Haptic Interaction with a Polygon Mesh Reconstructed from Images

Xingzi Zhang
School of Computer Science and Engineering
Nanyang Technological University
Singapore
e-mail: *ZHAN0388@e.ntu.edu.sg*

Michael Goesele
TU Darmstadt
Germany
e-mail: *goesele@cs.tu-darmstadt.de*

Alexei Sourin
School of Computer Science and Engineering
Nanyang Technological University
Singapore
e-mail: *assourin@ntu.edu.sg*

Abstract*—*Multi-view reconstruction methods are able to produce polygon meshes of a complex scene with millions of triangles which are well suited for visualization purposes. However, these large-scale, dense meshes normally cannot be haptically rendered directly with readily available APIs. In this paper, we present a method to extend meshes reconstructed from images to visual-haptic applications by using images for visual display while an improved hybrid collision detection method is used to meet the real-time requirements for haptic rendering. Moreover, three main imperfections (holes, outliers and degenerated facets) inherent in reconstructed models are also handled to help provide a smooth and consistent force feedback. Given a reconstructed model and the corresponding image, the proposed method provides a way to virtually explore the scene in the image, making it possible to appreciate art works and historic monuments within a touching distance.

Keywords-haptic interaction; image-based visualization; imperfect polygon mesh

I. INTRODUCTION

Multi-view reconstruction methods such as [16] are able to produce polygon meshes of a complex scene which are sufficient for visualization. However, because of imperfections inherent in passive computer vision techniques, meshes reconstructed in this way usually contain holes, bumpy surfaces and outliers, which cannot be handled easily in haptic interaction. Besides, these meshes often have more than one million triangles, which are unevenly distributed based on the image content. For example, if there are several close-up photos of a part of the scene, it is likely to have more details and thus more polygons in this region, resulting in a partially dense model. Collision detection with a large-scale, dense mesh is a challenging task in haptic interaction which requires a high update rate (1 kHz). Simplifying the mesh by decimation can accelerate the collision detection process, but at the cost of losing geometry details on the reconstructed surface. For applications which require simulated tangible perception that exactly matches the displayed geometry, it is necessary to find a haptic rendering algorithm which can maintain both the visual-haptic interactive speed and the accuracy of the haptic feedback.

Interactive rendering of large-size meshes is problematic since the combined visual and haptic rendering time for the meshes would significantly increase as the number of polygons goes up and reaches the limit of the graphics renderer. Although mesh simplification methods and acceleration techniques can effectively reduce the rendering time, image-based visualization is a better choice instead of photorealistically rendering a scene with millions of polygons.

Therefore, to be able to interact with complex models reconstructed from images, we propose to use images to replace visual display of these models while still aligning them with the respective parts of the images for haptic interaction. In applications, such as interactive panoramas and virtual street walkthroughs, we believe this could provide a way to immerse in a visual-haptic environment with models of original high resolution. An improved hybrid collision detection method which combines precomputed connectivity information and spatial partitioning is also proposed in this paper. It is suitable for fast collision detection and is able to handle the artificial elements inside the meshes. In Section II, we survey the relevant works. In Section III, the main algorithms and system architecture are described. Comparison with a few relevant approaches is done in Section IV, followed by the conclusion in Section V.

II. RELATED WORK

Regarding the features of reconstructed meshes, two main problems that we encounter in the interaction are (i) how to achieve fast and accurate collision detection and (ii) how to provide photorealistic visual feedback for the interaction.

A. Point-based Collision Detection with Polygon Meshes

Many methods proposed for haptic rendering of polygon meshes so far detect collision with the whole polygon mesh

in each haptic frame. The haptic rendering time thus depends on the number of polygons. For example, in widely-used haptic rendering methods such as God-Object [1], Ruspini [2] and CHAI3D [3], active constraint polygons need to be found first from all the polygons in each haptic frame, and then the constraint polygon with the shortest distance to haptic cursor is determined as the contact polygon. OpenHaptics HLAPI [4] utilizes OpenGL Depth Buffer and Feedback Buffer in a smart way and can automatically detect collision with shapes rendered in graphics based on the shape geometry and depth information. In this way, Depth Buffer and Feedback Buffer's performance are not influenced by the size of polygons. However, the Feedback Buffer has a limited size (storing up to 65536 vertices) and using the Depth Buffer results in discontinuities in the computed haptic force due to the fact that 3D geometry is saved as an image in the Depth Buffer.

There are a number of methods that have been proposed to reduce the computational time using spatial partitioning and hierarchical structures, such as H-COLLIDE [5] and ActivePolygon [6]. In the ActivePolygon algorithm, polygons are stored in an octree data structure. Only the polygons stored in the cells that the haptic cursor passes by between frames are used for collision detection. These methods could effectively reduce the haptic rendering time, however, they could not handle the situation when the mesh is too dense, because the computation complexity of these algorithms depends on the number of polygons in the cells that the haptic cursor passes from frame-to-frame. Thus, if the haptic cursor moves very fast and passes several cells within one cycle, only the first cell (obtained from the cursor position in last frame) and the last cell (obtained from the cursor position in current frame) are known while the in-between cell information is lost. To avoid leaving out the actual contact polygon, all the cells that the cursor might pass need to be considered and this would lead to a significant expansion in the searching range, even if when the cell size is optimized. For example, in Geomagic Touch its maximum velocity is 2.5 mm/ms, so all the polygons in those cells within the distance of 2.5 mm to the previous position of haptic cursor need to be checked. If the mesh is dense and has a few hundred polygons within a 2.5 mm cubic space, fast and accurate collision detection cannot be maintained.

