A Collision Detection Framework for Deformable Objects

Rynson W.H. Lau[†‡]
rynson@cs.cityu.edu.hk

Oliver Chan[†]
four4@cs.cityu.edu.hk

Mo Luk[†]
mo@cs.cityu.edu.hk

Frederick W.B. Li[!]
borbor@cs.cityu.edu.hk

[†] Department of Computer Science, City University of Hong Kong, Hong Kong
[‡] Department of CEIT, City University of Hong Kong, Hong Kong
[!] Research Center for Media Technology, City University of Hong Kong, Hong Kong

Abstract

Many collision detection methods have been proposed. Most of them can only be applied to rigid objects. In general, these methods precompute some geometric information of each object, such as bounding boxes, to be used for run-time collision detection. However, if the object deforms, the precomputed information may not be valid anymore and hence needs to be recomputed in every frame while the object is deforming. In this paper, we presents an efficient collision detection framework for deformable objects, which considers both inter-collisions and self-collisions of deformable objects modeled by NURBS surfaces. Towards the end of the paper, we show some experimental results to demonstrate the performance of the new method.

Categories and Subject Descriptors

I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Geometric algorithms, languages, and systems; Curve, surface, solid, and object representations*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Interaction techniques*

Keywords

Deformable objects, NURBS surfaces, collision detection, interference detection

1 Introduction

Although deformable objects are important in modeling clothing, facial expression, and animal characters, they are rarely used in interactive applications such as virtual environments and computer games. This is mainly due to the high computational cost of tessellating the deformable surfaces into polygons for rendering as the objects deform. One of our current research directions is to develop techniques to support real-time handling of deformable objects. Earlier, we proposed an incremental method for efficient rendering of deformable NURBS surfaces [16]. In this paper, we investigate the collision detection problem of deformable objects.

There are two problems that we need to address in detecting collisions of deformable objects, *inter-collision detection* and *intra-collision detection* (also referred as *self-collision detection*). Inter-collision detection concerns if two objects are colliding with each other, while self-collision detection concerns if an object is colliding with itself. For rigid objects, since their shape does not change, only inter-collision detection needs to be considered. In this paper, we propose an efficient collision detection framework for deformable objects, which addresses both problems. The framework is based on incremental updating and integrates well with our deformable NURBS rendering method [16]. The main contributions of this paper are as follows:

- a framework for collision detection of deformable objects.
- an efficient method for detecting self-collisions of deformable objects.
- an incremental mechanism for the self-collision detection method.

The rest of the paper is organized as follows. Section 2 describes previous work on collision detection and briefly reviews our deformable NURBS rendering method. Section 3 presents an overview of our collision detection framework for deformable NURBS objects. Sections 4 and 5 discuss how we detect inter-collisions and self-collisions of deformable NURBS objects, respectively. Section 6 evaluates the performance of the proposed method through experimental results. Finally, section 7 briefly concludes the paper.

2 Related Work

Many collision detection methods have been proposed. Most of them are for rigid objects only, and very few of them may handle deformable objects. In this section, we briefly look at these two kinds of methods. In addition, we also briefly review the deformable NURBS rendering method that we developed earlier.

2.1 *Collision Detection of Rigid Objects*

There are three approaches for detecting collisions of rigid objects, *hierarchical space subdivision*, *hierarchical object subdivision* and *incremental distance computation*. In hierarchical space subdivision, the environment is subdivided into a space hierarchy. Objects in the environment are clustered hierarchically according to the regions that they fall into. When an object in the environment changes its position and moves into another region, only objects (or primitives) within the new region need to be checked for collision with the incoming object. There are many ways of subdividing the environment. In [19], the environment is subdivided into an octree. In our earlier work [3], a walkthrough environment is subdivided into a quadtree for quick determination of visible objects. In [9], the environment is subdivided into a balanced binary tree called the *K-d tree*, while in [5], the environment is subdivided into a BSP tree.

In hierarchical object subdivision, each object is subdivided into a bounding hierarchy, i.e., a hierarchy of bounding volumes. Checking for collisions is by determining if the bounding hierarchies of the objects intersect each other. Methods of this approach including using spheres [11] and axis-aligned bounding boxes (AABB's) [5] as the bounding volume. Although both methods are efficient in determining if two bounding hierarchies intersect each other, they do not produce tight bounding volumes. In the case of AABB, the bounding volumes may need to be recomputed even if the object only performs a rigid transformation such as rotation. In [6], oriented bounding boxes (OBB's) are used as the bounding volume. The idea of this method is to find the best orientation in fitting each bounding box to minimize the bounded volume, and hence this method produces much tighter bounding volumes and does not require the bounding hierarchy to be rebuilt when the object performs a rigid transformation. However, the limitation is that it takes a much longer time to construct the bounding hierarchy and to determine if two bounding boxes intersect due to the need for realignment. Other methods of hierarchical object subdivision include using discrete oriented polytopes [13], quantized oriented slabs with primary orientation [8], and the combination of spherical shells and oriented bounding boxes [14].

