5. *Benefits of our continuous collision detection algorithm over discrete methods. The left image shows two successive configurations of the avatar revealing a fast arm motion. No collision is detected at these discrete time steps. The middle image shows the interpolating path used to detect a collision between these two configurations. The right image shows the backtracking step used to compute the time of collision and the avatar position at that time. It highlights the time interval over which there is no collision with the virtual environment.*

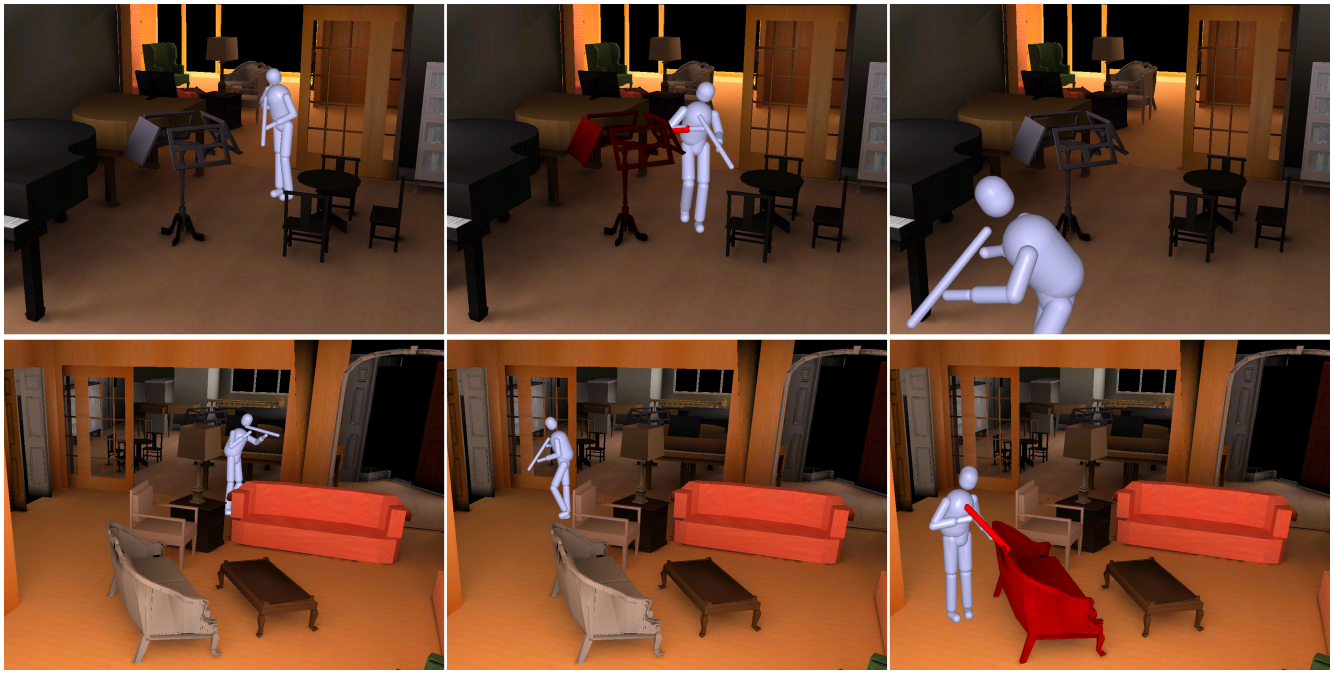

Figure 6. *The benchmark environment and the avatar model used to test the performance of our algorithm. Top row: the avatar visiting the music room. In the middle image, its lower right arm collides the music stand. Lower row: the avatar in the living-room colliding with the sofa in the rightmost image.*