Geometry connectivity information was first used by Chih-Hao Ho et al. in their "neighborhood watch" algorithm [7] to predict the next active primitive based on the previous active primitive. Here, active primitive refers to the primitive that the haptic cursor is in contact with. Before haptic rendering, the connectivities among vertices, lines and polygons of the mesh are predefined and stored. After the first collision is detected, only the neighbors of the previous contact primitive are checked. Using an iterative approach one can track the trace of haptic cursor and find the closest primitive at the current position. In this way, the haptic rendering time is independent of the number of polygons except for every first collision with the mesh.

As discussed above, spatial partitioning cannot handle interactive collision detection with a large mesh. Despite this limitation, it is suitable to be used for checking whether the haptic cursor is inside or outside the mesh since only polygons within one cell need to be checked. Inspired by the "neighborhood watch" algorithm [7], we propose a hybrid collision detection method which combines precomputed connectivity information and spatial partitioning. In this way, the computational time is fully independent of the polygon number. One of the main differences between our proposed method and Chih-Hao Ho's method is that we are not dealing with perfect CAD polygon meshes. The geometry information obtained from the meshes can be incomplete, may contain redundant vertices and facets, or may even be wrong. A more general criteria for searching for the active primitive is thus needed.

B. Visual Rendering in Visual-haptic Interaction Environment

In a visual-haptic interactive scene, polygon meshes, as well as the haptic cursor, are usually displayed for visual feedback. While GPUs can process millions of polygons interactively for visual rendering, displaying the haptic cursor at the same time is problematic since the haptic cursor position is computed by the CPU (together with other haptic rendering tasks) at the rate of 1 kHz. As the combined rendering time increases with increasing mesh size, the movement of the haptic cursor in the scene slows down and would finally become clumsy. To reduce the graphics rendering time for visual models, we need to either speed up the rendering process or to reduce the size of the models.

Common graphical renderers in visual-haptic interaction, such as OpenGL and Direct3D, utilize rasterization-based rendering due to the real-time requirement. With powerful graphics hardware and the use of acceleration structures for culling, a complex interactive scene can be rendered in real-time. However, the rendering effect of a rasterization-based rendering algorithm heavily depends on the lighting techniques applied to the scene and the manual effort of designers, which poses an obstacle to realistic immersion. Compared to rasterization-based algorithms, ray tracing provides a more realistic visual effect, but it is costly in computation. With the emergence of high-performance rendering engines like Brigade [8], it has become possible to incorporate ray tracing into real-time rendering. However, it is still far from being applied in interactive visual-haptic scenes with millions of polygons.

In order to display a more realistic scene, there are works which combine ray tracing with rasterization-based rendering in visual-haptic interaction environment. For example, Morris and Joshi propose to display pre-processed raytraced images to simulate a static-viewpoint scene [9]. Depth information is extracted here along with the image for proper occlusion with other objects rendered in real-time. In this way, large computation is avoided to be done in the rendering loop and realism is improved.

Based on the examples illustrated in [9], we know that images can be a promising alternative to displaying the models in some real-time applications. Previously, image-based visualization is mainly used in haptic interaction where images are the interactive objects. Images are displayed as a guidance to the perceivable contours and textures of the objects, and the force feedback is generated based on the image properties such as grayscale or color values of the pixels [10, 11]. To allow the users to perceive the whole geometry of the object in the image, some works augment images with haptic models which match the image content. The augmented models can be simplified geometry

models, depth maps, or even mathematical functions and procedures [12-14].

Since our goal is to interact with complex meshes reconstructed from multi-view reconstruction methods such as MVE [16], corresponding images to the mesh are readily available and camera parameters are already reconstructed in the reconstruction pipeline using structure-from-motion algorithms. Therefore, image-based visualization is an ideal choice in this case. Using the reconstructed camera parameters, the haptic display is easily registered with the image and a high-resolution visual feedback can be achieved without complex computations.

III. System Overview

A. Hybrid Collision Detection Algorithm

One of the main goals for this paper is to find an algorithm that is suitable for handling collision detection with large-scale, dense reconstructed meshes. Inspired by [7], we incorporate spatial partitioning into the "neighborhood watch" algorithm, narrowing down the detection range to triangles within one cell to get the first active polygon and then using this polygon as starting point to track the next active primitive. A mesh has three kinds of primitive types: vertices, line segments and polygons. We consider a primitive active if the projection of the haptic cursor on the mesh is right inside or on this primitive.