In incremental distance computation, the minimum distance (or the closest features) between each pair of objects is incrementally tracked so that their collisions can be determined efficiently. Methods of this approach include [7, 18, 21]. Related methods include [4], which maintains a separating plane between each pair of objects, and [17], which incrementally update a list of intersecting cells.

2.2 *Collision Detection of Deformable Objects*

Only a few methods have been proposed for detecting inter-collisions of deformable objects. Even fewer methods consider self-collisions of these objects. Most of these collision detection methods are based on the hierarchical object subdivision approach and can roughly be divided into two types, one targeted for objects represented by polygon meshes and the other by parametric surfaces.

Objects represented by polygon meshes are usually deformed by updating the positions of the polygon vertices. In [23], a hierarchical octree method was proposed. Each node of the octree is a rectangular bounding box aligned to the global axes. Although the method is simple, it needs to update the complete bounding hierarchy if the object deforms. No performance results were given. In [2], a method to build a local AABB tree for each object was proposed, which is essentially a binary tree of bounding boxes aligned to the local coordinate space of the object. The intersection of two bounding boxes is determined by a separating axis [6]. Instead of rebuilding, bounding boxes are refitted as the object deforms. While this simplifies the bounding box updating, it often increases the overlapping areas of the bounding boxes. This method also requires all the nodes to be updated if the object deforms.

In [22], a self-collision detection method was proposed. A hierarchy of polygon patches is first created. The polygon normals and the projected patch contours are analyzed to determined if self-collision occurs. This method may run efficiently as only simple bitwise operations are involved in analyzing polygon normals. In addition, experimental testings show that the patch contour test may be ignored without affecting the result of self-collision detection.

Objects represented by parametric surfaces are usually deformed by updating the surface control points. Such a deformation often leads to a retessellation of the surfaces, if the output quality of the surfaces are to maintain. In [10], a method was proposed to determine the collision between two deformable parametric surfaces, but it requires the surfaces to have computable Lipschitz values. The major limitation of the method is the computational cost. In [12], a collision detection method for Bézier and B-spline surfaces was proposed. The method constructs an AABB tree for each surface. It uses the pseudo-normal patch and Gauss map to detect self-collisions and the sweep-and-prune method to detect other collisions. However, constructing the pseudo-normal patch, calculating new surface points and updating the complete AABB tree must be performed in every frame during surface deformation. In addition, this method needs to resolve a large number of algebraic equations to determine if a surface is self-colliding. All these processes are computationally expensive.

In general, if the deformation of an object is based on updating a polygon model, the corresponding collision detection methods can be more efficient because they involve only updating the bounding hierarchy as the object deforms. However, these methods usually assume a fixed model resolution. Hence, the amount of deformation is restricted or the object may appear polygonal. On the other hand, if the deformation of an object is based on updating a surface model, surface points may need to be dynamically computed or removed. The corresponding collision detection methods are usually less efficient because all the bounding boxes need to be reconstructed as the object deforms. The method proposed here is for objects modeled by NURBS surfaces and it solves both problems discussed above. First, the object may deform freely because it is modeled by NURBS surfaces. Second, the surface points can be incrementally computed in a very efficient way. Third, the updating of the bounding hierarchy when the object deforms is also very efficient because the update is limited to the affected region of the deformation as will be described later.

2.3 *Deformable NURBS Rendering Method*

In [16], we proposed a very efficient method for rendering deformable NURBS surfaces. This method considers the incremental property of deforming NURBS surfaces. Initially, we precompute a polygon model that represents the shape of each NURBS surface. For each vertex of the polygon model, we also precompute a set of deformation coefficients. As the surface deforms, this precomputed polygon model is not recomputed, it is incrementally updated to represent the deforming surface. This updating is based on two fundamental techniques: *incremental polygon model updating* and *resolution refinement*. As a surface deforms, incremental polygon model updating dynamically computes the new positions of the polygon model to represent the deforming surface, while resolution refinement monitors the local curvature of the surface and refine the resolution of the polygon model in order to maintain its smoothness.

Our earlier results show that this rendering method is roughly 5 to 15 times faster than existing methods [16]. Recently, we have extended the rendering method for different types of parametric surfaces [15]. In addition, we have also considered the fact that an object may be modeled by many surface patches. We propose the hierarchical surface technique, which combines adjacent surface patches hierarchically to form a multi-resolution polygon model to represent the whole object. This allows us to produce a polygon model with optimized resolution for each deformable object. This hierarchical surface technique also helps simplify the implementation of our collision detection method proposed here.

3 Collision Detection of Deformable Objects

The main objective of this project is to develop a collision detection method for deformable objects suitable for real-time applications. This implies that the new method must be able to update all the data structures, in particular the bounding hierarchy, within a time much shorter than a frame time, as the object deforms. Hence, we have chosen the local AABB method, which fits bounding boxes to an object at its local coordinate space. Its advantages are that the bounding hierarchy is fast to compute and there is no need to update the bounding hierarchy if the object performs a rigid transformation. We have also considered the fact that very often only a small part of the object is actually deforming at any particular time. We may save a lot of computation time if we update only the region of the bounding hierarchy affected by the deformation, provided that if we can identify this region. Our collision detection method addresses this issue.

Similar to our NURBS rendering method, our collision detection method has two stages, the *preprocessing stage* and the *run-time stage*. Figure 1 shows an outline of our method and the interactions between the collision detection method and the rendering method. There are two main tasks in the preprocessing stage executed by Procedure **PreprocessingTasks**. The first task is to construct a local AABB bounding hierarchy for each deformable object. This will be described in this section. The second task is to compute the bitfield for each node of the bounding hierarchy. We will discuss this in section 5.2.

During run-time, some objects may need to perform a (rigid or non-rigid) transformation. Procedure **RunTimeTasks** identifies objects that undergo transformation. For each of these objects, we update its data structure with the transformation information. If the object is undergoing a non-rigid transformation, we determine the deformation region, the region of the object affected by the deformation, and then perform a self-collision detection of the object. Finally, we check both types of objects for inter-collision with each of the other objects.

For the rest of this section, we briefly describe how to construct the bounding hierarchy, perform transformation update and determine the deformation region. In the next two sections, we describe the inter-collision detection process and the self-collision detection process.

```
Procedure PreprocessingTasks
{
  construct local AABB bounding hierarchy;
  compute the bitfields of the hierarchy;
}

Procedure RunTimeTasks
{
  FOR each object in the environment DO
      IF object undergoes rigid transformation THEN
          perform object transformation update;
          perform inter-collision detection with other objects;
      ELSE-IF object undergoes non-rigid transformation THEN
          determine the deformation region;
          perform object transformation and bitfield update;
          perform self-collision detection;
          perform inter-collision detection with other objects;
      ENDIF
  ENDFOR
}
```

Figure 1: The outline of our collision detection framework in pseudo-code.

3.1 *Construction of the Bounding Hierarchy*

As mentioned earlier, our method constructs a local AABB bounding hierarchy for each object. When the NURBS rendering algorithm constructs the initial polygon model for each deformable object during the preprocessing stage, it also constructs a quad-tree of the polygon model, with the leaf nodes representing individual polygons and the root node representing the whole object. To construct the bounding hierarchy, we simply start from the leaf nodes of the quad-tree and determine the bounding box of each polygon based on the local coordinate of the object. Once the bounding boxes of all the leaf nodes have been determined, the bounding box of each parent node can be computed recursively from the bounding boxes of its four child nodes.

3.2 *Object Transformation Update*

There are two kinds of object transformation, rigid transformation and non-rigid transformation. Rigid transformation includes object translation and rotation. In our method, if an object performs a rigid transformation, there is no need to update the surface model, the polygon model nor the bounding hierarchy of the object as they are all maintained in the local coordinate of the object. We only need to update the transformation matrix associated with the object, which will be used to determine the actual position of the object in world coordinate. On the other hand, if an object performs a non-rigid transformation, we need to update the surface model, the polygon model and the bounding hierarchy. Updating the surface model and the polygon model of the object is performed by the NURBS rendering algorithm. Updating the bounding hierarchy is performed by our method. It involves updating all the affected bounding boxes and bitfields of the bounding hierarchy to reflect the deformation.

3.3 *Determining the Deformation Region*

We notice that when an object deforms, very often only a small part of the object is affected. The deformation is normally achieved by moving one or more control points of the objects. We refer to the region of the object affected by the displacement of a single control point as the *deformation region*. Both the bounding hierarchy updating process and the self-collision detection process will benefit from this optimization. Hence, if an object deforms during a particular frame, we first determine the deformation region. In the NURBS rendering algorithm, we only need to update and refine the resolution of the polygon model within this deformation region. We then update all the nodes of the bounding hierarchy that are within the deformation region, and check to see if any of these nodes may collide with any other nodes of the same object.

NURBS surfaces have the local modification property [20]. If the position of a control point $P_{i,j}$ is changed, only the shape within the parameter region $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$ of the surface is affected, where p and q are the degrees of the NURBS surface along u and v parameter directions, respectively. This is the deformation region. Hence, if a NURBS surface deforms, the updating processes only need to be applied to this region. In order to quickly determine which nodes of the bounding hierarchy are inside a given deformation region, we perform the comparison on the Bézier patches generated through knot insertion. We notice that every Bézier patch has a well-defined parameter boundary. If a Bézier patch does not intersect with the deformation region, its quad-tree nodes will also not intersect with it. Hence, we may store the parameter range in each Bézier patch rather than in each quad-tree node to reduce both computational and memory costs.

4 Inter-Collision Detection

When two objects collide in a dynamic environment, at least one of the objects must be undergoing either rigid or non-rigid transformation. Instead of checking each object within a region for possible collision with other objects, we identify objects that undergo rigid or non-rigid transformations during a frame time and test each of them with each of the other nearby objects for possible collision.