Before applying this collision detection algorithm, there are three preprocessing steps. Reconstructed models are likely to have duplicate vertices, e.g., the city wall model used in our experiments has 1883 groups of duplicate vertices. Our first preprocessing step is to delete the duplicate vertices and zero-area polygons in the mesh, if any. All the vertices are traversed to form a list of duplicate vertex groups, and in each group the vertex with the smallest index is considered as effective while the others are deemed duplicates. Then, the polygons with duplicate vertices are divided into two groups. Those with two or more duplicate vertices from the same group (i.e. zero-area facets) are deleted directly, while the others have their duplicate vertices replaced by the effective vertices of the same group. For example, suppose p_1, p_2, p_3 are overlapped vertices and vertices p_4, p_5 don't overlap with each other. Since p_1 has the smallest index in this overlapped vertex group, it is an effective vertex. For triangle facet $\triangle p_1p_2p_4$, we should delete it because p_1 overlaps with p_2, while for triangle facet $\triangle p_2p_4p_5$, we should replace p_2 with p_1.

After removing all the duplicate vertices and zero-area polygons, the second step is to build the connectivities among vertices, line segments and polygons and store all the neighbors for each primitive. We define the neighbors for the three primitive types referring to the definitions in [7]: for a polygon, the neighbors are its line and vertex components; for a vertex and a line, the definition of neighbors is extended to all the polygons connected to it and all the lines and vertices that comprise these polygons. Fig. 1 illustrates an example of how neighbors are defined for a vertex, a line segment and a polygon.

Based on the connectivities between the vertices and polygons, the vertex normals are recalculated by summing up the weighted normal of the neighboring polygons and normalizing the sum [15] as in (1).

$$n_v = \frac{\sum_i \alpha_i * n_{f,i}}{\left\| \sum_i \alpha_i * n_{f,i} \right\|} \quad (1)$$

Here, the weight is each neighboring polygon's inner angle at this vertex. Besides, we also check and store whether a line is on a convex or concave surface. The lines with only one adjacent polygon are marked as edges. These lines may be the edges of the outer contour or the edges of holes on the surface of the mesh.

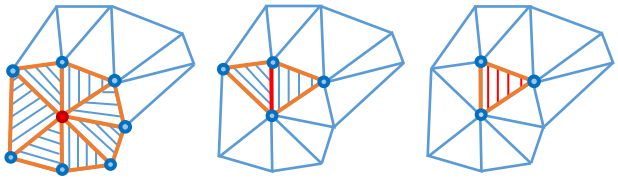


Figure 1. A neighboring vertex is marked as a small circle, a neighboring line segment is marked in orange color, and a neighboring polygon is marked with stripes. (a) The red vertex has 7 neighboring polygons, 14 neighboring line segments and 7 neighboring vertices. (b) The red line segment has 2 neighboring polygons, 4 neighboring line segments and 4 neighboring vertices. (c) The red polygon has 3 neighboring line segments and 3 neighboring vertices.

In the final preprocessing step, we apply a uniform partition to the space within the bounding box of the polygon mesh and divide this space into cells. A polygon is considered as belonging to one cell if the polygon has one or more vertices in this cell, the polygon has edges intersecting with the bounding box of this cell or the bounding box of this cell intersects with the polygon. The criteria is the same as that in [6]. The size of the cell is determined by the size of the bounding box and the density of the model. To guarantee the detection accuracy, it should not be smaller than the maximum distance that the haptic cursor can move in the camera frustum in one frame. Therefore, the way that the model and the haptic cursor are mapped into camera workspace would also have an impact on the cell size determination.

After all three preprocessing steps, the collision detection phase starts. The whole process is illustrated as pseudocode in Fig. 2(a). It can be divided into two parts based on the collision detection status in the last frame. If there is no collision detected in the last frame (i.e. *collision == FALSE*), then we follow the procedures below to check whether the haptic cursor (Haptic Interface Point, HIP) collides with the mesh in this frame. First, we check whether the HIP is inside the bounding box of the mesh. If it is so, since the bounding box has been split into a grid, we continue to find out in which cell the HIP is contained. After that, we proceed to check whether the HIP collides with the polygons within this cell. If any of the polygons in this cell has a positive distance to the previous HIP and a negative distance to the current HIP, it means that the path of the HIP intersects with this polygon (i.e. collision happens). The collided polygon is then tagged as an active primitive and the collision detection status is set to *TRUE* (i.e. *collision = TRUE*). The computation complexity of this part is only related to the number of polygons in the cell and is independent of the size of the mesh.

Correspondingly, if the HIP was in collision with the mesh in the last frame (i.e. collision == TRUE), then what we need to check is whether the HIP is still inside the mesh in this frame. Here, our strategy is to first find out which

primitive is active in this frame and then check whether the HIP is behind this primitive seen from the camera. If the HIP is underneath this primitive (i.e. the HIP has a negative distance to this primitive), it means the HIP is still in contact with the mesh. The procedures for finding the active primitive in this frame are implemented in the repeat until loop operation: with the active primitive in the last frame as the starting point, the neighbors of the previous active primitive and itself as well are checked to find the new active primitive in each iteration. If the new active primitive is the same as the previous one, then it would be considered as the active primitive in this frame. Otherwise, the loop continues with the obtained new active primitive assigned as previous active primitive in the next iteration. After the active primitive is found through iterations, the projection of the HIP on the active primitive is assigned as the proxy. The dot product between the vector from the current HIP to the computed proxy position and the normal of the active primitive is calculated to determine whether the cursor is inside or outside. The new collision detection status would be saved for the next frame.