4.1 *Bounding Hierarchy Collision Test*

To determine if two objects collide with each other, we check if the bounding hierarchies of the two objects intersect each other. We start from the root nodes and check if the bounding boxes of the two root nodes intersect in space. If they do not, the two objects do not collide. If they do, the two objects may or may not collide and we move down the hierarchies and repeat the above operations with the child nodes. However, since each parent node has four child nodes in a quad-tree, if we test all the child nodes of the two parent nodes for collision, we will need to perform a total of 16 bounding box intersection tests. Instead, we subdivide only one of the parent nodes and test the other parent node with each of the child nodes of the subdivided parent node. Here, we choose the parent node with a lower subdivision level, i.e., the one nearer to the root of the hierarchy, for subdivision. If both parent nodes are at the same subdivision level, we just choose either one for subdivision. This approach helps quickly eliminate child nodes that do not intersect.

If the bounding box intersection test reaches the leaf nodes of the bounding hierarchies and the bounding boxes of two leaf nodes intersect each other, we will need to check whether the polygons bounded by the two bounding boxes intersect each other. If they do, the two objects are said to have collided and they collide at the position where the polygons intersect.

4.2 *Bounding Box Collision Test*

To determine if two bounding hierarchies intersect each other, we may need to perform a large number of tests to check if two bounding boxes intersect each other. These bounding boxes have arbitrary orientations since they are constructed in the local coordinates of the two objects. Here, we simply apply the SAT lite method proposed in [2] for testing if two bounding boxes intersect each other. Basically, this method aligns the two bounding boxes to the local coordinate system of each of them and test for overlapping in each axis. If the two bounding boxes do not overlap in any of the six axis test, they do not intersect each other. Otherwise, we assume that they intersect each other. (Note that there is still a small possibility here that the two bounding boxes do not intersect each other. See [2] for detail.)

5 Self-Collision Detection

Our self-collision detection method improves on the method proposed by [22] in several ways. First, we identify all possible node testings in advance. Given one or two nodes to test for self-collision in run-time, we can immediately determine all the testing operations needed to be carried out, as shown in section 5.2. Second, we propose a technique to quickly identify if two nodes are adjacent to each other, as shown in section 5.3. Finally, we propose a self-collision list to support incremental self-collision detection, as shown in section 5.4.

5.1 *Node Self-Collision Test*

This is a basic operation to determine if a node may be self-colliding. Our method here is essentially the same as [22]. It is based on the property that a surface is not possibly self-colliding if there exists a vector that has positive dot product with all the polygon normals of the surface and the 2D projection of the surface's contour does not intersect itself. This property is also applied to two adjacent surfaces, which have at least one shared vertex. To implement this test efficiently, we adopt the vector direction sampling method to perform self-collision detection. We represent the direction space by a unit sphere, and we sample the sphere to get 14 well-distributed normalized sample vectors. For each node in the bounding hierarchy, we associate a bitfield of 14 bits. A bit of the bitfield is set to one if the corresponding sampling vector in the sphere has a positive dot product with the polygon normal of the node. At each frame, we update the bitfield of each leaf node which is inside the deformation region and propagate the result up the bounding hierarchy through logically **AND**'ing the bitfields. If the resulting bitfield of a particular node is not zero, the node has no self-collision. Otherwise, the node may or may not be self-colliding. This test can also be applied to adjacent node pairs through logically **AND**'ing the bitfields of the two adjacent nodes.

5.2 *Object Self-Collision Test*

To perform self-collision detection of a deformable NURBS object, we recursively traverse the quad-tree of the polygon model representing the deformable object starting from the root node, and categorize the quad-tree nodes into three types of test units. Each test unit may contain a single node, two adjacent nodes with at least one shared vertex, or two non-adjacent nodes. These test units are categorized and processed as follows:

- A *Type 1* test unit contains only a single quad-tree node. We determine if the node is possibly self-colliding by the node self-collision test. If the node may be self-colliding, we will check for self-collision on its child nodes, which will generate four *Type 1* and six *Type 2* test units as shown in Figure 2.

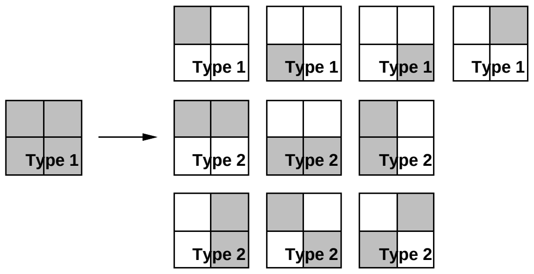

Figure 2: Subdivision of a type 1 test unit.

- A *Type 2* test unit contains two adjacent quad-tree nodes, which have at least one shared vertex. First, we determine if the adjacent node pair is possibly self-colliding by the node self-collision test. If the node pair may be self-colliding, we then combine the node of smaller curvature with each of the four child nodes of the other node to generate either two *Type 2* test units and two *Type 3* test units as shown in Figure 3(a) or one *Type 2* test unit and three *Type 3* test units as shown in Figure 3(b).

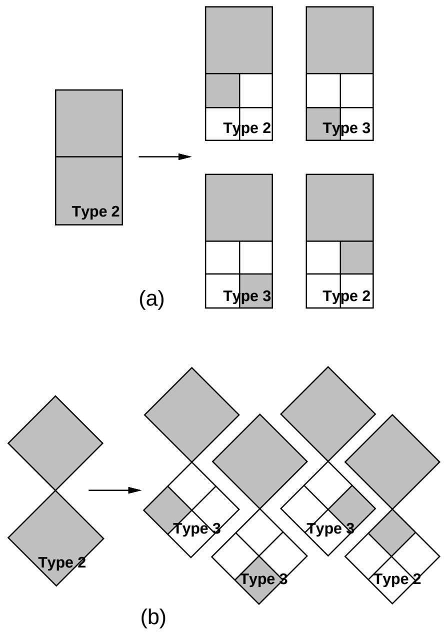

Figure 3: Subdivision of a type 2 test unit.

- A *Type 3* test unit contains two non-adjacent quad-tree nodes. We perform bounding box collision test on the two nodes to determine if they may be colliding. If they may, we will combine the node of smaller curvature with each of the four child nodes of the other node to generate four *Type 3* test units as shown in Figure 4.

As a summary, to determine whether an object is self-colliding, we initially check the root quad-tree node as a *Type 1* test unit. If the node may be self-colliding, we then check its child nodes recursively, until a node is found to have no self-collision, or we have reached the leaf nodes. In the case of leaf nodes, if they are of *Type 1* or *Type 2*, there is no self-collision. However, if the leaf nodes are of *Type 3*, we need to perform an exact polygon collision test to determine if they are self-colliding.

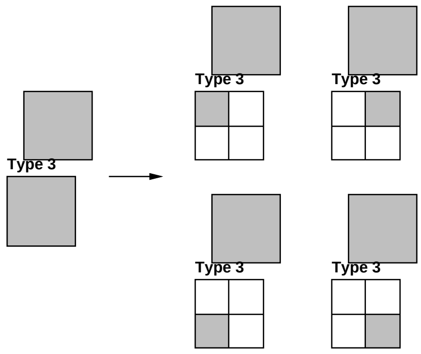

Figure 4: Subdivision of a type 3 test unit.

5.3 *Patch Adjacency Test*

To determine if a deformable NURBS object is self-colliding, our method needs to categorize the test units into different types based on the adjacency of quad-tree nodes. In [22], a circular list that contains the adjacent nodes around the contour of each node is maintained to help determine if two nodes are adjacent to each other. This method, however, requires $O(n)$ storage space and has $O(\log n)$ search time complexity. Here, we propose a simple method to determine node adjacency with no additional memory cost and only $O(1)$ computation complexity. We first assign unique indices all nodes within each level of the quad-tree. This index basically indicates the 2D location of a node in the grid of nodes at the level. Figure 5 shows the node indices for two levels of nodes.

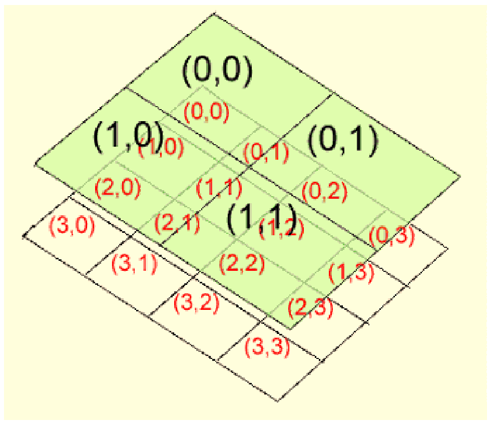

Figure 5: Indexing of each level of nodes.

Suppose that we have a test unit of two nodes, M and N, located at level l_M and l_N, respectively, as shown in Figure 6, where N is at a deeper level, i.e., $l_M < l_N$. To determine if M and N are adjacent to each other, we first calculate their level difference, $d = |l_N - l_M|$. We then calculate an adjacency region, R, with (x_1, y_1) and (x_2, y_2) being the minimum and maximum corners, respectively, as follows:

$$\begin{aligned}(x_1, y_1) &= ((x \ll d) - 1, (y + 1) \ll d) \\ (x_2, y_2) &= ((x + 1) \ll d, (y \ll d) - 1) \qquad (1)\end{aligned}$$

where (x, y) is the node index of N. Since the object self-collision detection test has already guaranteed that N will not be a child node of M, we can simply check if N is inside R, which involves only four comparison operations. If it is, M and

N are adjacent nodes; otherwise, they are not.

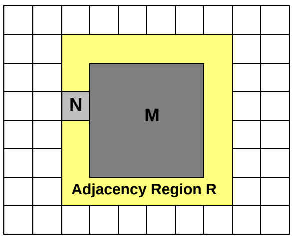

Figure 6: The adjacency region.