Algorithm 1 Algorithm for collision detection

```
if collision == FALSE then
  find the Cell that HIP is in
  for all polygons in this Cell do
    if the path of HIP passes polygon_j then
      collision ← TRUE
      activeprimitive ← polygon_j
    end if
  end for
else
  pos ← the position of the HIP
  repeat
    activeprimitive_prior ← activeprimitive
    set A = {activeprimitive_prior, neighbors of activeprimitive_prior}
    activeprimitive = FindActiveNeighbours(A, pos)
  until activeprimitive == activeprimitive_prior

  vector ← vector from current HIP to proxy
  normal ← the normal of activeprimitive
  if vector * normal < 0 then
    collision ← FALSE
  else
    collision ← TRUE
  end if
end if
```

(a)

Algorithm 2 activeprimitive = FindActiveNeighbours(A,pos)

```
for all polygons ∈ A,
if there exists a polygon which has the projection of HIP onto it inside its
range then
  distmin = min{|dist_i| :the projection of HIP on polygon_i plane is inside
  polygon_i, polygon_i ∈ A}
  return polygon with distmin to the HIP
else
  for all line segments and vertices ∈ A,
  if min{dist_i : line_i ∈ A} < min{dist_j : vertex_j ∈ A} then
    distmin = min{dist_i : line_i ∈ A}
    return line segment with distmin to the HIP
  else
    distmin = min{dist_j : vertex_j ∈ A}
    return vertex with distmin to the HIP
  end if
end if
```

(b)

Figure 2. (a) Algorithm for collision detection. (b) Algorithm for obtaining the active primitive in the neighborhood.

In Ho et al. [7], the criteria for determining the active primitive in the neighborhood are based on the distance. The primitive with the shortest distance to the cursor is considered as the active primitive. However, there can be two different primitives which have the same distance towards one point at the same time, leading to an unclear case. For example, suppose we have a piece of mesh like the one shown in Fig. 3. Point *P* is the current HIP and the previous active primitive is line segment *AB*. According to the neighbor definition and distance criteria described in [7], when we search for an active primitive in line segment *AB*'s neighbors, polygon △*ABD* would be chosen as an active primitive. Since none of its neighbors (line segments *AB*, *AD*, *BD* and vertices *A*, *B*, *D*) have a shorter distance to point *P* compared to it, it would still be the active primitive in the next iteration. Then in the next iteration the new active primitive would be the same as the previous one, and it would lead to the result that polygon △*ABD* is the active primitive in this frame, which is incorrect.

To resolve the ambiguity in the distance criteria, as illustrated in Fig. 2(b), we expand the definition of the neighbors and use the distance combined with the orthogonal projection of HIP on the active primitive as criteria for iteration. We go through the neighbors with the following sequence. First, all the neighboring polygons are checked to find whether there is any polygon that has the projection of the HIP onto it within its range. If such polygon exists, then the polygon that fulfills the condition and has the smallest distance to the HIP would be returned. If no such polygon exists, we proceed to compare the distances between the HIP and the other neighbors (line segments and vertices). The neighbor with the smallest distance to the HIP is then the return value. If our criteria is applied to the case as in Fig. 3, the iteration result is very clear. With line segment *AB* as the active primitive in the last frame, the iteration result would go in this way: *AB*→*AD*→△*ADE*.

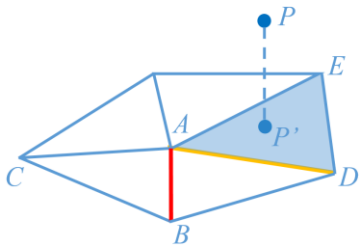


Figure 3. An example to illustrate difference between the distance criteria and our criteria. The projection of *P* onto △*ABD* is *P'*.

B. *Force Rendering*

In our paper, we assume that the interactive models are hard and stiff objects, therefore we apply proxy-based haptic rendering: we compute a proxy to represent the haptic cursor so that the cursor is always visible. When the HIP is moving in free space, the position of the proxy matches the HIP. When there is a collision, the active primitive is known and the proxy is assigned as the projection of the HIP on the active primitive.

We use a simple spring force model. The magnitude of the force feedback is proportional to the penetration depth of the HIP into the active primitive, which is exactly the *distmin* that we obtain in the iteration loop of Algorithm 2. Normally, the force is computed in the same direction as the facet normal. In our method, we use this approach if the active primitive is a polygon. When the active primitive is a line segment, the force is applied along the direction opposite to the movement which is from the proxy to the HIP position. In this way, we can effectively prevent the haptic cursor from crossing the edges. Thus, if the cursor

slides to a hole on the mesh, it would not fall into the hole. The disadvantage of this strategy is that if the cursor slides along a ragged edge, there are frequent changes in the force direction, since we always give the cursor a resistant force perpendicular to the edge. If the force direction is in the same direction as the velocity, this may lead to a cursor jump.