5.4 *Incremental Self-Collision Detection*

We note that if a node is detected to be self-colliding at one frame, it is likely that either the same node and/or some nearby nodes will be self-colliding in the next frame. Likewise, if a node is detected to have no self-collision at one frame, it is likely that either the same node and/or some nearby nodes will have no self-collision in the next frame. Hence, we may reduce the number of nodes to check for self-collision by considering this frame coherency property. Our method to do this is to maintain a *self-collision list* to keep track of the nodes that are detected to have either self-collision or no self-collision. This is similar in concept to the separation list [17] for detecting object inter-collisions.

During the first frame, we traverse the quad-tree of a deformable object starting from the root node. As soon as we have found a node without self-collision, we will stop traversing down the branch and store the node in the self-collision list. On the other hand, if we reach a leaf node and conclude that it is self-colliding with another leaf node, we store both nodes in the self-collision list also. At the end of the first frame, the self-collision list will contain the highest level nodes at which no self-collision is detected and the leaf nodes at which self-collision is detected. In the next frame, we no longer need to traverse the complete quad-tree. We may simply check the self-collision list from the previous frame. We traverse this list and check the status of each node. If a node is found to have no self-collision in the current frame, we recursively move up the quad-tree and check its parent node for self-collision. The lowest level node that we find without self-collision is placed in the self-collision list, replacing all its child nodes found in the list.

If a node is found to be self-colliding in the current frame but no self-collision in the previous frame, we recursively move down the quad-tree until we have found nodes that indicate no self-collision or until we have reached the leaf nodes that indicate self-collision. We will then replace the original node in the list with the new nodes. However, if a leaf node is found to be self-colliding in both the current and the previous frames, we may just leave the node in the list. After processing all the nodes in the list, the updated list may then be carried to the next frame and the process repeats for the new frame. The decision on updating the collision list is summarized in Figure 7.

6 Results and Discussions

We have implemented both the NURBS rendering method and the proposed collision detection method in C++ and OpenGL. In this section, we show and analyze the performance results of the inter-collision as well as the self-collision detection algorithms. All the experiments presented here were performed on a PC with

Previous Frame / Current Frame	No Self-Collision	Self-Collision Detected
No Self-Collision	Recursively check parent nodes to find nodes without self-collision.	Recursively check parent nodes to find nodes without self-collision.
Self-Collision Detected	Recursively check child nodes to find highest node without self-collision or leaf nodes with self-collision	Leave the nodes in the collision list.

Figure 7: Update decision on the collision list.

a Pentium III 450MHz CPU and 128M bytes of RAM.

6.1 *Inter-Collision Detection*

To study the performance of the inter-collision detection algorithm, we compare it with the AABB method proposed by UNC. We have created a fish model for use in our experiments. The fish model is composed of two NURBS patches. Figures $8(a)$ and $8(b)$ show the fish model with mapped texture and in wireframe, respectively.

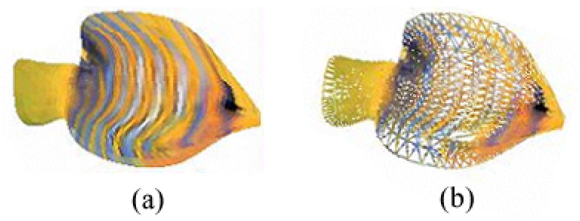

Figure 8: The fish model used in our experiments, (a) with mapped texture and (b) in wireframe.

Figure 9 shows the performance comparison of the two methods in updating the bounding hierarchy in each frame as the fish model deforms. Since the performance of our method in updating the bounding hierarchy depends on the size of the deformation region, we have tested it when the sizes of the deformation region are 6.25%, 25% and 100%. During the experiments, the number of polygons in the fish model varied between 4,200 and 4,600. When the size of the deformation region was 100%, the performance of our method was about 32% faster than the AABB method. However, when the size of the deformation region dropped to 6.25%, it was almost 6 times faster.

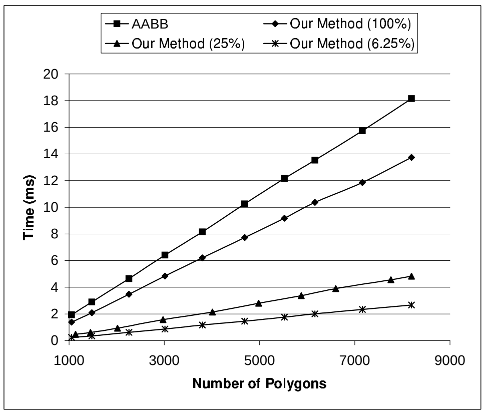

Figure 9: The update costs of the AABB and our methods when the fish model deforms.

Figure 10 shows the performance comparison of the two methods in detecting collisions when two fishes are deforming as they swim towards each other. During the experiment, the number of polygons in the two fish models varied between 4,100 and 4,600. We have tested our method when the sizes of the deformation region are 20% (only the fish tails are deforming) and 100% (the whole fishes are deforming). The computation times shown in the diagram include the updating and the inter-collision detection of the bounding hierarchies. From the measurements, our method is 31% (at frame 3) to 53% (at frame 7) faster than the AABB method when the size of the deformation region is 100%. When the size of the deformation region is reduced to 20%, our method is 2.8 times (at frame 3) to 1.2 times (at frame 7) faster than the AABB method.