C. Coordinate System Alignment

When using images to replace visual display of the meshes, we need to align the haptic models with the respective parts of the images so that the image content matches the haptic display. In a multi-view reconstruction process, camera parameters of the images can be estimated based on structure-from-motion techniques. Therefore, given a target image and corresponding reconstructed model, the estimated camera parameters could be used to calculate the modelview and projection matrices for projecting the model in the camera frustum. Suppose R_C is the orientation matrix of the virtual camera with respect to the world coordinate system, T_C is the column vector which defines the location of the virtual camera in the world coordinate system, f is the focal length of the camera, *img_width* and *img_height* are the width and the height of the given image, pp_x and pp_y are x, y coordinates of the principal point offset of the camera in pixel coordinate system, z_{near} and z_{far} are z coordinates of the near and far clipping planes, then the 4*4 modelview and projection matrices M_{mol} and M_{proj} can be obtained as follows:

$$M_{mol} = \begin{pmatrix} R_C & T_C \\ 0 & 1 \end{pmatrix} \tag{2}$$

$$M_{proj} = \begin{pmatrix} 2f\alpha_x & 0 & 2(pp_x - 0.5) & 0 \\ 0 & 2f\alpha_y & 2(pp_y - 0.5) & 0 \\ 0 & 0 & \frac{z_{far}+z_{near}}{z_{far}-z_{near}} & \frac{-2\, z_{far}\, z_{near}}{z_{far}-z_{near}} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3}$$

$$aspect = img\_width / img\_height \tag{4}$$

$$\alpha_x = \begin{cases} 1, & if\ aspect > 1 \\ 1/aspect, & if\ aspect \le 1 \end{cases} \tag{5}$$

$$\alpha_y = \begin{cases} aspect, & if\ aspect > 1 \\ 1, & if\ aspect \le 1 \end{cases} \tag{6}$$

The example models used in this paper are produced by the MVE [16]. Note that the origin of the pixel coordinate system of MVE is at the top-left corner of the image while that in OpenGL is at the bottom-left corner of the image, therefore when displaying MVE models in OpenGL, the y-axis needs to be inverted to match the image. This could be done by inverting all elements in the second row of either M_{mol} or M_{proj}.

There are three workspaces involved in the visual-haptic interaction: the camera workspace (defined during the structure-from-motion process), the haptic workspace, and the world coordinate system. The whole mapping and transformation process behind the interaction scene is illustrated in the flowchart in Fig. 4. The procedures enclosed by the blue dashed lines are for visual rendering. In real 3D scenes, the haptic cursor would be hidden when moving to the back of the objects. To simulate such occlusion effect with displaying only 2D images, we write the reconstructed models to the depth buffer and then disable writing to the depth buffer right after the writing operation. The depth buffer writing is kept disabled in the following rendering loop. Afterwards, with depth test enabled and *glDepthFunc* depth comparison function set to GL_LEQUAL, the depth values of the models rendered in real-time (e.g., haptic cursor) are compared with the depth values stored in the depth buffer. A pixel of the haptic cursor is only drawn if the incoming depth value at this pixel is less than or equal to the stored depth value. In such a way, if the haptic cursor goes to the back of the reconstructed model (i.e. the incoming depth value is greater than the stored depth value), it is not drawn and the occlusion effect is thus achieved.