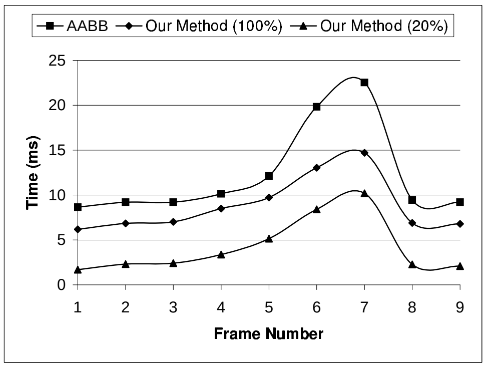

Figure 10: The performances of the AABB and our methods in detecting collisions when the fish models are deforming.

6.2 *Self-Collision Detection*

To analyze the efficiency of our self-collision detection method, we have studied the low-level operations involved and compared the complexity of our method with other relevant methods. The operations involved during run-time are shown as follows:

- *Deformation Updates:* As the surface deforms, we need to update the information of each node within the deformation region. This will require 24 **ADD** or **SUBTRACT** operations to update a bounding box and 4 **AND** operations to evaluate the bitfield.

- *Self-collision Test:* Given a test unit, we need to determine if the test unit is self-colliding. If it is a *Type 1* test unit, the cost is only 1 **IF** operation to check the bitfield. If it is a *Type 2* test unit, the cost is 1 **AND** and 1 **IF** operations. If it is a *Type 3* test unit, we need to perform a bounding box collision test. However, since all the bounding boxes of the object are already aligned to the same local coordinate system, there is no need to perform the axis alignment operations. Hence, in the worst case, the cost of a *Type 3* test unit is 6 **ADD** or **SUBTRACT** operations and 6 **COMPARISON** operations.

- *Patch Adjacency Test:* From our experiments, we note that we need to perform the patch adjacency test on approximately half of the test units. Each check involves 4 **IF**, 9 **COMPARISON**, 5 **ADD** or **SUBTRACT**, and 4 **BIT-SHIFT** operations.

We can see that most of the tasks involved in detecting self-collision are very efficient. The most expensive test is the processing of *Type 3* test units, which checks if two non-adjacent nodes are colliding. Figure 11 shows the algorithm complexity of our method as compared with two other methods [12, 22]. We can see that our method has a lower complexity on the patch adjacency test. It also requires to process less test units, due to the introduction of the self-collision list.

Operations	Our Method	Volino's Method [22]	Hughes' Method [12]
Vector direction sampling	O(n)	O(n)	N/A
Patch adjacency test	O(1)	O(log n)	O(log n)
Node self-collision test	O(1)	O(1)	$O(d! * m) + O(m^{d/2})$
Total test units processed	~ $^2/_3$A	A	A

n = Total number of quad-tree nodes
m = Number of vertices in a node
d = Dimension of the environment space
A = Total number of test units

Figure 11: The comparison of the complexity of different methods.

Figure 12 shows the performance of the self-collision process when applied to a deforming NURBS object. The object is a flat surface containing 7,000 to 8,500 polygons, depending on the surface curvature. During the experiment, the surface collides with itself four times. In the first, the third, and the fourth collisions, the surface simply touches itself and then bounces away. Hence, the computational costs involved during these three collisions are relative light as shown in the diagram. However, in the second collision, a large patch of the surface touches another patch of the same surface. This generates a large number of *Type 3* test units, resulting in a surge in computation time. During the collision period, the maximum number of test units found in the self-collision list in a frame is 24,000, while 96% (or 23,000) of the test units are of *Type 3*. (A video showing this self-collision experiment can also be found in "www.cs.cityu.edu.hk/∼rynson/projects/srender/srender.html".)

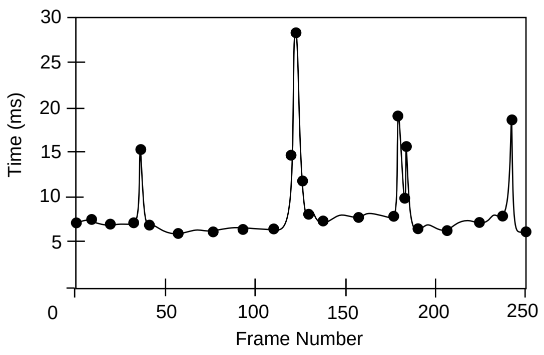

Figure 12: Time spent on self-collision detection for the testing video.

7 Conclusions

In this paper, we have proposed a collision detection framework for deformable objects. Our method has considered both inter-collision detection and self-collision detection. We have conducted a number of experiments on the proposed method. In inter-collision detection, results show that the performance of our method can be optimized by considering the deformation region of each deformable object.