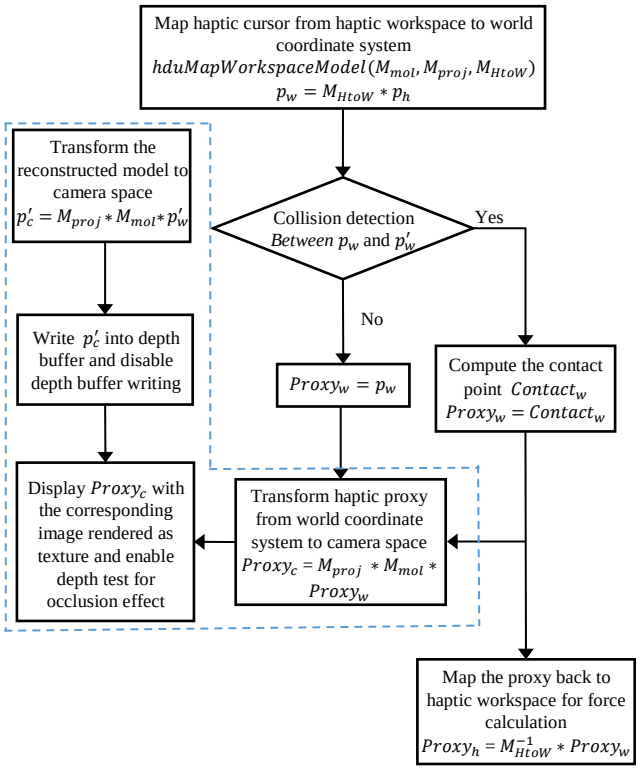


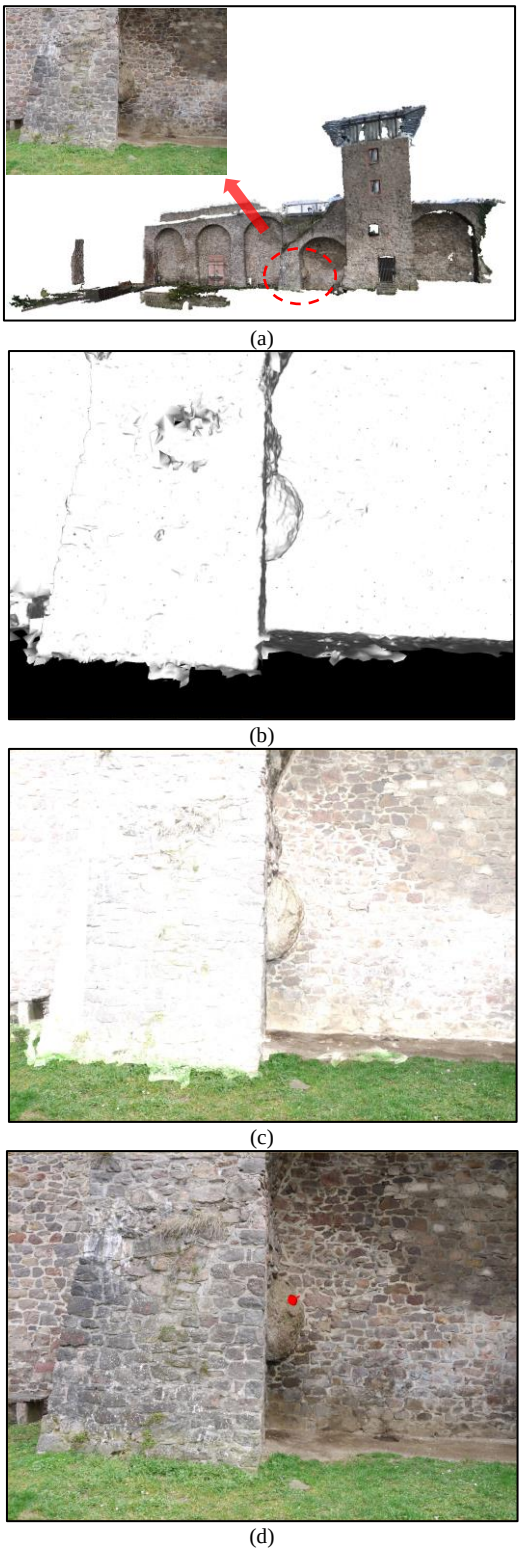
Figure 4. Flowchart of the mapping process.

In the haptic servo loop thread, the position of the haptic cursor is mapped to the world coordinate system for collision detection and then mapped back to the haptic workspace for force rendering if the collision happens. The generated proxy position is transformed to the camera workspace and sent to the client thread for displaying.

D. Examples

The images in Fig. 5 illustrate how the concepts introduced in the previous three sub-sections are implemented given a reconstructed model. Fig. 5(a) shows the original reconstructed city wall model, while the small image in the left upper corner is the image to be used for visual display in the interactive scene. Based on the reconstructed camera parameters of this image, we transform the model to the camera workspace and obtain the part in Fig. 5(b) after clipping. We can see that the clipped model matches with the content of the image (Fig. 5(c)). After alignment, the haptic cursor is able to interact with the

city wall in the image as displayed in Fig. 5(d). The red ball in Fig. 5(d) represents the proxy of the haptic cursor. A red line pointing to the normal direction is also shown, indicating that the cursor is in contact with the model now.



(a)

(b)

(c)

(d)

Figure 5. (a) the original reconstructed model. (b) the transformed model displayed in simulated camera frustum. (c) the alignment of the transformed model and the image. (d) a snapshot of the interactive scene.

The examples of haptic interaction with the models reconstructed from images (Fig. 6) can be seen in the companion video, which is also available at https://youtu.be/6_tHrG9q3H8.

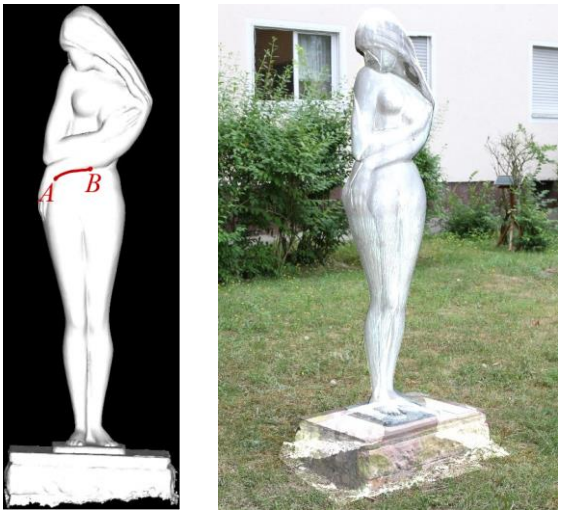
Figure 6. Examples of interactions with the models reconstructed from images.

IV. COMPARISON AND EVALUATION

In this section, we compare our haptic rendering algorithm with God-object renderer [1] provided by H3D API and OpenHaptics HLAPI [4] using feedback buffer. These two renderers are selected because they are commonly used in haptic interaction with meshes. Besides, we present the performance of our algorithm under two conditions: one where meshes are displayed for visualization and another one where image-based visualization is applied.

In the comparison experiments with the other two haptic rendering algorithms, we display simplified versions of a reconstructed statue model (Fig. 7(a)) for visual feedback: one has 51766 triangles (around 52k) and another one has 103534 triangles (around 103k). We did not use the original model (around 9.5 million triangles), because both HLAPI feedback buffer and God-object renderer could not handle collision detection with such large meshes. In our experiments, God-object renderer works with the default collision detection algorithm provided by HAPI (a library

of H3D API) and Oriented Bounding Box of the model is created for broad-phase collision detection inside.



(a) (b)

Figure 7. (a) The target model is displayed using OpenGL in the comparison experiment. (b) The image is shown instead of displaying the model.

Given the same mesh, the comparison is done in terms of average combined rendering time, the smoothness of the cursor movement, and whether the algorithm works as intended. To make sure that the combined rendering time is recorded under the same conditions, we use the same renderer (OpenGL) for displaying and move the haptic cursor along the same path *AB* as shown in Fig. 7. Path *AB* is chosen because it is a depression curve on the surface of the model and thus it is easy to follow. Here, the combined rendering time refers to the time for rendering the proxy of the haptic cursor and the visual medium (meshes or images) in the graphics loop. Since the position of the proxy is calculated at the haptic update rate, the haptic rendering performance would also have an influence on the combined rendering time. Therefore, the combined rendering time is determined by both visual and haptic rendering performance. In our experiments, the combined rendering time is collected in display function of each frame while the cursor sliding from *A* to *B* and then an average is computed for comparison. As for the second comparison item, the smoothness of the movement, it is a visual reflection of the combined rendering time. When the combined rendering time exceeds 1 ms, the movement of the cursor would not be smooth any more. Note that the combined rendering time and the smoothness of the movement only make sense if we are able to touch the model (i.e. the third comparison item). If the model is not touchable, we consider that this algorithm as failed.

Based on the results shown in Table I, HLAPI feedback buffer gives the worst performance on combined rendering time among the three algorithms. It also reflects visually on the cursor movement, which is why the cursor could not move very smoothly when using HLAPI. We should note that HLAPI feedback buffer normally is able to handle a mesh of 52k triangles. The reason why it fails here is that the interactive model is partially dense. Specifically, the average amount of triangles within a 0.5*0.5*0.5 cubic space along line *AB* in the 52k statue mesh is more than 50. God-object renderer shows rather good results with the smaller mesh, but it could not work properly with the bigger mesh, so the obtained combined rendering time is not comparable. Comparing to these two rendering algorithms, our algorithm can work with both of the two simplified models and provide effective results.

TABLE I. The results of the performance comparison.

Renderer	Mesh size (T)	Average combined rendering time (ms)	Movement	Touchable
God-object	Statue (52k)	0.012	Smooth	Yes
HLAPI	Statue (52k)	80	Less smooth	Yes (not in real-time)
Ours (mesh)	Statue (52k)	0.001	Smooth	Yes
Ours (image)	Statue (52k)	0.0098	Smooth	Yes
God-object	Statue (103k)	0.011	Smooth	No
HLAPI	Statue (103k)	130	Less smooth	Yes (not in real-time)
Ours (mesh)	Statue (103k)	0.0093	Smooth	Yes
Ours (image)	Statue (103k)	0.0092	Smooth	Yes
Ours (mesh)	Statue (9506k)	330	Clumsy	Yes (not in real-time)
Ours (image)	Statue (9506k)	0.6	Smooth	Yes
Ours (mesh)	City wall (17600k)	820	Clumsy	Yes (not in real-time)
Ours (image)	City wall (17600k)	0.017	Smooth	Yes

(a) (b)

Figure 8. (a) the city wall image used in the test, with path AB marked red. (b) the corresponding model.

The results of our algorithm working with two different means of visualization (mesh-based and image-based) are also shown in Table I. When the mesh is small, the difference is very subtle. Take the statue mesh of 103k triangles as an example. The average combined rendering time of our algorithm with mesh-based and image-based visualization are 9.3E-6 and 9.2E-6 seconds respectively. Both of the two times are on microsecond level. Considering the unavoidable impact of noise, we can say that they are almost the same. However, when it comes to the statue model of the original size (around 9.5 million triangles), if we display meshes in the graphics loop, the cursor's movement would become clumsy. On the contrary, if we display the image instead, our algorithm would still work, with the combined rendering time below 0.001 second. The largest model that we test with our algorithm is the city wall model shown in Fig. 5(a) with 17.6 million triangles. Fig. 8(a) is the image of the city wall that we use in the test. As shown in Table I, with image-based visualization the average combined rendering time is largely cut off and we are able to slide along path AB (marked red in Fig 8(a)) smoothly.