In self-collision detection, we have analytically compared our method with two other self-collision detection methods [12, 22]. Results show that our method has a lower computation complexity than the other two methods. We have also demonstrated the performance of our method in detecting self-collisions of a deforming NURBS surface. In general, our self-collision detection method is very efficient most of the time, except when large surface patches of an object come into contact with each other. This will generate a large number of potentially intersecting non-adjacent bounding boxes, which are expensive to check for collision. We are currently investigating this issue.

Future work of this project includes extension of the method for any deformable parametric surfaces [15].

Acknowledgements

We would like to thank the anonymous reviewers for their useful comments on this paper. The work described in this paper was partially supported by a SRG grant from City University of Hong Kong (Project Number: 7001391).

8 References

[1] L. Balmelli, J. Kovačević, and M. Vetterli. Quadtrees for Embedded Surface Visualization: Constraints and Efficient Data Structures. In *Proc. of IEEE ICIP*, volume 2, pages 487–491, Oct. 1999.

[2] G. Bergen. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools*, **2**(4):1–13, 1997.

[3] J. Chim, R. Lau, A. Si, H. Leong, D. To, M. Green, and M. Lam. Multi-Resolution Model Transmission in Distributed Virtual Environments. In *Proc. of ACM VRST'98*, pages 25–34, 1998.

[4] K. Chung and W. Wang. Quick Collision Detection of Polytopes. In *Proc. of ACM VRST'96*, pages 125–132, 1996.

[5] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. I-Collide: an Interactive and Exact Collision Detection System for Large-scale Environments. In *Proc. of ACM Symp. on Interactive 3D Graphics*, pages 189–196, 1995.

[6] S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: a Hierarchical Structure for Rapid Interference Detection. In *Proc. of ACM SIGGRAPH'96*, pages 171–180, 1996.

[7] L. Guibas, D. Hsu, and L. Zhang. H-Walk: Hierarchical Distance Computation for Moving Convex Bodies. In *Proc. of ACM Symp. on Computational Geometry*, pages 265–273, 1999.

[8] T. He. Fast collision detection using QuOSPO trees. In *Proc. of ACM Symp. on Interactive 3D Graphics*, pages 55–62, 1999.

[9] M. Held, J. Klosowski, and J. Mitchell. Evaluation of Collision Detection Methods for Virtual Reality Fly-throughs. In *Proc. of Canadian Conf. on Computational Geometry*, pages 205–210, 1995.

[10] B. Herzen, A. Barr, and H. Zatz. Geometric Collisions for Time-Dependent Parametric Surfaces. In *Proc. of ACM SIGGRAPH'90*, pages 39–48, 1990.

[11] P. Hubbard. Approximating Polyhedra with Spheres for Time-Critical Collision Detection. *ACM Trans. on Graphics*, **15**(3):179–210, July 1996.

[12] M. Hughes, C. DiMattia, M. Lin, and D. Manocha. Efficient and Accurate Interference Detection for Polynomial Deformation. In *Proc. of Computer Animation Conference*, 1996.

[13] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Trans. on Visualization and Computer Graphics*, **4**(1):21–36, Jan. 1998.

[14] S. Krishnan, M. Gopi, M. Lin, D. Manocha, and A. Pattekar. Rapid and Accurate Contact Determination Between Spline Models using ShellTrees. In *Proc. of Eurographics'98*, pages 315–326, Sept. 1998.

[15] F. Li and R. Lau. Real-Time Rendering of Deformable Parametric Free-Form Surfaces. In *Proc. of ACM VRST'99*, pages 139–144, 1999.

[16] F. Li, R. Lau, and M. Green. Interactive Rendering of Deforming NURBS Surfaces. In *Proc. of Eurographics'97*, pages 47–56, Sept. 1997.

[17] T. Li and J. Chen. Incremental 3D Collision Detection with Hierarchical Data Structures. In *Proc. of ACM VRST'98*, pages 131–138, 1998.

[18] M. Lin and J. Canny. Efficient Algorithms for Incremental Distance Computation. In *IEEE Conf. on Robotics and Automation*, pages 1008–1014, 1991.

[19] M. Moore and J. Wilhelms. Collision Detection and Response for Computer Animation. In *Proc. of ACM SIGGRAPH'88*, pages 289–298, 1988.

[20] L. Piegl and W. Tiller. *The NURBS Book*. Springer-Verlag, 1995.

[21] M. Ponamgi, D. Manocha, and M. Lin. Incremental Algorithms for Collision Detection Between Polygonal Models. *IEEE Trans. on Visualization and Computer Graphics*, **3**(1):51–64, Jan. 1997.

[22] P. Volino and N. Magnenat-Thalmannhen. Efficient Self-Collision Detection on Smoothly Discretized Surface Animations Using Geometrical Shape Regularity. In *Proc. of Eurographics'94*, pages 155–166, Sept. 1994.

[23] Y. Yang and N. Thalmann. An Improved Algorithm for Colision Detection in Cloth Animation with Human Body. In *Proc. of Pacific Graphics Conference*, pages 237–251, 1993.