If we check the last and the third items from the last row of Table I, we find that, despite the fact that the original statue model is only half size of the city wall model, its average combined rendering time is longer than that of the city wall model. This is due to the selection of image for display. With different images selected, the camera workspace, the haptic workspace, and the world coordinate system are aligned in a different way. This would has an impact on the haptic rendering performance of our algorithm and lead to unsatisfactory result with some images. Fig. 9(a) is such an example. As we can see, the depth range (i.e. the distance between near and far clipping planes) of the frustum defined by Fig. 9(a) is long. Since the haptic workspace is fixed, it means when the haptic cursor moves one unit distance in haptic workspace, it corresponds to at least several unit distances in camera workspace. In this way, more computations are required to track the active primitive, and this results in many pop-through effects when the cursor moves fast.

(a) (b)

Figure 9. (a) an image with which our algorithm fails. (b) the corresponding model.

V. CONCLUSION

We have presented our approach to creating a visual-haptic interactive environment with multi-view reconstructed models. To deal with large-size, partially dense reconstructed meshes, we propose an improved hybrid collision detection method. By preprocessing the mesh with uniform partitioning and building connectivities among the vertices, lines and polygons, we are able to handle collision detection with meshes of over ten million triangles. In this approach, instead of visually displaying the mesh, image-based visualization is applied to meet the requirements of realism and high interactive speed. We align the haptic models with the images so that the haptic display would match the visual content. Occlusion of the haptic cursor is simulated as if it was interacting with a real 3D scene.

With the presented method, we add a new modality into interaction with images. Besides viewing an image, this method enables us to appreciate the image content within a touching distance and complements our viewing experience. Potentially, it could also be combined with interactive applications such as virtual street navigation and virtual shopping, providing additional information for the users. Currently, only stiff objects are simulated in this paper. By incorporating soft objects in the future, the online shoppers would be able to feel the texture and the softness of the sofa from a sofa picture.

ACKNOWLEDGMENT

This research is supported by the National Research Foundation, Prime Minster's Office, Singapore under its International Research Centers in Singapore Funding Initiative, joint PhD Degree Program NTU-TU Darmstadt, and MOE Singapore Funding RG17/15 "Haptic Interaction with Images and Videos".

REFERENCES

[1] C. B. Zilles and J. K. Salisbury, "A constraint-based god-object method for haptic display," in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1995, pp. 146-151, doi: 10.1109/IROS.1995.525876.

[2] D. C. Ruspini, K. Kolarov, and O. Khatib, "The haptic display of complex graphical environments," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997, pp. 345-352, doi: 10.1145/258734.258878.

[3] *CHAI 3D*. (2014). Available: http://www.chai3d.org/index.html

[4] Geomagic. (2016). *OpenHaptic Toolkit Overview*. Available: http://www.geomagic.com/en/products/open-haptics/overview

[5] A. Gregory, M. C. Arthur, et al. "A framework for fast and accurate collision detection for haptic interaction," *ACM SIGGRAPH 2005 Courses*, No. 34, 2005, doi:10.1145/1198555.1198604.

[6] T. Anderson and N. Brown, "The activepolygon polygonal algorithm for haptic force generation," in *Proceedings of the sixth PHANToM Users Group Workshop*, USA, October 27-30, 2001.

[7] C.H. Ho, C. Basdogan, and M. A. Srinivasan, "Efficient point-based rendering techniques for haptic display of virtual objects," *Presence*, vol. 8(5), 1999, pp. 477-491, doi:10.1162/105474699566413.

[8] OTOY. Brigade. (2016). Available: https://home.otoy.com/render/brigade/.

[9] D. Morris and N. Joshi, "Hybrid rendering for interactive virtual scenes," *Stanford University Technical Report CSTR*, 2006, vol. 6.

[10] J. Li, A. Song, and X. Zhang, "Image-based haptic texture rendering," in *Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and its Applications in Industry*, 2010, pp. 237-242, doi:10.1145/1900179.1900230.

[11] H. Vasudevan and M. Manivannan, "Tangible images: runtime generation of haptic textures from images," in *haptics symposium on Haptic interfaces for virtual environment and teleoperator systems*, 2008, pp. 357-360, doi: 10.1109/HAPTICS.2008.4479971.

[12] M. A. Otaduy, N. Jain, A. Sud, and M. C. Lin, "Haptic display of interaction between textured models," in *Visualization IEEE*, 2004, pp. 297-304.

[13] S. Rasool and A. Sourin, "Tangible images," in *SIGGRAPH Asia 2011 Sketches*, 2011, p. 41, doi:10.1145/2077378.2077430.

[14] S. Rasool and A. Sourin, "Towards Tangible Images and Video in Cyberworlds--Function-Based Approach," in *International Conference on Cyberworlds (CW)*, 2010, pp. 92-96, doi: 10.1109/CW.2010.19.

[15] G. Thürrner and C.A. Wüthrich, "Computing vertex normals from polygonal facets," *Journal of Graphics Tools*, vol. 3(1), 1998, pp.43-46, doi:10.1080/10867651.1998.10487487.

[16] S. Fuhrmann, F. Langguth, N. Moehrle, M. Waechter and M. Goesele, "MVE – An image-based reconstruction environment," in *Computer&Graphics*, vol. 53, 2015, pp. 44-53, doi: 10.1016/j.cag.2015.09.003.